Ankle Breakers Final Paper

Brian Cleary, Kevin Ellis, & Derrell Gottwald

May 12, 2016

INTRODUCTION

The Rossmann Company currently operates over 3,000 drug stores in 7 European countries. Dirk Rossmann founded the first self-service drugstore in Germany in 1972 and since then the company has experienced remarkable growth. Current business operations require store managers to predict daily sales for up to six weeks in advance. With the number of store managers each using different statistical methods, the accuracy of results has been largely varied. Reliable sales forecasts enable stores managers to create effective staff schedules with the goal of improving productivity and motivation. It becomes apparent that store managers could benefit from a centralized prediction model thus alleviating that requirement and allowing them to focus their efforts on their customers and staff.

Objective

The objective of this project is to develop a robust predictive model which predicts 6 weeks of daily sales for 1,000 stores located across Germany. The Rossmann Company has supplied data with basic factors such as School or State holidays to facilitate predictive model development.

Problem Statement

Using the software program *R*, develop a robust and accurate predictive model capable of predicting 6 weeks of daily sales for 1,000 stores located across Germany whilst developing a process guide for detailing key aspects related to model development such as exploratory data analysis and feature engineering. The accuracy of the predictive model is evaluated using Root Mean Square Percentage Error (RMPSE).

Model Evaluation

For this project, a predictive model is developed using training data where sales is provided. The resulting predictive model is used to predict sales on a test data set which is then evaluated against answers which are known but not provided using RMPSE which is given by:

$$RMPSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2}$$

where y_i denotes the sales of a single store on a single day and $\hat{y}_i$ denotes the corresponding prediction. Any day and store with zero sales is ignored in scoring, such as days when the store is closed.

EXPLORATORY DATA ANALYSIS

Data was provided in the form of three comma-separated value (.csv) files called: train.csv, test.csv, and store.csv. The file store.csv provides general information about the 1,000 stores whereas the other two contain daily information such as sales and the number of customers. The first step was to combine general store information with the training and testing data to facilitate data analysis using the *merge* function.

```
#Import Files as Data.Tables
    train = fread('train.csv',header=TRUE,stringsAsFactors=FALSE) #870,191 Observations
    test = fread('test.csv',header=TRUE, stringsAsFactors=FALSE) #42,000 Observations
    store = fread('store.csv',header=TRUE, stringsAsFactors=FALSE) #1,000 Observations
#Merge Store Information With Training & Test Data
    train = merge(train,store,by='Store') #Overwriting Previous Train Data.Table
    test = merge(test,store,by='Store') #Overwriting Previous Test Data.Table
```

Each .csv file was imported into *R* and placed into a data table structure. The data table structure is ideal for managing "big data" where one has hundreds of thousands of rows with hundreds of columns. The functionality provided by the data table structure facilitates data manipulation orders of magnitude faster than other structures in *R* such as data frames. This "speedup" is critical in managing this data set, especially with regard to feature engineering. The resulting merged data sets represent the *baseline* or *stock* data sets with which to begin exploratory data analysis. From this point forward, *stock* data or features will refer to fields present in the original data sets.

Initial Evaluation of *Stock* Data Set

Stock features from the original data set include: *Store*, *DayOfWeek*, *Date*, *Sales*, *Customers* (not included in test data set), *Open*, *Promo*, *StateHoliday*, *SchoolHoliday*, *StoreType*, *Assortment*, *CompetitionDistance*, *CompetitionOpenSinceMonth*, *CompetitionOpenSinceYear*, *Promo2*, *Promo2SinceWeek*, *Promo2SinceYear*, and *PromoInterval*. *Store* represents the ID number for the store, in this case 1 to 1,000. *DayOfWeek* indicates a value from 1 to 7 where a value of 1 represents Monday, 2 represents Tuesday, and so forth. *Date* indicates the specific date. *Sales* and *Customers* represent the total sales and customers respectively. *Open*, *Promo*, *SchoolHoliday*, and *Promo2* are bit columns, as they contain the value 0 or 1 where 0 indicates closed and 1 indicates open for *Open*, 0 indicates that a store is not running a promotion or a continuing and consecutive promotion and 1 indicates that they are for *Promo* and *Promo2* respectively, and 0 indicates that there is not a school holiday and 1 indicates there is for *SchoolHoliday*. *StateHoliday* indicates the type of state holiday where 'a' is a public holiday, 'b' indicates Easter Holiday, 'c' indicates Christmas, and '0' is no holiday. *StoreType* and *Assortment* describe the type of store based on four different store models and on different assortment levels respectively. *CompetitionDistance*, *CompetitionOpenSinceMonth*, and *CompetitionOpenSinceYear* describe the distance in meters to the nearest competition and the approximate month and year as to when the competitor opened respectively, Lastly, *Promo2SinceWeek*, *Promo2SinceYear*, and *PromoInterval* describe the approximate week and year when that specific store began running a continuing and consecutive promotion (*Promo2*) as well as the intervals the promotion is started anew respectively. As mentioned previously, a store that is running a *Promo2* (indicated by a 1) is running a continuing and consecutive promotion that began for a specific store on the approximate week and year provided by *Promo2SinceWeek* and *Promo2SinceYear*. A *PromoInterval*, for example, indicated by "Feb,May,Aug,Nov", means each round begins in February, May, August, and November and continues for that month.

After obtaining *baseline* data sets, it is necessary to validate the quality of data. The objective of this project is to predict sales, thus the first thing to ensure is that each store has a valid data point for every day in both *baseline* data sets. The training set spans January 1, 2013 to June 19, 2015 whereas the test set ranges from June 20, 2015 to July 31, 2015. For each store, it is expected that there will be 900 data points for 900 days in the training set and 42 data points for 42 days in the test set. The data provided includes line items for days when stores are closed so we can infer that the number of data points is equal to the number of days. Interestingly, it was found that there were 162 stores in the training data set where data was not recorded between July 1, 2014 and December 31, 2014 (Figure 1).

Correlation Method	**Correlation Coefficient**
Pearson Method	0.8268285
Spearman Method	0.8344475
Kendall Method	0.8276364
Average	**0.8296375**

Table 1: Calculated correlation coefficients between *Sales* & *Customers* using different methods.

