Udacity Machine Learning Engineer Nanodegree: Capstone Project Report

Date: 02/05/2020
Author: Steven Vuong

Project Overview, Problem Statement, Metrics & Benchmark

For Udacity's Machine Learning Engineer Nanodegree Capstone Project, I have opted to challenge myself by tackling a Kaggle competition. Specifically, "Predict Future Sales" (https://www.kaggle.com/c/competitive-data-science-predict-future-sales/overview). A snippet from the description tells us 'This competition also serves as the final project for the "How to win a data science competition"'.

This challenge and dataset involved specifically requires time series forecasting. Essentially, using past data to try and "predict the future". In the context of a company trying to predict sales, this is very important as it can assist company executives and decision-makers make better informed high level business decisions. With the example of stores, aid guidance on where to invest or cut back spending in order to make optimisations that can benefit the store as a whole. With new techniques from Machine Learning research, we can use powerful models to enable forecasting. Below are some academic references to having used machine learning techniques for time series forecasting:

- 25 years of research into time series forecasting: https://www.sciencedirect.com/science/article/abs/pii/S0169207006000021
- Time series forecasting using Hybrid ARIMA model: https://www.sciencedirect.com/science/article/abs/pii/S0925231201007020
- Neural Network forecasting for seasonal and trend forecasting: https://www.sciencedirect.com/science/article/abs/pii/S0377221703005484
- Sales Forecasting using neural networks: https://ieeexplore.ieee.org/abstract/document/614234
- Linear, Machine learning and probabilistic approaches to time series forecasting: https://ieeexplore.ieee.org/abstract/document/7583582

The dataset for this challenge can be found here:
https://www.kaggle.com/c/competitive-data-science-predict-future-sales/data
This data is kindly provided by one of the largest Russian software firms - 1C Company, with a granularity of recordings daily from the period of January 2013 to December 2015.

There are multiple CSV files with sales data of stores and items sold with multiple data fields. We have performed exploratory data analysis on the datasets provided to investigate (and visualise) the data, identify anomalies and deal with them, as well as engineer features for modelling. To reiterate, we want to predict future sales numbers for items sold in store. Data files and fields are listed below.

Data Files:

- sales_train.csv - the training set. Daily historical data from January 2013 to October 2015.
- test.csv - the test set. It is required to forecast the sales for these shops and products for November 2015.
- sample_submission.csv - a sample submission file in the correct format.
- items.csv - supplemental information about the items/products.
- item_categories.csv - supplemental information about the items categories.

- Shops.csv - supplemental information about the shops.

Data Fields:

- 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

In our solution, we have used all data features, split into training, validation and testing (after preprocessing and feature engineering).

For an official problem statement, it can be said that we want to predict total sales for every product and store in the next month, given historical sales data (above).

For our metric, the numeric predicted sales count is compared against the actual sales count (hidden and has to be submitted to get the value of) by means of Root Mean Squared Error (RMSE). The equation for that is:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

Figure 1. RMSE Equation, where N is the total number of data points.

For tackling this problem, I have approached this in the following steps and sub-steps:

1. Exploratory Data Analysis (EDA) & Preprocessing
    a. Explore dataset for trends
        i. Visualise features and correlations
        ii. Identify and deal with anomalies
        iii. Identify and deal with missing values
        iv. Wrangle, wrangle, wrangle until data provides us with enough direction as to how we might want to proceed with modelling
    b. Identify features for modelling and prepare data accordingly (Feature Engineering)
        i. Encode Categorical features

    ii. Separate columns if necessary
    iii. Create new features that may assist in time series prediction (such as time lag)
    iv. Group into months
    v. Visualise and inspect
    vi. Save intermediary data
2. Model & Evaluate features
    a. Separate data into train, validation and test sets
    b. Select model and hyper-parameters
    c. Evaluate model(s)
    d. Ensemble models and fit second layer model on first model layer outputs
    e. Evaluate model and fitt to test set for kaggle submission.

As Kaggle is a free platform with the idea of learning and sharing data science techniques through competitions and discussion, there are a plethora of fantastic kernels/notebooks to learn from. A few I would like to give specific thanks to are listed below. These have been fantastic in helping me guide my solution, as I have replicated a number of visualisations / methods from each in the process of building my own unique solution and approach to this problem.

- Dennis Larionov: Feature Engineering , XGBoost https://www.kaggle.com/dlarionov/feature-engineering-xgboost
- Konstantin Yakovlev: 1st Place Solution - part 1: “Hands on Data” https://www.kaggle.com/kyakovlev/1st-place-solution-part-1-hands-on-data
- DimetreOlivera: Model tacking, Feature Engineering and EDA https://www.kaggle.com/dimitreoliveira/model-stacking-feature-engineering-and-eda
- Jagan: Time Serie Basics, Exploring Traditional TS https://www.kaggle.com/jagangupta/time-series-basics-exploring-traditional-ts/data#Single-series
- TrieuChau: A beginner for sale data prediction https://www.kaggle.com/minhtriet/a-beginner-guide-for-sale-data-prediction

I would personally recommend checking out their works as further readings into solving this problem and giving a thumb up on Kaggle if you enjoyed reading their approaches as I have done. There are also other resources I have used throughout which I will give a nod to through this report.

From examining previous solutions, it is expected a solution achieving a RMSE score of 1.2 or lower will be achieved with techniques such as applying XGBoost algorithm and model stacking (https://www.kaggle.com/arthurtok/introduction-to-ensembling-stacking-in-python).

Data Exploration, Visualisation and Data Preprocessing

Our first step in approaching this problem is understanding our goal, the data we have available and tools at our disposal. In exploratory data analysis, an open approach to learning more information about the data is taken to find the possibilities as well as constraints.

Firstly, the csv data is loaded in. I have opted for Pandas library to load as a Pandas dataframe, and casted integers to int16 and float values to float32 to save memory (helpful for tackling constraints

explained later). As there is connected information across csv files, I have opted to aggregate all the information to one dataframe early to help find insights or if there are missing values earlier than later.

The next step is to do some surface level checks, probe the number of rows and columns. Pandas thankfully has several handy functions we can use. By casting dataframe.info(), we can see:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 10 columns):
 #   Column              Dtype
---  ------              -----
 0   date                datetime64[ns]
 1   date_block_num      int16
 2   shop_id             int16
 3   item_id             int16
 4   item_price          float32
 5   item_cnt_day        int16
 6   item_name           object
 7   item_category_id    int16
 8   shop_name           object
 9   item_category_name  object
dtypes: datetime64[ns](1), float32(1), int16(5), object(3)
```

Figure 2. dataframe.info() for our loaded dataframe

From Figure 2. We can see we have 2935849 rows and 10 columns, which is quite large (good thing we casted down!). Also investigating missing and null values, there were none which is fantastic! Luckily this dataset turned out to be quite clean which may not always be the case in dealing with real world data sets.
We can also sanity check the time period of the data.

```
# look at time period of data
print('Min date from train set: %s' % train['date'].min().date())
print('Max date from train set: %s' % train['date'].max().date())

