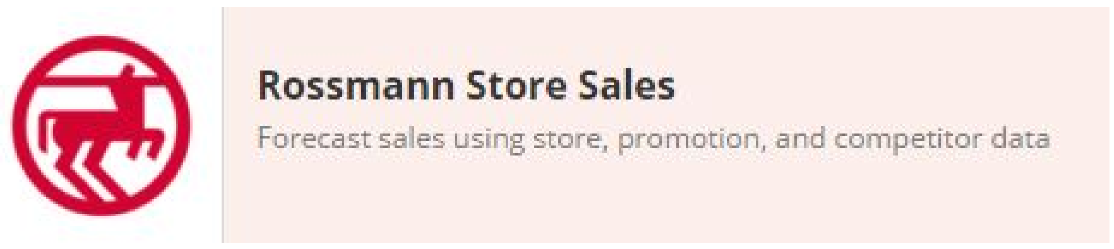
# Kaggle: Rossmann Store Sales



Rossmann Store Sales

Forecast sales using store, promotion, and competitor data

**<u>Team</u>**:

Uday Mittal, Shonali Balakrishna, Shibani Konchady

University of California, Irvine

## Index

Problem Statement
Previous Work
Project Decomposition
- Data Cleanup and Exploration
- Feature Engineering
- Store ID
  - Effect of closing of store
  - Day of the week
- Responsibilities & Credits
  - Tech Stack
  - Coding Experience

Results
- Linear Regression
- Neural Networks
  - Feature Set 1
  - Feature Set 2
  - Feature Set 3
- Xgboost

Comparison between Algorithms
Conclusions
Bibliography

## Problem Statement

**Forecast sales using store, promotion and competitor data.**

Rossmann operates over 3,000 drug stores in 7 European countries. Currently, Rossmann store managers are tasked with predicting their daily sales for up to six weeks in advance. Store sales are influenced by many factors, including promotions, competition, school and state holidays, seasonality, and locality. With thousands of individual managers predicting sales based on their unique circumstances, the accuracy of results can be quite varied.

In their first Kaggle competition, Rossmann is challenging you to predict 6 weeks of daily sales for 1,115 stores located across Germany. Reliable sales forecasts enable store managers to create effective staff schedules that increase productivity and motivation.

## Previous Work

To prepare for the Kaggle competition, we read a number of old posts from Kaggle giving us various approaches on how to improve the scores on the algorithms, and how to finetune the best performing algorithms. Very few people had explored Neural Networks for the problem[7], which seemed a feasible solution and we decided to explore how well Neural Networks would perform against the other techniques that were being used commonly to solve the challenge.
https://www.kaggle.com/c/walmart-recruiting-store-sales-forecasting
https://www.kaggle.com/c/rossmann-store-sales/scripts

## Project Decomposition

We started off with Data Cleanup and Exploration - trying to visualize the various features of the data, to gauge the importance of each feature, and derive some insights on the most valuable features to use.

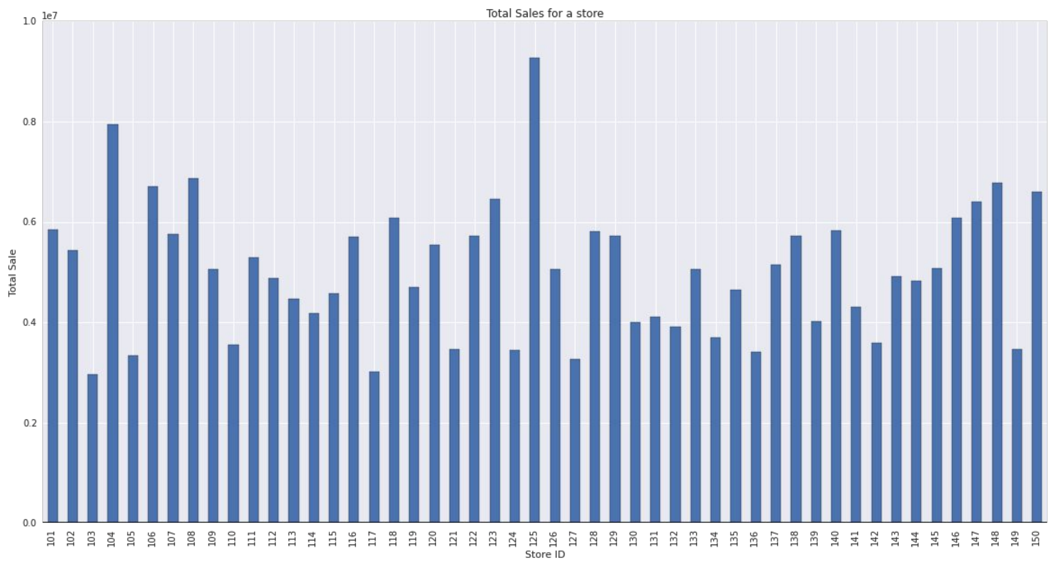
### Data Cleanup and Exploration

With large real-world data sets such as the Rossman store data, it is practical to expect to not have data for all features uniformly available across the given stores. In the absence of valid/available data for a specific feature, the corresponding data tends to be replaced by a filler instead such as Nan. To be able to utilize the given data across the given features meant cleaning up these by either replacing the filler values with the mean for that column or disregarding the given feature all together.

