PREDICTING FUTURE SALES

Group Project Report (3253)

Abstract

Predicting Future Sales based on time-series data for next month to help 1C improve commerce

Christopher Mills. Ayo George. Sangramjit Bhowal

Table of Contents

Background:

One of the largest Russian software firms (1C Company) has provided records of the sales of many products at many stores from January 1st, 2013 to October 31, 2015. This is challenging partial-time-series data that is part of the Predict Future Sales competition on Kaggle.com.

Objective:

The objective is to predict total sales (counts, not value) for specific combinations of product and store in November of 2015 (the next month).

Data Analysis:

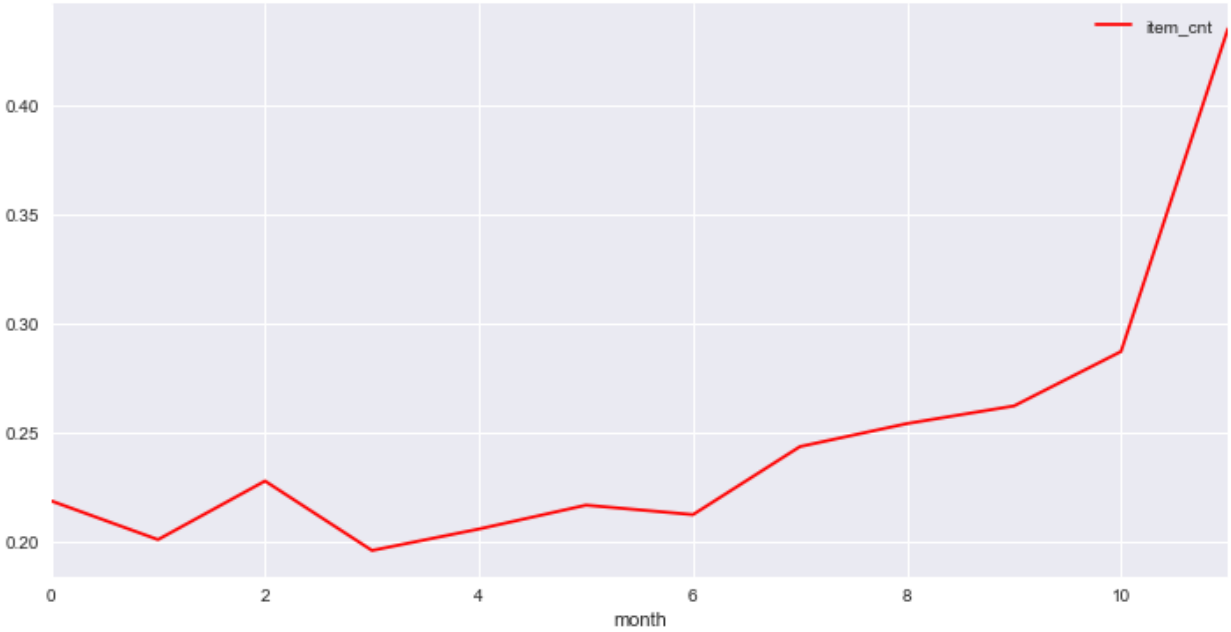
The files are obtained from the Kaggle web site (see Reference section) and their columns and the number of rows are given in Table 1 below. The shops dataset has 60 unique shop_name and corresponding shop_id value. The item_categories dataset has 84 unique item_category_name and corresponding item_category_id values. The item dataset has 22,170 unique item_name and corresponding item_id. The item dataset has corresponding item_category_id that the item_id belongs to. The training sales dataset has total 2,935,849 historical daily (date-wise) sales – price (item_price) and item-count (item_cnt_day) for item_id and shop_id. The given training sales data runs from Jan 01st, 2013 to Oct 31th, 2015 for the shop_id and item_id combinations. There are 34 date_block_num values which indicates individual months in chronological order. The test dataset has the 214,200 records of shop_id and item_id combinations that we need to make predictions for. It's worth pointing out that no price information is provided for the test period, just shop_id and item_id.