Min date from train set: 2013-01-01
Max date from train set: 2015-12-10
```

Figure 3. Minimum and maximum date period

It is as we expect, from January 1st 2013 to December 2015. The goal is to predict the sales for the month after, so for January 2016. We can also see some sample values.

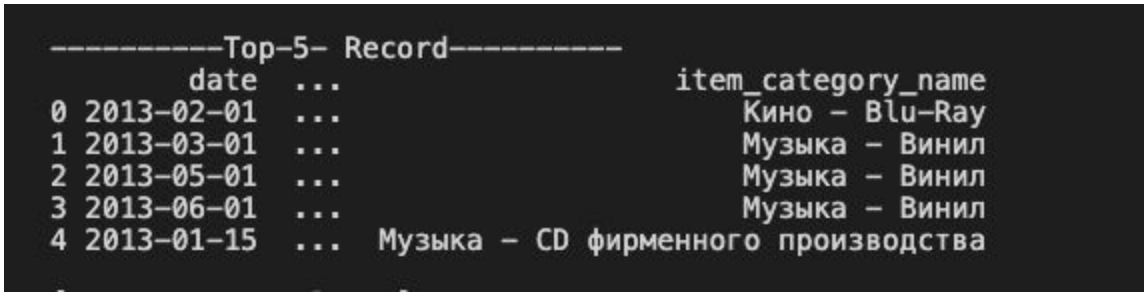

Figure 4. dataframe.head(). As the data is from stores in Russia, the language is cyrillic as we might expect.

After investigating the test set, which contains shop_id and item_id for the values we expect to predict, we find out there are rows of shop_id and item_id not present in our training set and vice-versa. This could be because items no longer on sale within the shops or shops have closed down/opened up or moved addresses etc.. As our goal is to predict data in the test set, we will keep these rows in our training set and deal with values that are not present in our training set that we need to predict test set rows of.

For the purpose of using this outside the context of a kaggle competition and continuously predicting future sales data, it may be better to keep all rows, allowing for more training (which may produce better results). Unfortunately, because of memory constraints we will only focus on relevant rows (those present in the test set). And can cull unwanted rows.

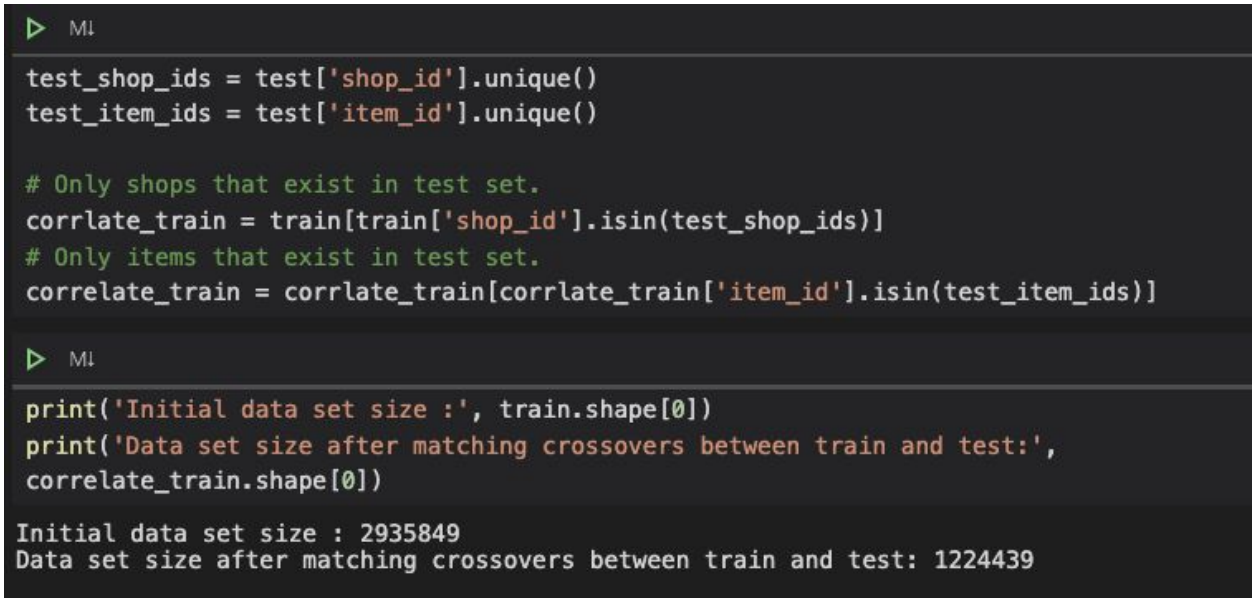

Figure 5. Remove rows in our training set that do not match with our test set. We can see the number of rows after is 1224439, a reduction from 2935849.

Now we will deal with duplicated rows. It turns out we have 5 duplicates, as seen in figure 6.

```
train[train.duplicated()]
```

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day	item_name	iter
1435367	2014-02-23	13	50	3423	999.00	1	Far Cry 3 (Classics) [Xbox 360, русская версия]	
1496766	2014-03-23	14	21	3423	999.00	1	Far Cry 3 (Classics) [Xbox 360, русская версия]	
1671873	2014-01-05	16	50	3423	999.00	1	Far Cry 3 (Classics) [Xbox 360, русская версия]	
1866340	2014-12-07	18	25	3423	999.00	1	Far Cry 3 (Classics) [Xbox 360, русская версия]	
2198566	2014-12-31	23	42	21619	499.00	1	ЧЕЛОВЕК ДОЖДЯ (BD)	

Figure 6. Duplicate rows

In figure 6, we can see the item_id values are similar and the price for them. However, the dates appear different, so may not be a mistake. As there are only 5 rows, we have left them in and may deal with them later.

We also want to identify anomalies, especially in our target metric. Figure 7 below shows a 1D plot of item_cnt_day (item sales for a day) and item_price.

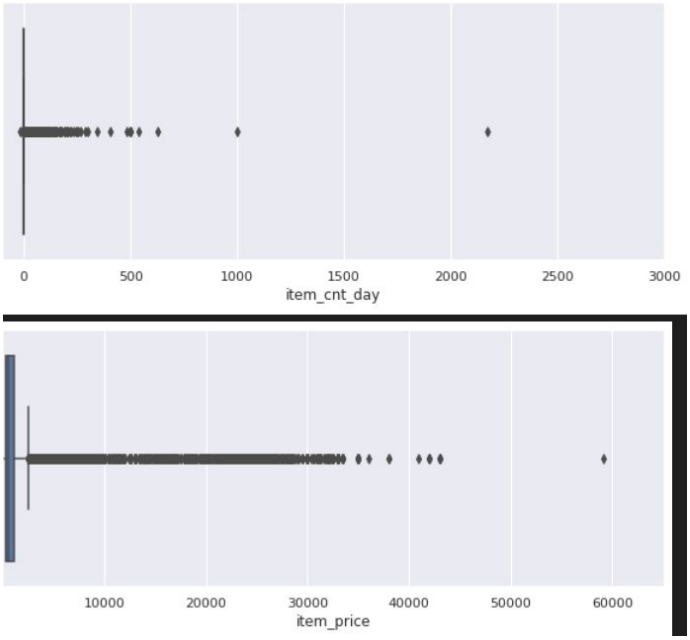

Figure 7. Item count and item price 1d plot.

We can see there is a very high value for item count day, we will remove rows with item_cnt_day > 1000, and item_price > 100000 (none in principle). Also, we fill in any item price < 0 (turned out to be less than 10 rows) with the median item price, as a negative price is implausible. From Figure 7., we can also see there are values with item_cnt_day < 0. In counting them, there are 2941rows which are too many to be anomalou and could contain important information, so we have left these in our data. Perhaps another way to deal with these is set values to 0.

In investigating shop_id, we find some shops are duplicates of each other and are fixed by adjusting the id' to corrected values. Additionally, we find that the shop_name contains the city name in the first part of the string, so from that we create an additional feature: 'city'. Following this we resolve typo and label encode the city, as it is categorical and we want to convert strings to float values or integers which is easier for our model later on to fit.

```
# Create a column for city
train['city'] = train['shop_name'].str.split(' ').map(lambda x: x[0])
train.head()

# Encode the city name into a code column
train['city_code'] = LabelEncoder().fit_transform(train['city'])
train.head()
```

Figure 8. Creating a new feature, 'city' and encoding it to create the feature 'city_code'. Note that intermediary steps involve dealing with typos and erroneous inputs.

We also apply the same approach with category name, as it contains the item type and subtype within the name. So is also encoded with label encoding. In hindsight, we could have also used another encoding method, such as mean encoding.

```
# Create separate column with split category name
train['split_category_name'] = train['item_category_name'].str.split('-')
train.head()

# Make column for category type and encode
train['item_category_type'] = train['split_category_name'].map(lambda x : x[0]
.strip())
train['item_category_type_code'] = LabelEncoder().fit_transform(train
['item_category_type'])

# Do the same for subtype, make column wiht name if nan then set to the type
train['item_category_subtype'] = train['split_category_name'].map(
    lambda x: x[1].strip() if len(x) > 1 else x[0].strip()
)
# Make separate encoded column
train['item_category_subtype_code'] = LabelEncoder().fit_transform(train
['item_category_subtype'])
train.head()
```

Figure 9. Creating encoded feature columns for item category type and subtype. Note that for subtype, we deal with missing/na values by filling in with its respective item category type.

Having captured and encoded the necessary information, we can proceed with dropping the following intermediary columns:

- Shop_name
- Item_category_name
- City
- Split_category_name
- Item_category_type
- Item_category_subtype

Investigating further, we find that we have 4716 unique item names and want to reduce this. From looking at the actual values, we realise many have similar first words within their strings and so categorise by taking the first word of item_name and encoding this in the same method we have used to encode above. In doing so, we reduce the number of unique first words to 1590 and encode this into a column ‘item_name_code’ with the same approach as above. Finally, we also drop the column ‘item_name’ and ‘item_name_split’ (an intermediary feature).

Figure 10 now displays the following columns for the first two rows. These columns will be useful in modelling our data later on.

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day	item_category_id	city_code	item_category_type_code	item_category_subtype_code	item_name_code
0	2013-02-01	0	59	22154	999.00	1	37	27	7	1	1587
10	2013-03-01	0	25	2574	399.00	2	55	12	9	2	145

Figure 10. Columns after label encoding features from original dataframe.

The next step is to group into months as we want to predict the monthly sales data, therefore this aggregation is a necessary step. In addition to summing the item count, we will also take the sum of the item price as well as the mean of the item count and the item price. There currently exists a column ‘date_block_num’ which tells us which month from January 2013 the row is taken from and so can sort by this feature. Following all these steps, we now have the columns as is seen in figure 11.