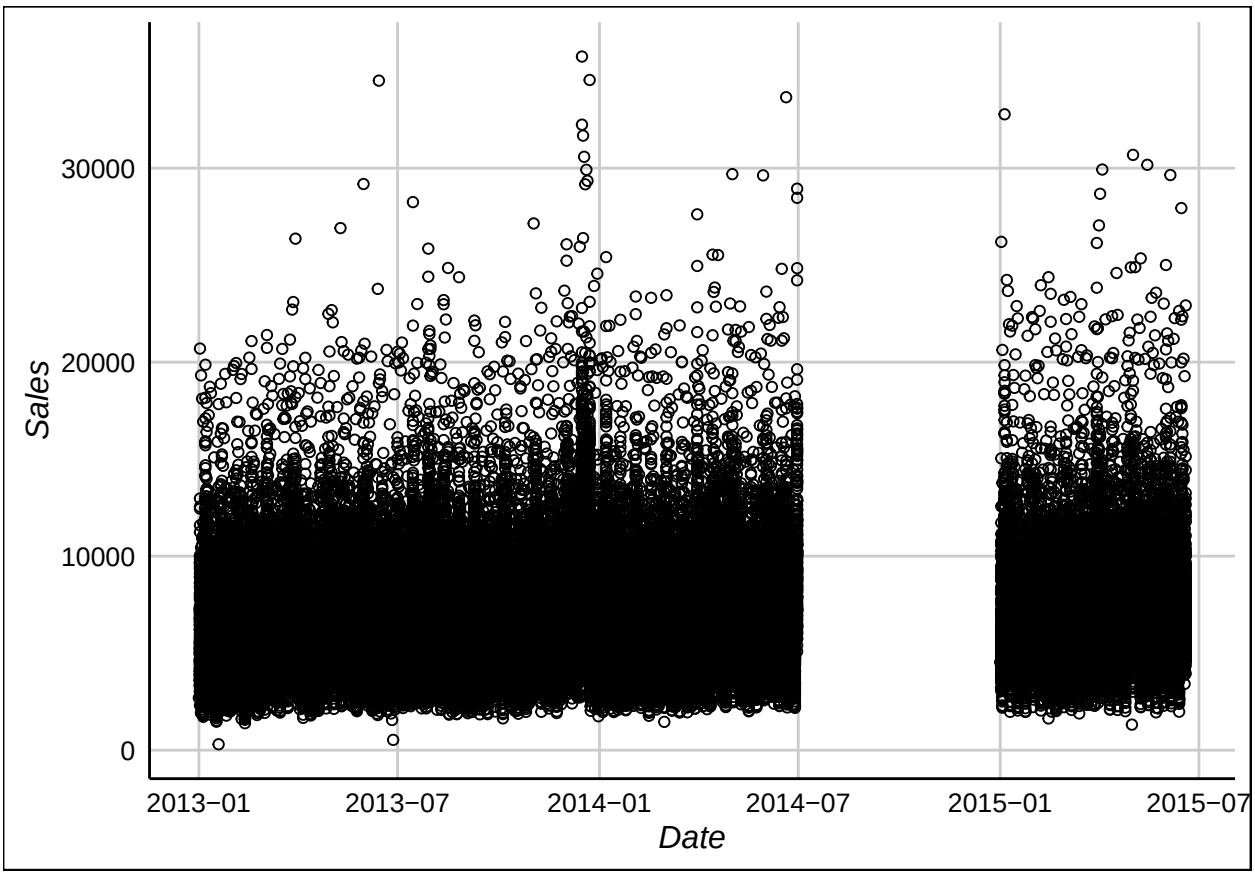

Figure 1: 162 stores have missing *Sales* data in training set between July 1, 2014 & December 31, 2014.

The scatterplot shown in Figure 1 details potential difficulties in predicting sales since no data is available in this time frame for these stores. It should be noted that there is no data provided for any of the *stock* features in this time frame for these specific stores and that the issue is not simply relegated to *Sales*. Meanwhile, the test set has all 1,000 stores where each store has 42 line items *stock* features are populated.

In problems involving sales, the number of customers is typically highly correlated. To check this notion, observe the results shown in Table 1. The correlation coefficients between sales and customers were calculated for three different methods and produced an average of 0.83 which indicates that sales and customers are highly linearly correlated. However, *Customers* is not a field present in the test set, i.e. it would need to be predicted like sales. However, as shown later, it was believed that features could be created using this field which would improve model accuracy.

Calculation of correlation coefficients between *Sales* and the remaining *stock* features provided little insight into possible linear relationships. Nearly all features, aside from *Promo* and *DayOfWeek*, had correlation coefficients close to zero indicating no linear relationship. The features *Promo* and *DayOfWeek* had correlation

coefficients of -0.18 and 0.39 respectively, where the negative sign simply means that as one variable increases the other decreases. This result indicates that date-related and promotional-related features are likely to be more important. To gain further insight, consider the histograms shown in Figure 2.

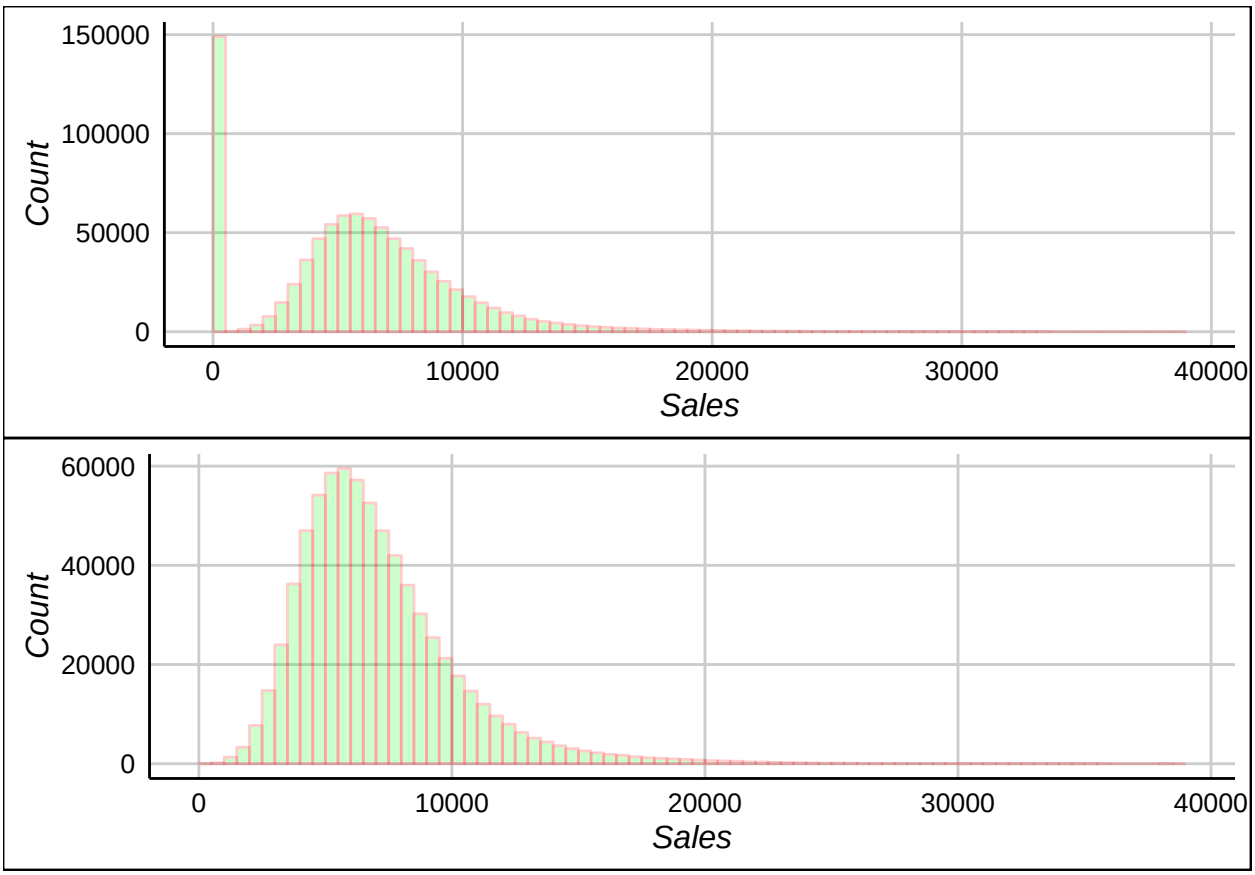

Figure 2: Basic histograms for *Sales* when store is both open & closed and open.

The histograms shown in Figure 2 reveal a number of interesting characteristics with regard to sales. The first histogram was created by including days where stores were closed and *Sales* were 0, whereas the second histogram was created by including only days where stores were open. Interestingly, with stores being open, we are still witnessing a number of instances where *Sales* is 0. Further research reveals there are 51 observations that fall into this category. More importantly there is no pattern to them, such as if the instances were occurring more frequently with a specific store or on a certain day of the week. Thusly a key assumption for model development is to include only data points where *Sales* is not equal to 0. Additionally, the second histogram is skewed positively indicating the mass of the distribution is concentrated to the left and the right tail is longer, potentially hinting at the presence of outliers. Further investigation reveals there are 816 observations with *Sales* greater than 25,000 and indeed some of these instances seem to be outliers when compared with the rest of the sales for that particular store. Now consider a histogram of the mean of *Sales* for each of the 1,000 stores where *Sales* is not equal to 0 (Figure 3).

