Gradient Boosted Trees to Predict Store Sales

Maksim Korolev, Kurt Ruegg
mkorolev@stanford.edu, kruegg@college.harvard.edu

We aim to minimize prediction error for a data science competition involving predicting store sales. We implement baseline models that are surpassed by XGBoost implementation of gradient boosting trees. We further use SigOpt Bayesian Optimization to modify the hyperparameters of the gradient boosted trees for further reduction of prediction error.

Introduction

Machine learning forecasting techniques have a number of applications for businesses. In particular, these methods can work quite well for large chain stores that are able to gather significant amounts of data. Using this data to make informed decisions for the future can have large financial consequences when scaling to a large number of stores. Specifically, this information can inform businesses on optimal staff levels, product shipments, and sales promotions at each branch. On the data science competition website Kaggle, the German pharmacy Rossmann posted a challenge to the data science community to predict the sales of their stores for a 6 week interval in the Fall of 2015 using data gathered from their stores from the previous 30 months [5].

In detail, each training example consisted of the following features and response variable:

Features

Date discrete, non-ordinal

Day of Week discrete, non-ordinal

Store ID discrete, non-ordinal

Customers discrete, ordinal

Open binary

State Holiday discrete, non-ordinal

School Holiday binary

Store Type discrete, non-ordinal

Assortment discrete, non-ordinal

Competition Distance continuous

Competition Open Since Month/Year discrete, ordinal

Promo binary

Promo2 binary

Promo2 Since Year/Week discrete, non-ordinal

Promo Interval discrete, non-ordinal

Response Variable

Sales continuous

A brief explanation of the non-obvious features: Store Type indicates 1 of 4 store designs. Assortment is 1 of 3 levels (basic, extra, and extended) of how large the store assortment is. Competition Distance gives the distance in meters to the closest competing drug store. Competition Open Since gives the year and month when the closest competitor opened. Promo indicates if the basic promotion is active. Promo2 indicates if a store is participating in a cyclic promotion that runs for 1 month, then takes 2 months off. Promo2 Since Year / Week indicates when the store started participating in Promo2. PromoInterval gives the months that Promo2 is active for a participating store [5].

The training set consisted of data on 1115 stores from Jan. 1, 2013 to Jul. 31, 2015. With the exception of a 180 stores that had an extended closure from Jul. 1, 2014 to Dec. 31, 2014 all stores had entries for every day in the time period.

The test set consisted of data from all 1115 stores for the time range from Aug. 1, 2015 to Sep. 17, 2015. We did not have access to the actual values of the response variables in the test set, but we did have access to our score on the test set. For the contest, we were allowed to make 5 submissions per day.

Contest Background

Concretely, all our models $h(x)$ attempt to map our feature space x, where d is the number of features, to our response variable y

$$h(x) \rightarrow \hat{y}$$
$$x \in \mathbb{R}^d, \hat{y} \in \mathbb{R}$$

The competition utilizes Root Mean Square Percent Error as the scoring method:

$$RMSPE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\frac{y^{(i)} - \hat{y}^{(i)}}{y^{(i)}})^2} \tag{1}$$

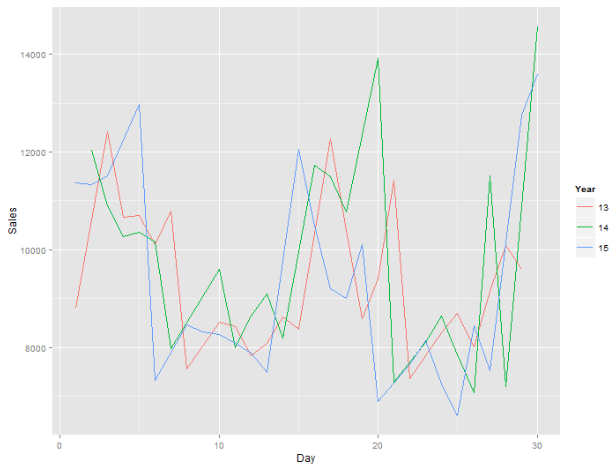
Linear Regression Model

The first model we tried was a simple linear regression model, which we made by transforming all discrete non-ordinal features into binary encoding and plugging our features into the built-in multiple linear regression function in R. As expected, our initial model was fairly inaccurate:

$$\boldsymbol{RMSPE : .25412} \tag{2}$$

Mean Guess: A Simple Decision Tree

We noticed that most stores had similar sales across years:



