Rossmann Store Sales

David Beam and Mark Schramm

December 2015

1 Introduction

The objective of this project is to forecast sales in euros at 1115 stores owned by Rossmann, a European pharmaceutical company. The input to our algorithm is a feature vector (discussed in section 3) of a single day of data for that store. We tried using a number of algorithms, but mainly gradient boosting, to output the predicted total sales in euros for the given store that day. Rossmann provides a massive 1.1 million example data set as part of a sponsored (by Rossmann) contest to see who can best predict their sales. Over 3500 groups have entered the contest, with the best group achieving 8.9% root mean squared error.
This project is also being used for CS 221, and use different parts of the project for that class that includes K-means to group the data and an Markov decision process (MDP) to model store output over a period of time. For this class, our algorithm focused around regressive algorithms and decision-tree based regression algorithms like gradient boosting. As a baseline algorithm for both classes we used support vector regression (SVR) over an unprocessed feature set. However, to improve performance we used an augmented feature set. We also experimented with linear regression and neural nets with less success.

2 Related Work

1) Exploratory Analysis Rossman - posted by Christian Thiele on the Kaggle competition board. This resource useful insights for augmenting our features such as using the log of the distance between stores and competitors instead of the actual distance.
2) Introduction to Boosted Trees - Tianqi Chen University of Washington We used gradient boosting because many competitions used and endorsed xgboost, a python library for decision-tree based gradient boosting. This link gave a useful explanation of how gradient boosting works:
3) Microsoft Time Series Algorithm - MSDN Another alternate approaching to sales data is Microsoft Time Series which is based on a core algorithm called autoregressive integrated moving average (ARIMA). One component of ARIMA is the autoregressive model, which models a predicted value X(t) as a linear combination of values at previous times, for example A(1)*X(t-1) + A(2)*X(t-2) plus a random error term. The other component of ARIMA is the moving average model, which only depends on past errors (errors before time t).
4) Different Approaches for Sales Time Series Forecasting - Bohdan Pavlyshenko This post on Kaggle compared the performance of linear regression with regularization, ARIMA, conditional inference trees, and gradient boosting. From this resource, we ultimately determined that no single approach behaves the best on all stores, convinced us to focus our efforts on algorithms that assume the data points are i.i.d, instead of using time series algorithms, which would be harded to work with and optimize.
5) We also gained a lot of insight from the 229 poster session, as there were several groups working on the Rossmann competition. This included insights such the importance of parameter tuning and an idea to train on individual stores.

3 Datasets and Features

Store	Day Of Week	Date
Customers*	Open	Promo
State Holiday	School Holiday	

Table 1: Features: *Customer data was provided, and was the most useful feature, however it is important to be able to predict sales without using customer data, as future customer numbers are unknown, while some initial trials used customers as a fetaure final results and tuning were done without it as discussed below.

The full dataset provided by Rossmann includes approximately 1.1 million training examples consisting of the features shown in Figure 1. There was also meta-data for each store, some entries were incomplete, however every entry listed the store type, assortment (inventory) and distance

to closest competitor (two stores were missing this, so we filled them in with a large value to approximate a very far competition distance). These three meta-data features were used as well. The open feature was a special case, if the store was closed, sales were always 0, we did not train on any examples where open was 0 and we always predicted 0 sales when we saw this at test time. Some data pre-processing was necessary. positive entries for the holiday fields, as well as the store type were given as 'a', 'b', 'c', or 'd', we converted these values to numbers 1-4, for compatibility with our algorithms. The date field was mapped to the day of the year, 1-365, and the year was turned into a separate feature (not used in our final experiments as it ended up slightly hurting performance). Lastly, competition distance was converted to the log distance as recommended by Exploratory Analysis Rossmann, and this helped performance.

For almost all training and testing, we did not use the full 1.1 million point data set, but a smaller set of 65,000 training examples, 21,000 development samples, and 21,000 test samples, all randomly selected once, then saved into excel for consistency. We employed simple cross validation, tuning on the dev set, then finally testing on the test set. For some final testing, we did ran on the full 1.1 million entry dataset, here we randomly split off 80% of the data to use for training and the remaining 20% for testing. No dev set was used as there was no tuning done at this point, and in general we saw very little variance between dev and test results, most likely due to the similarity of the data.