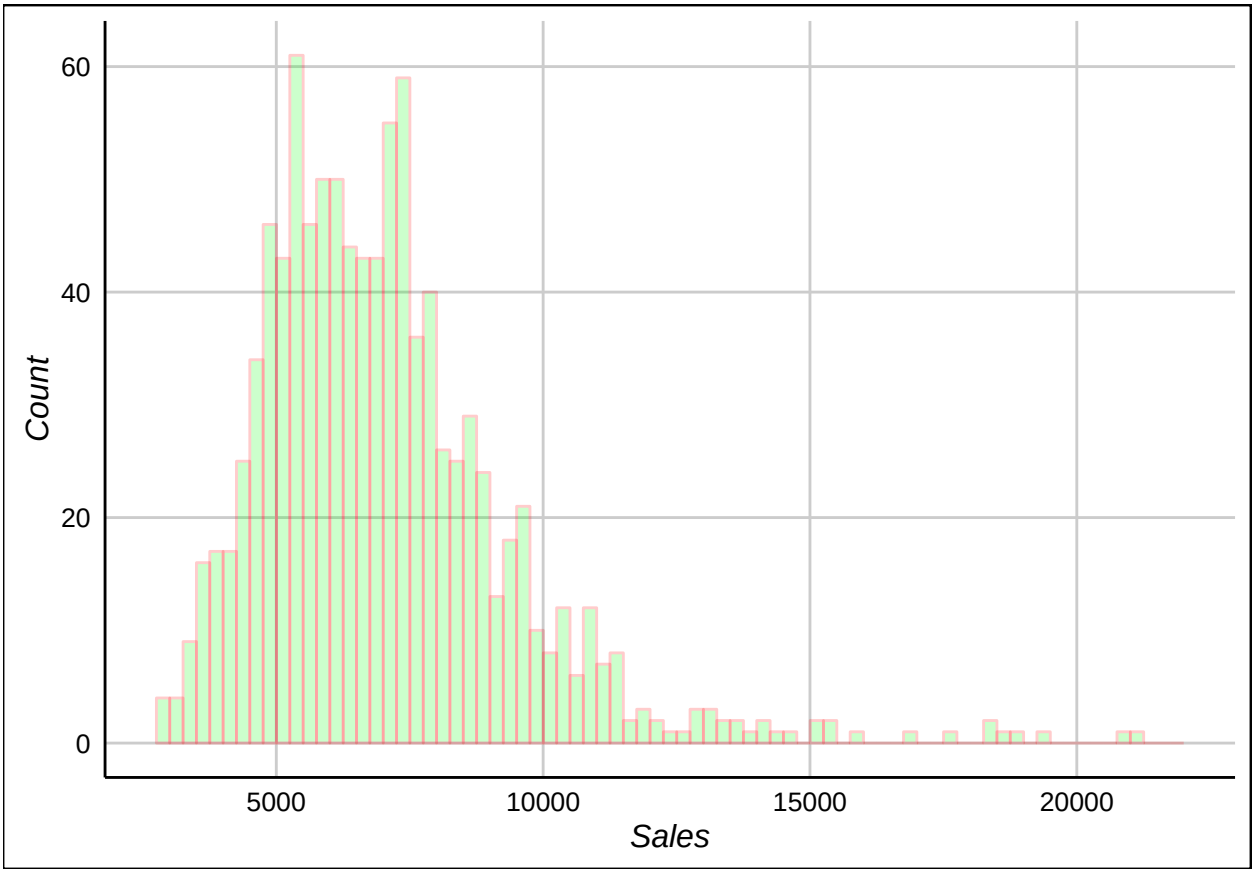

Figure 3: Histogram for the mean of sales by *Store* when sales is not equal to 0.

The histogram shown in Figure 3 reveals a more unstable nature and is once again positively skewed. The mean of sales for most stores appears to range from 5,000 to 7,500.

Unusual Features of *Stock* Data

Investigation of linear relationships between *Sales* and other *stock* features revealed some unusual aspects. The most unusual being that the correlation coefficient between *Sales* and *Promo* was 0.39 and yet *Sales* and *Promo2* was -0.12. Intuition suggests that promotional events would affect *Sales* and yet the correlation coefficient revealed no such relationship. As a result of merging data sets, if a store was running a continuing and consecutive promotion, the value for *Promo2* was 1 for every single date. *Promo2* was a feature of the store.csv file, and so when training data was merged with general store information, the value for *Promo2* was copied for every entry in the training set and thus for every date for that particular store. The same is true as well for the following *stock* features: *CompetitionDistance*, *CompetitionOpenSinceMonth*, *CompetitionOpenSinceYear*, *Promo2SinceWeek*, *Promo2SinceYear*, and *PromoInterval*. It makes sense that these features, which include *Promo2* would not be correlated with *Sales* because they do not change with regard to the *Store*. Research revealed that certain stores would start running a continuing and consecutive promotion in the Summer of 2014 and yet *Promo2* would be 1 for the year 2013. The same was true with regard to competition-related features. It became imperative to create new features that would capture the importance of these variables by explicitly determining when a store was running a continuing and consecutive promotion or if a competitor had opened nearby.

Time Series Models

Auto regressive integrated moving average (ARIMA) methods were used for exploratory data analysis as well as initial prediction. ARIMA methods are ideal for time series data with seasonal components. The models are simple and easy to implement as compared to more complex machine learning methods and do not require ancillary features to improve model accuracy. ARIMA methods require defining parameters specific to the time series which are found through observation of historical patterns in data. The immediate challenge is that each store exhibits a different trend for *Sales*, thus the most obvious approach is to develop 1,000 separate models specific to the store. Should this approach be adopted, there may be patterns that influence the behavior of *Sales* when stores are observed in groups. To investigate this potentially commonality among stores, two approaches were attempted: k-means clustering and grouping methods based on aggregated statistics. Each provided deeper insight into the relationships between *Sales* and the *stock* features, but no significant patterns were revealed.

A naive ARIMA model was created for an initial prediction effort. The daily average of sales for all stores was collected from the training set, equating to 900 data points (900 consecutive days). An ARIMA model was fit with parameters chosen by investigating graphs of the autocorrelation (ACF) and partial autocorrelation functions (PACF). The predictions, as a result of the model, were evaluated with an RMPSE of 0.5354. Overall, a good score for such a naive approach.

PREDICTIVE MODEL DEVELOPMENT

The following sections, though appearing separate in this paper, were performed in conjunction with one another. Typically, model selection followed by baseline model development using *stock* features must occur a number of times before a firm grasp of the nature of the problem is understood. At that stage, final model selection and baseline model development occurs, followed by rounds of feature engineering and hyper parameter tuning peaking at final model development.

Feature Engineering

Exploratory data analysis revealed that *Sales* could be influenced by a number of individual features or by a combination of features. For this reason a large number of additional features were created to explore those influences. As described in greater detail in **Model Development**, converting non-ordinal numeric features to factors, creating an explicit breakout of *Promo2* and competition-related features, and adding running totals and count-related features each contributed the most to improving model accuracy as compared to the *baseline* model.

Converting Non-Ordinal Numeric Features To Factors

The desire to convert non-ordinal numeric features to factors results from the presence of seasonality. Consider Figure 4 which details the sales of store 679.

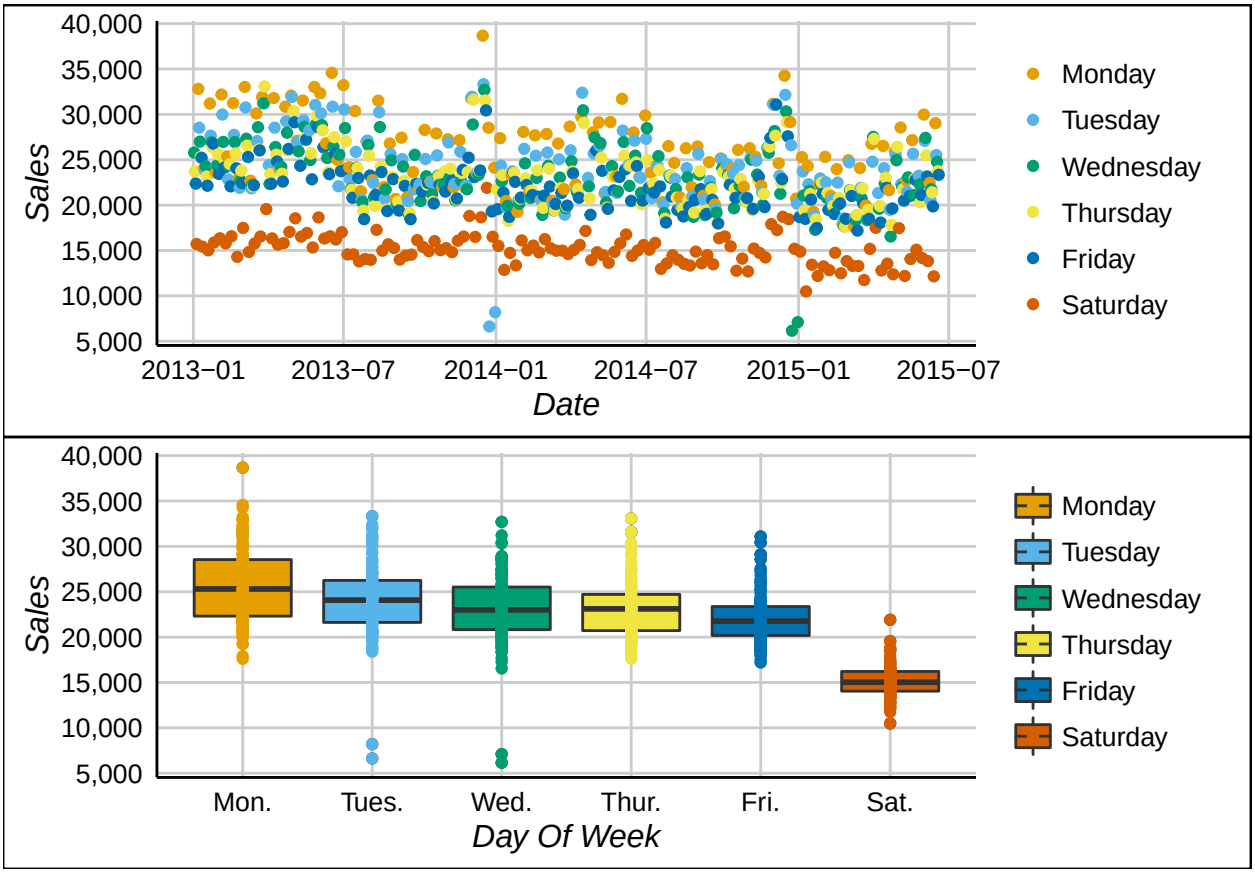

Figure 4: Scatter plot & box plot of sales of store 679.

