ROSSMANN

Store Sales Prediction

Predicting daily sales up to six weeks in advance across 3,000 locations in Europe

Final Report

Paul Marten
July 2021

1. Introduction

1.1 Context

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

1.2 Problem

How can we best predict the Rossmann stores' daily sales up to six weeks in advance, through means of using data about each location and past sales to formulate a regression model that will minimize percentage error?

1.3 Data

The data is obtained from the Kaggle competition titled "Rossmann Store Sales." The data can be found here: https://www.kaggle.com/c/rossmann-store-sales/data

- **train.csv** - historical data including sales.
- **store.csv** - supplemental information about the stores.

2. Data Wrangling

2.1 Loading

Being as that both files were already supplied and in csv (comma separated values) format, they were both read into dataframes. The train.csv file had a date column, so I read this file in with the index being the parsed dates, and named the dataframe 'train'. The store.csv file was read in using a standard index, and was named 'store' accordingly.

2.2 Cleaning

- Checking out the initial raw dataframes, ‘train’ on top and ‘store’ on the bottom:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1017209 entries, 2015-07-31 to 2013-01-01
Data columns (total 8 columns):
 #   Column         Non-Null Count    Dtype
---  ------         --------------    -----
 0   Store          1017209 non-null  int64
 1   DayOfWeek      1017209 non-null  int64
 2   Sales          1017209 non-null  int64
 3   Customers      1017209 non-null  int64
 4   Open           1017209 non-null  int64
 5   Promo          1017209 non-null  int64
 6   StateHoliday   1017209 non-null  object
 7   SchoolHoliday  1017209 non-null  int64
dtypes: int64(7), object(1)
memory usage: 69.8+ MB
----------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1115 entries, 0 to 1114
Data columns (total 10 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Store                      1115 non-null   int64
 1   StoreType                  1115 non-null   object
 2   Assortment                 1115 non-null   object
 3   CompetitionDistance        1112 non-null   float64
 4   CompetitionOpenSinceMonth  761 non-null    float64
 5   CompetitionOpenSinceYear   761 non-null    float64
 6   Promo2                     1115 non-null   int64
 7   Promo2SinceWeek            571 non-null    float64
 8   Promo2SinceYear            571 non-null    float64
 9   PromoInterval              571 non-null    object
dtypes: float64(5), int64(2), object(3)
memory usage: 87.2+ KB
```

2.2.1 ‘train’

Features:

- **Store**: The location (unique id for each store).
- **DayOfWeek**: Starting on Monday as 1, and ending on Sunday as 7.
- **Sales**: The turnover, or gross revenue, on a given day (target variable).
- **Customers**: The amount of customers on a given day.
- **Open**: Indicator of whether the store was open or closed (0 = closed, 1 = open).

- **Promo**: Indicates if the store was running a promotion that day (0 = no, 1 = yes).
- **StateHoliday**: Indicates if it was a state holiday. Normally all stores, with few exceptions, are closed on state holidays. Note that all schools are closed on public holidays and weekends (a = public holiday, b = Easter holiday, c = Christmas, 0 = None).
- **SchoolHoliday**: Indicates if the (Store, Date) was affected by the closure of public schools.

- There were no missing values in any feature, which is always great to see. This dataframe's version of missing data is wherever there are 0 sales. Days with no sales will be ignored in model evaluation, given the nature of the root mean squared percentage error metric being used, but that will be explained further in the modeling section of this report.
- Doing some investigation, about 17% of the records accounted for days when a store was closed and sales were 0, and about 0.005% of records indicated when a store was open but had no sales. These rows were dropped from the data.
- For further analysis later in the notebook, I used the datetime nature of the index to create new features by extracting the year, month, day, and week of year.
- The final shape of 'train' was (844338, 12).

2.2.2 'store'

Features:

- **Store**: The location (unique id for each store).
- **StoreType**: Differentiates between 4 different store models: a, b, c, d.
- **Assortment**: Describes an assortment level: a = basic, b = extra, c = extended.
- **CompetitionDistance**: Distance in meters to the nearest competitor store.
- **CompetitionOpenSince[Month/Year]**: Gives the approximate year and month of the time the nearest competitor was opened.
- **Promo2**: A continuing and consecutive promotion for some stores: 0 = store is not participating, 1 = store is participating.
- **Promo2Since[Year/Week]**: Describes the year and calendar week when the store started participating in Promo2.
- **PromoInterval**: Describes the consecutive intervals Promo2 is started, naming the months the promotion is started. E.g. "Feb,May,Aug,Nov" means each round starts in February, May, August, November of any given year for that store.

- Checking for missing values, there were quite a bit, 3 rows having almost half of their entries missing. After further investigation, it turned out to be very clear why.

```
# Check missing values

store.isnull().sum()
```

```
Store                          0
StoreType                      0
Assortment                     0
CompetitionDistance            3
CompetitionOpenSinceMonth    354
CompetitionOpenSinceYear     354
Promo2                         0
Promo2SinceWeek              544
Promo2SinceYear              544
PromoInterval                544
dtype: int64
```

- For the 3 rows missing value for **CompetitionDistance**, there was really no way to infer the values from the data, and being as that it was only 3 of 1115 rows, I imputed these missing values with the mean of the column.
- For the two features **CompetitionOpenSinceMonth** & **CompetitionOpenSinceYear**, there were a significant amount amount of NaN values, there was one thought that came to mind to impute these values. I could possibly observe the sales for the stores that had missing values for these, and see where the sales had some variance that could indicate new competition. The issue with this is that I had not yet explored the data well enough to know if this was even true, so to save the headache and possible error, I just imputed these with 0s so as to not lose a nice chunk of the data dropping these rows.
- Looking at the next three features that were missing a lot of values, **Promo2SinceWeek**, **Promo2SinceYear**, & **PromoInterval**, I was able to see that these were only missing values for records where **Promo2** was equal to 0. This made total sense, and with a quick check, I concluded that this was the case and also imputed these NaNs with 0s.

2.2.3 'train_store'

- Now that both dataframes were cleaned up, it was time to merge them into one. I used an inner join, and joined the two on the 'Store' column. The resulting dataframe had 21 features and 844,338 records.

```
# Join train and store on the 'Store' column, using an inner merge

train_store = pd.merge(train, store, on='Store', how='inner', sort=True)
print(train_store.shape)
train_store.head(10)
```

```
(844338, 21)
```

3. Exploratory Data Analysis

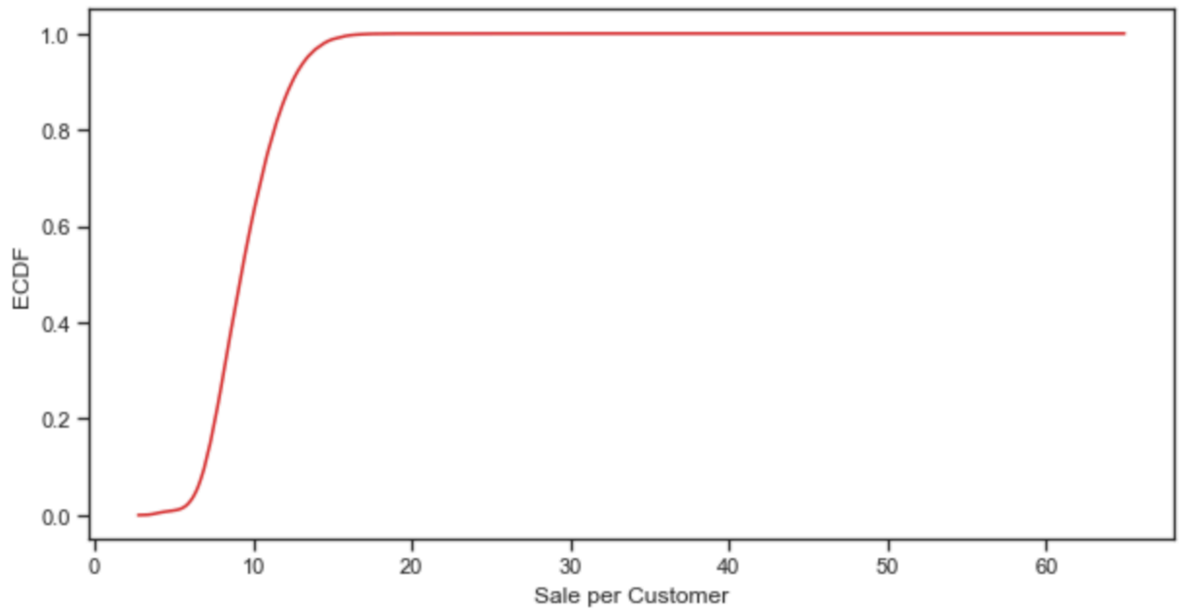
- The first thing I did in the EDA portion of the notebook was create a new feature, **SalePerCustomer** in order to aid in the exploration of data. Calling the describe() method on the new feature, I was able to see that the gross revenue per customer per day was about $9.50. The upper limit was approximately $65, with the lowest being about $2.75.