```
# Rename columns
train_by_month.columns = ['date_block_num',
                          'item_category_type_code',
                          'item_category_subtype_code',
                          'item_name_code',
                          'city_code',
                          'shop_id',
                          'item_category_id',
                          'item_id',
                          'sum_item_price',
                          'mean_item_price',
                          'sum_item_count',
                          'mean_item_count',
                          'transactions']
```

Figure 11. Renaming column features after aggregation and calculating averages by month.

As we mentioned previously, there are values of shop_id and item_id within the test set that are not covered within the training set. In order to obtain these values, we will create an empty dataframe with all possible combinations of item_id and shop_id in the test set and then merge with our main data frame after. We fill missing values with 0, perhaps there may be a better way to deal with these, however.

```
# Get all unique shop id's and item id's
shop_ids = test['shop_id'].unique()
item_ids = test['item_id'].unique()
# Initialise empty df
empty_df = []
# Loop through months and append to dataframe
for i in range(34):
    for item in item_ids:
        for shop in shop_ids:
          empty_df.append([i, shop, item])
# Turn into dataframe
empty_df = pd.DataFrame(empty_df, columns=['date_block_num','shop_id','item_id'])
```

▷ M↓

```
# Merge monthly train set with the complete set (missing records will be filled
with 0).
train_by_month = pd.merge(train_by_month, empty_df, on=['date_block_num','shop_id',
'item_id'], how='outer')
len(train_by_month)
```