4 Methods

In total we tried four algorithms, we used SVR as a baseline, gradient boosting as our main algorithm, and also tried linear regression and a feed forward neural net. The last two algorithms showed poor initial results and were abandoned, all four however, will be discussed in some detail below.

4.1 SVR

SVR is just the regression version SVM's which we studied in class. After looking at visualizations of the sales over time on Kaggle, and determining the effects of the features on sales, we opted to restrict ourselves to a linear SVR kernel because the features seemed only to relate linearly to sales. The linear program of our SVR was therefore minimize:

$$\frac{1}{2}||w||^2 + C\sum_{i=1}^{l}(\zeta_i + {\zeta_i}^*) \text{ s.t}$$

$$y_i - < w, x_i > -b \leq \epsilon + \zeta_i$$

$$< w, x_i > +b - y_i \leq \epsilon + {\zeta_i}^* \zeta_i, {\zeta_i}^* \geq 0$$

4.2 Feed Forward Neural Net

We briefly tried using a Feed Forward Neural Net. Neural nets used a series of, weight matrices as well as a nonlinear function (we settled on tanh) to take an input feature vector and output a prediction, in our case a regression sales value. The feed forward hypothesis function of a neural net with weight matrices U and W, an input vector x, and two bias vector b_1 b_2 would be calculated as follows (note this is based off an assignment from cs224).

$$p_\theta(x^{(i)}) = h_{(\theta)}(U(tanh(Wx + b_1) + b_2)$$

Where $h_{(\theta)}$ is the sigmoid function. We can then use the least squared loss function.

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(p_\theta(x^{(i)}) - y^{(i)})^2$$

And minimize this loss with SGD to optimize all of our parameters. We were not able to get good results though using neural nets, despite trying a few different hyper parameter values, our best rms was a disappointing 2880 or 49.9% with customer data included. This is likely due to the low dimensionality of the data set, neural nets our generally at their best with large input vectors (50 or 100 at least) to tune and influence weight matrices. Neural nets had not really been connected to this problem at all, but it was still interesting to try a deep learning approach.

4.3 Linear regression

We also briefly tried linear regression to see how it compared to SVR, despite running quickly it performed worse than SVR on initial runs, so we decide to stick with boosting as the main algorithm. Regression played a small part in our project and was studied in class so it is only discussed very

briefly. It uses the same least squares cost function as neural nets, with the difference being rather than using the hypothesis function $p_\theta(x^{(i)})$, we just use the sigmoid function.

4.4 Gradient Boosting

Our primary algorithm was gradient boosting. Gradient boosting is a very powerful supervised learning algorithm that uses regression trees. Each regression tree splits the data at each non-leaf node based on a constraint on a feature (i.e. distanceToCompetitor ¡ 1000). The leaf node value of a data point determines its score, and given multiple regression trees, a data point is predicted by summing the scores of the leaf node it belongs to across all of the regression trees. The functions generated by these regression trees are actually step functions the jump where the data splits. Assuming we have K trees, the generic model for the regression tree is

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i)$$

and the objective function we wish to minimize is

$$\sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where l is our loss function and Ω is the measure of function complexity. Gradient regression uses square loss for l but a number of different function complexity measures that typically incorporate the number of leaves and the L1 norm of the leaf scores/weights Gradient boosting, or additive training, expresses a function in terms of a sum of the previously attempted functions. So

$$\hat{y_i}^{(t)} = \sum_{k=1}^{t} f_k(x_i)$$

Substituting into the objective function, using square loss, using a Taylor series expansion, and using a standard definition of function complexity, our new objective function becomes

$$\sum_{j=1}^{T} [G_j w_j \frac{1}{2}(H_j + \lambda) {w_j}^2] + \gamma T$$

Where $G_i = \sum_{i \in I_j} g_i$, and $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ That is the sum over all training instances i contained in leaf j of a tree. The ws are the weights of each leaf and λ is some constant of the L1 norm of the weights. The optimal weight of each leaf and resulting objective value are ${w_j}^* = -\frac{G_j}{H_j + \lambda}$ and $Obj = -\frac{1}{2} \sum_{j=1}^{T} \frac{{G_j}^2}{H_j + \lambda} + \lambda T$. In practice, the tree is grown greedily using the change in the objective function value based on a split: $\frac{{G_L}^2}{H_L + \lambda} + \frac{{G_R}^2}{H_R + \lambda} - \frac{{G_L + G_R}^2}{H_L + H_R \lambda} - \gamma$. The terms here are the score of the left child, right child, score if we don't split, and complexity cost of an additional leaf, respectively. We can do a linear scan over a sorted instance, calculating the above expression each time to greedily determine the optimal split over our data.