```
# Create a new column 'SalesPerCustomer'

train_store['SalePerCustomer'] = train_store['Sales']/train_store['Customers']
train_store['SalePerCustomer'].describe()
```

```
count    844338.000000
mean          9.493641
std           2.197448
min           2.749075
25%           7.895571
50%           9.250000
75%          10.899729
max          64.957854
Name: SalePerCustomer, dtype: float64
```

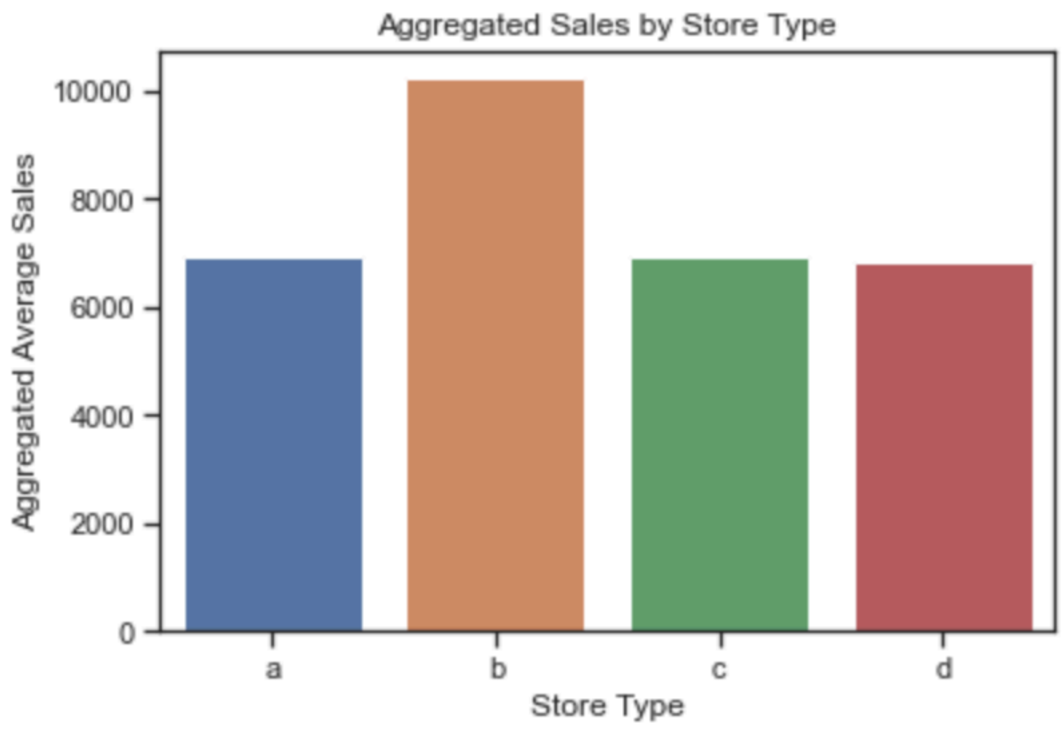
- Following this, I plotted the empirical cumulative distribution function (ECDF) of the the new variable, **SalePerCustomer**. The purpose of this was to have a visual method for understanding the distribution of **SalePerCustomer**. Its value at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value.

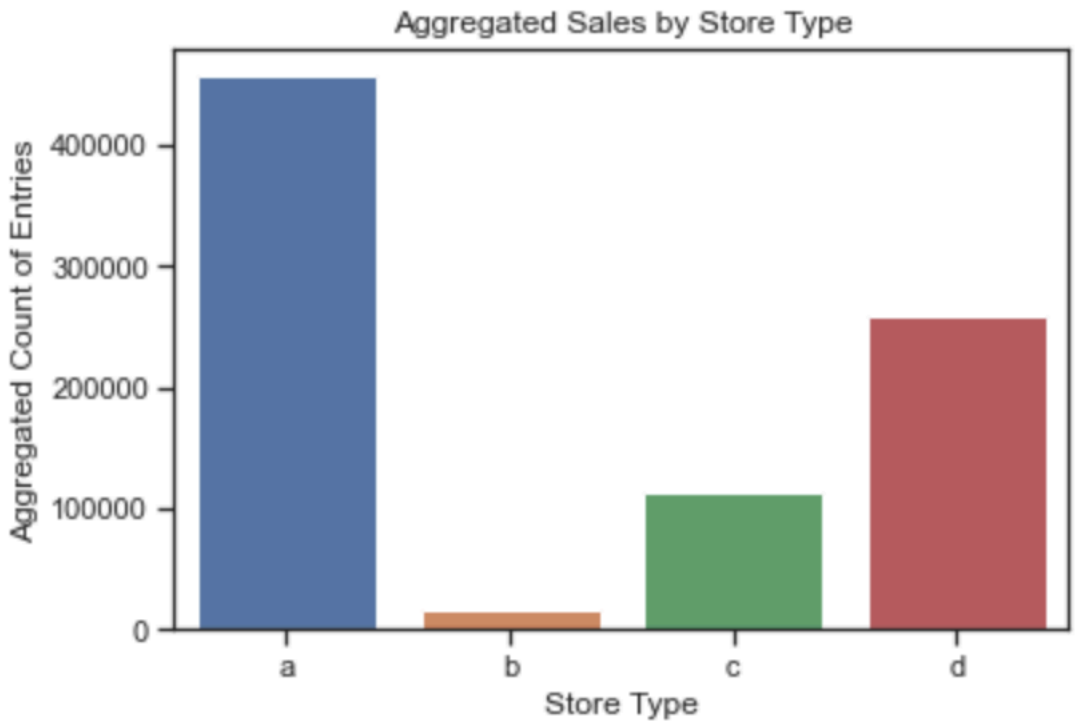


So although the maximum sale per customer was $65, it is very rare for a customer to spend that much, indicated by the fact that almost 100% of customers spend less than that. As a matter of fact, almost 100% of customers spend just $15 or less. This is indicated by the flattening of the curve being around the point (15, 1.0).