Using this insight we made our simple Mean Guess Model. In order to predict the sales of a day in 2015 we took the average of 2013 and 2014 for our given day and given store.

The following equation is a general formulation of our model. The index set A gives the indices of the features that must match when we make a prediction. Our hypothesis function takes the average of all the training examples that match all the features that are specified in A.

$$h(x) = \frac{\sum_{i=1}^{N} y^{(i)} \prod_{j\in A} 1\{x_j^{(i)} = x_j\}}{\sum_{i=1}^{N} \prod_{j\in A} 1\{x_j^{(i)} = x_j\}} \tag{3}$$

$$\boldsymbol{RMSPE : .18237} \tag{4}$$

From here we examined machine learning literature and realized that we had created a decision tree, albeit a very simple version. A decision tree is a tree where each node splits the input space of the node among its children. In our simple mean guess decision tree, we performed a multi-way split along 1115 stores and then did a 365 way split on each day of the year to produce our result. We then took the mean of the members of each leaf to make our predictions. We next chose to pursue an algorithm called Gradient Boosted Machines, which have been shown to perform the best out of all decision tree learning methods, assuming model parameters have been optimized [11].

Gradient Boosted Trees

The gradient boosted trees method is an ensemble learning method that combines a large number of decision trees to produce the final prediction.

The “tree” part of gradient boosted trees refers to the fact that the final model is a forest of decision trees. A decision tree is defined as a model where each non-leaf node represents a split on a particular feature, each branch represents the flow of data based on the split, and each leaf represents a classification. An individual decision tree is grown by first ordering all features and checking each possible split for every feature. The split that results in the best score for some objective function becomes the rule for that node. The splitting process continues until some termination condition is met. Termination can be caused by running out of features to split on or reaching pure sets of training examples, but usually an early termination condition will be met before this. Early termination of tree growth is important to prevent overfitting, which will be discussed later. Finally, in order to make a prediction, each leaf must have an associated value. The response of the leaf will usually be the majority response of its training examples for classification problems and the mean of training examples for regression problems.

“Boosted” means that the model is built using a boosting process. Boosting is build on the principle that a collection of “weak learners” can be combined to produce a “strong learner,” where a weak learner is defined as a hypothesis function that can produce results only slightly better than chance and a “strong learner” is a hypothesis with an ”arbitrarily high accuracy” [3]. The hallmark of all boosting methods is the additive training method which adds a new weak learner to the model in each step. In the case of gradient boosted tree, the weak learner is a new decision tree. This is shown in the below equation, where $F(x)$ is our full model after $t-1$ rounds and $h(x)$ is the new tree we are adding to the model.

$$F_0 = 0$$
$$F_t(x) = F_{t-1}(x) + h(x)$$

The “gradient” portion of gradient boosted trees refers refers to the method by which the new function is added to the model. The key idea is that each new function is an attempt to correct the errors of the model built in previous rounds. Thus, this new function $h(x)$ should be fit to predict the residual of $F_{t-1}(x)$. For XGBoost, this insight is used during the derivation of the the final objective function.

In the XGBoost package, at the t^{th} step we are tasked with finding the tree F_t that will minimize the following objective function:

$$Obj(F_t) = L(F_{t-1} + F_t) + \Omega(F_t) \tag{5}$$

Where $L(F_t)$ is our loss function and $\Omega(F_t)$ is our regularization function.