Table 1: Data File Details

#	Data File	Field Name	Records
1	shops.csv	-shop_id: 0 to 59 -shop_name	60
2	item_categories.csv	-item_category_id: 0 to 83 -item_category_name	84
3	items.csv	-item_id: 0 to 22 169 -item_name -item_category_id: key to item_categories data	22,170
4	sales_train_v2.csv	-date -date_block_num: 0 (Jan 2013) to 33 (Oct 2015) -shop_id: key to shops data -item_id: key to items data -item_price -item_cnt_day: how many of this item sold at this store on this date (sometimes 0 or negative due to refunds)	2,935,849
5	test.csv	-ID: required in submission file, along with prediction -shop_id -item_id	214,200

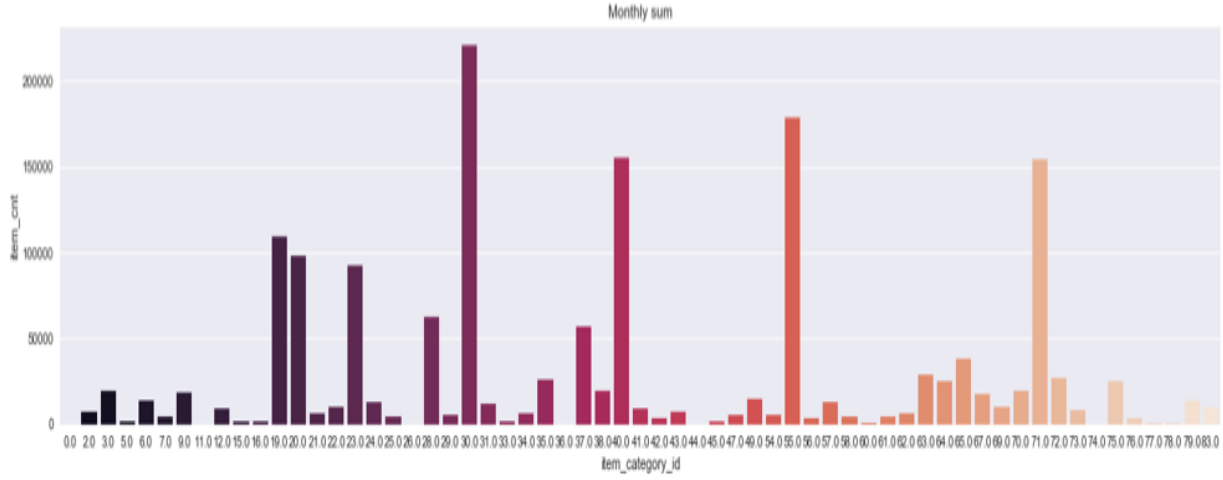
We checked, month-wise, the mean item count across the date range and it seems that the trend of sales is increasing towards the end of the year, as you can see in Plot 1 below. People are buying more items during the end months of the year than the rest of the months. Note that 0 indicates January and 11 indicates November.