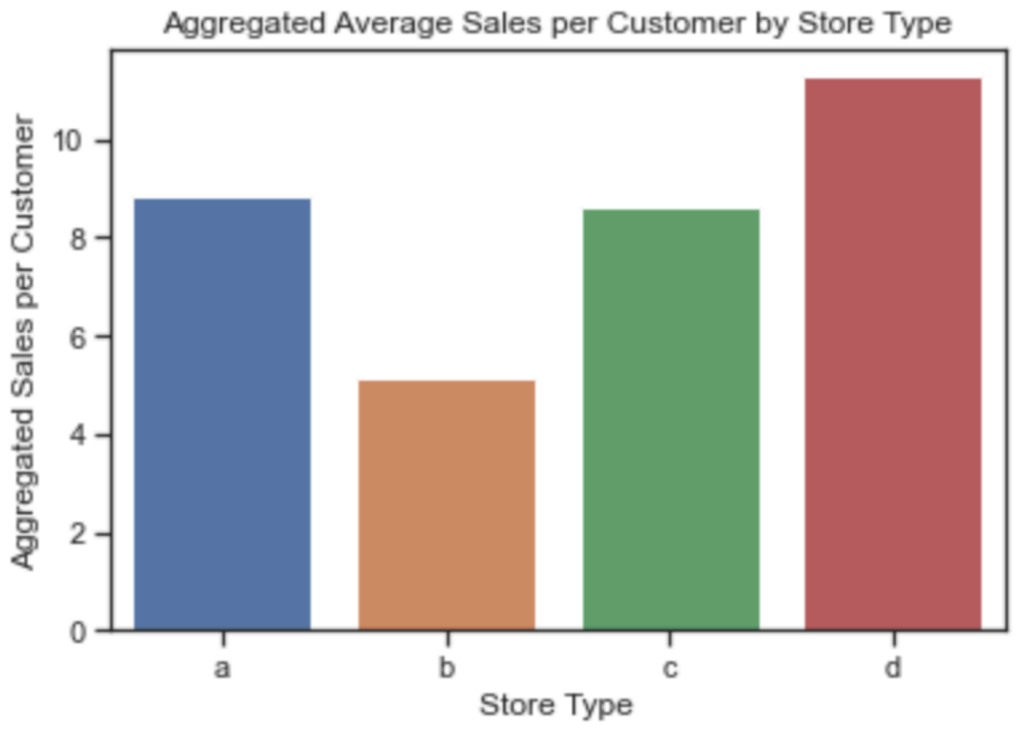
3.1 Store Type Analysis

- Being as that there are about 3,000 stores, aggregated on each store would be quite messy. Luckily, **StoreType** only has four distinct values, a-d, and groups every location into one of these categories. I did a few different types of aggregations on these types, which resulted in the following bar charts:

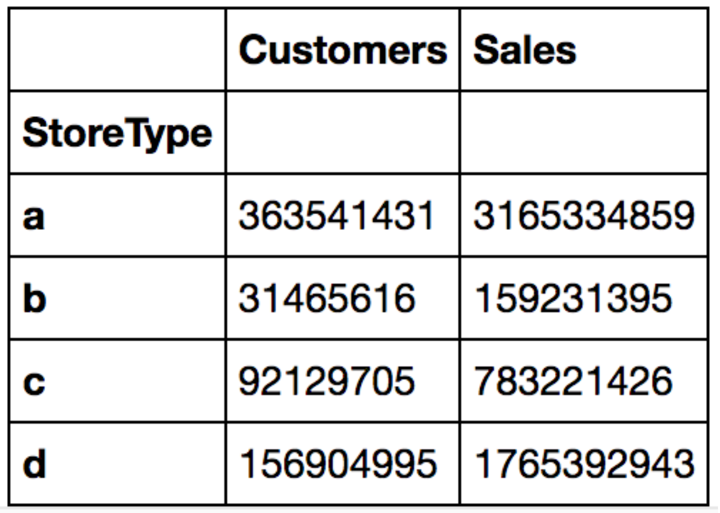




- Looking at these two graphs, we see that store type b has the highest average sales, but the least amount of data entries. This could mean that it still has the highest frequency of customers and sales comparatively, there is just not as many stores of type b.

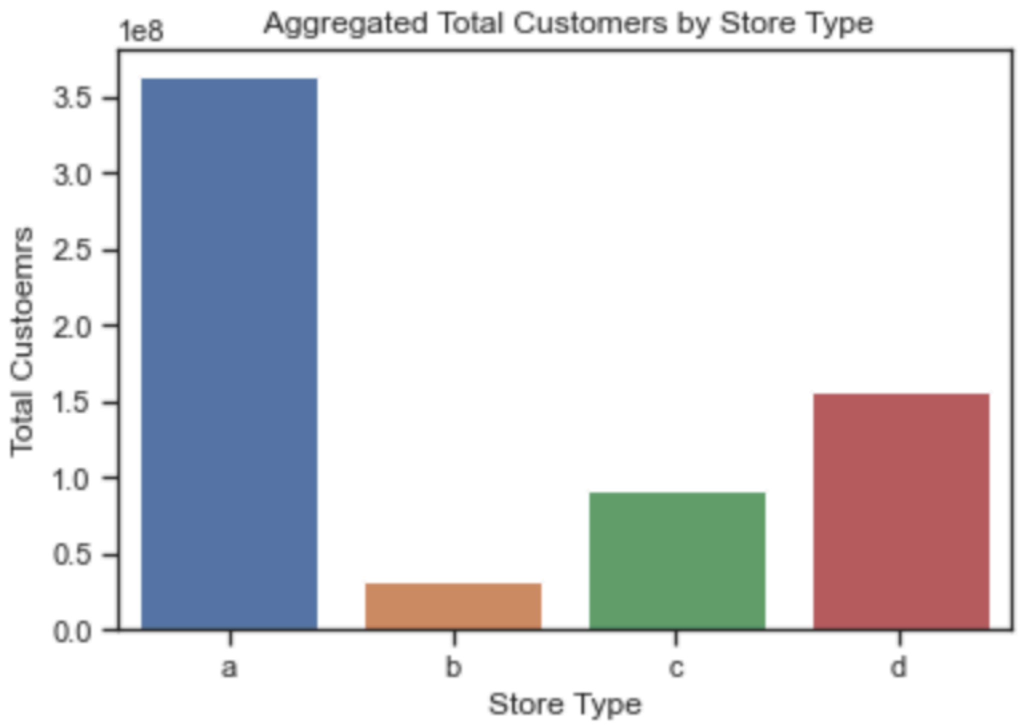


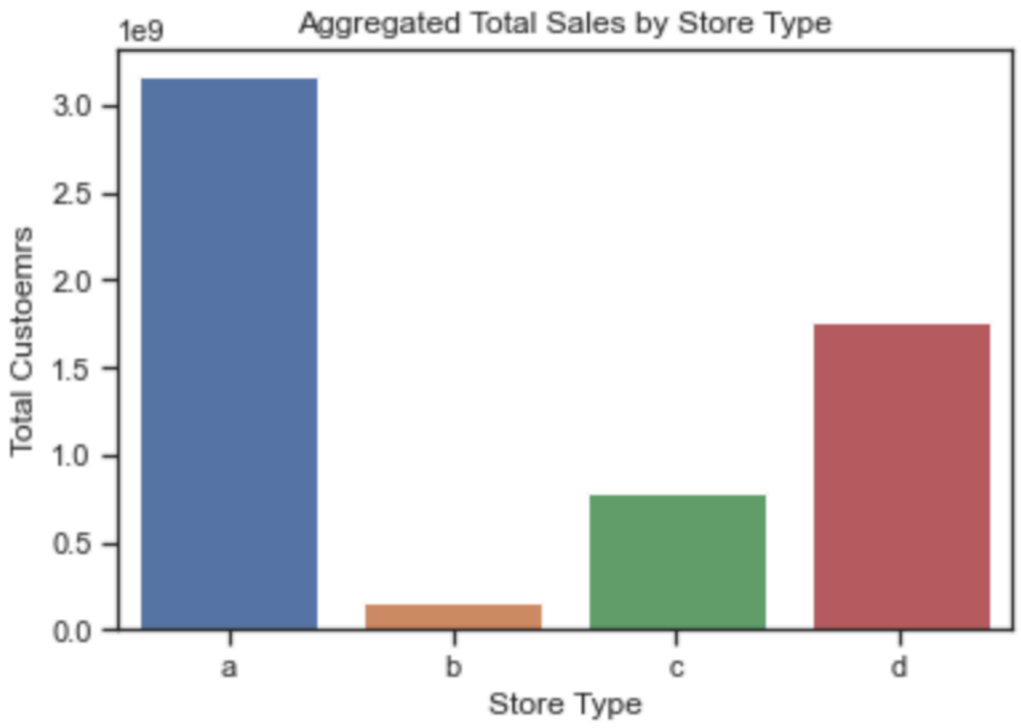
- Looking at the sales per customer aggregated by the mean per store type, its clear that store type d has the highest average sales per customer. This is counterintuitive because it ranked lowest on total average sales.



	Customers	**Sales**
StoreType		
a	363541431	3165334859
b	31465616	159231395