In Figure 4, the first graph represents a scatter plot of sales for store 679. It is obvious that the store is closed on Sundays, but more importantly the smallest amount of sales occurs on Saturdays and the most on Mondays. This disparity is clearly revealed in the boxplot shown in the second graph where average sales for Saturday is much lower compared to Monday. Additionally, there appear to be spikes in sales at similar months of the year (around July and towards the end of December). These observations support the notion that seasonality is present. This analysis can be extended to other stores and the presence of seasonality is even more apparent in some cases.

With the presence of seasonality it is imperative to establish features able to detect it. By converting non-ordinal numeric features to factors we are creating new features. For example, by converting the numeric variable *Store* to a factor we are creating 1,000 new features. During model development, each store is treated individually which more effectively reveals seasonal trends specific to that store. The same premise can be applied to date-related and store-related features as well. In order to convert numeric variables to factors the *as.factor()* function was used. When converting non-ordinal numeric features to factors, it is important to ensure that the number of levels in the both the training and test set are equal. Consider converting *Year* to a factor. In the training set there are three levels: 2013, 2014, and 2015. However, in the test set there is only one level, 2015. Additional levels must be added to unify the features between the training and test set.

Determining Store Location

Using only the *stock* features *StateHoliday* and *SchoolHoliday*, the German state each store is located in can be determined with a strong degree of confidence. A unique property in terms of either the *StateHoliday*

German State	Abbr.	Method	No. Days	No. Stores
Saxony	SN	Unique *STH* - 11/20/2013		68
Thuringia	TH	Unique *SCH* - 02/18/2013 - 02/23/2013		33
Bavaria	BY	*SCH* (*NCW*) - 07/25/2013 - 09/16/2013	31	162
Saarland	SL	*SCH* (*NCW*) - 07/08/2013 - 08/16/2013	30	36
Baden-Württemberg	BW	*SCH* (*NCW*) - 07/25/2013 - 09/06/2013	32	59
Hamburg	HH	*SCH* (*NCW*) - 03/04/2013 - 03/15/2013	10	27
Saxony-Anhalt	ST	*STH* - 01/06/2014		51
Hesse	HE	*SCH* (*CW*) - 10/11/2013 - 10/28/2013	12	103
Lower Saxony	NI	*SCH* (*NCW*) - 01/24/2013 - 02/10/2013	2	18
North Rhine-Westphalia	NW	*SCH* (*NCW*) - 07/22/2013 - 09/03/2013	32	262
Berlin	BE	*SCH* (*NCW*) - 06/19/2013 - 08/02/2013	33	81
Schleswig-Holstein	SH	*SCH* (*NCW*) - 10/04/2013 - 10/18/2013	11	100
			Total	**1,000**

Table 2: Unique properties defined for each German state and then applied to stores. Under **Method**, *SCH* stands for school holiday, *STH* stands for state holiday, *NCW* stands for not counting weekends, & *CW* stands for counting weekends.

or *SchoolHoliday* was defined for each state and then applied to each store. If the store met the conditions for a property, it was inferred that the store belonged in that state. The properties defined for each state are outlined in Table 2. It is important to note that the order in which the property is applied to stores is important. For example, determining stores that reside in Saxony is always first, followed by those that reside in Thuringia, and so forth. All state and school holidays in Germany are provided in detail from the website: http://www.schulferien.org/.

Based on Table 2 determining stores that reside in Saxony are simple because that state celebrates a unique holiday which no other state celebrates. In this case, a store resides in Saxony if on November 20, 2013 the value for *StateHoliday* is equal to 'a' which is defined as a public holiday. Consider the more complicated case of Saarland. From Table 2, the unique property for that state is that a school holiday is celebrated between July 8, 2013 and August 16, 2013. In total there are 40 days within that range. Through significant research it was discovered that *SchoolHoliday* is mainly 0 for Saturday and Sunday (hence the *NCW* notation in Table 2), even during a school holiday. So the number of days when *SchoolHoliday* is 1 becomes 30 (**No. Days** in Table 2). To determine the stores that reside in Saarland simply sum the number of instances where *SchoolHoliday* is 1 and determine if the total is 30 for each store which has not already been associated with a state. If so, the store resides in Saarland. The number of stores in German states is shown in Figure 5.

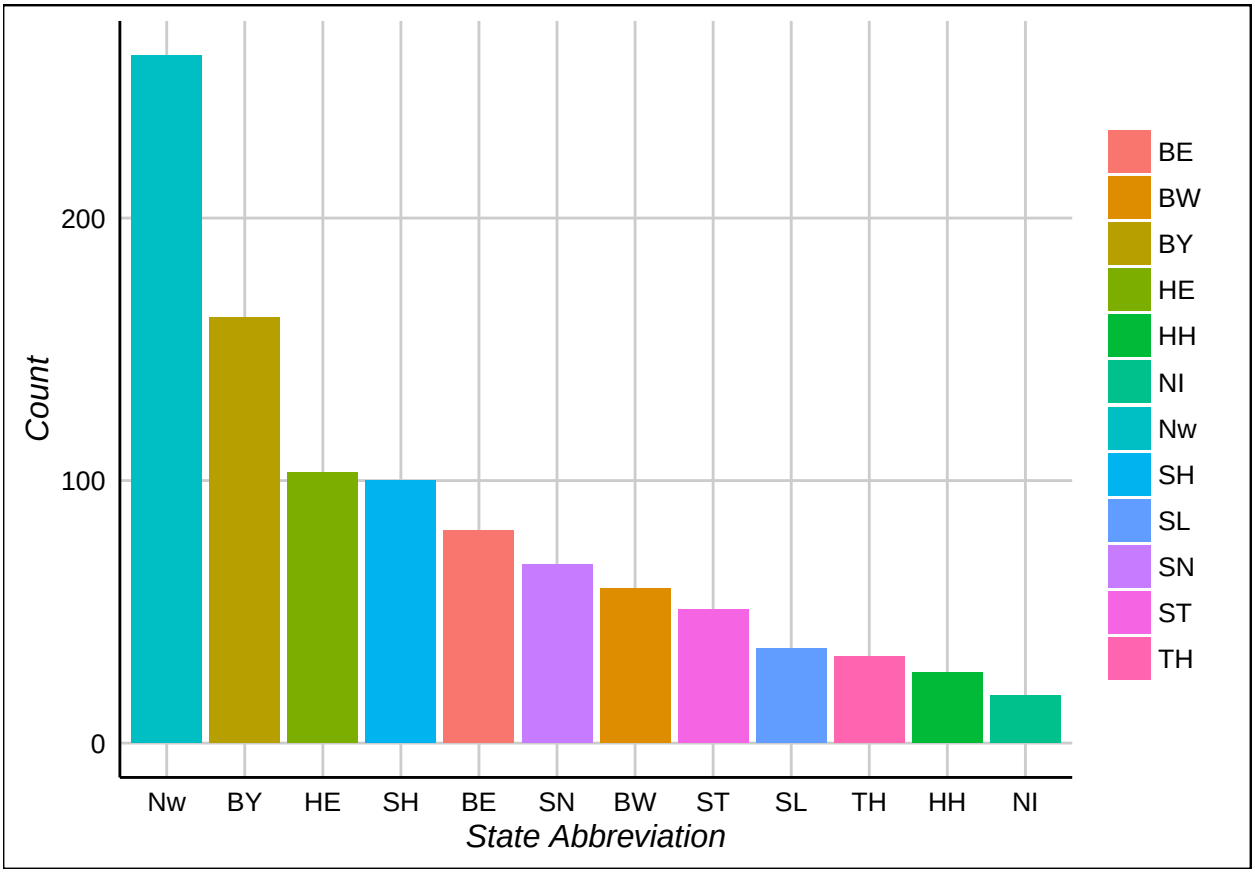

Figure 5: Number of stores by German state.