Plot 1: Mean Item Count by Month of the Year



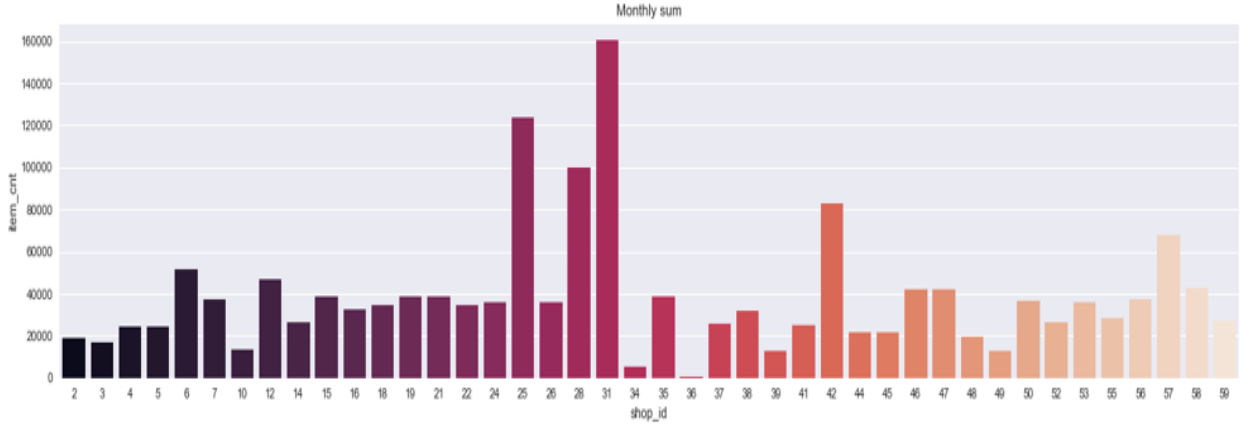
While analyzing counts of items with respect to category, it seems certain item_categories are sold more than others, as shown in Plot 2 below. Example item_category ids are 30, 40, 55, 71. These item categories are popular to the people with high demand. Also, price of the item might be reasonable in comparison to its competitors. However, it must be pointed out that the number of items per category varies. For example, the largest three categories have 5035, 2365 and 1780 items, but there are three categories with just a single item.

Plot 2: Sum of Item Counts by Item Category



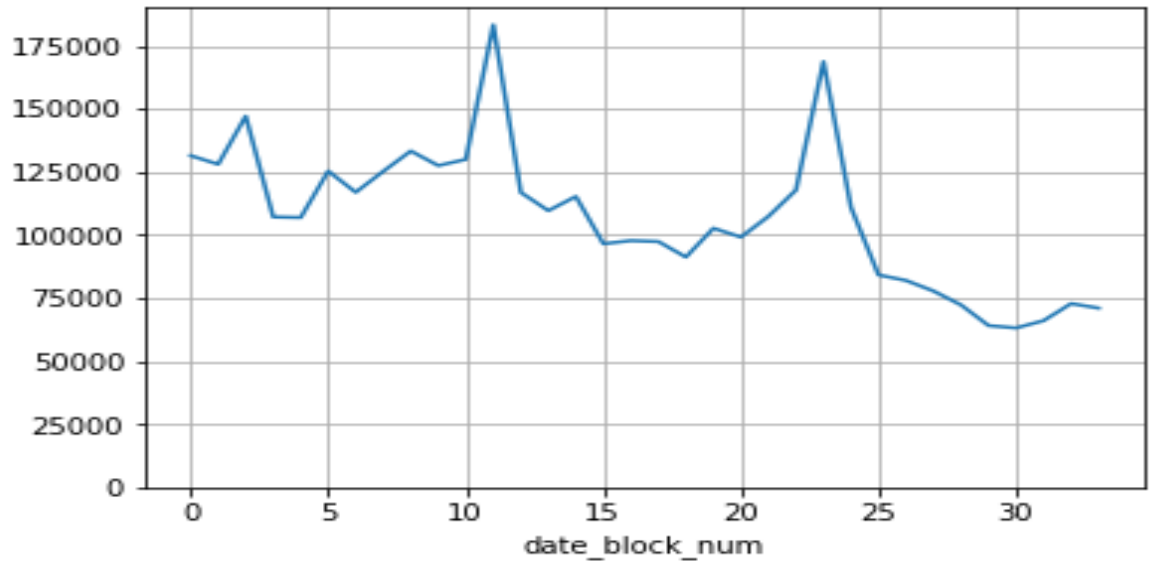
We have observed that counts of items for certain shops (example shop ids – 31, 25, 28, 42) are higher than others, as shown in Plot 3 below. It is possible that these shops are larger or in crowded areas, or people in that region have a higher buying capacity.