When we were looking at the competition data, we came across the fields for CompetitionOpenSinceYear (354), CompetitionOpenSinceMonth (354), PromoInterval (544), Promo2SinceYear (544) and Promo2SinceWeek (544), where out of the 1115 stores that the data was provided for, about one-third to half the stores did not have valid values for. Thus, the fields were deemed to be impractical criteria for comparison in the absence of wider applicability across the stores and it was determined that these could be discarded for current evaluations. Another related field- ‘CompetitionDistance’ also had some Nan values but since this data was missing only for three stores, it was decided that we would replace the Nan value by the mean value for the feature, ensuring that this change of value would not vary the results by much.
We explored each of the features to understand what the data looks like, and which of the features can be used with each algorithm.

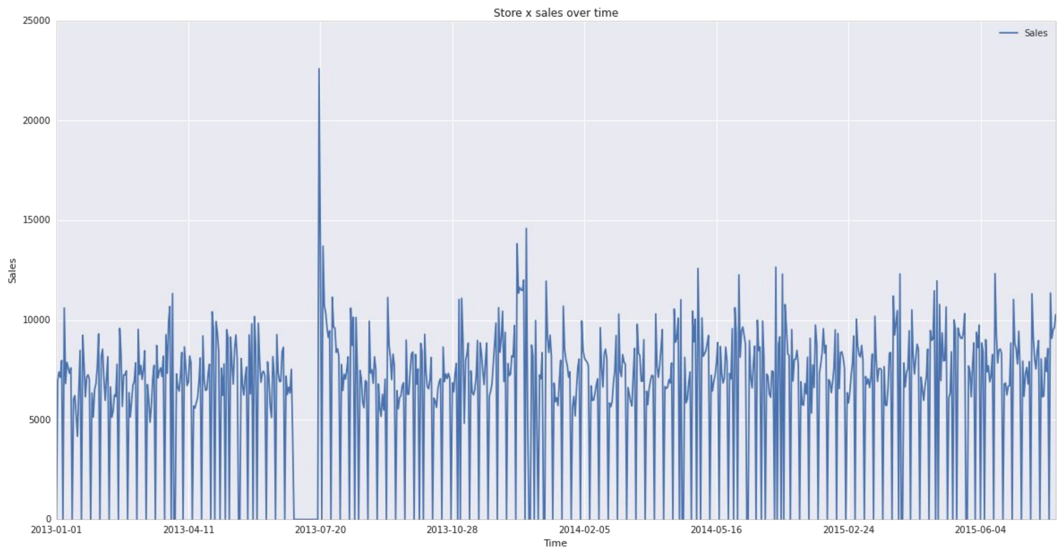
## Feature Engineering

### Store ID



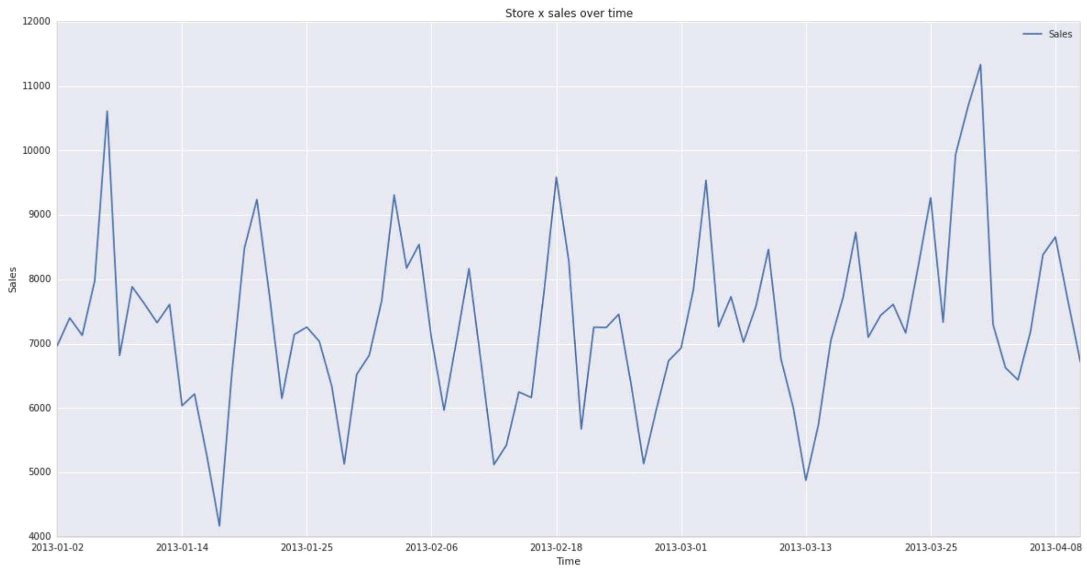
Total sales for each store were observed to be different. The graph above shows total sales for stores with IDs 100 through 150. This suggested that we use store ID as one of the features for model.