```
7282800
```

Figure 12. Ensuring training set has required test set row outputs.

After, we find 6682642/7282800 na/missing. This is quite concerning, as it implies perhaps many more item_id and shop_id are in the test data than the training. Nonetheless, we will attempt to proceed by filling in missing records with 0. Perhaps there are other ways of dealing with this or a method to infer shop_id and item_id that could be useful.

The next step we take is to create a separate column for year and month. We do this to investigate the seasonality of our data, and find out how sales may change year by year. These could therefore turn out to be important features in modeling our data. The figures directly below are a series of plots in investigating more trends of seasonality and item correlation.

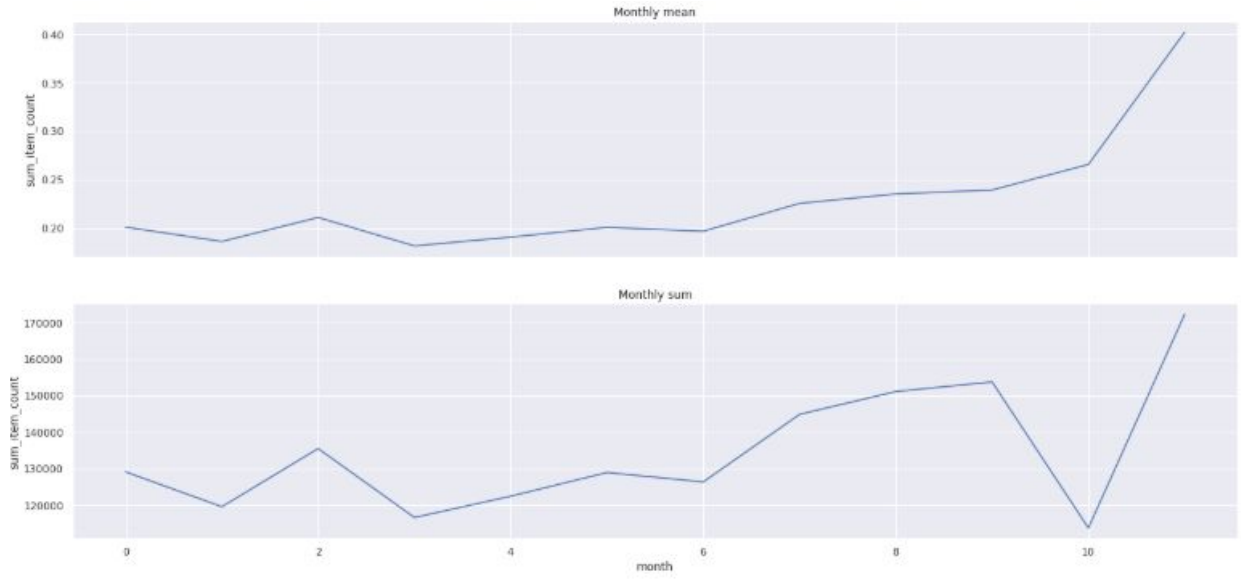

Figure 13. Overall item count mean average and sum by month. We can see that there is an increase towards the end of the year which could correlate with the festive period and a lull in the mid months. In the lower graph, the monthly sum, we can see there is a dip on the 10th month and 3rd which also corresponds with the monthly mean.

Breaking item count sales by year, we can look towards Figure 14.

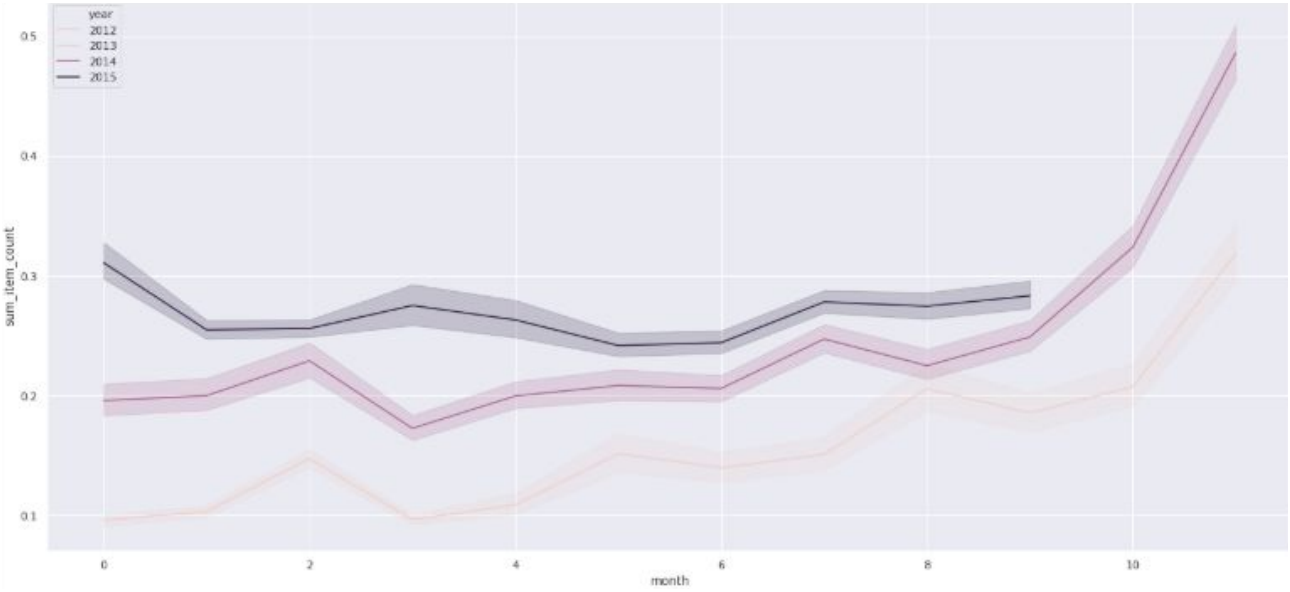

Figure 14. Item count mean sales by year. It appears sales improve year on year; with 2014 outperforming 2013 and 2015 outperforming 2014. Here we can also see seasonal trends in our datta, observing dips in month 3, which does not occur in 2015. Notably, the data for 2015 is incomplete, so stops at month 9. We can, however, make inferences from m2013 and 2014 and so expect the value to rise for later months in accordance with 2015.

Looking at more categorical data, we see plots by item category id and shop id in figures 15 and 16 respectively.

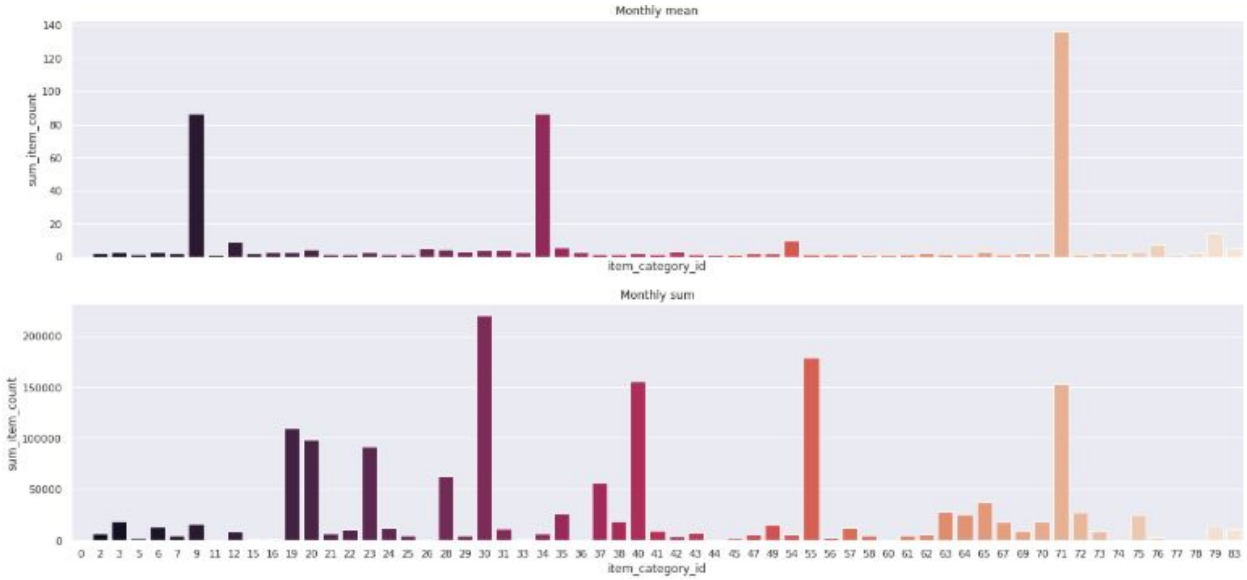

Figure 15. Item count sales for category ids. We can see that a few items sell very well and others very little especially in the monthly mean. On the other hand, the monthly sum bar plot shows more variability in distribution of item sales by category id.

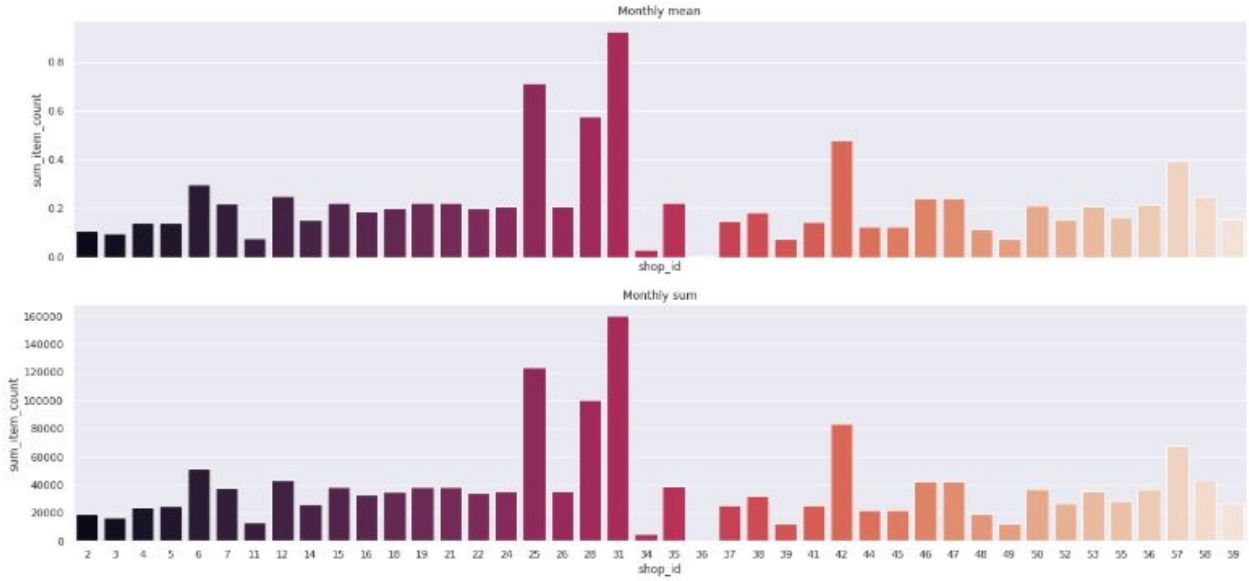