Plot 3: Sum of Item Counts by Shop



There are 34 months sales data given in the training data set. Plot 4 below shows the sum of all item counts by date_block_num, which indicates months of 2013, 2014 and 2015. The spikes occur for November of 2013 and November of 2014. There is also a clear downward trend over time.

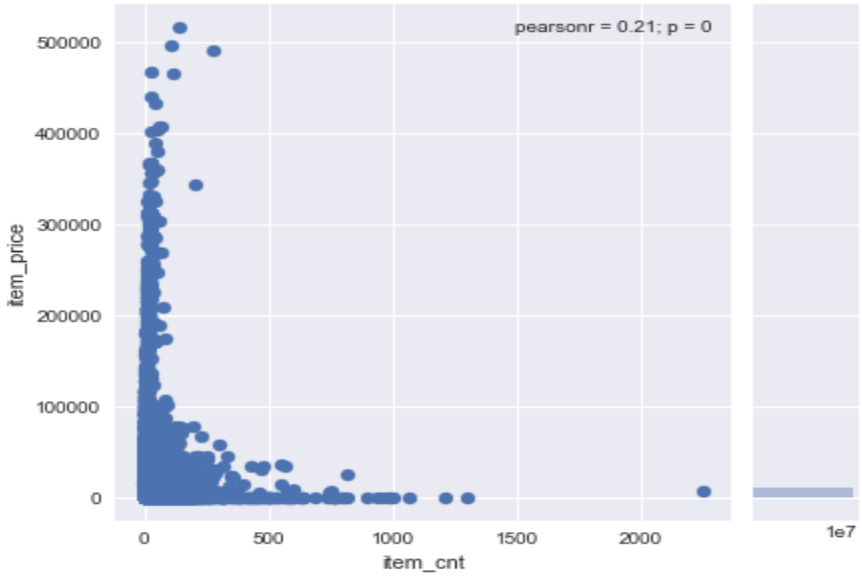
Plot 4: Sum of All Item Counts by Month



Plot 5, on the next page, is a scatter plot of item_cnt_month--which is the total count per item_id per shop_id per month--versus the mean price for that item and shop and month. As you can see, both variables are skewed to the right. The mean of item_cnt_month is especially low: of the 1.61 million values, 1.06 million are one, 265 thousand are two and only 281 thousand are greater than two. 2583 values are zero (equal number of refunds

and sales) and 915 are negative (more refunds than sales). This is very important, because item_cnt_month is the variable we are trying to predict.

Plot 5: Scatterplot of Monthly Item Count vs Mean Item Price



Three Key Assumptions:

Assumption 1: the software firm has shared accurate and complete historical sales data—every item that was sold or refunded at every store in the given date range. Then every missing combination of item_id, shop_id and date_block_num was a case of zero count. Therefore, we created records for each missing combination, giving them a count (item_cnt_month) of 0. This expanded our dataset from 1.6 million rows to 45.2 million rows, 96.5% of which were cases of zero count.

We observed that the test dataset includes every combination of 5100 distinct item_ids and 42 distinct shop_ids. In the non-expanded (1.6-million-row) dataset, each of the 5-6 months leading up to the test month had roughly this many unique item_ids and shop_ids. This exploratory analysis lead to our next assumption.

Assumption 2: the test dataset is every combination of an item that was sold/refunded in November 2015 and a shop that sold/refunded an item that month. However, if the preceding 5-6 months are any indicator, most of these shops didn't sell most of the included items, for a ratio of roughly 85% zero-count of the given items at the given stores. Therefore, our training and validation data should be selected to have a similar ratio of zero-count, which we achieved after feature engineering.

Assumption 3: we should adjust the values of item_cnt_month to the range [0, 20], because that is what has been done for the test set answers according to the Kaggle

competition's web page. Therefore, every value less than zero was increased to zero and every value greater than twenty was reduced to twenty.