### Effect of closing of store



The graph above shows the sales time series for a particular store over 3 years [Store 100]. It was observed that the sales shoot up most of the times after a store holiday. This suggested deriving ‘PreviousDayHoliday’ as one of the features by checking for ‘StateHoliday for the previous day with respect to the current date being considered.

## Day of the week



The store sales showed a periodicity - a weekly trend as seen in the above graph. To incorporate this, we used the day of the week as one of the features. We did a 1 of K hot encoding to incorporate this feature.

## Algorithms

We explored four regression algorithms, starting with Linear Regression, and then working extensively with Neural Networks, and then onto ensemble methods like Random Forests and Gradient Boosted Machines.

### Linear Regression

Taking ordinary least squares linear regression as a starting point, we wanted to determine a baseline value for RMSPE that could be utilized for comparisons with the other relatively unfamiliar methods of evaluation that we would use as models and check for performance.

The Scikit Learn's implementation for LinearRegression was used for this purpose, and after rudimentary analysis of the features and a few test runs, we determined that we would take a subset of 46 features (an approximate number being considered simultaneously for one of the feature sets in Neural Networks as well) and obtained an RMSPE value of 0.277753103723.

### Neural Networks

We extensively worked on Neural Network regression for this project, and compared its performance for different feature sets. We used the Adam[1] optimizer with the root mean square as our loss function and we used dropout[2] layers in between to prevent overfitting.

We each worked on different subsets of features to find which features give us the best performance. We trained for many different epochs and compared performance on validation data set and then on the Kaggle dataset. The training and validation split was kept at 90:10.

### Random Forests

Random Forests Regression are an ensemble method that operate by constructing multiple decision trees, and then considering the mean predictions of the decision trees, thus countering the effect of overfitting that decision trees otherwise are susceptible to. Here, we SciKit Learn's implementation of RandomForestRegressor, and tuned it using parameters like n_estimators(number of trees in the model) and max_features(number of random features to take).

### Gradient Boosted Machines(XGBoost)

Gradient Boosting uses an ensemble of weak decision trees which are added in a stage by stage fashion, where the output is the weighted sum of predictions of individual trees, providing more accuracy with lesser trees.
Here, we used the XGBoost library, an optimized general purpose gradient boosting library, on the GPU instances. We chose this over Scikit Learn's implementation, as it is parallelized, and also provides an optimized distributed version.

## Responsibilities & Credits

The responsibility was split up almost uniformly, with each team member working on a number of things based on interest and skillset:
Shibani: Data Cleanup, Data Exploration, Feature Engineering and Derivation, Linear Regression, RMSPE Computation, Neural Networks with a Feature Set
Shonali: Data Exploration & Parsing, RMSPE & Loss computations, Feature Engineering, Neural Networks with a Feature Set, XGBoost, Random Forests
Uday: Data Exploration, Feature Engineering, Template code for prediction with NNs, RMSPE Computation, Neural Networks with a Feature Set, Setting up parallel processing environment and multi-threaded computations on GPU, Chunk-by-chunk data training

## Experience in Coding Up

### Tech Stack

We coded everything in python. Our tech stack involved the following:
**Models and Training:** Keras on Theano(Neural networks with GPU processing), XGBoost, Scikit-Learn
**Plotting:** Seaborn, Matplotlib
**Data Handling:** Numpy, Pandas

### Coding Experience

**Runs on CPU:** Initially we were training our neural networks on CPUs. This took a lot of time (~10 hours for a 1116 dimension feature vector and a single hidden layer, for about 100 iterations). To make things work faster, we first switched to mini-batch training with batch sizes of upto 100 data points. This helped us get our first few results in a couple of hours, but the results were not that great since the number of iterations were low. The neural networks would predict the same value irrespective of the input and needed more training.

**Runs on GPU:** To conduct experiments faster, we moved to training on the GPU. We configured Keras/theano to run on GPU and setup the environment on an AWS EC2 g2.2xlarge instance. Though setting up theano to run on GPU took time with all the dependency issues, once it was up, we could run experiments much faster (~25 seconds for each iteration for a 1116 dimension feature vector and a single hidden layer)

**Chunk by chunk data training:** On switching to GPU training, we were not able to train the entire data in one go. The data size was around 9GB after processing and feature extraction. It was not possible to fit this data on the GPU memory for training. Therefore, we trained the models getting data chunk by chunk

**Coding environment:** Most of the experiments and prototyping was done on ipython notebooks, since it is easier to explore and get feedback faster that ways.