Regularization is essential to prevent overfitting to the training set. Without any regularization, the tree will split until it can predict the training set perfectly. This will usually mean that the tree has lost generality and will not do well on new test data. In XGBoost, the regularization function looks like this:

$$\Omega(F_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{6}$$

Where T is the number of leaves in the tree, w_j is the score of leaf j, λ is the leaf weight penalty parameter, and γ is the tree size penalty parameter.

Determining how to find the function to optimize the above objective function is not clear. In order to create a concrete way of selecting our next tree, we must make an assumption in order to manipulate the objective function. First, we assume a fixed tree structure in order to produce an objective function that will allow us to compare different tree structures resulting from different tree splits. An extended derivation that can be read in detail in a presentation given by the author of XGBoost [4]. we produce the following objective:

$$Obj(F_t) = -\frac{1}{2}\sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{7}$$

$$\begin{aligned} G_j &= \sum_{i \in I_j} \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \\ H_j &= \sum_{i \in I_j} \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \\ I &= \{i | q(x_i) = j\} \end{aligned}$$

Where $q(x)$ maps input features to a leaf node in the tree and $l(y_i, \hat{y})$ is our loss function. This objective function is much easier to work with because it is now gives a score that we can use to determine how good a tree structure is. From here we can choose splits that will minimize the objective function. Also, once we have no splits that can further decrease the objective function, we will have our termination point. This equation can be manipulated into a gain function, which is the final equation that does the dirty work of picking each feature splits in XGBoost at each step in tree growth.

$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda)} - \gamma \tag{8}$$

Where R and L represent the right and left child nodes, respectively, that would be created in a given split. Gain is then maximized over all possible split options in all the nodes in the current tree.

Parameters

XGBoost requires a number of parameters to be selected. The following is a list of all the parameters that can be specified:

η(eta) Shrinkage term. Each new tree that is added has its weight shrunk by this parameter, preventing overfitting, but at the cost of increasing the number of rounds needed for convergence.

γ(gamma) Tree size penalty

max depth The maximum depth of each tree

min child weight The minimum weight that a node can have. If this minimum is not met, that particular split will not occur.

subsample Gives us the opportunity to perform "bagging," which means randomly sampling with replacement a proportion specified by parameter of the training examples to train each round on. The value is between 0 and 1 and is a method that helps prevent overfitting.

colsample bytree Allows us to perform "feature bagging," which picks a proportion of the features to build each tree with. This is another way of preventing overfitting.

λ(lambda) This is the L2 leaf node weight penalty

Target Variable Transformation

The scoring method of the competition differed from the loss function included in the gradient boosted tree library (Root Mean Square Percent Error vs. Square Loss). This meant that we either had to transform our target variable or write our own loss function. It turns out that doing a simple log transformation of the target variable is theoretically sound as long as the difference between our target values and prediction values is small. See Appendix A for a confirmation of correctness.

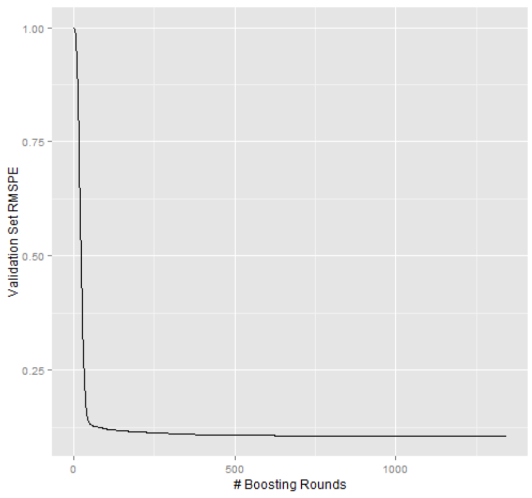
$$RMSPE(y^{(i)}, \hat{y}^{(i)}) \approx \sqrt{\frac{1}{n}\sum_{i=1}^{n} (\ln y^{(i)} - \ln \hat{y}^{(i)})} \tag{9}$$

Cross Validation

We noticed sales are similar across years. Therefore, we theorized that a good validation set that could mimic the test set would be the sales in 2014 during the same time period as the competition test set (Aug. 1, 2015 - Sep. 17, 2015). While rolling window validation is the recommended approach to time series, we believed that the consistent pattern of sales and the computational benefits (rolling window validation takes significantly longer) outweighed the cons.