Figure 16. Item count sales according to shop ids. There appears to be about 4 shops that sell far above the others, which have a similar amount of sales. It is possible there are other reasons for this such as shop location or size which could be a future factor to investigate.

In figure 17, we can also see item count count against item price by year and density. The negative correlation observed could be useful in pricing an item according to reach a desired number of sales, in initial pricing, discounting or otherwise.

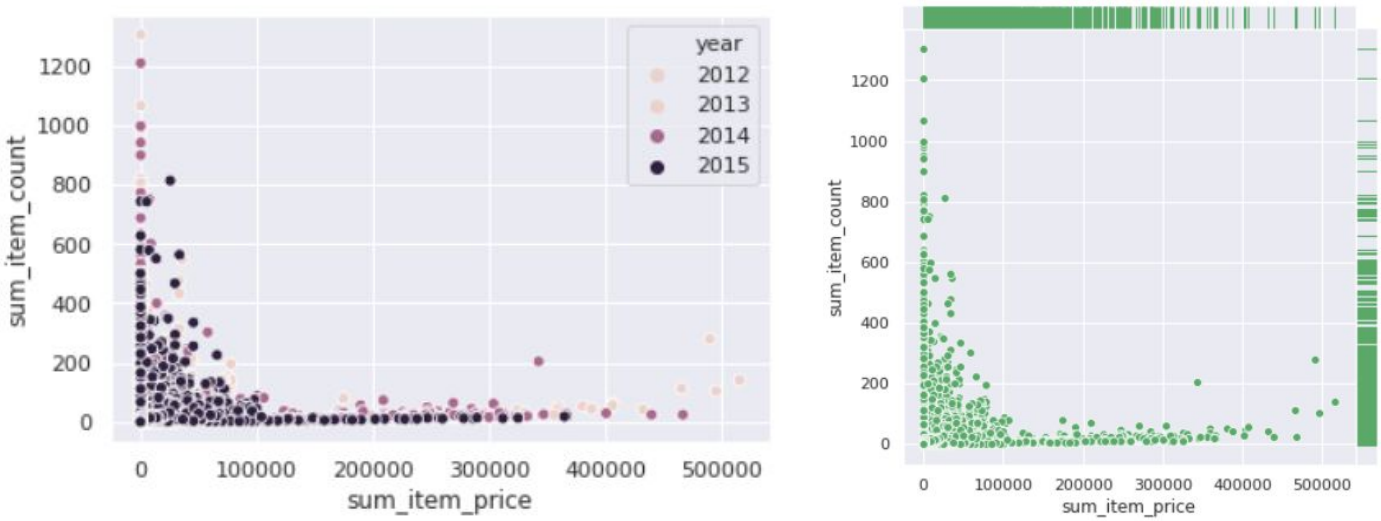

Figure 17. Left (scatter plot), Right (jointplot) of item price against item count. As expected, we can see a high iietm price corresponds with a low number of item sales count and the same vice versa with a low item price having a high item count for sales. This is seen from multiple points hugging the axes near the top left and bottom right, indicating a negative correlation. The majority of points are clustered towards the bottom left.

In figures 18 and 19 below, we can observe other categorical variables (item_name_code, city_code) respectively according to the target metric of sum_item_count.

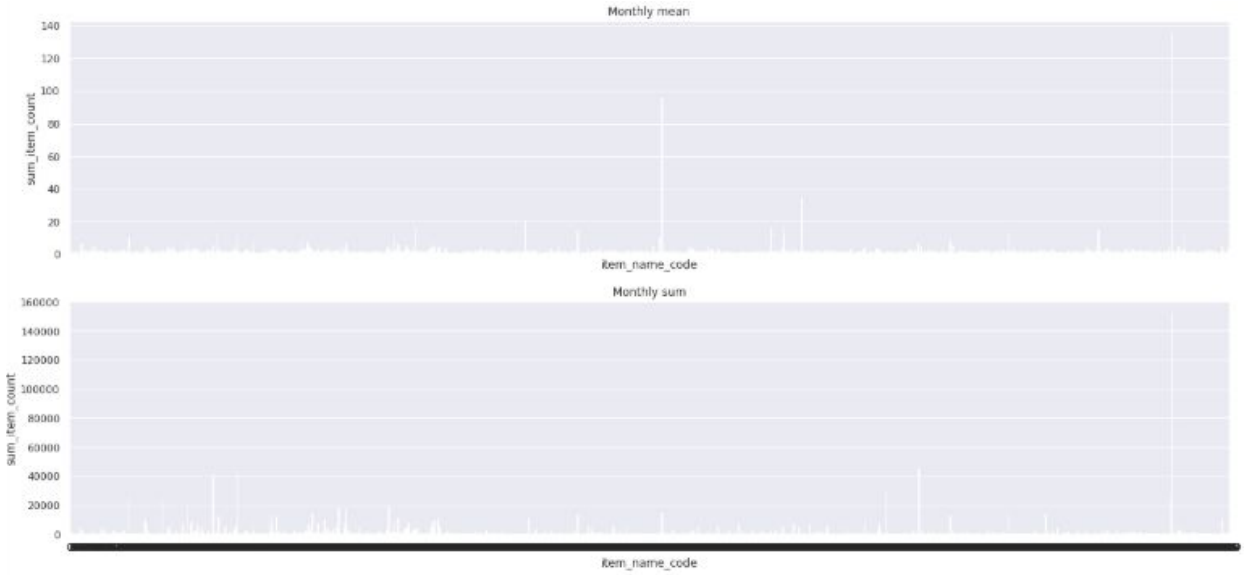

Figure 18. Item_name_code by target metric. It seems there are only a few, perhaps 2 item name codes that significantly apart from the rest and have very high sales.

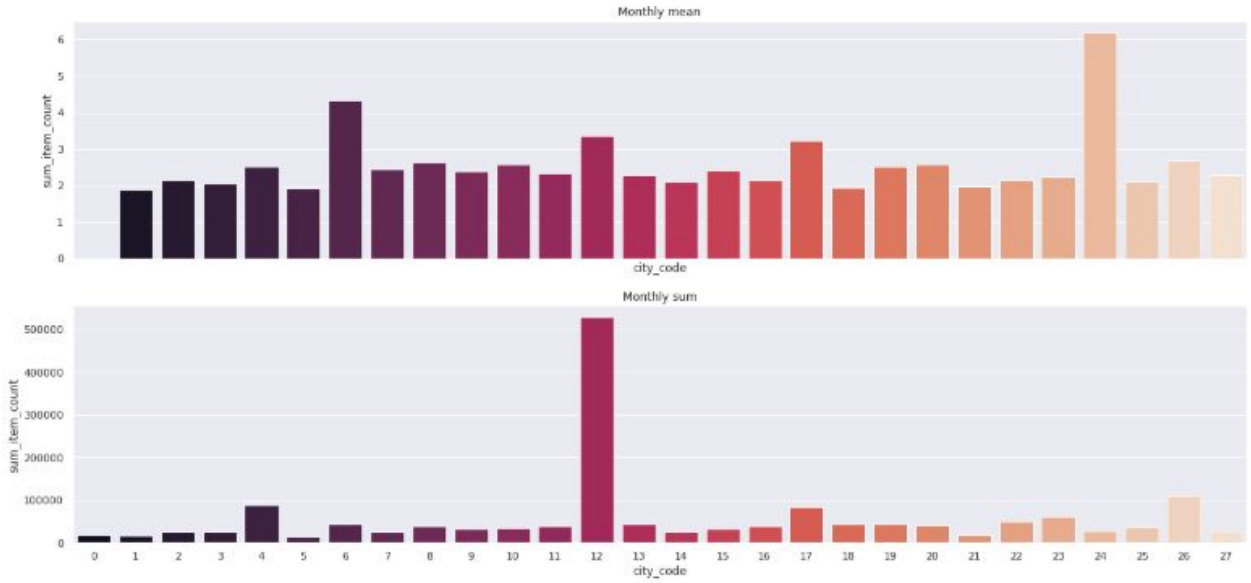

Figure 19. City_code vs sum_item_count, we can see there are two cities stand out, with label coding 12 and 24 respectively.

In Figure 20. We can see the mean item price vs transactions by year.

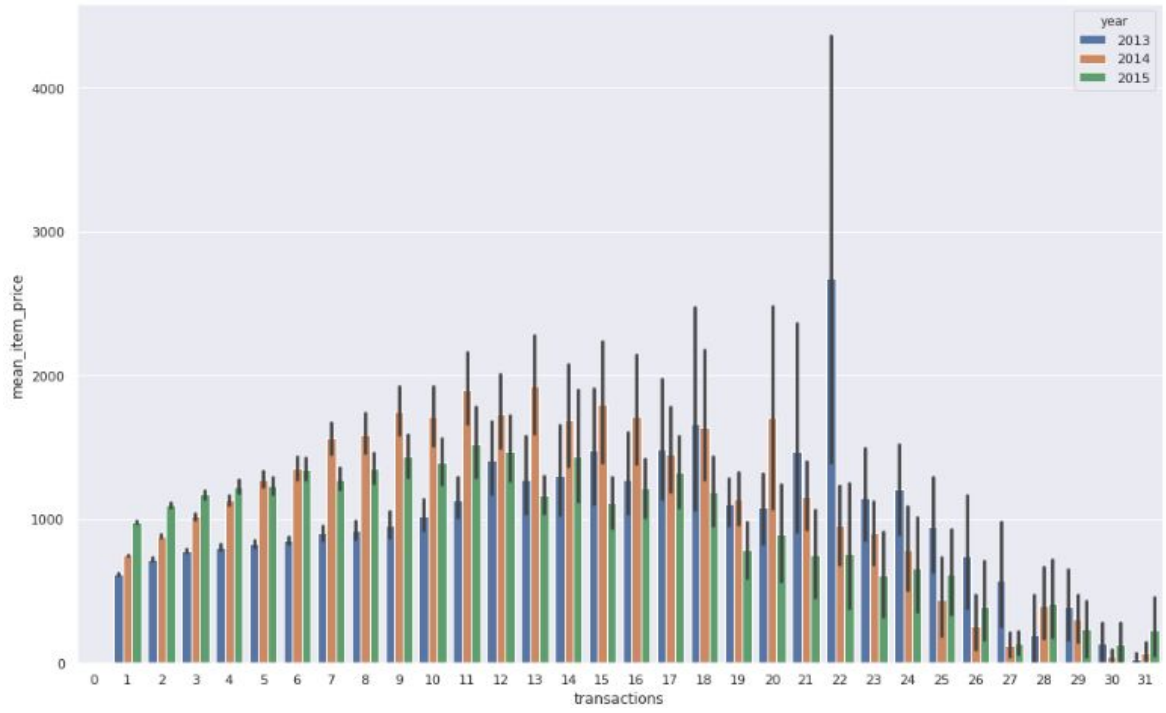

Figure 20. Mean_item_price vs transactions. It appears a large number of transactions occur for 2013, whereas the rest of the year looks steady for the mean item price across each year.

Ultimately this is a forecasting problem. We will approach this by creating a target (Y) column which is the item count sales of the next month for a given item, and is done for each row. By doing this, we can train a model to predict the item count sales of our test data which is the month following our last month in the training data.