## Results

### Linear Regression

Baseline value obtained for a feature set with 46 features:
RMSPE=0.277753103723

### Neural Networks

We tried three different sets of feature vectors with Neural Networks. We conducted multiple experiments with each to find the best performing configuration. We used the Adam[1] optimizer with the root mean square as our loss function. We also used dropout[2] layers in between to prevent overfitting.

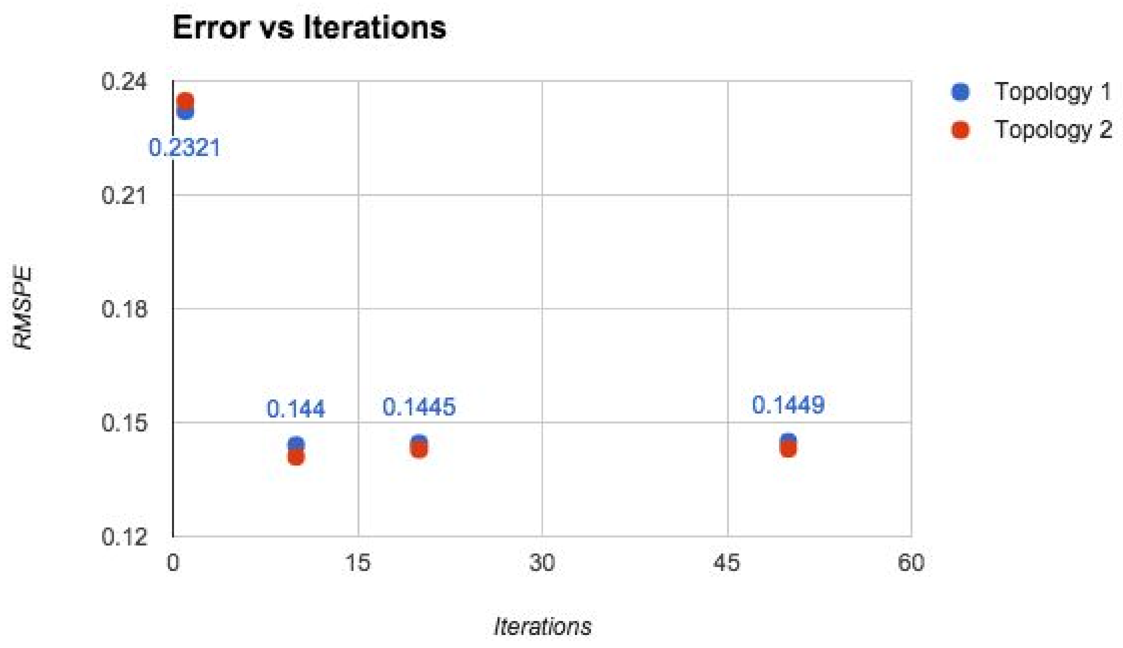
#### Feature Set 1

**Feature vector description:** Store Ids as 1-of-K (1115 inputs), and day of the week (1 input)
**Topology 1:** One hidden layer: 1500 neurons + One dropout layer [30% drop]

**Topology 2:** Two hidden layers: L1 - 800, L2 = 800 neurons + Two dropout layers [40% drop each]

| | Iterations = 1 | Iterations = 10 | Iterations = 20 | Iterations = 50 |
|---|---|---|---|---|
| Topology 1 | 0.2321 | 0.1440 | 0.1445 | 0.1449 |
| Topology 2 | 0.2347 | 0.1410 | 0.1429 | 0.1431 |

The table above is shown in the graph below



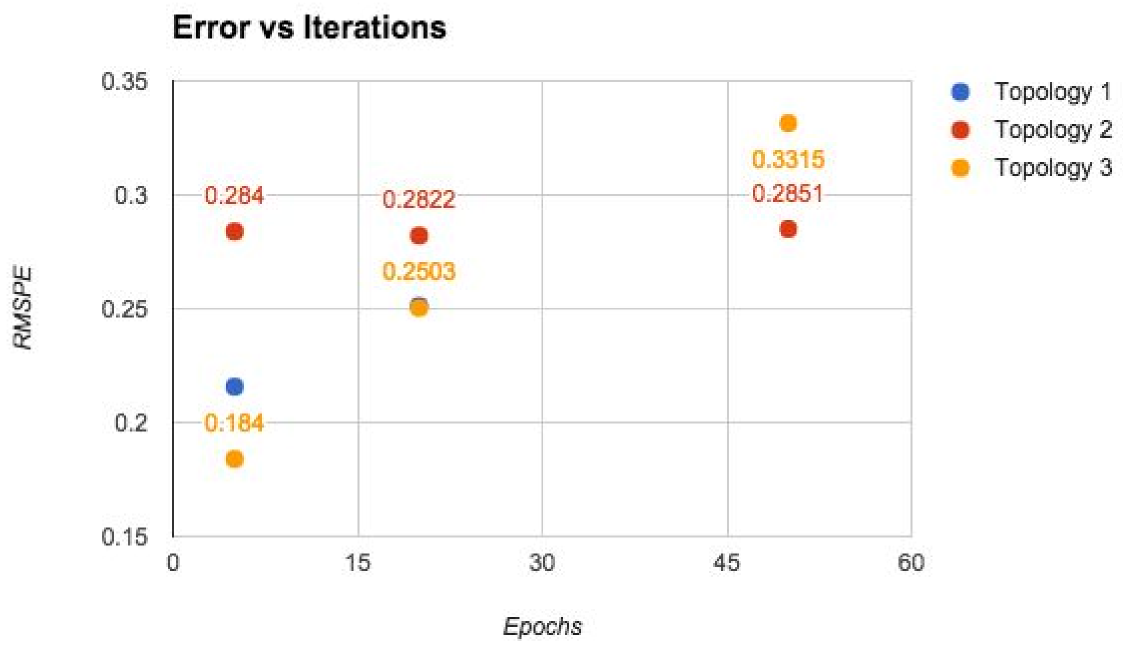
We could not observe any significant difference on increasing the number of layers to two. The RMSPE did not vary much after 10 iterations over the entire dataset and hence, increasing the number of epochs lead to no improvement in the performance. The lowest scorer on the validation set performed the best on Kaggle. This suggested that we were starting to overfit, even after 5 iterations.