c	92129705	783221426
d	156904995	1765392943

- By grouping the data by **StoreType** on the **Customers** & **Sales** columns and computing the sum, we see the above table. It's a little hard to interpret at first glance, so the following two graphs make it easier to interpret:



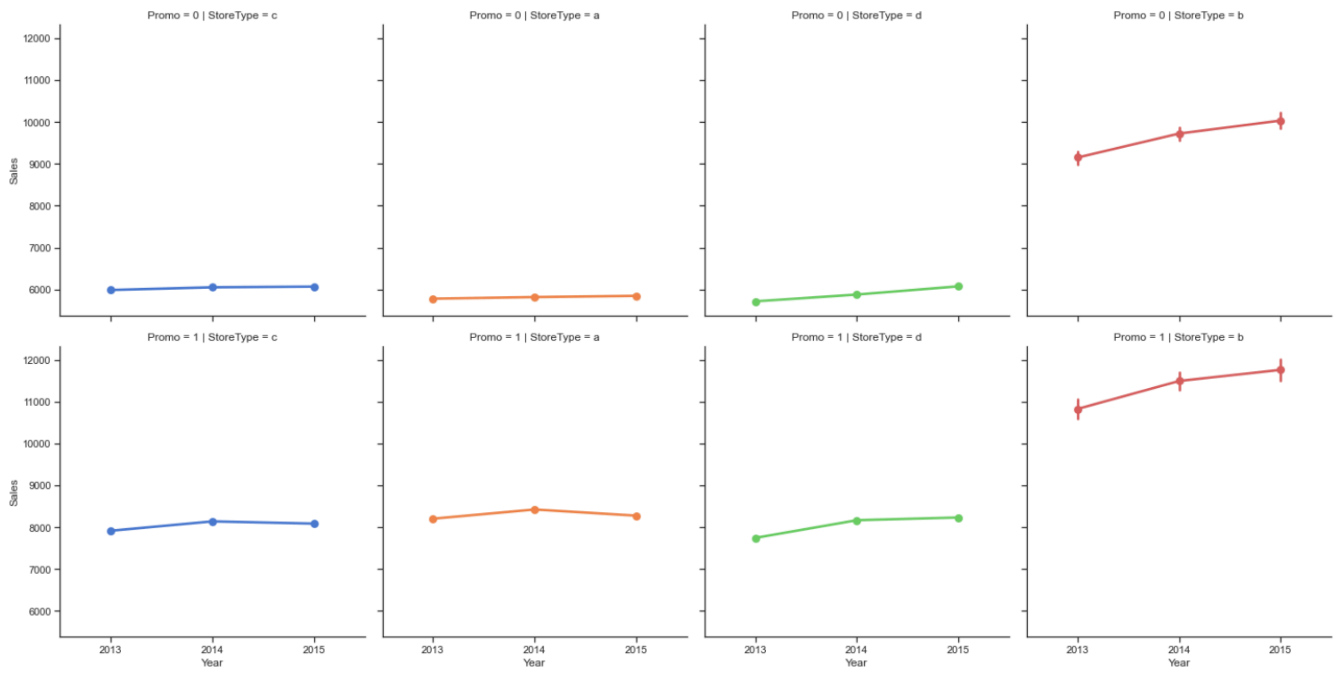


Conclusions**:**

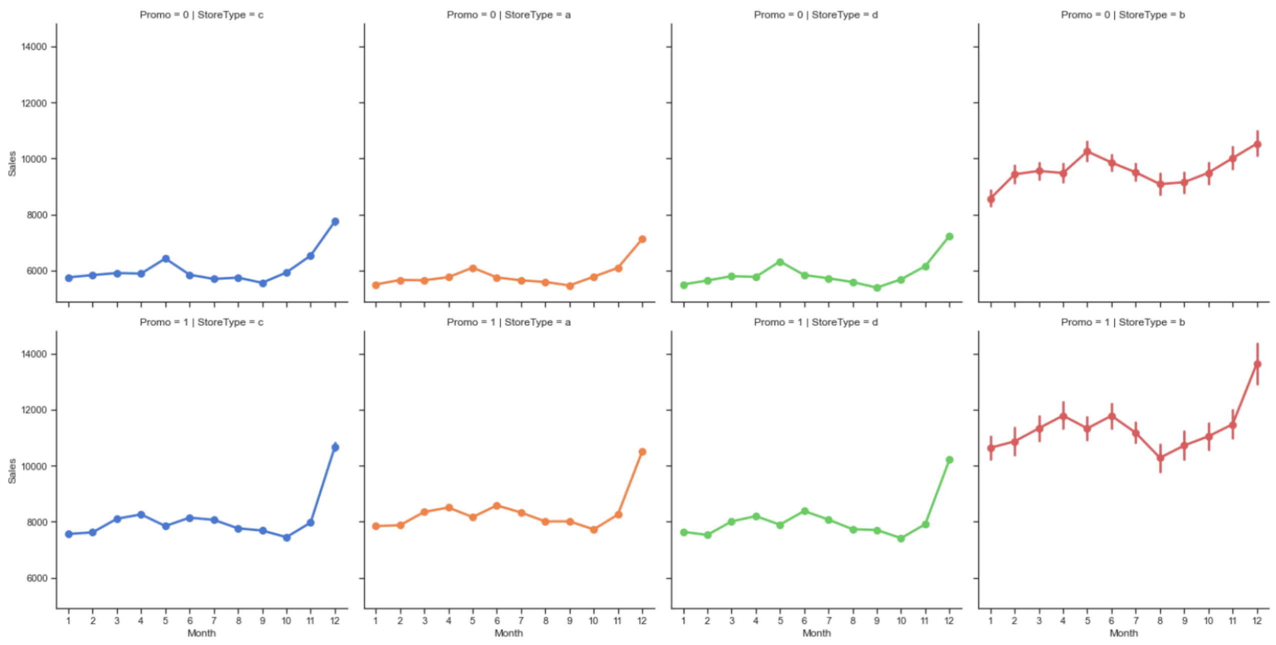
- Type **a**: First in total number of customers and sales, third in average sales value, second in average sales per customer.
- Type **b**: Last in total customers and sales, first in average sales value, last in average sales per customer.
- Type **c**: Third in total sales and customers, second in average sales value, third in average sales per customer.
- Type **d**: Second in total number of customers and sales, last in average sales value, first in average sales per customer.

- These are quite general summary statistics, and although useful, they don't paint the whole picture. We still have promos and holidays to worry about. Let's see how the sales trend when promos are considered. I followed by using seaborn's factorplot, which is very helpful when plotting with categorical features (**StoreType** & **Promo**).