Weather Data

Weather data was collected via a Kaggle Forum (https://www.kaggle.com/c/rossmann-store-sales/forums/t/17058/weather-at-berlin-us-airport). In the forum 16 .csv files were posted, one for each state, which provided weather data measured daily at local airports. Measured values include: temperature, humidity, visibility, wind speed & direction, and precipitation. Weather data measured locally was assumed to be representative of the entire state.

The process in which the 16 .csv files were generated involved using the *weatherData* package which requires that the *devtools* package to be installed as well. The *weatherData* connects to a server via an API where the user supplies a weather station ID (*KBML* is the weather station ID for the Berlin International Airport) and a date range. The weather data for that station is imported directly into *R* and then exported as a .csv file. This process is time-consuming, so the 16.csv files were downloaded directly from the forum.

Weather data was appended to the training and test sets by first importing all 16 .csv files into *R* and then merging the data sets using the location of the store, in this case the German state, and the *Date*, independently for each of the 16 .csv files (one weather data file for each state).

Economic Data

Economic data was collected with the thought that trends related to the German economy would affect sales. Economic data was collected from the website: www.quandl.com. Economic data could be imported directly

from *Quandl* into *R* via an API call, but it was easier to download the .csv files for different metrics and then import each of the files and merge the data via *Date* (similar process defined in the section **Weather Data**). The metrics of interest include: crude oil prices (WTI, Brent, & OPEC), exchange rate as compared to USD, inflation, producer price index, consumer sentiment, composite leading indicator, and business confidence index. During model development, when the importance of different features was being evaluated, it quickly became clear that the economic data was not near as important as other features. More importantly, these features are similar to *Customers* in the sense that they would need to be predicted in the future state which is not useful.

Adding Running Totals & Count Features

Creation of these features resulted from the need to explicitly determining when a store was running a continuing and consecutive promotion or if a competitor had opened nearby. To solve this need, two features were added: *In_Promo2* where when this variable is 1 a store is running a continuing and consecutive promotion on that specific date and if 0 it is not and *With_Competition* where when the variable is 1 a particular store has competition some meters away as specified by *CompetitionDistance* and 0 if there is no nearby competition. The detailed process for adding *In_Promo2* is outlined below. The process is similar for adding *With_Competition*. The most efficient way to add these features is by first merging the training and test sets so the process need only be completed once. Begin by initializing the feature to 0, then for *In_Promo2*, anytime the *PromoInterval* is populated set *In_Promo2* equal to 1 for the months specified in *PromoInterval*. Finally for each store, determine the exact date for when either a store began running a continuing and consecutive promotion or if a competitor had opened nearby. Finally set the value for the respective feature to 0 for all dates that occur before the calculated start date.

```
#Merge Train & Test Set Together - MUST ENSURE THAT COLUMN ORDER IS THE SAME IN BOTH SETS!
    #Sort Train & Test By Date & Store
        test=test[order(Date,Store)]
        train=train[order(Date,Store)]
    #Combine Test & Train - Bind Test Set To End Of Training Set
        DT=rbind(train,test)

#Create Feature In_Promo2
    #Initialize New Feature & Set Equal To 0
        DT$In_Promo2=0 #Accounts For Stores That Do Not Run A Promo2
    #Set In_Promo2 To 1 For Specific Months When PromoInterval Is Specified
        DT$In_Promo2[DT$PromoInterval=='Jan,Apr,Jul,Oct' & DT$Month %in% c(1,4,7,10)]=1
        DT$In_Promo2[DT$PromoInterval=='Feb,May,Aug,Nov' & DT$Month %in% c(2,5,8,11)]=1
        DT$In_Promo2[DT$PromoInterval=='Mar,Jun,Sept,Dec' & DT$Month %in% c(3,6,9,12)]=1
    #For Each Store, Determine Exact Date When A Continuing and Consecutive Promotion
    #Was Started & Set In_Promo2 To 0 For Dates That Are Less Than The Exact Date.
        for (i in 1:1000) {
            Subset_Store=DT[DT$Store==i,]
            if (unique(Subset_Store$Promo2) == 1) {
                NewDate=unique(DT$Date[
                          which(DT$Week==unique(Subset_Store$Promo2SinceWeek) &
                                DT$Year==unique(Subset_Store$Promo2SinceYear) &
                                DT$Day_In_Month==which.min(Subset_Store$Day_In_Month))])
                if (length(Date) == 1) {
                    DT$In_Promo2[DT$Store==i & DT$Date<NewDate] = 0
                }
            }
        }
    DT$In_Promo2[DT$Open == 0] = 0 #When The Store Is Not Open, Set In_Promo2 = 0
```

With the addition of these features we now have 4 features where the values are either 0 or 1 and include: *Promo*, *StateHoliday*, *In_Promo2* and *With_Competition*. To further quantify the importance of these columns, 4 additional features were added to each of the aforementioned on a store-by-store basis and include: a running total for when the feature is equal to 1, a running total for when the feature is equal to 0, a count of the successive iterations for when the feature is 1, and a count of the successive iterations for then the feature is 0. Running totals add either a value of 1 or 0 to the previous total depending on whether a 1 or 0 was detected in the feature. If a running total was designed to track when *Promo* was equal to 1, each time a value of 1 was encountered for a specific store, the total would increment by 1. If a 0 was encountered then the total would remain unchanged. Each of the running totals is initialized to 0 and begins to increment onward from January 1, 2013. The count is similar to the running total except the sum is reset to zero. Suppose there are 4 successive dates for a store when *Promo* is 1 before *Promo* changes to 0, a count of successive iterations for when *Promo* is 1 would add 1 to the total 4 times giving a total of 4 before resetting to 0 because the a value of 0 was detected in *Promo*. In this case, the count represents a number of successive days the store was running a promotion. The process for creating a running total and a count of successive iterations is given below and makes use of the functionality of the data table structure in *R* and will remain the same for the other features.

```
#Recall Section To Merge Training & Test Data set From Before
#Promo
    #Add Running Total Of Days In Promo (When Promo Is 1)
        #Adds Count
            DT[,Total_Count_Days_In_Promo:=1:.N,by=c("Store","Promo")]
        #Sets Values To NA When Promo = 0
            DT$Total_Count_Days_In_Promo[DT$Promo==0] = NA
        #Replaces NA Values With Closest Non-Zero Value
            DT[,Temp:=na.locf(Total_Count_Days_In_Promo,na.rm=FALSE),by=c("Store")]
        #Deletes Original Column
            DT[,Total_Count_Days_In_Promo:=NULL]
        #Renames Temp Column
            DT=rename(DT,c("Temp"="Total_Count_Days_In_Promo"))
        #Sets Remaining NA Values To 0 - Cause of na.locf()
            DT$Total_Count_Days_In_Promo[is.na(DT$Total_Count_Days_In_Promo)==TRUE]=0
    #When Promo == 1 Count Successive Iterations, But Return To 0
        #Sets Up Groups of Successive Iterations Where Promo Is 1
            DT[,group:=cumsum((Promo==1 & shift(Promo,n=1,type='lag')==0)),by=c("Store")]
        #Have to Zero Conditions Where Promo == 0
            DT$group[DT$Promo==0] = 0
        #Create a Basic Count But Subset By Store & Group
            DT[,Count_Days_In_Promo:=1:.N,by=c("Store","group")]
        #Have to Zero Conditions Where Promo == 0 - Essentially a Running Total Otherwise
            DT$Count_Days_In_Promo[DT$Promo==0]=0
        #Delete Secondary Group Column
            DT[,group:=NULL]
```