A key step to improving the performance of boosting was hyper parameter tuning. To do this we used a greedy version of sklearn's gird search, which given a grid of hyper-parameter options exhaustively tries each combination to see which performs best. To save a massive amount of run time, we only tuned one or two parameters at a time, greedily assumed that this would be the best value, and then moved on to the next parameter, adding in our newly tuned values. Of course the greedy method does not work perfectly, and some parameters were re-tuned downstream, for example initial tuning led us to a large learn rate of .9 however this was later decreased to .2. Despite having to occasionally re-tune greedy tuning saved dozens of hours of run time and still reduced our rms by almost 300%/

5 Results and Analysis

5.1 General Results

Most of our development was done working with gradient boosting, as previously mentioned, our boosting trials will be thoroughly discussed in section 5.2. That said there were still some general trends and results discovered from the other algorithms. In general our baseline SVR, out performed linear regression and neural nets in all head to head trials. It is entirely possible that we could have gotten results close to or as good as those of boosting had we decided to focus on SVR, however boosting's much faster run time

(28 minutes (SVR) vs 7 seconds (un-tuned GB) 10min (tuned)) and slightly better preliminary results made it a more desirable choice. The comparison of performances with and without customer data showed just how useful of a feature it was. its inclusion with no other changes doubled the performance of SVR and Boosting (as seen in Table 2). While, again, we couldn't use it on final trials as future customer data is of course unknown, it was a good indicator of how well an algorithm would work. For example the fact that we could not get a good prediction with neural nets using customer data told us that its prospects for final testing without customer data were not high. Figure 1, reinforces the importance of customer data by offering a visualisation showing the correlation between customer data, promotions, and sales (we chose the customer and promotions features for this graph as they were shown to be the two best by the sklearn's select k-best feature selector). The sales data is more or less linear with respect to customers, while an active promotion increases the slope of the line. The customer feature was so influential that we even tried first predicting the number of customers for a given day, then including that as a feature to predict sales, however this gave us slightly worse boosting performance (again shown in Table 2). Dataset size was one variable that had varying impacts on different algorithms, regression for example converged quickly, and showed the same results for a 65k set as the full 880k set. Boosting however showed a 3.6 % performance gain when increasing the amount of data (we did not run the full data on SVR due to the long run time, and neural nets showed no improvement moving from 1k train to 4k train so we stopped there).

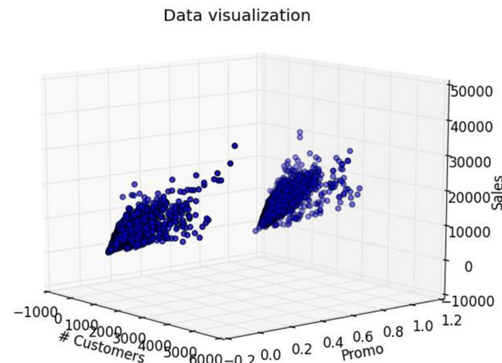

Figure 1: Data Visualisation: A plot of customers and promotions vs sales

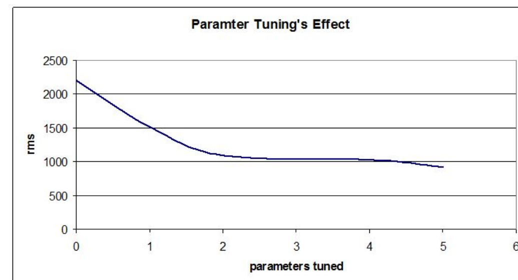

Figure 2: Tuning: this graph shows the effect of hyper-parameter tuning on rms using the 65k dataset