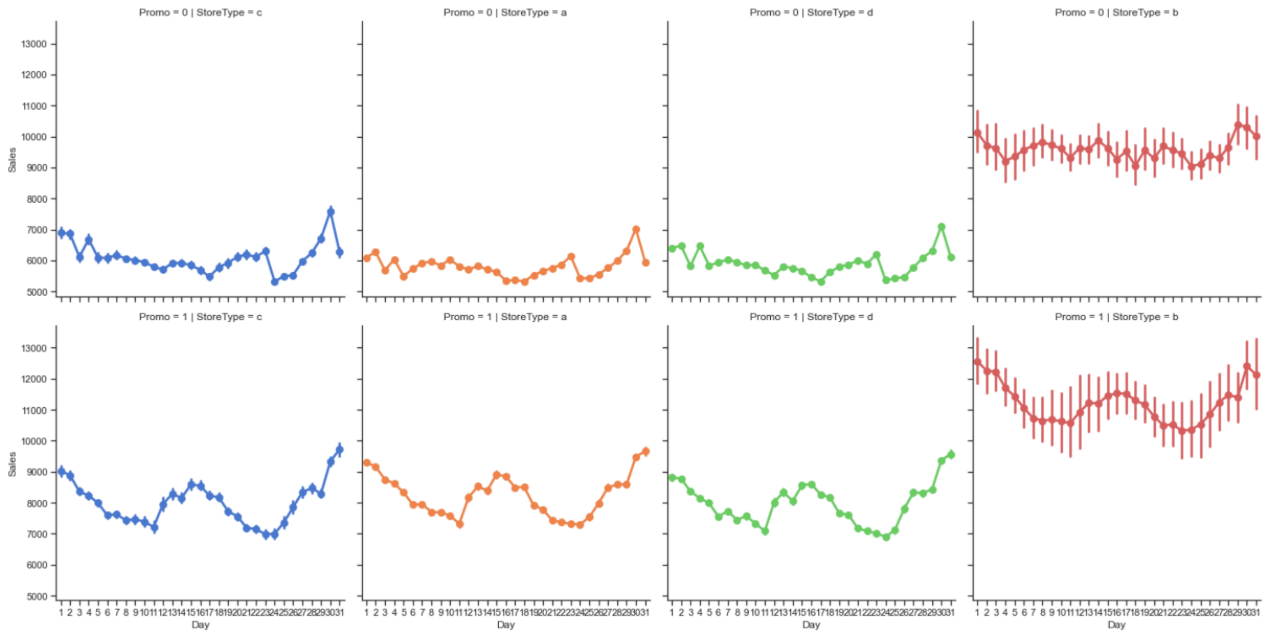
By Year



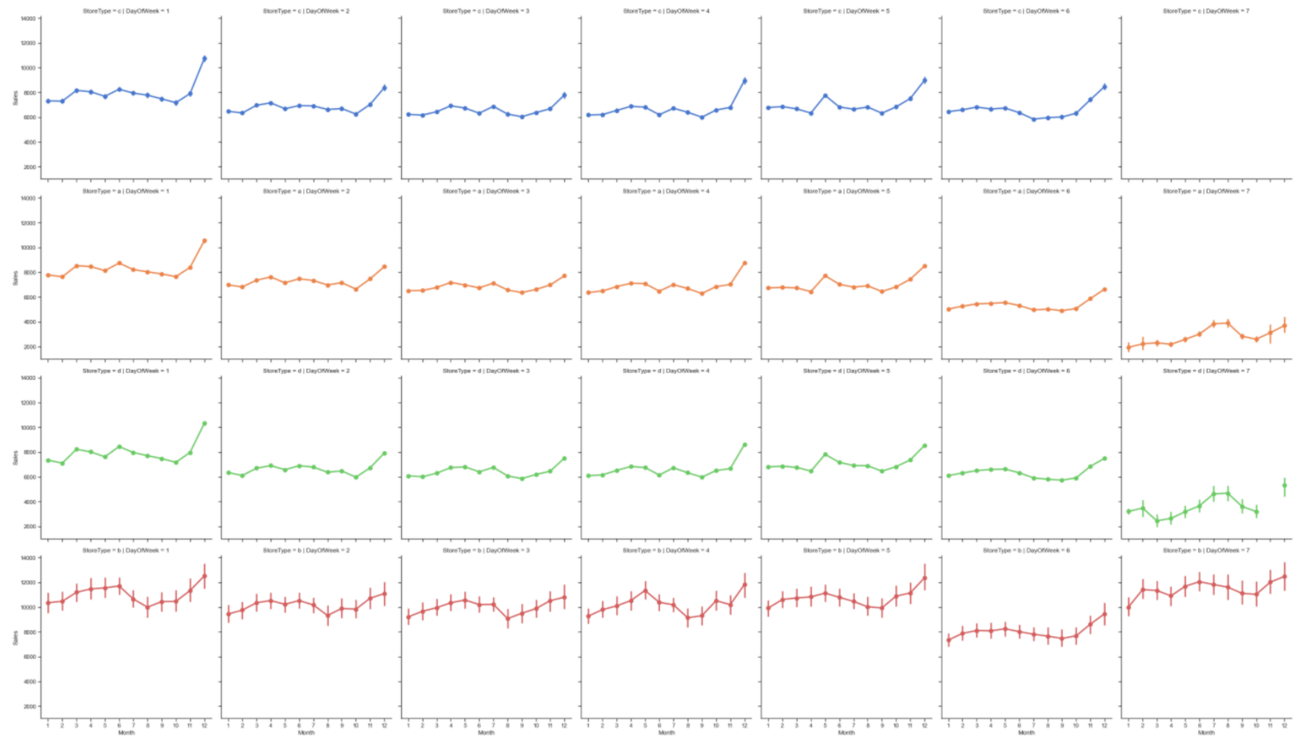
By Month



By Day



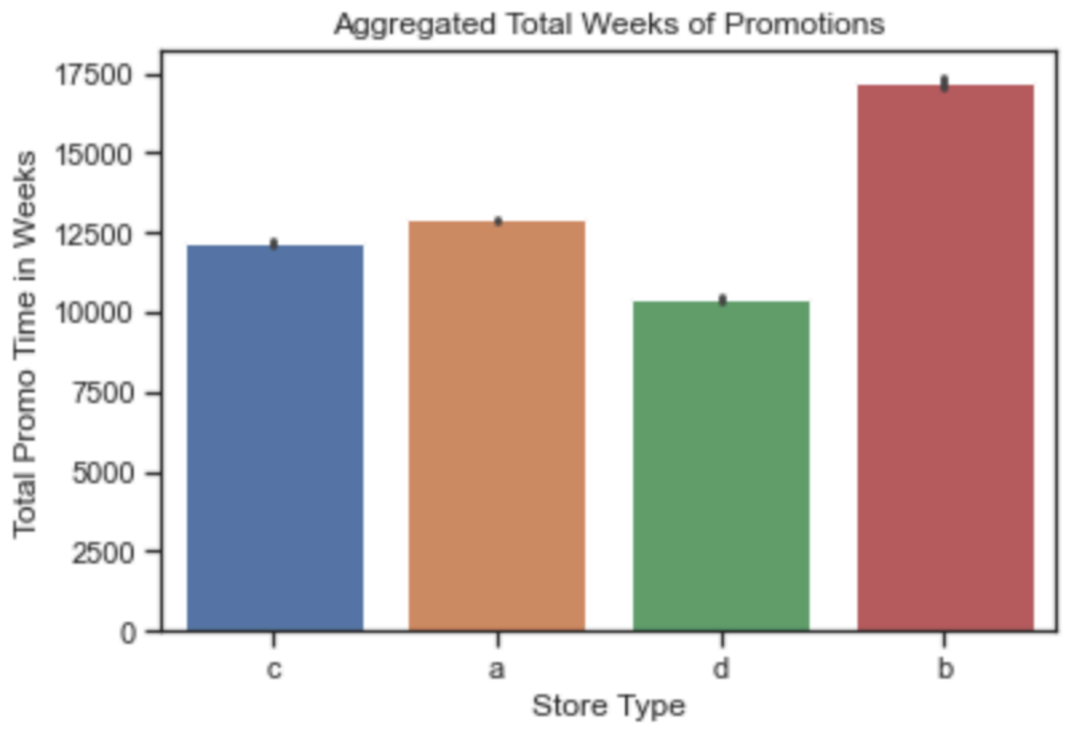
By DayOfWeek

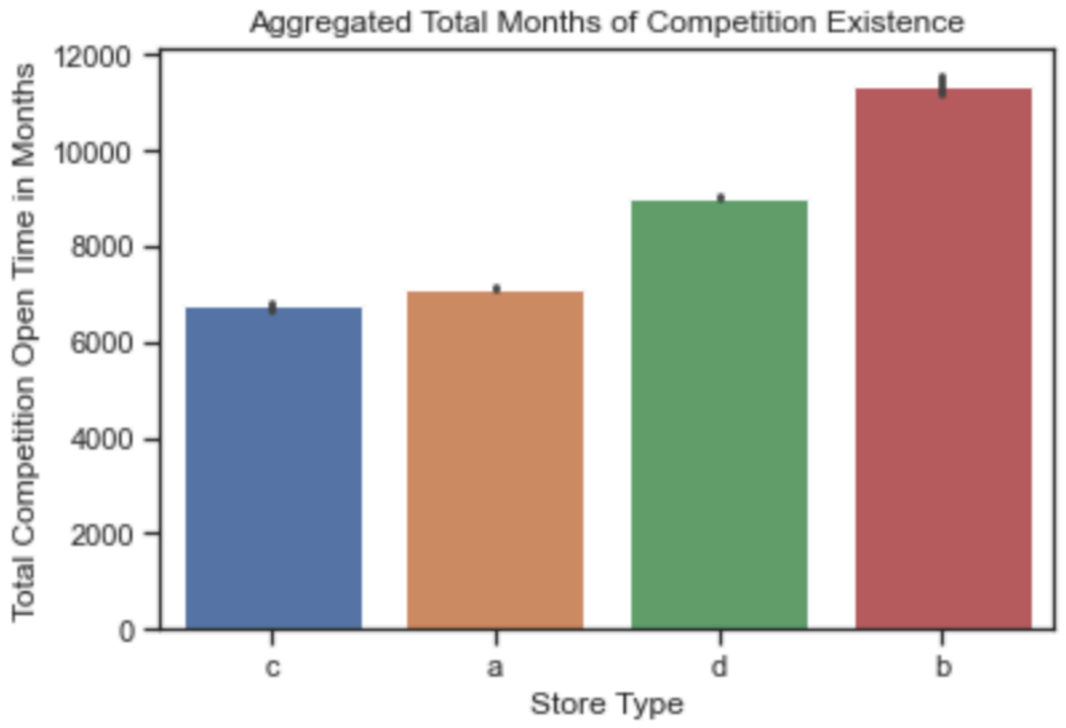


- Sales seem pretty consistent from day to day, with some obvious trends upwards towards the end of the year and minor dips around August/September in most cases. Apparently store type c is closed on Sundays, and some stores of type d are closed on Sundays in October, November, and December. After further investigation, I found that approximately 3% of stores are closed on Sundays.

- The final bar charts I plotted were in relation to the promotion durations, as well as how long competition has been open for.

	Sales	**Customers**	**PromoOpen**	**CompetitionOpen**
StoreType				
a	6925.697986	795.422370	12918.492198	7115.514452
b	10233.380141	2022.211825	17199.328069	11364.495244
c	6933.126425	815.538073	12158.636107	6745.418694
d	6822.300064	606.353935	10421.916846	9028.526526