## Feature Set 2

**Feature vector description:** Store, Promo, SchoolHoliday, StateHoliday(as 1-of-K), Year(as 1-of-K), Month(as 1-of-K), DayOfWeek(as 1-of-K), Promo2, Assortment(as 1-of-K), StoreType((as 1-of-K)) (44)

**Topology 1:** One hidden layer: 150 neurons + One dropout layer [40% drop]
**Topology 2:** Two hidden layers: L1 - 150, L2 - 80 + Two dropout layers [40% drop]
**Topology 3:** Three hidden layers: L1 - 150, L2 - 100, L3 = 50 + Two dropout layers [40% drop]

| | Iterations = 5 | Iterations = 20 | Iterations = 50 |
|---|---|---|---|
| Topology 1 | 0.2157 | 0.2512 | - |
| Topology 2 | 0.284 | 0.2822 | 0.2851 |
| Topology 3 | 0.184 | 0.2503 | 0.3315 |



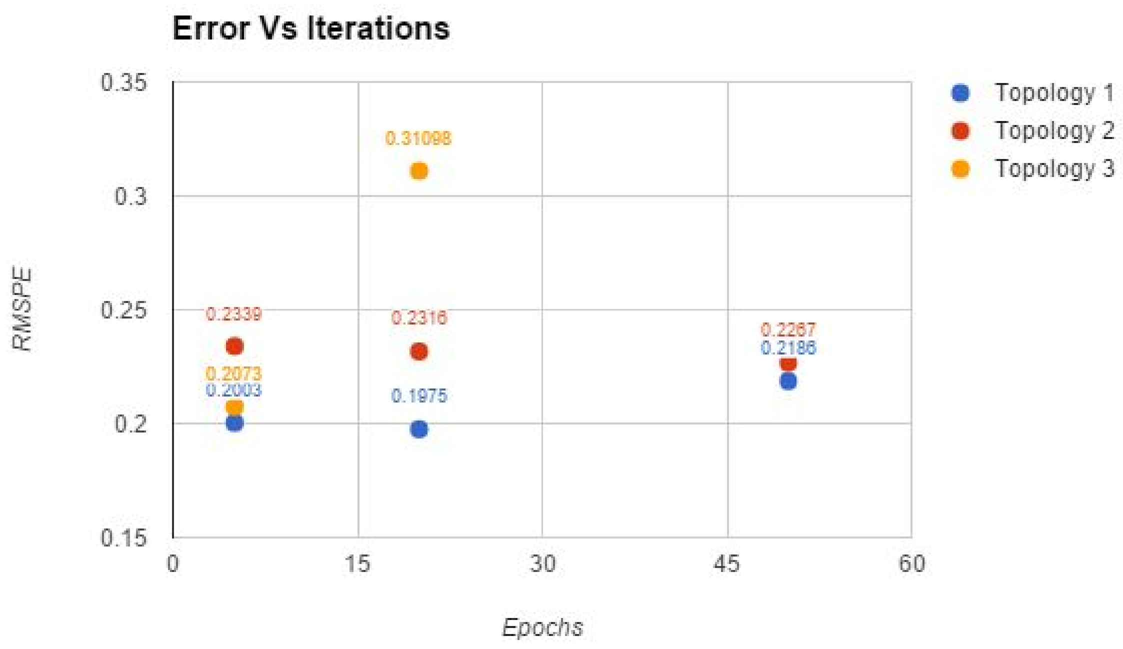
Increasing the number of layers (making the neural network deeper) did not improve the RMSPE, in fact made it worse, which shows that simpler models perform better for a dataset like this one.
Moreover, training the neural network for higher number of epochs did not improve the performance on the validation set or the Kaggle dataset, clearly displaying the problem of overfitting.