```
# Shift item count for next month
train['sum_item_cnt_next_month'] = train.sort_values('date_block_num').groupby(
['shop_id', 'sum_item_count'])['sum_item_count'].shift(-1)
train.head()
```

Figure 21. Creating a column of item count for the next month for each item id.

Another interesting feature is perhaps item price per unit sold, so we create this feature and deal with 0 items sold (creating infinite as division by zero is impossible) by filling those values with 0.

```
train['item_price_unit'] = train['sum_item_price'] // train['sum_item_count']

# Replace inf with Nan (occurs when division by zero)
train['item_price_unit'] = train['item_price_unit'].replace([np.inf, -np.inf],
np.nan)
# fillna witth 0
train['item_price_unit'].fillna(0, inplace=True)
```

Figure 22. Creating a column of item price per unit sold.

We also make the maximum and minimum item price a feature of our dataset. With that we can have additional columns in each row of how much the value deviates from the maximum and minimum values accordingly (price increase and price decrease).

In time series prediction, a powerful technique is using a rolling window. We create a 3 month rolling window for our features and calculate summary statistics (max, min, mean and standard deviation) of our target metric, the sum_item_count and append these as columns to our dataframe. 3 months is chosen as it is long enough to capture a season, hoping to capture seasonal trends without being too large making computation detrimental. If we had more historical data and greater access to computing resources, we could date a rolling window back further to capture longer term trends.

This has the advantages of also helping to assess and gauge the instabilities of our data in our model by comparing to past data, which may be useful (Reference: https://www.mathworks.com/help/econ/rolling-window-estimation-of-state-space-models.html).

Because of limited memory, we only group by primary categories of shop_id, item_category_id and item_id, filling in empty features with 0.

Another powerful feature in time series analysis is having lag features, these are variables carried over from prior time steps that may be useful to determine future time steps. This is a classical approach of depicting a time series problem as a supervised learning one, thus phrasing the problem in a differing light. In line with the rolling window duration, lag based features are calculated over a 3 month period to capture a season.

References / Further Reading:

- https://machinelearningmastery.com/basic-feature-engineering-time-series-data-python/
- https://www.analyticsvidhya.com/blog/2019/12/6-powerful-feature-engineering-techniques-time-series/

With lag based features, we can now calculate how item count for any given item has changed over this duration, thus capturing a trend.

Finally, we are ready to model our features. A list of all the features used can be found in the appendix. In total, we have 29 features, of which we will train on 27 (perhaps explaining why we would run into memory issues. Maybe a way to circumvent this is to do exploration and feature engineering on a smaller subset of data as proof of concept then apply functions in a more powerful compute instance).

Modelling: Algorithm & Technique, Implementation & Refinement

In total, we have 7282800 rows and 29 columns, of which we train on 27/29. Our test set data to predict is one month ahead of our training set, so using the 33rd date_block_num, corresponding to December 2015 we can try to predict sales data for the test data month and year which is January 2016 for matching shop_id and item_id as is required for the Kaggle competition submission.

For our training set, we will use the 3rd-29th month blocks. We start on the 3rd block as we use a 3 month rolling window as well as 3 month lag based features and so have to start on the 3rd block. Our validation blocks will be 30th-32nd months and the test is the last block (the 33rd block). These blocks correspond to when the data was taken how many months after the beginning of recording (January 2013). Ideally we have as large a training block as possible to have the lowest RMSE for our test set, so we want to have the smallest validation data possible, especially as the validation data is closest to the actual test set, something we will deal with later. Generally, this can raise the probability of a model overfitting. However, for a time series analysis, a training set with data closer to the live date is more ideal.

After splitting, we have 5783400 training rows, 642600 validation rows and 214200 test rows. Final checks are done for null values (there were none, which is a promising sign) and the test data is aligned with the required submission format.

Another pre-processing step that could have been conducted is to normalise categorical values within our dataset (sum_item_price, mean_item_price, sum_item_count, mean_item_count and transactions). Although this may alter predicted distributions and have knock-on effects as other features are based on the ones mentioned to engineer features such as those in the rolling window.

At this stage, I first tried to fit all the models to an XGBoost model. Unfortunately, this crashed my runtime. I wanted to pursue XGBoost as it is very powerful, quick and simple to implement. In addition, there is little trade off between bias and variance due to its nature in prioritising (assigning higher weightings to) harder to predict values as well as ensembling multiple smaller weak classifiers. Therefore, I opted to stick with this instead of switching to a lighter model such as LightGBM.

The workaround I went ahead with was to split up our dataset by features. As there were initially 27 features to train on, I partitioned this into 3 smaller training subsets of 9 features each and opted to fit a Linear Regression model on the outputs of each model. In effect, creating two layers of prediction, the first layer of 3 XGBoosted models with feature subsets and a secondary layer of a linear regression model on top of the predicted outputs from the XGBoosted models which hopefully produces a better prediction score.

In theory, there are 27! different ways to rearrange our features for selection, and is something perhaps to play with to try and produce a better result. Because this number is >10^20, we split the

data by sorting the features, dividing into 9 groups and taking the first from each into subset 1, second subset 2 and last value to subset 3. With that, we are ready to train our XGBoost models!

At this stage, it should be acknowledged that more variations in potential models can be attempted, such as Random Forest (potentially very powerful model) or CatBoost and perhaps is something to return to and could possibly add to values to ensemble in our second layer should we not achieve our desired RMSE score. Ultimately, this would in theory make for a better solution.