For our first result we simply converted all our features into numerics and fed them into the XGBoost algorithm and got the following score:

$$\boldsymbol{RMSPE : .1363} \tag{10}$$



The above figure shows the learning rate of the XGBoost training process. In order to pick our optimal model, we choose the boosting iteration that minimizes validation error.

External Data

Our data set also had room for improvement. There were many external factors that influence store sales that were not given to us in the original data set. We added information about the weather for a given day and region using the 'weatherData' R package, which reads data from Weather Underground [1]. We also gathered web search rates for "Rossmann" from Google Search Trends for each region and day [2]. Although stores are labeled without region information, there is variation in which days are holidays between the German states. Using the holiday information, it is possible to label which region of Germany each store is in. This allowed us to link each store the Google Search Trends and weather of its region [5]

Feature Variable Transformations

Gradient boosted trees are good at handling continuous and discrete ordinal data. However, unordered discrete data can be problematic because of the implied ordering that is created when the data is represented numerically. We had two options in this situation: one-hot encoding or imposing order. One-hot encoding simply creates a binary variable for each instance of a feature to remove the implied ordering. The second option was to put our own ordering on these features. We ultimately chose the second option because one-hot encoding can be prohibitively slow for features with a large number of possible values [6].

The order that we chose to impose on our discrete non-ordinal features was to take the mean of the response variables for all instances of each feature. For example, for each store we took the average sales for every example of that store in the training set and replaced the store label with that value. This resulted in unique values for each store, but introduced an ordering based on the average sales of that store. We did mean response variable replacement with Store, Day, and Month.

Although imposing order is often better than the random order of non-ordinal data, it can be problematic if the order introduced is incorrect and we artificially force the tree building algorithm to only pick splits that follow the imposed order. We did find an improvement in our model after introducing these transformations, which indicated that our imposed order was effective.

Time Series

A final consideration that we had to deal with was the fact that we had time series data. Time series data is any set of data over a sequential time interval. Data of this nature often exhibits autocorrelation, which is the correlation of our response variable with itself at different points in time. Part of this was taken into account by keeping track of the Day of Week, Month, and Year. These variables helped to explain cyclic variation in the data. In order to account for possible autocorrelation, we decided to add lagged variables that contained the sales for previous days into each our. This was problematic because our test set did not have concrete information about previous days sales, so we were required to do iterative prediction, feeding our predicted sales into subsequent days to serve as lagged variables. This introduced the danger of error accumulation. We also cross validated using iterative prediction to select a model that did not rely too heavily on the unstable information in the lagged variables [7].

Specifically, we cross validated on a date range that matched the size of our test set (48 days). We introduced lagged variables of the sales from the previous 7 days from the same store. This turned out to not be effective with our current set of parameters and will be discussed further in the results section.

Bayesian Optimization

In order to tune the parameters of our gradient boosted trees model, we chose to use Bayesian optimization. This method is much faster than grid search and is able to arrive at a more optimal solution than random parameter search. Bayesian Optimization is a black box method that takes a list of parameters and a function and finds the optimal parameter set for the function. The process uses the information from all previous parameter set guesses in order to determine the next set of parameters to try.

The use of Gaussian Processes for modeling loss functions has long been recognized as a good method for optimizing hyperparameters [9]. Gaussian Processes interpolate (i.e. curve fit) data with high regularity (i.e. non-erratically) using a proper choice of covariance kernels. Using bayesian optimization, a black box function that is costly to train can be optimized in the least amount of time, assuming a prior [8].