## Feature Set 3

**Feature vector description:** Feature Set 2, PreviousDayHoliday, and CompetitionDistance (46)
**Topology 1:** One hidden layer: 200 neurons
**Topology 2:** Two hidden layers: L1 = 500, L2 = 500 neurons (5 Epochs)
Two hidden layers: L1 = 200, L2 = 200 neurons (20 and 50 Epochs)
**Topology 3:** Three hidden layers: L1 = 100, L2 = 100, L3 = 100 neurons

| | Epochs = 5 | Epochs = 20 | Epochs = 50 |
|---|---|---|---|
| Topology 1 | 0.200307616385 | 0.197503426427 | 0.218623196942 |
| Topology 2 | 0.233899978353 | 0.231647212676 | 0.226754390496 |
| Topology 3 | 0.207277755094 | 0.310987087888 | - |

The table above is shown in the graph below:

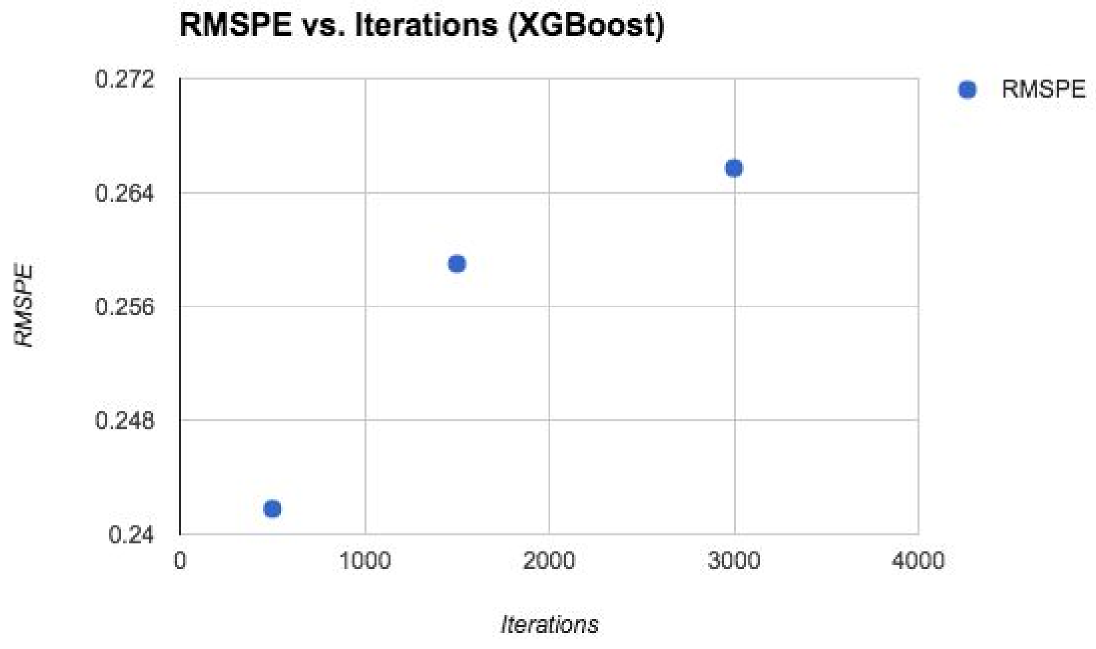


Results were visibly better with a smaller number of epochs. Higher number of epochs did not vary the RMSPE by much when it was done with a smaller number of hidden layers. But as we went on to making the network deeper with the number of layers, nodes and epochs, the performance drastically worsened with the rise in the error value. Thus the depth of the neural network appeared to be directly proportional to the possibility of overfitting the model.