For our XGBoost model, I decided to set the max depth to 8 (default is 6) to have a more complex model to have a stronger predictive ability. The min_child_weight was set to 300. This means there needs to be up to 300 rows/instances to make up a new node (not that large considering the size of our training set. As it is unlikely for values to be >50 in predictions, we clip values higher and values < 0 as this is implausible. Whilst these hyper-parameters can be modified and tweaked either manually, or with other methods such as grid-search or Bayesian Hyperparameter optimisation, we had to limit this because of the desire to avoid crashing our runtime session and keep performance just below the limit to maximise efficiency. The learning rate was left as default to 0.3. The full training parameters can be seen in the appendix.

References:

- https://machinelearningmastery.com/tune-number-size-decision-trees-xgboost-python/,
- https://xgboost.readthedocs.io/en/latest/parameter.html
- Note: We also saved our model as pickle files https://machinelearningmastery.com/save-gradient-boosting-models-xgboost-python/

We set early stopping rounds to 20, meaning if our model does not improve in RMSE values in predicting the target value, then it stops training after 20 iterations/epochs and saves the best model. Evaluating performance of our XGBoots models will be in the next section, for now we will focus on the training and model refinement process.

Moving forward, we have successfully trained three XGBoost models on subsets of our features. If we want to increase the power of each XGBoost model with the available computational resources, we could split our data into a lesser number of features and ensemble. However, this risks losing potential information between variables. Another approach is dimensionality reduction using techniques such as principal component analysis.

For a production level system, more tests may be required in preprocessing and feature engineering before training, as too many anomalous values or erroneous data entries may skew our model or negatively impact our resulting outcome. This would need to be considered if the company wished to predict future data every month, thus raising the need for pipelines etc.. For the purpose of a Kaggle competition, Jupyter Notebooks suffice for data discovery, proof of concept modelling and evaluation. Jupyter Notebooks also have the added benefit of telling the journey our data has made at each step, and if well documented, can highlight the thought process behind each one.

The next step is to make predictions from our first 3 models. We do this for the train, validation and test data to make predicted outputs. Once we aggregate these predictions, we are able to fit a linear regression model. Whilst a more complicated model can be fitted, I have opted for a very simple one as we only have 3 features which are numerical in value. A future endeavor may be to experiment with the secondary model used.

Our approach to fitting the linear model is to first fit on the train predicted values from first layer outputs, then the validation predicted values and finally produce predictions on our test set. This will be our final submission for the Kaggle competition. Note that we also investigated just fitting on

validation predicted and this performed worse than fitting on train, then validation. Figure 23 below shows an example dataframe of our model inputs and target.

	model_0	model_1	model_2	target
0	0.87	0.92	0.94	1.00
1	0.97	0.92	0.94	1.00
2	0.97	0.92	0.93	1.00
3	0.97	0.92	0.94	1.00
4	1.01	1.24	0.97	1.00

Figure 23. Predictions made from our three XGBoost models, in the first three columns and target values in the final column. We fit our Linear Regression model using this data as a basis.

Figure 24. depicts a high level overview diagram of the approach we have taken to training our model.

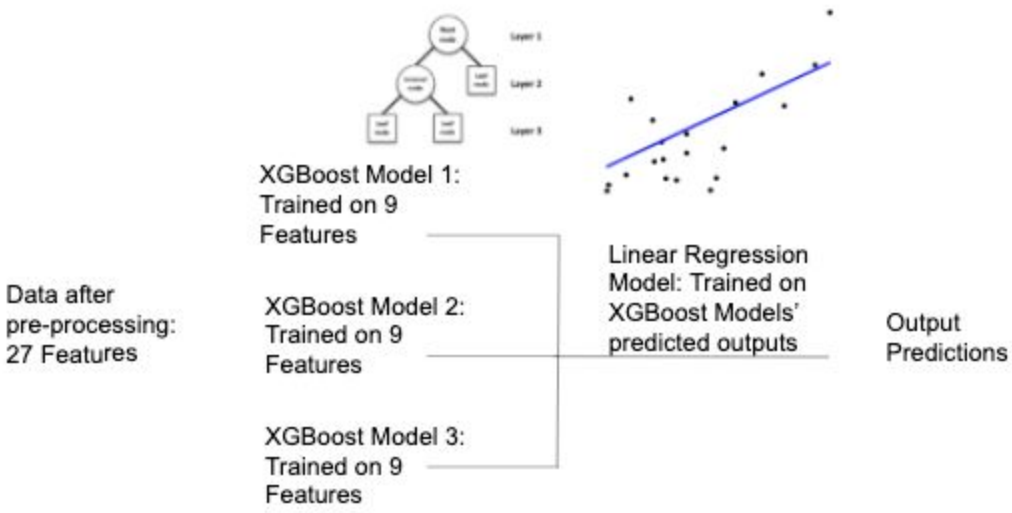

Figure 24. Overview of training process. We can see two levels of training, a form of model stacking whereby the first layer predictions (from apply XGBoost) are input into our second layer model (Linear Regression) to produce output predictions. As a reminder, we have split up our data into 3 subsets because we are unable to fit all 27 features to one XGBoost model to train.

Results: Model Evaluation, Visualisation & Project Justification

Firstly, we will assess our XGBoost models in layer 1 (see Figure 24.). We can do this by obtaining the train RMSE and validation RMSE, as well as the number of iterations in training required to reach that stage, as we have early stopping implemented if our model does not improve after 20 iterations. The performances of our first layer model can be seen in table 1.

Model	Num. Iterations	Train RMSE	Validation RMSE
XGBoost 1	26	0.765931	0.460401
XGBoost 2	15	0.857145	0.542111
XGBoost 3	19	0.80142	0.497807

Table 1. XGBoost model evaluation table with performance on train and validation data.

Interestingly, in table 1 we immediately see the validation RMSE is lower than the train RMSE. Whilst this is a good direction to head in, we also want to investigate what could be the cause. Reasons for this may be that our validation data consists of “easier” examples to diagnose. A future endeavour would be to perform cross-validation to get a more whole idea of how our models perform.

We also see that XGBoost model 1 is the best performing of the three models with the lowest train and validation RMSE of 0.766 and 0.460 (3 d.p) respectively compared to the other two XGBoost models. It is also the model that has trained for the longest number of iterations. Something else to perhaps try is to increase the number of rounds required before early stopping.

The worst performing model of the three is XGBoost model 2 with the highest train and validation RMSE of 0.857 and 0.542 (3 d.p) respectively. Below in figures 25, 26 and 27, we will visualise feature importance for each model with F-score as a metric. F-score combines precision and recall to gauge a model’s overall performance. In this context, we use it to assess a given model’s features importance. Reference: https://deepai.org/machine-learning-glossary-and-terms/f-score.

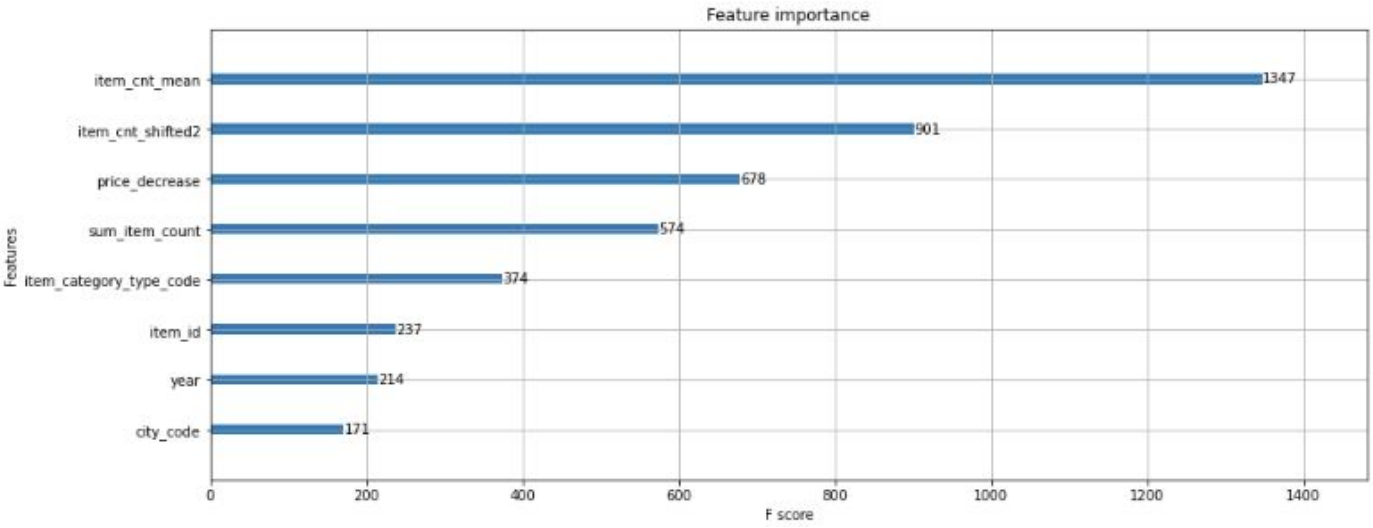

Figure 25. XGBoost model 1 Features vs F-score barplot . We can see the most important feature is item_cnt_mean with an F-score of 1347. Far behind is item_cnt_shifted2 and price_decrease with F-scores of 901 and 678 respectively. The least important feature is city_code with an F-score of 171.

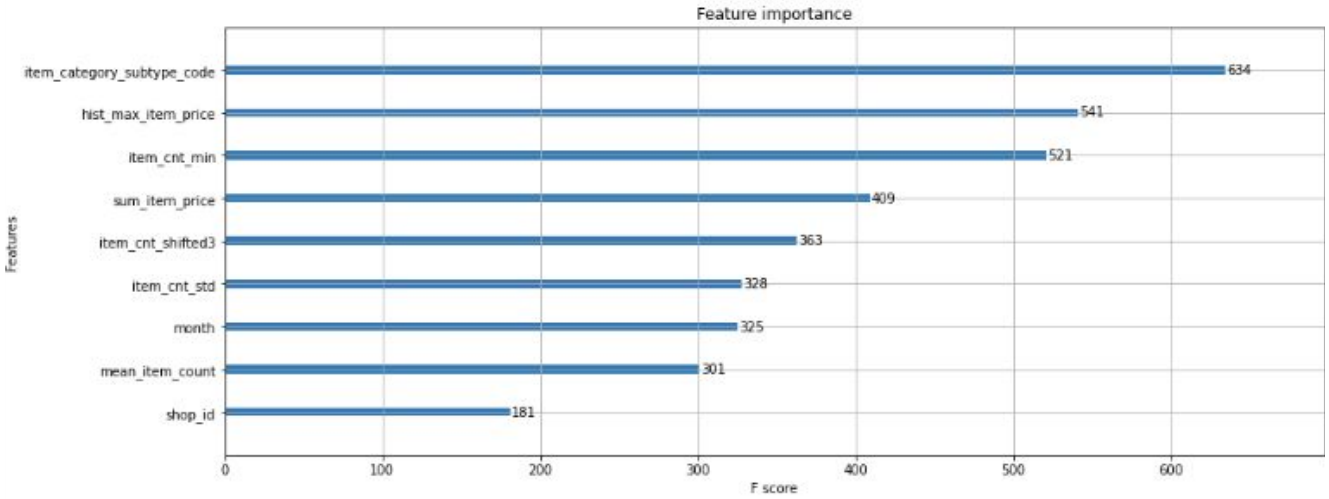

Figure 26. XGBoost model 2 Features vs F-score barplot. It seems the most important feature is item_category_subtype_code with an F-score of 634. Followed by hist_max_item_price and item_cnt_mean with values of 541 and 521 respectively. The weakest feature is shop_id with an F-score of 181.

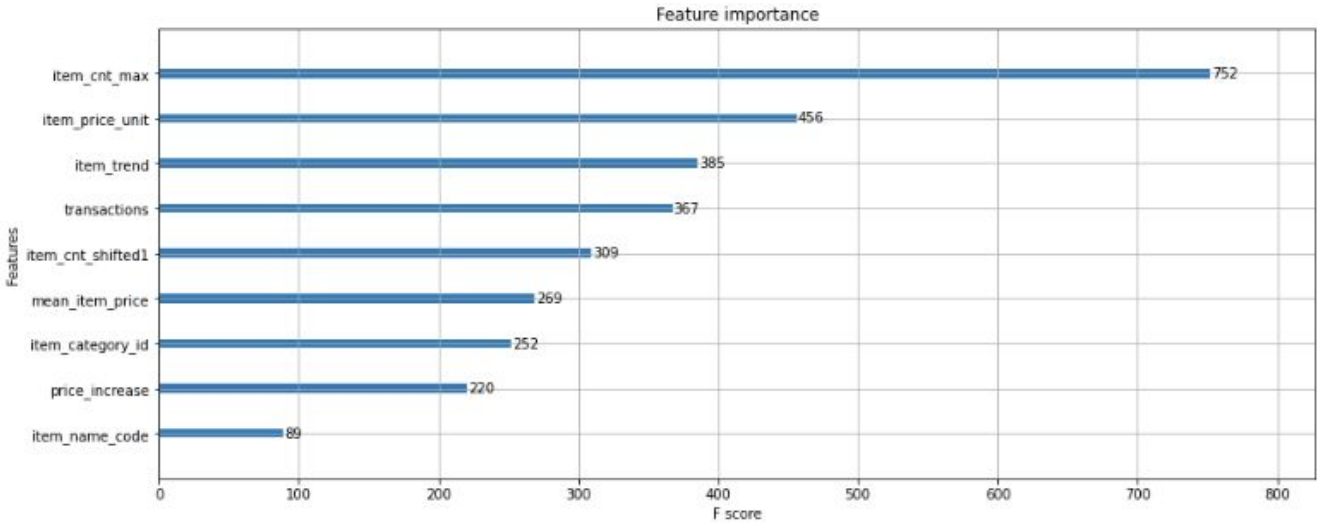

Figure 27. XGBoost model 3 Features vs F-score barplot. There is a large range between the most important F-score and the least, with the most important being item_cnt_max with an F-score of 752 and item_name_code of F-score 89.