Adding Lag Features

Creation of these features resulted from belief that somehow implementing *Customers* would improve model accuracy. The issue was how to integrate this feature into the test set since it is must be predicted like *Sales*. However, what if we were able to calculate statistics on historical data? Consider a specific date for a certain store in the test data set. What if we could aggregate values for *Customers* that were nonzero and average them for a 2 month period of time prior to that date? What if we aggregated for 3 months? 6 months? The average of that aggregated value could become an observation in the test data set. The difficulty of creating these features is that the averages must be calculated on a date-by-date basis specific to a certain store. If

not performed in an efficient manner, the process could take days to complete. Lag features were added for both *Customers* and *Sales* for the following time periods: 1 month, 2 months, 3 months, 6 months, 9 months, and a year. The procedure outlined below was able to accomplish feature creation in a matter of minutes using functionality of the data table structure.

```
#Merge Train & Test Set Together - MUST ENSURE THAT COLUMN ORDER IS THE SAME IN BOTH SETS!
    #Sort Train & Test By Date & Store
        test=test[order(Date,Store)]
        train=train[order(Date,Store)]
    #Combine Test & Train - Bind Test Set To End Of Training Set
        DT=rbind(train,test)
#Create Variables That Calculate A Date A Period Of Time Prior To The Current Date
    DT[,Lag_Month:=Date %m-% months(1)] #1 Month
    DT[,Lag_Two_Month:=Date %m-% months(2)] #2 Month
    DT[,Lag_Quarter:=Date %m-% months(3)] #3 Month
    DT[,Lag_Six_Month:=Date %m-% months(6)] #6 Month
    DT[,Lag_Nine_Month:=Date %m-% months(9)] #9 Month
    DT[,Lag_Year:=Date %m-% months(12)] #Year
#Wish To Investigate Average Sales Per Customer - Add Feature First
    DT$Sales_Per_Customer[DT$Sales!=0 &
                          is.na(DT$Sales)==FALSE]=DT$Sales[DT$Sales!=0 &
                                                           is.na(DT$Sales)==FALSE] /
                                                  DT$Customers[DT$Sales!=0 &
                                                               is.na(DT$Sales)==FALSE]
#Create 1,000 Data Frames - One for Each Store - Parsing Helps Improve Speed of Loop
    for (i in 1:1000){
        eval(parse(text =
                    paste("Store",i,"=subset(DT[DT$Sales!=0 & is.na(DT$Sales)==FALSE,",
                          "c(1,7,10,11,13,15,65,70,79,80,81,82,83,84,85),with=FALSE],",
                          "Store==",i,")",sep="")))
    }
#Calculate Statistics
    for (i in 1:1000) {
        y0 = as.symbol(paste0("Store",i)) #Creates Symbol "Store1" Or "Store345"
    #Lag 1 Month Where eval(y0)$Date Equates To Store1$Date And So Forth
        DT[Store==i,
            Historical_Average_Sales_Month :=
                mean(eval(y0)$Sales[eval(y0)$Date>=Lag_Month &
                                    eval(y0)$Date<=Date]),
                    by="Date"]
        DT[Store==i,
            Historical_Average_Customers_Month :=
                mean(eval(y0)$Customers[eval(y0)$Date>=Lag_Month &
                                        eval(y0)$Date<=Date]),
                    by="Date"]
        DT[Store==i,
            Historical_Average_Sales_Per_Customer_Month :=
                mean(eval(y0)$Sales_Per_Customer[eval(y0)$Date>=Lag_Month &
                                                 eval(y0)$Date<=Date]),
                    by="Date"]
        DT[Store==i,
            Historical_Average_Sales_With_Promo_Month :=
                mean(eval(y0)$Sales[eval(y0)$Date>=Lag_Month &
                                    eval(y0)$Date<=Date & eval(y0)$Promo==1]),
```

```
                by="Date"]
    DT[Store==i,
        Historical_Average_customers_With_Promo_Month :=
            mean(eval(y0)$Customers[eval(y0)$Date>=Lag_Month &
                                    eval(y0)$Date<=Date & eval(y0)$Promo==1]),
                by="Date"]
    #.......
```

As is apparent from the process outlined above, additional statistics were calculated aside from simple averages. Averages of *Sales* when *Promo* is equal to 1 or averages of *Customers* when *SchoolHoliday* is 1 for all of the aforementioned time periods were calculated as well. Upon completion of this phase, hundreds of features involving different statistics were created.

Model Development

As mentioned previously, model development was performed in conjunction with feature engineering and is typically an iterative process. Model selection followed by baseline model development occur a number of times before a firm grasp of the nature of the problem is understood. After final model selection, especially in the case of this problem, numerous rounds of developing models with different feature and hyper parameter sets had to occur before final mode development.

Model Selection

In exploratory data analysis a number of models were applied with the purpose of making sense of the relationships between *Sales* and *stock* features and include: linear regression, Generalized Additive Models, and time series models. Calculation of correlation coefficients and linear regression models revealed few features (and even interactions) with linear relationships to *Sales*. Generalized Additive Models were used to investigate the non-linear effects between features however, interactions could not be investigated. ARIMA models were used for initial prediction efforts. Though easy to implement, the low values of RMPSE sought required a more sophisticated method.

The methods mentioned previously were more suited to exploratory data analysis which coerced investigation of more advanced machine learning methods. For completeness, Neural Networks and Support Vector Machines were investigated for possible use in this project, however as these methods are more suited towards classification they were not pursued further.

Tree-based methods for regression are both simple and powerful which lends itself to their widespread use. Tree-based methods are conceptually easy to visualize as they simply partition the feature space and subsequently fit a simple model to each partition. Typically tree-based methods are visualized as a top-down structure where each split (or branch) is determined by a condition further splitting the feature space. This process continues until at the bottom there are the "leaves" which contain the prediction score. There are a number of packages in *R* that employ tree-based methods, however each is slightly different and offers its own advantages.

The first package is called *randomForest* and is based on the *random forests* procedure. *Random forests* is a modification of bagging (also known as bootstrap aggregation) which builds a large collection of de-correlated trees, and averages them. The bagging process has been shown to work well for high-variance, low-bias procedures such as tree-based methods as they are able to capture complex interactions and if grown sufficiently deep have a low bias. At this point it is worthwhile to briefly discuss bagging and how it is integrated with random forests. Before bagging we need to understand a method called the bootstrap. The bootstrap is an extremely powerful tool that quantifies the uncertainty associated with a given estimate or statistical method. Suppose we have a data set and we compute a statistic using that one data set. In this simple case we aren't able to see how variable that one statistic is (how much the statistic or statistical method changes). Rather

than repeatedly obtaining additional independent data sets from the population, we instead obtain distinct data sets by repeatedly sampling from the original data set. The bootstrap is typically a method of assessing accuracy of a parameter estimate or prediction however, it can be used to actually improve the estimate or prediction. Bagging exploits the fact that the bootstrap mean is approximately a posterior average (Bayesian Statistics). More explicitly, suppose we have a single prediction from a model that was fit to bootstrap sample (sample of the original data set), bootstrap aggregation or bagging, averages that prediction over a number of bootstrap samples (each bootstrap sample is random and distinct) thereby reducing the variance of the prediction (improved accuracy). For regression analysis with regard to *random forests*, we fit a regression tree a number of times to many different bootstrap sampled versions of the training data and average the result. There are many benefits to using *random forests*, but most importantly they are accurate with surprisingly little tuning and they are not prone to overfitting. However, *random forests* struggle with correlated features. Consider two features that are perfectly correlated. Suppose for constructing one specific tree, the algorithm needs at least one of those features which it will choose randomly. In *random forests* this choice will done for each tree because they are independent of one another and approximately 50% (as the process is random) of the trees will choose one feature while the other 50% choose the other. So, the importance of either feature is diluted thereby making it difficult to determine if this feature is important, whereas in boosting the importance will all be on one of the features, but not both. Overall, *random forests* were investigated for this problem and they are excellent for creating basic models using *stock* features. However, the moment additional features or numeric variables were converted to factors to attempt to capture seasonal components, the runtime required would increase significantly. It was decided this was not an effective method for the number of features we wished to include. Additionally, there is a significant body of literature that exists to support the claim that *random forests* are not nearly as accurate as boosting methods.