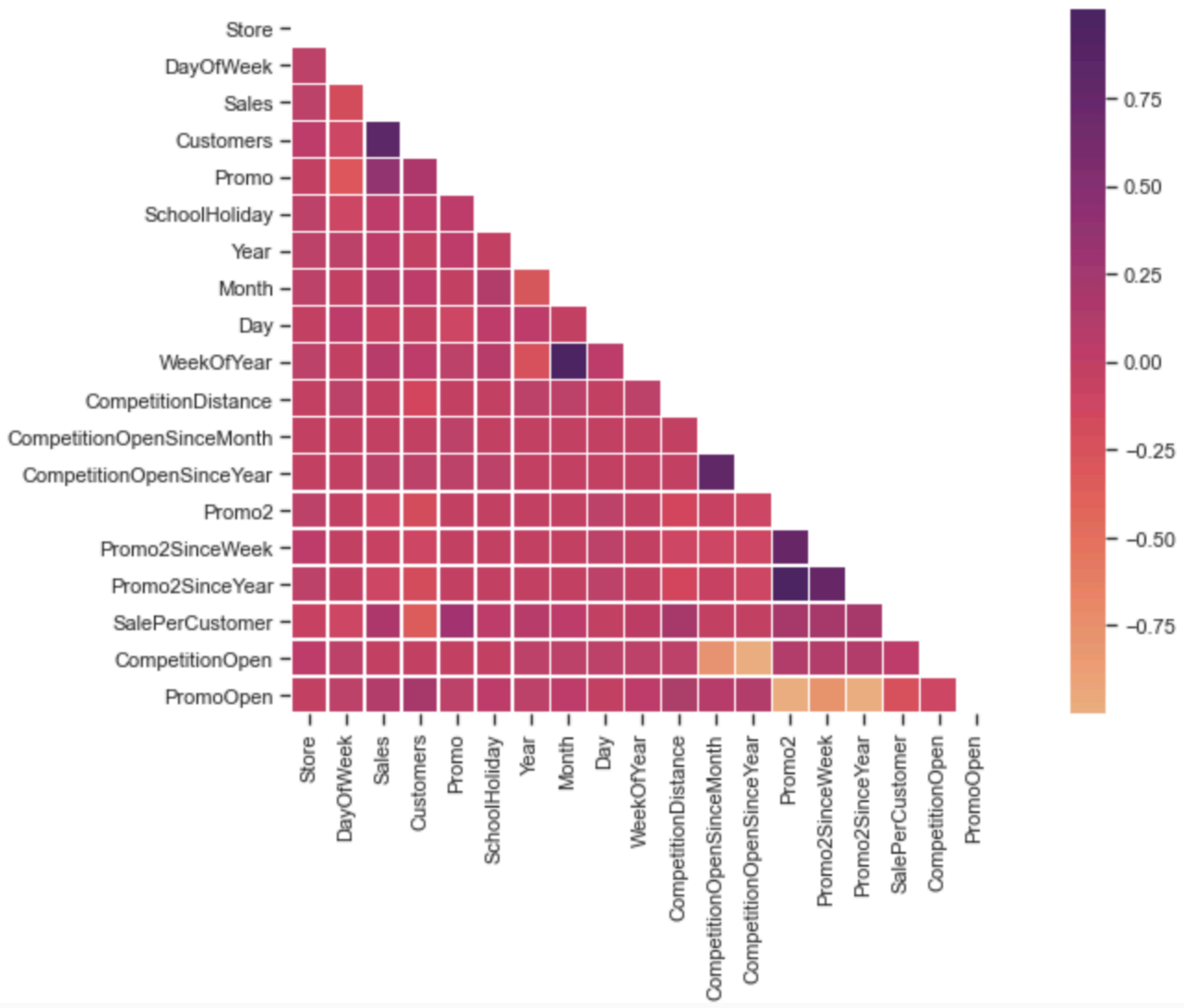




- Although store type a has the most sales and customers, it ranks third in exposure to competition but ranks second in total promotion time. Store type b has the most exposure to competition, as well as the longest running promotion time. This could explain why it has had the lowest total customers and sales.

3.2 Feature Correlations

- The next step to understanding the relationships within the data was understanding how the features were correlated with the target variable, **Sales**. To display this, I plotted a heat map, which uses different shades to identify how correlated features are with one another. The darker they are, the more correlated, and the lighter, the less correlated.

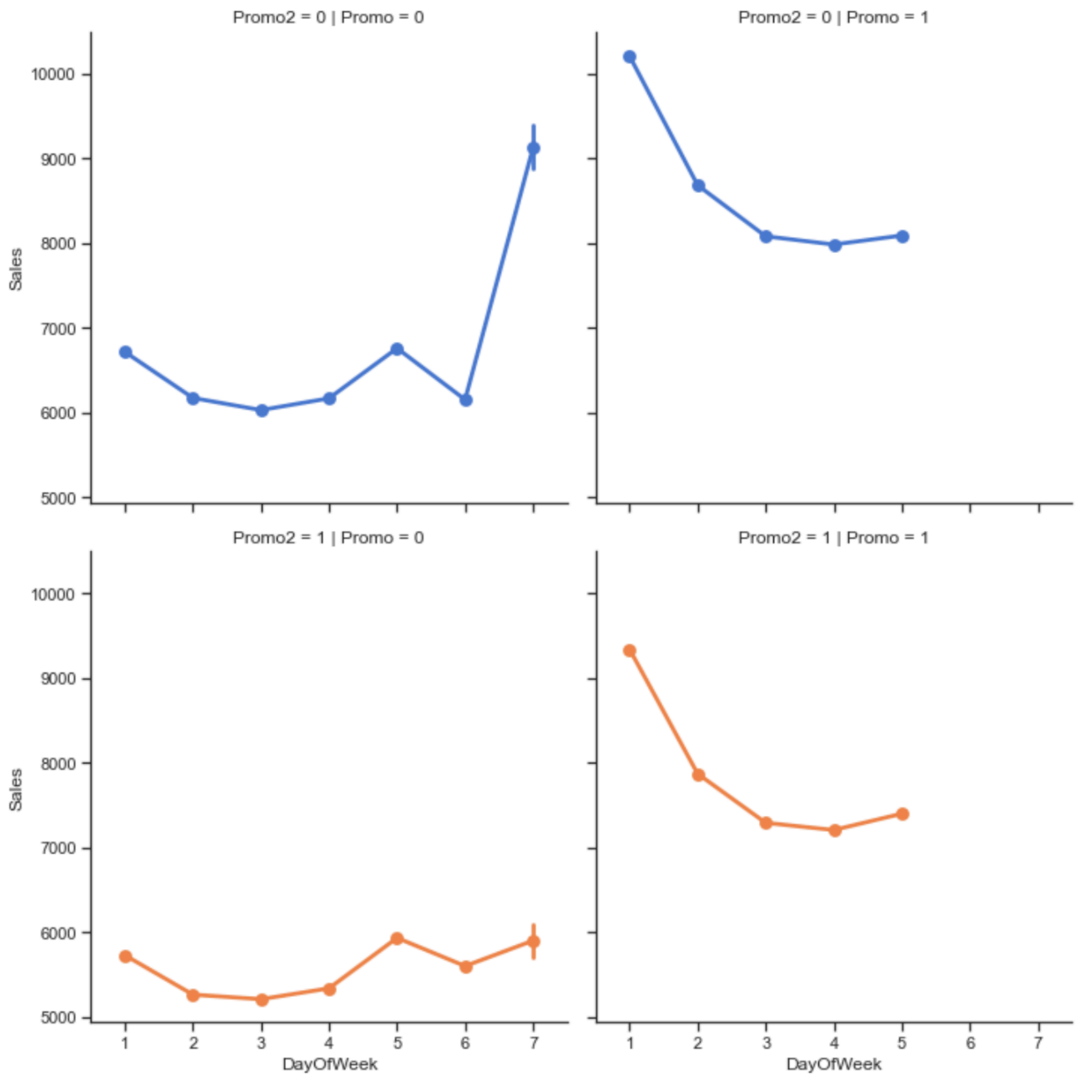


- By observing this chart, I noticed that the features highly correlated with **Sales** were not very abundant, although few had very negative correlations. In order to better understand this, I displayed some of the actual correlations numerically, basing it off of which features seemed darkest at first glance.