Figure 25 shows XGBoost model 1 contains the feature with the highest F-score, item_cnt_mean and a very large range between its maximum and minimum value. Figure 26 shows a smaller range for F-scores of feature importance. Figure 27 is similar to figure 25 as there is a large range between the values, though the scale is larger for model 1 in figure 25 compared to model 3 in figure 27.

Having looked at the feature scores, in hindsight perhaps we could have avoided modeling weaker features using techniques described earlier as PCA. And so a future task could be to model with stronger features only, which could produce better test results.

We can also plot predicted outputs vs target outputs for XGBoost model 1 and 2 (model 3 was attempted but crashed our Jupyter Notebook runtime). These can be seen in figure 28 below.

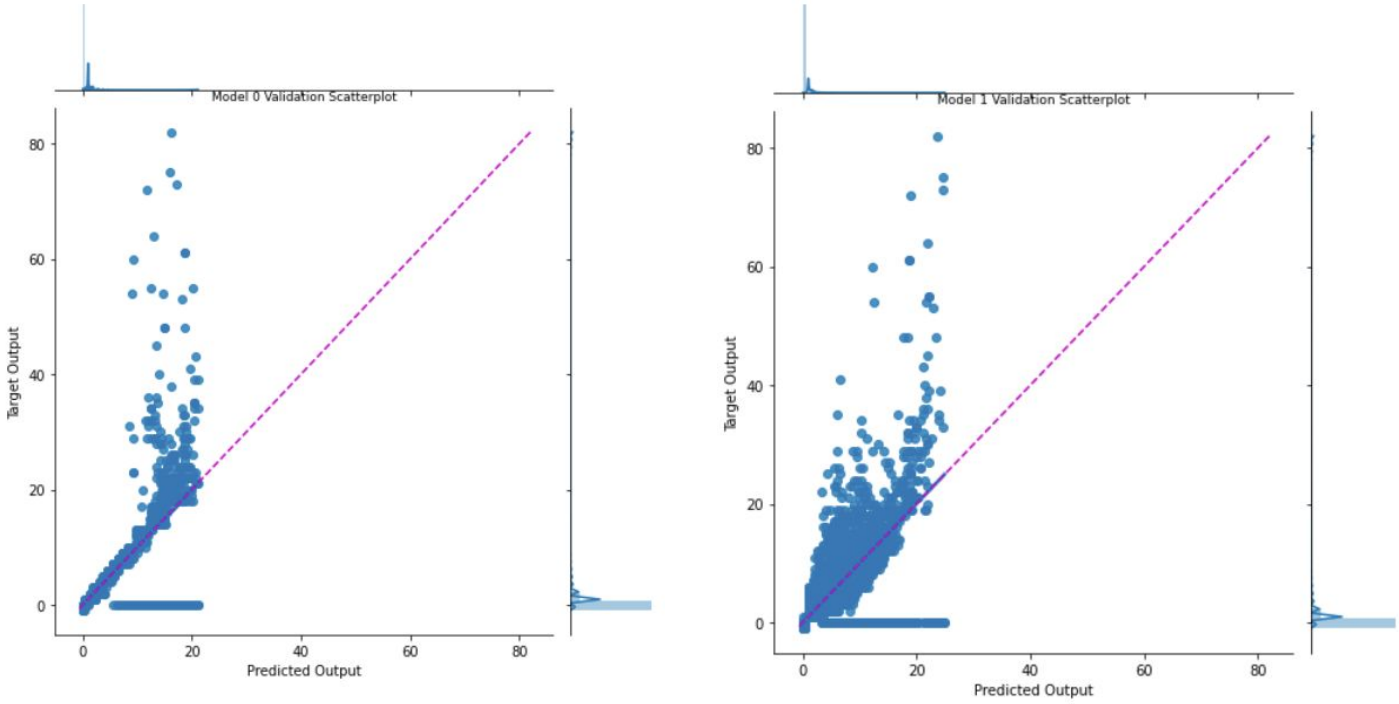

Figure 28. XGBoost model 1 (Left) and XGBoost model 2 (Right) Predictions vs Target output scatterplots.

In Figure 28. We can observe many points are close to a linear trend until predicted outputs reach 20, where predicted outputs do not exceed 20. This could be something to investigate, this occurs in both graphs (left and right), as well as a line of plots with target value 0 and predictions in the range of 0 to 30. A differing factor between plots is that XGBoost model 2 (right) has greater vertical spread than XGBoost model 1, which correlates to model 1 having a lower RMSE value from Table 1.

Finally, we can evaluate our linear regression model. Note that we used an open source library sci-kit learn to implement this (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

As mentioned in the previous section, we first fitted our model on train set predictions from our first layer models, then further trained by a very simple way of fine tuning on validation set prediction from first layer models. The results can be seen in Table 2. below.

Description	Train RMSE	Validation RMSE
Trained on train predictions	0.76094	0.46741
Trained on train predictions + Further trained on validation predictions	0.76677	0.45889

Table 2. Linear Regression model performance after fitting on train predictions then further on validation predictions.

From Table 2. We see that after training on train predictions, our linear regression model outperformed all of the first layer XGBoost models in our training set and the validation performance is only 0.007 RMSE higher than the best XGBoost model (model 1). Therefore, showing promise and validating the use of a stacked model approach.

The second row in table 2 shows a validation RMSE reducing to 0.45889, outperforming all first layer models. The train RMSE increases slightly as expected, However, our validation data for this time series problem is closer in time to the test data so will remain with this.

With our model trained and evaluated, we produce predictions for the test set and submit! Figure 29 below shows how we did in a live Kaggle competition (submitted to Kaggle on 26 April 2020).

Figure 29. Screenshot of kaggle submission.

And so we achieve the score desired of a RMSE lower than 1.2 as set in the benchmark section above. As we have finished 3789 out of 6576 submissions, there is definitely much room for improvement and many potential ideas as is mentioned in this report. So I will be glad about my first ever submission to Kaggle and simultaneously strive to aim higher in future.

Overall, I am glad this project and nanodegree has equipped me with the skills and confidence to approach data science problems and build my own unique solutions for these in a fun and collaborative way. So a big thank you to the team at Udacity (especially the person who has to read this) for this brilliant nanodegree program and hope it is the start of much more problem solving!

Appendix: Features used to train

- Date_block_num: (used to split to train/val/test, not going to be trained on)
- Item_category_type_code: Encoded item category type
- Item_category_subtype_code: Encoded item category subtype
- Item_name_code: Encoded item name code
- City_code: Encoded city code
- Shop_id: Encoded shop id
- Item_category_id: Encoded item category
- Item_id: Encoded item id
- Sum_item_price: Sum of monthly aggregated item price
- Mean_item_price: Mean of monthly aggregated item price
- Sum_item_count: Sum of monthly aggregated item count
- Mean_item_count: Mean of monthly aggregated item sum
- Transactions: Total number of item sales
- Year: Year of sale
- Month: Month of sale

- Sum_item_cnt_next_month: Target variable, item sales for next month
- Item_price_unit: Item price / item count
- Hist_min_item_price: Lowest item price in data history
- Hist_max_item_price: Highest item price in data history
- Price_increase: Difference between lowest item price and current item price
- Price_decrease: Difference between highest item price and current item price
- Item_cnt_min: Lowest item count over rolling window
- Item_cnt_max: Highest item count over rolling window
- Item_cnt_mean: Mean item count over rolling window
- Item_cnt_std: Standard deviation of item count over rolling window
- Item_cnt_shifted1: Lag feature month 1
- Item_cnt_shifted2: Lag feature month 2
- Item_cnt_shifted3: Lag feature month 2
- Item_trend: Item trend from lag features

Appendix: XGBoost Model Parameters

- max_depth: 8
- n_estimators: 500
- min_child_weight: 300
- Col_sample_bytree: 0.8
- Subsample: 0.8
- eta: 0.3
- seed: 0

Appendix: Project Rubric

Project Overview: Student provides a high-level overview of the project in layman's terms. Background information such as problem domain, project origin and related data sets or input

Problem Statement: Problem which needs to be solved is clearly defined. A strategy for solving the problem, including discussion of the expected solution has been made.

Metrics: Used to measure performance of a model/result are clearly defined. Justified based on problem characteristics.

Data Exploration: Features & Calculated statistics relevant to the problem are reported, discussed with a sampling of the data.Through description of inputs required. Abnormal characteristics need to be addressed

Exploratory Visualisation: Visualisations provided that summarise characteristics about dataset through discussion; Visual cues are clearly defined.

Algorithms & Techniques: Algorithms and techniques used in this project are thoroughly discussed and properly discussed based on characteristics of the problem.

Benchmark: Clearly define a benchmark result for comparing performances of solutions obtained

Data Preprocessing: All steps are clearly documented

Implementation: Clearly document metrics, algorithms and techniques. Complications that have occured are discussed.

Refinement: Improving algorithms and techniques are discussed. In initial and final solution, along with intermediate solutions, if necessary.

Model Evaluation and Visualisation: Model's final qualities, parameters are evaluated in detail. Analysis for validation of robustness of model's solution.

Justification: Compared against benchmark result. Whether it is significant enough to adequately solve the problem at hand.