The next two packages are called *gbm* and *xgboost* and both implement gradient boosting methods. The primary difference is that *xgboost* uses a more regularized model formalization to control overfitting, thus improving performance. The power of boosting methods is that unlike bagging where trees are grown independently, trees are grown sequentially which means that each tree is grown using information from previously grown trees. In gradient boosting we are fitting an *additive* model in a forward stage-wise manner where at each stage we are introducing a new regression to compensate the shortcomings of the existing model. These shortcomings are identified by negative gradients. Gradient boosting methods are a combination of gradient descent and boosting where gradient descent is a mathematical method designed to minimize a function (in this case a loss function such as Mean Square Error(MSE) or Root Mean Square Error (RMSE)) by moving in the opposite direction of the gradient (slope). Residuals are differences between the actual and predicted values at different points. Now to minimize a loss function such as MSE we want to adjust our predictions to be as close as possible to the actual values thus minimizing our residuals. Through some complicated math, it can be shown that the residuals are equivalent to negative gradients and the larger they are the greater the "shortcomings" are. This process is repeated until a stopping condition is identified. Gradient boosting methods are notorious for overfitting, however they are incredibly accurate if properly tuned and in our case could handle thousands of features with computationally minimal runtime. Though challenging, it was believed that gradient boosting methods would build the most accurate model.

In deciding between *gbm* and *xgboost*, *xgboost* was found to include an additional regularization term within the objective function. The regularization terms is designed to control the complexity of the model and help to avoid overfitting. However, with this complexity there are additional hyper parameters that require more detailed tuning. In terms of final model selection, *xgboost* (stands for extreme gradient boosting) was chosen because of its implementation of gradient boosting as well as safeguards against overfitting.

Baseline Model

Our baseline model included a number of *stock* features as well as three features related to date information as shown in Table 3. Features related to the date as well as *Store* and *StateHoliday* were all converted to factors to ensure capture of seasonal components. With regard to *LogSales* as shown in Table 3, if you recall, the objective of the project was to minimize RMPSE as defined in the section **Model Evaluation**. However, the loss function RMPSE is not supported as an objective function that you can minimize in machine learning

Features	Source	Data Type
Store	Stock	Factor
DayOfWeek	Stock	Factor
Promo	Stock	Numeric
StateHoliday	Stock	Factor
SchoolHoliday	Stock	Numeric
StoreType	Stock	Numeric
Assortment	Stock	Numeric
Promo2	Stock	Numeric
PromoInterval	Stock	Numeric
Month	Created	Factor
Year	Created	Factor
Week	Created	Factor
LogSales	**Dependent Variable**	**Numeric**

Table 3: Baseline model features with source & data types.

Hyper Parameter	Baseline Model	Description
objective	"reg:linear"	"reg:linear" - linear regression.
booster	"gbtree"	"gbtree" - tree based model.
max_depth	25	Maximum depth of tree. Controls overfitting.
eta	0.02	Learning rate.
gamma	0.50	Specifies the minimum loss reduction required to make a split.
subsample	0.60	Fraction of observations to be randomly sampled for each tree.
colsample_bytree	0.60	Fraction of columns to be randomly samples for each tree.
min_child_weight	Default	Minimum sum of weights of all observations required in a child.
max_delta_step	Default	Defines the tree's weight estimation.
lambda	0.50	L2 penalties are used to minimize prediction error.

Table 4: Description of hyper parameters & associated values for baseline model.

libraries. It turns out that RMSE is supported and that by taking the log transform of sales and devloping a model to minimize RMSE is the same as if we were minimizing RMPSE. Intuitively, RMPSE is simply removing the *scale* of the residual.

In *xgboost* the user can specify a number of parameters to control model complexity as well as prevent overfitting. The hyper parameters shown in Table 4 were found to be the most important. The learning rate (defined by *eta*) makes the model more robust by shrinking the weights on each step, effectively increasing the number of rounds required for the model to learn. The parameter *max_depth* is used to control overfitting as higher depth will allow the model to learn relations specific to a particular sample. Higher *gamma* makes model more robust to overfitting since a node is split only when the resulting split gives a positive reduction in the loss function. The parameter *subsample* defines the fraction of observations to be randomly sampled for each tree. Lower values make the algorithm more conservative and prevent overfitting but too small of values might lead to underfitting. Finally, the parameter *colsample_bytree* defines the fraction of features to be randomly sampled for each tree. These parameters were found to be the most critical for controlling model complexity and prevention of overfitting.

The procedure for creating the baseline model is shown in the following section of code.

```
#Log Transform Dependent Variable
    y = log(train_Sub$Sales)
#create Feature Set
    mtrain = train[,c("Choose Features"),with = FALSE]
    mtest = test[,c("Choose Features"),with = FALSE]
#Set Log Sales Column
```

```
    mtrain$logSales = y
    mtest$logSales = 1 #Set Value To 1 As Placeholder
#Remove Sales & Take Log Transforms Of Sales
    mtrain = mtrain[,-c(40),with=FALSE]
    mtest = mtest[,-c(40),with=FALSE]
#Create Sparse Matrix
    #Training Set
        outputVect_train = mtrain$logSales
        trainSparseList = sparse.model.matrix(logSales~.-1,data = mtrain)
        trainSparse = list("data" = trainSparseList, "label" = outputVect_train)
        str(trainSparse) #View The Sparse Matrix
    #Test Set
        outputVect_eval = mtest$logSales
        evalSparseList = sparse.model.matrix(logSales~.-1,data = mtest)
        evalSparse = list("data" = evalSparseList, "label" = outputVect_eval)
        str(evalSparse) #View The Sparse Matrix
#Create xgb.DMatrix
    dtrain = xgb.DMatrix(data = trainSparse$data, label = trainSparse$label)
    dtest = xgb.DMatrix(data = evalSparse$data, label = evalSparse$label)
#Set Model Parameters
    param = list("objective" = "reg:linear",
                 "booster" = "gbtree",
                 "max_depth" = "25",
                 "eta" = "0.02",
                 "gamma" = "0.50",
                 "subsample" = "0.60",
                 "colsample_bytree" = "0.60",
                 "lambda" = "0.50",)
#Run Model
    Model = xgboost(data = dtrain,
                    params = param,
                    nrounds = 8806, #Determined Through Model Training
                    maximize = FALSE)
#Make Predictions
    Predictions = predict(Model, dtest)
```

There are a couple of critical components in developing an *xgboost* model. The first is creation of a sparse matrix. The first step is creating feature set and appending the dependent variable (variable you wish to make predictions on) as the final column. For the baseline model the feature set is provided in Table 3 and the dependent variable is *LogSales*. A sparse matrix is a highly compressed, column oriented matrix which allows us to pass thousands of features to an *xgb.DMatrix*, but cannot contain character values. To pass character values, the field must be converted to a factor. As shown in the code, *LogSales* is appended to the end of the data being passed to *sparse.model.matrix* but is also passed to a vector (either denoted *outputVect_train* or *outputVect_eval*) which is used as a label in the resulting sparse matrix. In the *sparse.model.matrix* function the user must pass the data (either the training or test set) and provide the notation "logSales~.1" which indicates that the objective is to predict *LogSales* using all features in the set aside from the last, which, if you recall, is *LogSales* as it is appended to the end of the data frame. The result is a sparse matrix defined as either *trainSparseList* or *evalSparseList* and is subsequently combined as a list with the output vectors mentioned previously. These are then passed directly to the an *xgb.DMatrix*. The second critical component is use of the *xgb.DMatrix*. As is apparent from creation of the sparse matrix, there is a specific format required to create an *xgb.DMatrix*. The reason for using an xgb.DMatrix is to access advanced features that reside in *xgboost* and allow users to control model complexity and prevent overfitting. Creating an *xgb.DMatrix* is straightforward as the user is only required to pass features of the previously created sparse matrix (data &

label).

The final baseline model is created after passing a list of the hyper parameters, where the values provided in Table 4, the xgb.DMatrix for the training data set, and the number of rounds (*nrounds*) which is equivalent to the number of trees to grow and is simply another hyper parameter which was tuned during model training. Once the model is created, predictions can be made on the *xgb.DMatrix* for the test set and the results provided for evaluation. It is important to note that because we had to log transform *Sales* to begin, our predictions were still transformed, so it was necessary to take the exponential to transform back. Overall, our baseline model scored an RMPSE of 0.13756.

Model Improvements

The baseline model was simple to implement and represents an accurate, quick solution. However, for this project we created hundreds of additional features, not including those created through conversions to factors. The problem now is determining which of the additional features are important and which set of hyper parameters to use when determining feature importance. As new or different features are added the ideal hyper parameter set will change and so how can we effectively determine which features are important? Based on knowledge gained during exploratory data analysis, feature engineering, and creation of the baseline model it was decided to handpick specific features that "seemed" important and then tune the hyper parameters using a random search process. That set of tuned hyper parameters was then used for developing models with different sets of features to determine feature importance. Literature on extreme gradient boosting indicates that hyper parameters are important, but once you have a general sense of the range, tuning them further only contributes a small percentage to the improved accuracy.