Feature Engineering and Data Selection:

We created the following features for the purposes of training our machine learning model for the given datasets.

- item_cnt_month: sum of item sales by item_id and shop_id combinations in present month. *This is our target variable.*
- item_cnt_1_ago: the item_cnt_month value one month before.
- item_cnt_2_ago: the item_cnt_month value two months before.
- item_cnt_3_ago: the item_cnt_month value three months before.
- item_cnt_4_ago: the item_cnt_month value four months before.
- mean_month_shop: the mean of item_cnt_month with the given shop_id and the present month of the year (0-11, rather than num_block_id, because we can't calculate mean sales for Nov of 2015). We assumed that some shops have different seasonal patterns from others (consumer retail vs corporate neighbourhoods, etc.).
- mean_month_cat: the mean of item_cnt_month for the given item category and the present month of the year, since some categories (such as games) will sell more before Christmas than others (such as accounting or office software).
- mean_by_year: the mean of item_cnt_month in the given year, but with the months of November and December excluded because they're missing from 2015 and pull up the yearly average. This feature is intended to represent the downward trend in the company's sales, which is apparent: it's 0.085368 for 2013, 0.069746 for 2014 and 0.050812 for 2015.

The above features were calculated for all 45.2 million records in our expanded dataset, which consisted of every combination of date_block_num (0-33), shop_id (0-59) and item_id (0-22169). However, as noted after Assumption 2 above, we expect the test data to be about 85% cases of item_cnt_month = 0, so we felt we should train and validate our models on data with a similar ratio, rather than the roughly 96.5% 0s in the full dataset. Therefore, we constructed a new dataset according to the same structure as we assumed for the test set. For every value of date_block_num, every combination of an item_it sold in the month and a shop_id that sold an item in the month was included. This process selected 9.4 million rows, which turned out to be about 85.5% cases of zero: just what we were hoping for.

Machine Learning Models:

We made item_cnt_month the y values from the dataset and rest of the feature values the X values of the dataset. In some of our modeling, we split the data using train_test_split with test_size=0.30 (30%) into X_train, y_train, X_test and y_test. For later modeling we realized we should instead use StratifiedShuffleSplit because of our unbalanced data.

1. The Decision Tree Regressor Model without StratifiedShuffleSplit:

We fit the training dataset (X_train and y_train) to a DecisionTreeRegressor. Then we predicted X_test value using the same DecisionTreeRegressor model and determine the outcome (sales_prediction). Next, we calculated Root Mean Squared Error (RMSE) of sales_prediction and y_test value. The outcome was 1.10505.

2. The Random Forest Regressor Model without StratifiedShuffleSplit:

We fit the training dataset (X_train and y_train) to a RandomForestRegressor. Then we predicted X_test value using the same RandomForestRegressor model and determine the outcome (sales_prediction). Next, we calculated Root Mean Squared Error (RMSE) of sales_prediction and y_test value. The outcome was 0.96606.

Training and Validation sets with StratifiedShuffleSplit:

The data selected for training and validation is less unbalanced than the expanded dataset, but is still very unbalanced, being 85.5% zero-count. Because of its totally random nature, Scikit-Learn's train_test_split method is very unlikely to evenly distribute the 14.5% of non-zero-count observations. Therefore Scikit-Learn's StratifiedShuffleSplit class was used instead, with 25% used for validation. This class' split() method accepts one of the columns as an argument, and ensures that the split data has the same distribution of classes in the given column. By passing it the item_cnt_month column, we ensured that both our training and validation sets had nearly the same ratios of 0s, 1s, 2s, …, 20s. This close similarity was confirmed using value_counts() divided by the number of records.

3. The Random Forest Regressor Model with StratifiedShuffleSplit:

We fit the training data in RandomForestRegressor with n_estimators=30 and max_depth=7 on the training set. The mean prediction was 0.29385. Then we applied the same RandomForestRegressor model on the validation set. Results closely matched the result of training set. The mean prediction on the validation set was 0.29408.