For our problem, given a set of already sampled hyperparameters $\chi = \{x_1, ..., x_N\}$ in $\mathbb{R}^d$ and corresponding sampling function $y = \{y_1, ..., y_N\}$, such that $y_k = f(x_k)$

for $1 \leq k \leq N$ we want to find a continuous function s such that $s(x_k) = y_k$ for $1 \leq k \leq N$.

Here, the only choice that makes sense for our sampling function is the cross validation error for a given set of parameters. Because we cannot assume eta is normally distributed, we held it constant at a sufficiently low number (25% of the default value). We also varied number of boosting rounds internally as it scales based on the validation set (see learning rate). The rest of the hyperperamaeters were optimized by SigOpt.

Also, we estimate noise free observations as we only have a single validation set (k-fold or repeated cross validation would normally be a good estimate of this error). Because we assume a gaussian prior, we must also choose a covariance kernel $K(x, z)$ modeled by covariance matrix K

$$K = \begin{bmatrix} K(x_1, x_1) & \dots & K(x_1, x_1) \\ \vdots & \vdots & \vdots \\ K(x_N, x_1) & \dots & K(x_N, x_N) \end{bmatrix}$$

with basis

$$k(x) = \begin{bmatrix} K(x, x_1) \\ \vdots \\ K(x, x_N) \end{bmatrix}$$

and s defined as:

$$s(x) = k(x)^T K^{-1} y \tag{11}$$

we can calculate our posterior:

$$p(y|\epsilon; K) = ((2\pi)^N \det K)^{\frac{-1}{2}} \exp(-\frac{1}{2} y^T K^{-1} y) \tag{12}$$

Using this posterior, we can then use an acquisition function to calculate the next choice to sample. This function must properly balance the tradeoff between exploration and exploitation (i.e. choosing to sample where there is a lot of uncertainty or maximizing already known local optima). For instance, the expected improvement acquisition function determines the expected improvement over a target variable t evaluating the posterior at x:

$$EI(x) = \mathbf{E}[max(0, t - y)] = \int_{-\infty}^{\infty} (t - y)_+ p(y|x) d_y \tag{13}$$

The choice of a kernel for Gaussian fitting is not a trivial process; neither is the selection of an acquisition function an easy task. Fortunately, there is software available that focuses on optimizing these methods. We choose to use SigOpt for its well implemented API, integration for floats and integers, parallelism,and superb results.

Results

Our initial run with stock parameters and unmodified features of XGBoost resulted in a validation score of 0.1354 RMSPE. We wondered what feature engineering could do, and after implementing feature manipulation specified in methods, the RMSPE was reduced to 0.12013. With SigOpt optimization, we were able to get a 6% decrease in error to 0.1144 over 150 observations. Our results with these methods on the test set (leaderboard) resulted in 0.1363, 0.12112, and 0.1110 RMSPE, respectively. We saw variation between our validation set and test set. This small amount of variance, though, can be explained to random chance.

Table 1: *RMSPE Results*

Model	Validation	Test
Linear Model	*N/A*	0.2541
Mean Guess	*N/A*	0.1824
Boosted Tree Initial	0.1354	0.1363
Boosted Tree External Data	0.1201	0.1211
Boosted Tree Optimized	0.1144	0.1110
Boosted Tree Kitchen Sink	0.1050	0.1181

Next, we theorized that trees should be resilient to noise because each tree is built with optimal features based on the loss function. Therefore, we took a kitchen sink approach. We added all possible predictors that we had gathered. Number of boosting rounds and the time it took to train the model at each step significantly increased with this increase in complexity. Our validation error decreased to 0.10501. Unfortunately, due to resource constraints, we were not able to optimize this parameter set with SigOpt. Furthermore, our test error did not decrease from our previous low of 0.1110.