```
                    Sales  Customers     Promo  PromoOpen    Promo2  \
Sales            1.000000   0.823552  0.368199   0.127618 -0.127556
Customers        0.823552   1.000000  0.182859   0.202186 -0.202207
Promo            0.368199   0.182859  1.000000   0.000338 -0.000316
PromoOpen        0.127618   0.202186  0.000338   1.000000 -0.999999
Promo2          -0.127556  -0.202207 -0.000316  -0.999999  1.000000
Promo2SinceWeek -0.058493  -0.130864 -0.000795  -0.759489  0.759536
SalePerCustomer  0.186563  -0.323926  0.280027  -0.215719  0.215883

                 Promo2SinceWeek  SalePerCustomer
Sales                  -0.058493         0.186563
Customers              -0.130864        -0.323926
Promo                  -0.000795         0.280027
PromoOpen              -0.759489        -0.215719
Promo2                  0.759536         0.215883
Promo2SinceWeek         1.000000         0.198835
SalePerCustomer         0.198835         1.000000
```

- In order of correlation strength with **Sales**, these can be ranked **Customers**, **Promo**, **SalePerCustomer,** then **PromoOpen**. **Customers** has a decently strong correlation with **PromoOpen**, meaning that its likely that running a promo draws more customers. Interestingly, **Promo2** and related variables seem to have a negative correlation with sales. To further understand this, I plotted the following factorplot:



- From this, I could see that when there are no promotions at all (top left plot), the sales slowly rise through the week, dip slightly on Saturday, and shoot back up on Sunday. When both promotions are happening (bottom right), sales are highest on Mondays, but this trend does not differ from when promo is running alone without promo2. Sales for only promo2 are fairly low, with no major changes from day to day.

3.3 EDA Conclusions

- **StoreType a** has the most sales and customers in total.
- **StoreType** b has the lowest sales per customer on average, but highest average sales quantity across stores. This would mean that they most likely sell a lot of cheap things.
- **StoreType** d has the highest sales per customer, but is still beat by a by about twice as much in terms of total sales and customers. This eludes to them selling fewer, more expensive things.
- Customers seem to spend the most money on Sunday when there are no promotions, and the most on Mondays when there are promotions.
- **Promo2** does not seem to help sales in any clear way, while promo definitely does.

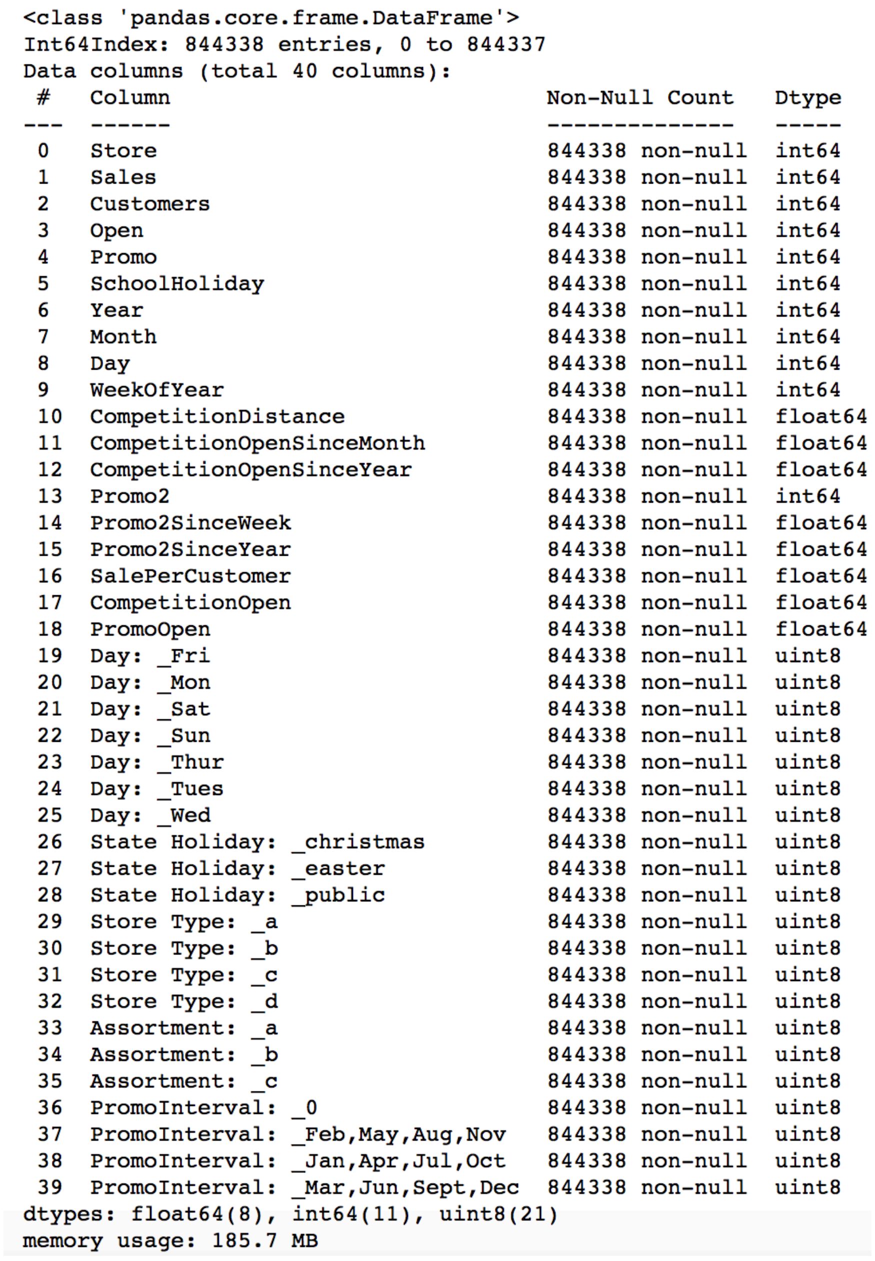
4. Feature Engineering and Preprocessing

4.1 Feature Engineering

- Being as that I preliminarily created a few new features, the main goal of this portion was to create dummy variables for the categorical features.

- Not to go too into the details here, but I basically renamed the data frame ‘df’ for simplicities sake, and created new features for each categorical column. For example, the **DayOfWeek** column had values 1-7 denoting each day of the week, so I used one-hot encoding to turn this into 7 boolean columns for each day, with the value 0 denoting it was not that particular day, and 1 denoting it was.

- The final dataframe was as follows:



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 844338 entries, 0 to 844337
Data columns (total 40 columns):
 #   Column                           Non-Null Count   Dtype
---  ------                           --------------   -----
 0   Store                            844338 non-null  int64
 1   Sales                            844338 non-null  int64
 2   Customers                        844338 non-null  int64
 3   Open                             844338 non-null  int64
 4   Promo                            844338 non-null  int64
 5   SchoolHoliday                    844338 non-null  int64
 6   Year                             844338 non-null  int64
 7   Month                            844338 non-null  int64
 8   Day                              844338 non-null  int64
 9   WeekOfYear                       844338 non-null  int64
 10  CompetitionDistance              844338 non-null  float64
 11  CompetitionOpenSinceMonth        844338 non-null  float64
 12  CompetitionOpenSinceYear         844338 non-null  float64
 13  Promo2                           844338 non-null  int64
 14  Promo2SinceWeek                  844338 non-null  float64
 15  Promo2SinceYear                  844338 non-null  float64
 16  SalePerCustomer                  844338 non-null  float64
 17  CompetitionOpen                  844338 non-null  float64
 18  PromoOpen                        844338 non-null  float64
 19  Day: _Fri                        844338 non-null  uint8
 20  Day: _Mon                        844338 non-null  uint8
 21  Day: _Sat                        844338 non-null  uint8
 22  Day: _Sun                        844338 non-null  uint8
 23  Day: _Thur                       844338 non-null  uint8
 24  Day: _Tues                       844338 non-null  uint8
 25  Day: _Wed                        844338 non-null  uint8
 26  State Holiday: _christmas        844338 non-null  uint8
 27  State Holiday: _easter           844338 non-null  uint8
 28  State Holiday: _public           844338 non-null  uint8
 29  Store Type: _a                   844338 non-null  uint8
 30  Store Type: _b                   844338 non-null  uint8
 31  Store Type: _c                   844338 non-null  uint8
 32  Store Type: _d                   844338 non-null  uint8
 33  Assortment: _a                   844338 non-null  uint8
 34  Assortment: _b                   844338 non-null  uint8
 35  Assortment: _c                   844338 non-null  uint8
 36  PromoInterval: _0                844338 non-null  uint8
 37  PromoInterval: _Feb,May,Aug,Nov  844338 non-null  uint8
 38  PromoInterval: _Jan,Apr,Jul,Oct  844338 non-null  uint8
 39  PromoInterval: _Mar,Jun,Sept,Dec 844338 non-null  uint8
dtypes: float64(8), int64(11), uint8(21)
memory usage: 185.7 MB
```