We applied Cross validation using the negative mean squared error with 3 folds provided a mean score of 0.9478 and standard deviation of ~0.00364. However, that was just using the training data, and these folds did not ensure stratification. Next, we tried to find the optimal parameters for model using GridSearchCV. This resulted in max_features of 6 and n_estimators of 100.

A Review of Gradient Boosting Regressors:

Gradient boosting regression trees are ensembles that use many decision tree estimators, sometimes referred to as weak learners, to make predictions. However, each estimator is trained on the residue (error) of the preceding estimator's scaled prediction, and then the scaled estimates are summed to give the final prediction. The scaling is

determined by the learning rate, which is some value less than or equal to 1, and influences the training of every subsequent estimator.

We used Scikit-Learn's GradientBoostingRegressor (GBR) class, which has a default learning_rate of 0.1. Therefore, by default the first decision tree estimator's predictions of the target values are multiplied by 0.1. The residues--the differences between the target values and the scaled predictions of the first estimator--are what the second estimator is trained to predict. Its predictions are also multiplied by the learning rate and the new residues are what the third estimator is trained on, and so on. The idea is that each subsequent estimator produces smaller residues, making up for the deficiency in the preceding estimators. Smaller learning rates slow down the reduction in the residues, necessitating more estimators, but have been found to result in models that generalize better.

4. The Gradient Boosting Regressor Model:

Besides arguments for n_estimators and learning_rate, GBR can take all the same hyperparameters as a decision tree and will pass these arguments to the individual estimators in the ensemble. Perhaps the most important such hyperparameter is max_depth, as this controls the maximum complexity of each tree. In our modelling experiments, this is the only decision tree argument we provided, so all other values were their defaults. The class' loss hyperparameter specifies what loss function to optimize, and we experimented using only the "ls" value, which refers to least squares regression. The subsample hyperparameter is a float that permits only a random fraction of the samples to be used to train each estimator, resulting in stochastic gradient boosting, but we also left this at its default of 1.0, disabling that functionality.

We ran two large rounds of experiments, both using nested for-loops and early stopping implemented with the GBR's warm_start argument set to True. With this setting, the model keeps any prior training when the fit() method is called, and if n_estimators has been increased, only the additional estimators get trained. This allows repeated training and evaluation in order for the optimal number of estimators to be determined. GBR actually has this functionality built in using the validation_fraction, n_iter_no_change and tol arguments, but because of our unbalanced data this built-in early stopping mechanism was unsuitable and external iteration was used instead.

The first round of experiments used an outer loop "for max_depth in [3, 5, 7]:" and an inner loop "for learning_rate in [0.200, 0.150, 0.100, 0.075, 0.050, 0.025]:". Within these was initialization of the GBR and some variables, and then a third loop "for n_estimators in range(1, 301):". Within this the GBR's n_estimators was set, it was trained on the training set, and then it made predictions on the validation set. The mean_squared_error of these predictions was taken as a variable test_error and compared to this value from the previous time through the loop. If there was improvement greater than 0.0001, the values of n_estimators and test_error were saved as a new "best". If the improvement was not greater than 0.0001, a counter was incremented and when the counter reached 5, a break statement was used to end the innermost for-loop, providing our early stopping.

The total training time for each combination of max_depth and learning_rate was also saved.

It's worth pointing out that these early-stopping loops take far longer to run than training all estimators at once. For example, fitting a stock GBR (with default n_estimators=100) to the training data took 8 minutes, but an early stopping process with the same settings took 19.8 minutes, even though it settled on only 84 estimators. So, it took more than twice as long to train fewer estimators.