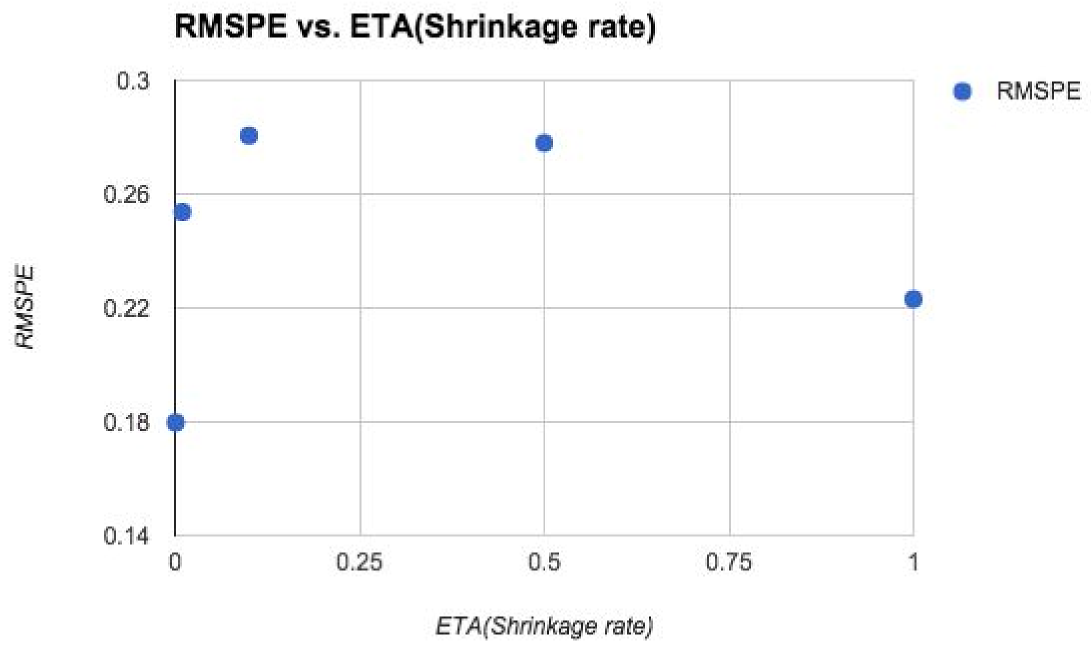
## Xgboost

We ran the XBoost algorithm with Feature Set 3, and ran it for different number of iterations, as shown below:

| Iterations: | 500 | 1500 | 3000 |
|---|---|---|---|
| RMSPE: | 0.241776242683 | 0.259024942825 | 0.265726191392 |



The 'eta' or shinkage rate here was kept at 0.01, while varying the epochs from 500 to 3000. As shown above in the table, we again see the effects of overfitting, as the RMSPE progressively increases as we increase the number of iterations.



Keeping the number of iterations constant at 1000, we tried tuning the shrinkage rate for optimal performance, and found that shrinkage rates lower than 0.01 give us the best performance. Therefore, we found that eta or shrinkage rate, which determines the rate of change of the weights, is an important tuning parameter, and provides better accuracy at lower values.

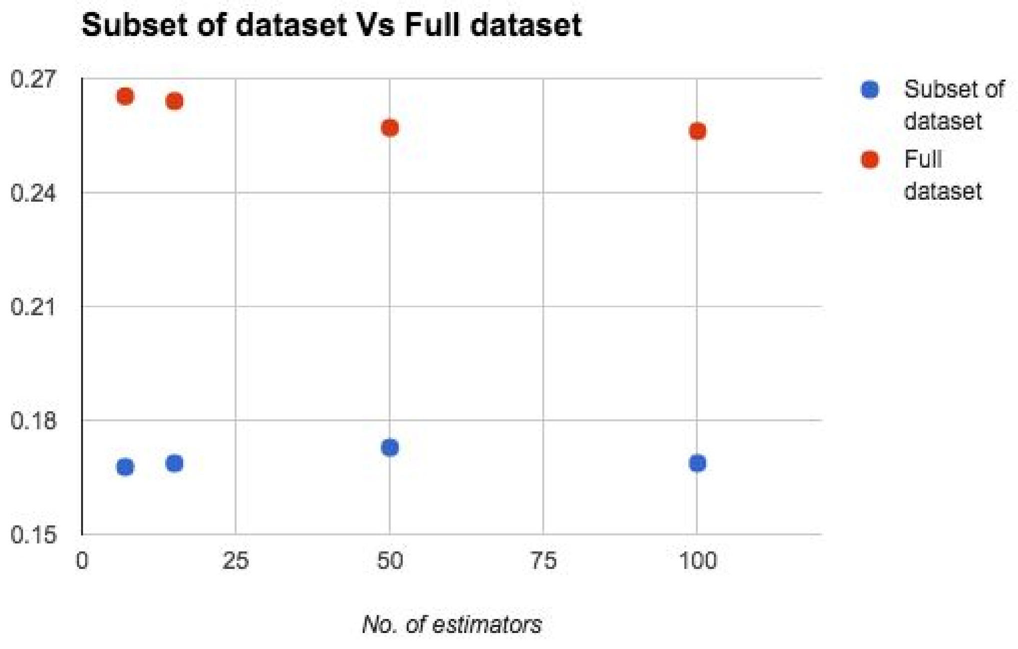
GBM tries to find an optimal linear combination of trees with relation to any given training data, and this additional tuning makes it a difficult algorithm to tune, and also highly prone to overfitting to the data, hence performing worse on real world data, even while giving low losses while training. But once tuned, it has been known to give excellent results.