The feature set chosen includes *stock* features such as *Store*, *DayOfWeek*, *Promo*, *StateHoliday*, *SchoolHoliday*, *StoreType*, and *Assortment*, date-related features such as *Week*, *Month*, and *Year*, all features created in the **Adding Running Totals & Count Features** section, a single feature for the abbreviation of the German state (*GermanStateABBR*), and finally, specific weather-related features which include the minimum, maximum, and mean of temperature (°C), dew point, and humidity. In total there are 39 features, of which *Store*, *Assortment*, *StoreType*, *GermanStateABBR*, *DayOfWeek*, *Day_In_Month*, *Week*, *Month*, *Year*, and *StateHoliday* were all converted to factors.

The process for tuning the hyper parameters using a random search process is straightforward, however it requires running hundreds of models and we are not able to use the test set as we do not know the values for *Sales*. The only option is to partition the training set into a smaller training set and test set where we would know the values for *Sales*. If you recall from the **Exploratory Data Analysis** section, the original test set covers a period of 42 days or 6 weeks. So, it follows that we would want to use the last 6 weeks of the training data set as our new test set and to avoid confusion call it the holdout set. The new training set will not include the last 6 weeks of data. Typically, data scientists would use methods such as cross-validation to randomly choose which data points would be in the training set and the holdout set, however our problem is time series in nature and due to this fact we have to keep the partition sequential which is why the holdout set includes the **last** 6 weeks of data. Once the holdout and new training set have been created the process is nearly identical to the process outlined in creation of the baseline model with one key difference. Instead of passing our data and list of hyper parameters to *xgboost* we instead pass those arguments to *xgb.train*. Use of the *xgb.DMatrix* allows us to access advanced features of the extreme gradient boosting method, most importantly the *watchlist* and *early.stop.round*. The *watchlist* is a parameter that allows users to specify a validation set that is monitored during model training. For example, in one iteration of the extreme gradient boosting method a tree is grown using training data and with a *watchlist*, predictions are made using the validation set. Those predictions are compared to the actual values within the validation set through a loss function such as RMSE. Since the trees are grown sequentially, the method realizes its shortcomings and produces a more accurate model in the second iteration. The process continues until the model can no longer improve in minimizing the loss function for a specified number of iterations as given by *early.stop.round*. Consider the following section of code.

Hyper Parameter	**Baseline Model**	**Engineered Model**	**FINAL Model**
objective	"reg:linear"	"reg:linear"	"reg:linear"
booster	"gbtree"	"gbtree"	"gbtree"
max_depth	25	13	14.8904985644
eta	0.02	0.10	0.0891883073198
gamma	0.50	0.28	0.0962115154038
subsample	0.60	0.50	0.827053601542
colsample_bytree	0.60	0.70	0.198988867033
min_child_weight	Default	Default	24.1254067832
max_delta_step	Default	Default	18.3337122008
lambda	0.50	Default	Default
RMPSE	0.13756	0.11987	0.11679

Table 5: Values for hyper parameters by model & associated scores for RMPSE.

```
#Procedure for Training XGBoost Model
    #Create xgb.Dmatrix By Passing A Sparse Matrices for The Training & Test Sets
        dtrain = xgb.DMatrix(data = trainSparse$data,
                             label = trainSparse$label)
        dtest = xgb.DMatrix(data = evalSparse$data,
                            label = evalSparse$label)
    #Specify a Watch List
        watchlist = list(validation = dtest, #Specify A validation Set
                         train = dtrain) #Specify The Training Set
    #Pass Arguments To xgb.train
        Model = xgb.train(params = param,
                          data = dtrain,
                          nrounds = 25000,
                          watchlist = watchlist,
                          maximize = FALSE,
                          early.stop.round = 25)
```

Similar to creation of the baseline model, sparse matrices are constructed for the training and holdout sets and passed to *xgb.DMatrix*. The *watchlist* is then specified by passing an *xgb.Dmatrix* for both the training and holdout sets as a list to a data frame. Finally, we pass the set of hyper parameters, the training data, and the watchlist to the arguments in the *xgb.train* function. Additionally, we set the value of *nrounds* to be high so the model will grow a sufficient number of trees to determine best possible configuration and set the *early.stop.round*, in this case to 25, to indicate that if the model cannot improve on minimizing the loss function for 25 sequential iterations than stop the training process. In our case the validation set is the holdout set. Now tuning hyper parameters using a random search process requires running hundreds of these models while randomly selecting values for the hyper parameters provided in Table 4 and then storing the final values for the loss function (in our case RMSE) as well as for *nrounds*. After running hundreds of these models, the ideal set of hyper parameters will have the lowest value for the loss function. In this loop the hyper parameters are each randomly sampled across a different range of values. The parameter *eta* ranges from 0 to 1 whereas *max_depth* ranges from 1 to 35. Tuning hyper parameters using random search is typically done iteratively. Hundreds of models are run initially to determine a set of models with hyper parameters that are performing best, then the range with which parameters are sampled is constricted. This process is repeated until the hyper parameters are properly tuned. The hyper parameters for this **Engineered Model** are shown in Table 5.

Comparing values for hyper parameters from the **Engineered Model** and **Baseline Model** (Table 5) reveals they are quite different by comparison. Additionally, the RMPSE score for the **Engineered Model** was 0.11987 which is a significant improvement over the baseline model. The primary reason is not so much the tuned hyper parameters but the addition of a number of features. Now, with a tuned set of hyper

parameters, feature engineering could be performed. Hundreds of models were run where different features were either integrated into or removed from the model. A number of different procedures were developed to investigate feature importance such as loops that randomly selected features at each iteration as well as handpicking features believed to correlate well with *Sales*. Overall, nothing scored better than the feature set used in developing the **Engineered Model**. It is believed that fixing the hyper parameters was affecting feature importance as one set of hyper parameters ideal for one feature set may not be ideal for another. Due to time constraints this could not be investigated further.

Since feature engineering revealed little in terms of model improvements, we decided to investigate the use of Bayesian Optimization to finely tune hyper parameters for the feature set used in the **Engineered Model**. Bayesian Optimization is a powerful tool for optimizing the objective functions which are costly or slow to evaluate. This is ideal for hyper parameter tuning as the function space (set of hyper parameters) can become quite complex. The mathematics behind Bayesian Optimization are outside the scope of this paper, however there are numerous packages and companies available to perform this method of hyper parameter tuning. In our research of the method, the company SigOpt was found to have an ideal implementation of Bayesian Optimization in *R*. SigOpt has developed a package in *R* that connects with SigOpt via an API whereby an experiment can be setup. In the experiment, a model is created based on a set of hyper parameters with the results being sent to SigOpt which can optimally suggest a set of new hyper parameters for the next iteration. This process is repeated until the ideal set of hyper parameters is reached. SigOpt firmly believes that this process greatly reduces the number of iterations required to tune hyper parameters that are common in processes such as random search and states that the more hyper parameters passed for tuning, the more effective the procedure is. Due to time constraints, these beliefs could not be tested, however, using the method provided by SigOpt, the set of hyper parameters shown in Table 5 produced a model that scored an RMPSE of 0.11679 (**FINAL Model**). A definite improvement over the set of hyper parameters found using random search.

CONCLUSION

The objective of this project was to develop a robust predictive model which predicts 6 weeks of daily sales for 1,000 stores located across Germany. The **FINAL Model** scored remarkably high considering the simplicity of the feature set. With a simple feature set, integration of the predictive model into standard business practice is easier. Though the **FINAL Model** was successful in terms of its robustness and predictive accuracy, there are number areas that which require further investigation and could potentially lead to the development of an even better predictive model than the one presented here. Future work should be focused on developing a more robust method for performing feature engineering in conjunction with hyper parameter tuning, it is the belief that adding different sets of features requires a different set of hyper parameters and by fixing one we are effectively diminishing the importance of certain features. If such a method were developed investigation into creating model ensembles could begin. However, the greatest hurdle is overcoming the amount of runtime associated with this task and perhaps the method provided by SigOpt is the ideal alternative as multiple experiments could be run in parallel. Secondly, though not discussed in detail in the **Model Development** section, the effect of missing values was a constant problem because an *xgb.DMatrix* cannot handle missing values. Literature suggests that missing values be imputed or set to extremes in the hope the model will identify these outliers and deal with them accordingly. However, more research is required to investigate if there is not a more efficient way of handling missing values in the data sets.