Unsurprisingly, increases in max_depth (causing more complicated estimators in the ensemble) generally resulted in greater training times and improved validation error at stopping time, but it had an inconsistent effect on n_estimators at stopping time. Decreases in learning_rate increased not only n_estimators and training time but also the error, contrary to the conventional finding that validation error improves for lower learning rates. The best n_estimators at stopping time ranged from 58 to 188 (both for max_depth=3), and the mean squared errors ranged from 0.85 to 0.89. Training times ranged from 13.3 minutes to 188.6 minutes. The whole round of experiments took 1200.7 minutes, just over 20 hours.

The second round of experiments was structured very similarly to the first, but with several changes. Seeking improved error at stopping time, the range of max_depth was increased to [3, 5, 7, 9, 11, 13] and the range of learning_rate was changed to [0.4, 0.3, 0.2, 0.1]. To improve loop run time (at the expense of precision), the innermost loop was changed to "for n_estimators in range(20, 301, 3):" rather than "range(1, 301)", so it is incremented by 3 each time rather than by 1. The error improvement threshold was also changed from 0.0001 to 0.00025, since three new estimators should make a bigger difference than one. Lastly, the code was modified so that the counter was reset to zero every time there was improvement greater than 0.00025 and the counter threshold was changed from 5 to 3. These had the effect of triggering early stopping only after three consecutive changes in n_estimators without enough improvement, rather than after five total instances of this happening since training the model began.

The changes to the early stopping mechanism had a clear impact: the n_estimators settled on was consistently higher in the second round of experiments than in the first, with it more than doubling for max_depth=5 and max_depth=7. But because n_estimators was being incremented by 3 rather than 1, training time was sometimes shorter despite this. Exploring the higher values of max_depth yielded better results, with the lowest MSE of 0.825 achieved with max_depth=11, learning_rate=0.1 and n_estimators=137. It was surprising that max_depth=13 produced worse results, however the process settled on lower values of n_estimators for every one of those searches. The longest search training time was over 15 hours for max_depth=13 and learning_rate=0.2, and the total run time for this round of experiments was nearly 91 hours.

Conclusion:

First let us summarize the results on validation data. The RMSE of the DecisionTreeRegressor model was 1.10505 and we did not feel it was acceptable. Then we thought to check with RandomForestRegressor where RMSE value was 0.96606, which looks better. With the expectation of getting a model that was more reliable, we applied a RandomForestRegressor after using StratifiedShuffleSplit and observed a RMSE of 0.97357. Using the same StratifiedShuffleSplit, the best GradientBoostingRegressor produced an RMSE of 0.90830, beating all our other models.

Our training and validation data were then combined into the full training set of 9.4 million rows. Our best random forest regressor and gradient boosting regressor were then trained on the full set, and then a LinearRegression was trained using their predictions as inputs (model stacking). Predictions from all three of these models were submitted to Kaggle and the results are presented in Table 2 below. Both the RandomForestRegressor and LinearRegression had lower errors on the training set than the GradientBoostingRegressor, and higher errors on the test set, so those two were more overfit. Our best model, the GBR, gives a ranking of 902nd place out of 1929 competitors.

Table 2: Final Model Results

Model	Non-Default Hyperparameters	Training RMSE	Kaggle Submission RMSE
RandomForestRegressor	n_estimators=100 max_features=6	0.52183	1.05691
GradientBoostingRegressor	n_estimators=137 max_depth=11 learning_rate=0.1	0.73140	1.03955
LinearRegression	None	0.50632	1.08329

Future Work:

The given data is timeseries sales dataset. We can apply model - ARIMA which stands for Autoregressive Integrated Moving Average. ARIMA is one of the commonly used methods for time-series forecasting. The model uses three parameters - seasonality, trend, and noise in dataset.

We can analyze the timeseries data using a Moving Average of sales, that means calculating the average of the last n days of sales data. Also, the dataset can be analyzed using an exponential smoothing technique where we can start weighting all available sales data while exponentially decreasing the weights as we move further back in time.

Reference:

For more information on this initiative please refer to Active Kaggle Competition:

[1] https://www.kaggle.com/c/competitive-data-science-predict-future-sales