Algortihm	Train Data Size	Test Data Size	Cust Used	% rms
SVR with meta	65k	21k	Yes	22.1
SVR no meta	65k	21k	No	46.6
Nueral Net	4k	1k	Yes	49.9
Linear Regression	65k	21k	Yes	23.6
Linear Regression	880k	220k	Yes	23.7
Boosting no tuning or meta	65k	21k	Yes	22.1
Boosting no tuning	65k	21k	Yes	17.3
Boosting no tuning	65k	21k	No	39.7
Boosting no tuning	65k	21k	predicted	41.6
Boosting tuned store by store	65k	21k	No	82.5
Boosting tuned	65k	21k	No	15.9
Boosting tuned	880k	220k	No	12.3

Table 2: Full Results, test error is reported, test and dev error never varied by more than 3 %. Boosting

always used store meta data unless otherwise specified. % rms was calculated by taking total rms and dividing by the average sale amount, 5771 euros

5.2 Boosting

We spent the most time focusing on optimizing our boosting. Early on (even when working with SVR) we saw that the addition of store meta-data as features improved performance, and indeed we saw a 5% rms gain when we added this. Other small gains came from the aforementioned data set expansion, as well as taking the log of the competition distance rather than the full distance (this was used in tuned trials, and on its own provided roughly a 3% increase). By far the biggest gain however, came from hyper parameter tuning, once we started tuning we got our rms down from 41.6 % to 1.5% and finally 12.3%. Using the grid search method discussed in section 4.4 we settled on a learn rate of .2, max recursion depth of 5, on minimum split of 1, and 4000 estimators. The effects of tuning can be visualized in Figure 2, it is important to note that this graph is somewhat biased. The tuning gain versus parameters tuned trade off was more of a function of some parameters leading to much larger gains than others. Had we made this graph in a different order the data would look somewhat different, however it serves its purpose in showing that initially we saw large performance gains that tapered off as we got further into tuning. We found that the more estimators we added the longer the run time and the better the performance, as far as we know this trend would continue for quite some time. We ultimately stopped pushing the number estimators higher due not only to the increasing run time, but more importantly a fear of over fitting to the train data. While our validation still showed low variance, the train dev and test data are all very similar as they were taken from the same 1.1 million data set, and while this large represents 3 years of data, meaning its should contain a large variety on its own, its still not guaranteed to maintain low variance on unseen future results. One failed experiment was an attempt to train an individual boosting model for each store, instead of one large one. Other work on this problem as well as the poster session implied that this should improve performance; however, we had a large error using un-tuned values(about 50%) and a staggering 82.5% error using values tuned for one large booster. This high amount of error leads us to believe there may have been some subtle implementation error in our code, however no bug was apparent upon inspection. Regardless we were still happy with our final 12.3% number and probably could have pushed it even lower if we had continued increasing the number of estimators.

6 Conclusion

We were able to achieve 12.3% rms error using gradient boosting, our main algorithm, to predict Rossmann sales without using customer data. Boosting worked very well for this data set, most likely due to the fact that this data is extremely dense, which is known to be ideal for boosting. We also got decent baseline results using SVR, given next to no tuning was done for this algorithm, given more time we could explore SVR in more depth. Other future work would include continuing to push the upper bound on the number of estimators used on the data, and trying to get better results using the individual store method. We could also more finely tune hyper parameters using a more algorithmic method such as Bayesian optimization, rather than the brutish grid search. Any of these paths could push our error down to current state-of-the-art levels of 9% rms error.

7 Bibliography

1) "Microsoft Time Series Algorithm." Microsoft Time Series Algorithm. Microsoft Developer Network, June 2015. Web. 11 Dec. 2015.

2) Chen, Tianqi. "Introduction to Boosted Trees." (2014): n. pag.
Https://homes.cs.washington.edu/ tqchen/pdf/BoostedTree.pdf. University of Washington, 22 Oct. 2014. Web. 09 Dec. 2015.

3) Thiele, Christian. ”Dashboard.” Rossmann Store Sales. Kaggle, 8 Oct. 2015. Web. 11 Dec. 2015.

4) From sklearn we used LinearRegression, SVR, and GradientBoostingRegressor. We used pybrain for our forward-feed neural net.

5) Manning, Christopher. Neural Networks for Named Entity Recognition. Stanford University, n.d. Web. 11 Dec. 2015.

6)Pavlyshenko, Bohdan. ”Rossmann Store Sales.” Different Approaches for Sales Time Series Forecasting -. Kaggle, 30 Nov. 2015. Web. 09 Dec. 2015.