## Random Forests

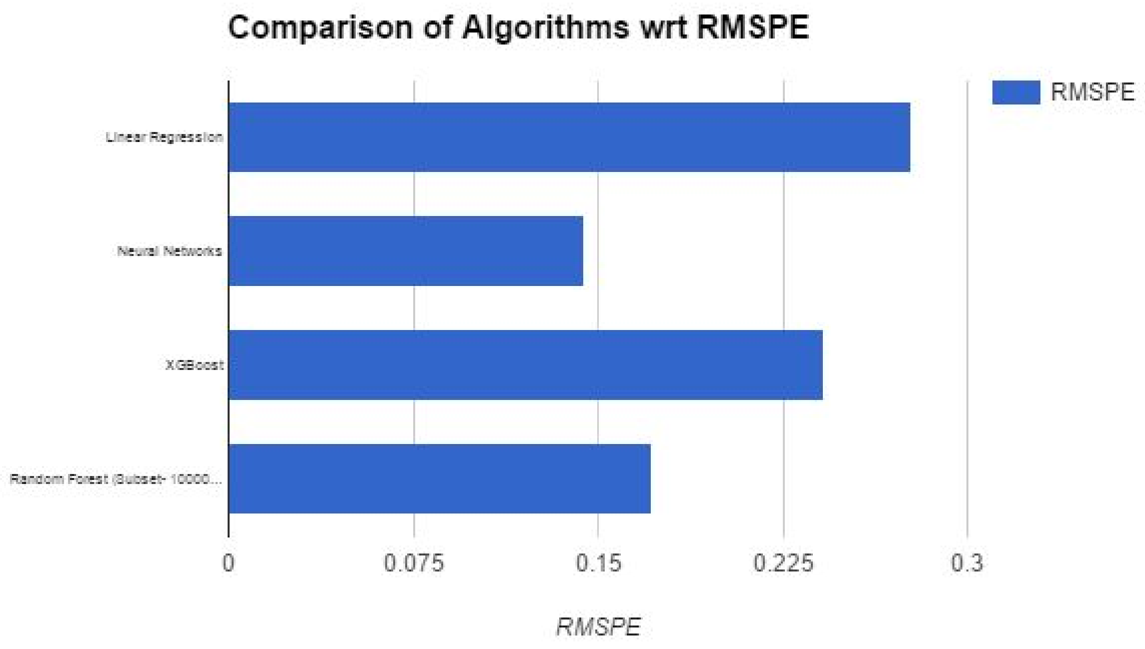
We tried Random Forests, using the Scikit-Learn library, and worked with two different datasets, first a subset of the dataset and then the entire dataset. This was for a couple of reasons - firstly as we were facing memory issues training with the entire dataset and had to move to an Amazon instance to train and obtain results and decided to train on a subset until the parallel environment is set up. Secondly, we wanted to explore the possibility of overfitting due to using the entire dataset.

As shown below, we did obtain better performance with a subset of the dataset than on the entire dataset. We also noticed that as we increased the number of trees or estimators, the performance increased only slightly.

| No. of estimators | 7 | 15 | 50 | 100 |
|---|---:|---:|---:|---:|
| Subset of dataset | 0.1676965776 | 0.1686845783 | 0.1728229458 | 0.1687029273 |
| Full dataset | 0.2653390755 | 0.2640653414 | 0.257051049 | 0.2561545176 |



## Comparison between Algorithms



Of all the algorithms we tried, we were able to get our best results from Neural Network. We worked extensively with Neural networks, with combinations of different feature sets and epochs, for different topologies (both deep and long neural networks). Our best Kaggle submission, with Kaggle RMSPE score of 0.23 was also using Neural Networks.

All of the methods used gave us better results than what we had started out with, with linear regression. Between the ensemble methods, we found XGBoost harder to tune due to the many parameters that need to adjusted for best performance. We also found that XGBoost is more prone to overfitting on the training data. Random forests were straightforward when it comes to tuning, gave us good performances with fewer parameters to tinker with. In terms of performance comparison as well, we were able to obtain better performance with Random Forests.

## Conclusions

We worked on Neural Networks extensively to evaluate how well simple neural networks would perform in time series prediction. We found that neural networks, with simpler topologies and training for fewer iterations gave us better performances on Kaggle. Our approach involved systematic experimentation with features, topologies and optimizers to evaluate the neural networks. We also learned that it is hard to train neural networks, since they are computationally intensive. The time taken to train NNs was impractical on CPU's and training on GPU is the only feasible solution to experiment with large neural networks. Other methods(Random Forest) perform almost the same with lesser computation in the time series domain.

## Bibliography

[1] Adam - A method for stochastic optimization: http://arxiv.org/pdf/1412.6980v8.pdf
[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (January 2014), 1929-1958.
[3] Keras Documentation: http://keras.io/
[4]Understanding XGBoost model: https://www.kaggle.com/tqchen/otto-group-product-classification-challenge/understanding-xgboost-model-on-otto-data
[5] Getting started with random forests: https://www.kaggle.com/c/titanic/details/getting-started-with-random-forests
[5] Bernard, Simon, Laurent Heutte, and Sébastien Adam. "Influence of hyperparameters on random forest accuracy." *Multiple Classifier Systems*. Springer Berlin Heidelberg, 2009. 171-180.
[6] Taieb, Souhaib Ben, and Rob J. Hyndman. "A gradient boosting approach to the Kaggle load forecasting competition." *International journal of forecasting*30.2 (2014): 382-394.
[7]https://www.kaggle.com/c/rossmann-store-sales/forums/t/16928/neural-network-models/96975 : A post on exploration of Neural Netwokrs for the model