Finally, we noticed autocorrelation in our training set. Therefore, we theorized that addition of lagged variables would help explain the data. We attempted adding one lagged variable as well as one week worth (7 days), which would account for the spikes seen in Figure 1. We tested with a few parameters of the successful trees found for the data without the one day lagged variables. Performance was similar, although one day lagged variables were far more costly to compute. One week of lagged variables, on the other hand, performed far worse (around 0.20 RMSE). We account this for the fact that the algorithm is weighing the lagged variables extremely heavily, but when we forecast them we are exponentially multiplying error. Refitting the parameters using Bayesian Optimization for the model with the lagged variables turned out to be highly time intensive. We hope to rerun SigOpt to see if the lagged variable model outperforms our current best with optimal parameters when we have the time to recompute.

Another future avenue is to implement rolling window cross validation. Currently our cross validation set is quite small and leads to overfitting on the narrow window of time that it covers. Rolling window cross validation would increase the size of cross validation and thereby improve the generality of our model. The problem lies in the amount of time required for rolling window cross validation models.

References

[1] http://www.wunderground.com/

[2] https://www.google.com/trends/

[3] Shapire, Roberty E. “The Strength of Weak Learnability.” Machine Learning 5 (1990): 197-227.

[4] Chen, Tianqi. “Introduction to Boosted Trees.” University of Washing Computer Science. University of Washington, 22 Oct. 2014. Web.

[5] “Rossmann Store Sales.” Kaggle. Web. https://www.kaggle.com/c/rossmann-store-sales.

[6] Prettenhofer, Peter. “Gradient Boosted Regression Trees.” PyData. 14 Apr. 2014. Lecture.

[7] Bontempi, Gianluca. ”Machine Learning Strategies for Time Series Prediction.” Machine Learning Summer School. ULB, Brussels. Lecture

[8] J. Mockus, V. Tiesis, and A. Zilinskas. The application of Bayesian methods for seeking the extremum. In L.C.W. Dixon and G.P. Szego, editors, Towards Global Optimization, volume 2, pages 117?129. North Holland, New York, 1978.

[9] Snoek, Jasper. “Practical Bayesian Optimization of Machine Learning Algorithms.” University of Toronto, 29 Aug. 2012. Web

[10] “SigOpt Blog.” SigOpt. Web. http://blog.sigopt.com/post/134931028143/sigopt-in-depth-profile-likelihood-vs-kriging.

[11] Hastie, Trevor, Robert Tibshirani, and J. H. Friedman. “Chapter 15.” The Elements of Statistical Learning: Data Mining, Inference, and Prediction. 2nd ed. New York: Springer, 2009. 589-91. Print.

Appendix

A

Show that RMSPE is the same as RMSE of log transformed response variable:

$$RMSPE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\frac{y^{(i)} - \hat{y}^{(i)}}{y^{(i)}})^2}$$

$$RMSE = \sqrt{\frac{1}{n}(y^{(i)} - \hat{y}^{(i)})^2}$$

In order to make $RMSPE$ equal $RMSE$ we need to find a function $f(y)$ that will make:

$$f(y^{(i)}) - f(\hat{y}^{(i)}) = \frac{y^{(i)} - \hat{y}^{(i)}}{y^{(i)}} \quad (1)$$

We can approximate $f(p)$ using Taylor Expansion, with the assumption that δ (the difference between our prediction and response variables) is small:

$$\delta = y^{(i)} - \hat{y}^{(i)}$$
$$f(\hat{y}^{(i)}) = f(y^{(i)} + \delta) \approx f(t) + f'(t)\sigma + ...$$

Drop the higher order terms of the Taylor Expansion and plugging into equation (1) and simplifying:

$$f(y^{(i)}) + f'(y^{(i)})\delta - f(y^{(i)}) \approx \frac{\hat{y}^{(i)} - y^{(i)}}{y^{(i)}}$$
$$f'(t)\delta \approx \frac{\delta}{t}$$
$$f'(t) \approx \frac{1}{t}$$
$$f(y) = \ln(y) + C$$

This shows that the correct way to make RMSPE and RMSE approximately equal is to take the log transformation of our prediction and target variables.