- I did a quick check to make sure there no missing entries, which there weren't, and the final dataframe had 40 features in total.

3.2 Preprocessing

- I split my data into a train/test set, using a test size of 25%.

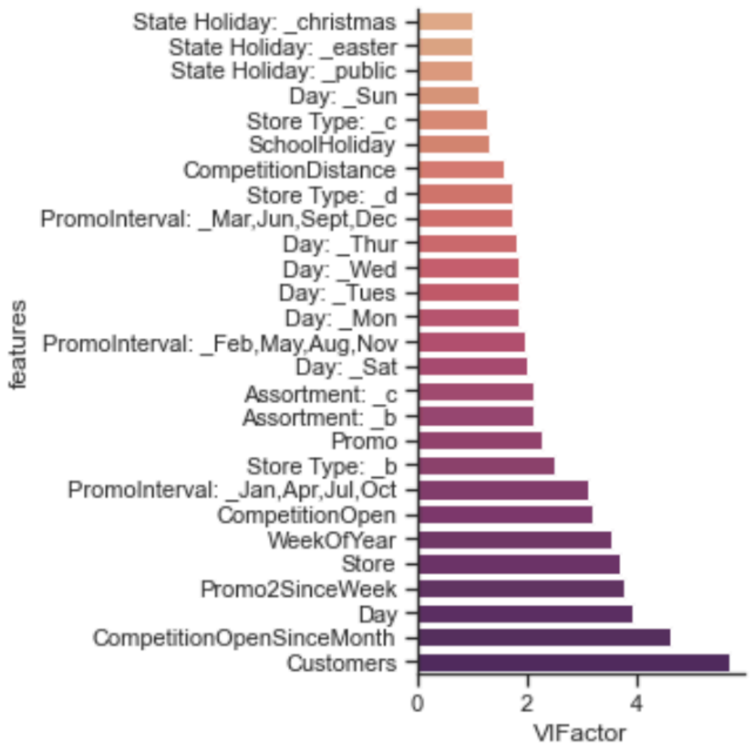
- My next task was to use the variance inflation factor (VIF) to remove multicollinearity from the data. The purpose of this is to determine which predictor variables are correlated with one another enough to hinder the regression model. The VIF estimates how much of the variance of the model is inflated due to this multicollinearity. Using the following function I was able to weed out the important features I would be using for modeling. Typically, anything over a VIF threshold of 5 is removed due to high correlation, but being as that my main predictor **Customers** had a VIF of 5.67, I adjusted this threshold to 5.7. Luckily, **Customers** was the only feature that was between 5 and 5.7, so I didn't end up with a slew of additional variables in making this adjustment.

```
def iterate_vif(df, vif_threshold=5.7, max_vif=6):
  count = 0
  while max_vif > vif_threshold:
    count += 1
    print("Iteration # "+str(count))
    vif = pd.DataFrame()
    vif["VIFactor"] = [variance_inflation_factor(df.values, i) for i in range(df.shape[1])]
    vif["features"] = df.columns

    if vif['VIFactor'].max() > vif_threshold:
      print('Removing %s with VIF of %f' % (vif[vif['VIFactor'] == vif['VIFactor'].max()]['features'].values
[0], vif['VIFactor'].max()))
      df = df.drop(vif[vif['VIFactor'] == vif['VIFactor'].max()]['features'].values[0], axis=1)
      max_vif = vif['VIFactor'].max()
    else:
      print('Complete')
      return df, vif.sort_values('VIFactor')

final_df, final_vif = iterate_vif(X_train)
```

- This function iterated 13 times, and left me with the following features and their corresponding VIFactor:



	VIFactor	features
1	5.675332	Customers
7	4.606351	CompetitionOpenSinceMonth
4	3.932738	Day
8	3.781941	Promo2SinceWeek
0	3.674038	Store
5	3.549141	WeekOfYear
9	3.202915	CompetitionOpen
25	3.116132	PromoInterval: _Jan,Apr,Jul,Oct
19	2.486271	Store Type: _b
2	2.249827	Promo
22	2.120728	Assortment: _b
23	2.097133	Assortment: _c
11	1.990614	Day: _Sat
24	1.943642	PromoInterval: _Feb,May,Aug,Nov
10	1.860659	Day: _Mon
14	1.857320	Day: _Tues
15	1.827611	Day: _Wed
13	1.796533	Day: _Thur
26	1.748758	PromoInterval: _Mar,Jun,Sept,Dec
21	1.734930	Store Type: _d
6	1.566222	CompetitionDistance
3	1.301851	SchoolHoliday
20	1.261911	Store Type: _c
12	1.127813	Day: _Sun
18	1.010524	State Holiday: _public
17	1.006615	State Holiday: _easter
16	1.005395	State Holiday: _christmas

- I redefined the train and test set to only include these features, and it was time to begin modeling.

5. Modeling

- I used 4 models here, and tested their predictive power to see which model performed the best. These models are linear regression, lasso regression, random forrest regression, and xg boost regression. The metric of choice for comparison is the RMSPE or **R**oot **M**ean **S**quared **P**ercentage **E**rror, and is defined as

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

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 0 sales is ignored in scoring. I defined the following function to implement this metric in my code:

```python
def rmspe(y_true, y_pred):
    '''
    Compute Root Mean Square Percentage Error between two arrays.
    '''
    loss = np.sqrt(np.mean(np.square(((y_true - y_pred) / y_true)), axis=0))

    return loss
```

- Fitting and predicting with all four models, I found their RMSPE scores and added them to a dataframe for easy comparison:

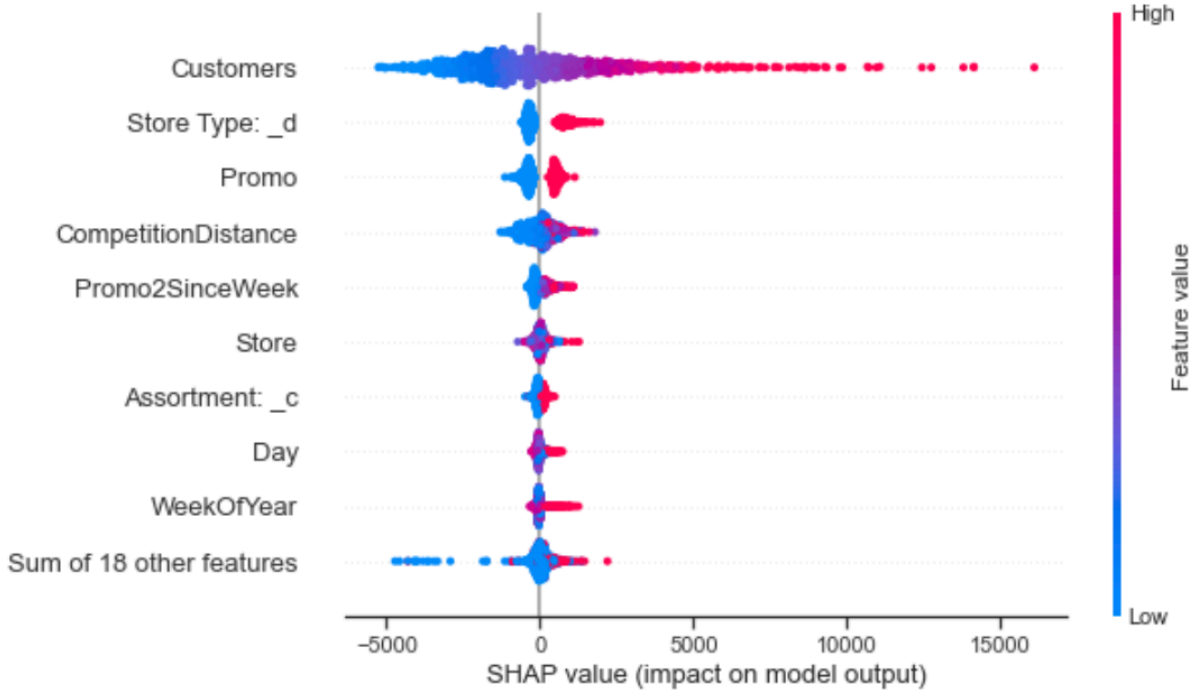
	Linear Regression	**Lasso Regression**	**Random Forest Regression**	**XGBoost**
Metrics/ Model				
RMSPE (Test data)	0.193	0.364	0.142	0.083

- The XG Boost regression model clearly performed the best, almost twice as good as the next best model. (Further hyperparameter tuning via cross validation with random search will be implemented at a later time).

6. SHAP Analysis

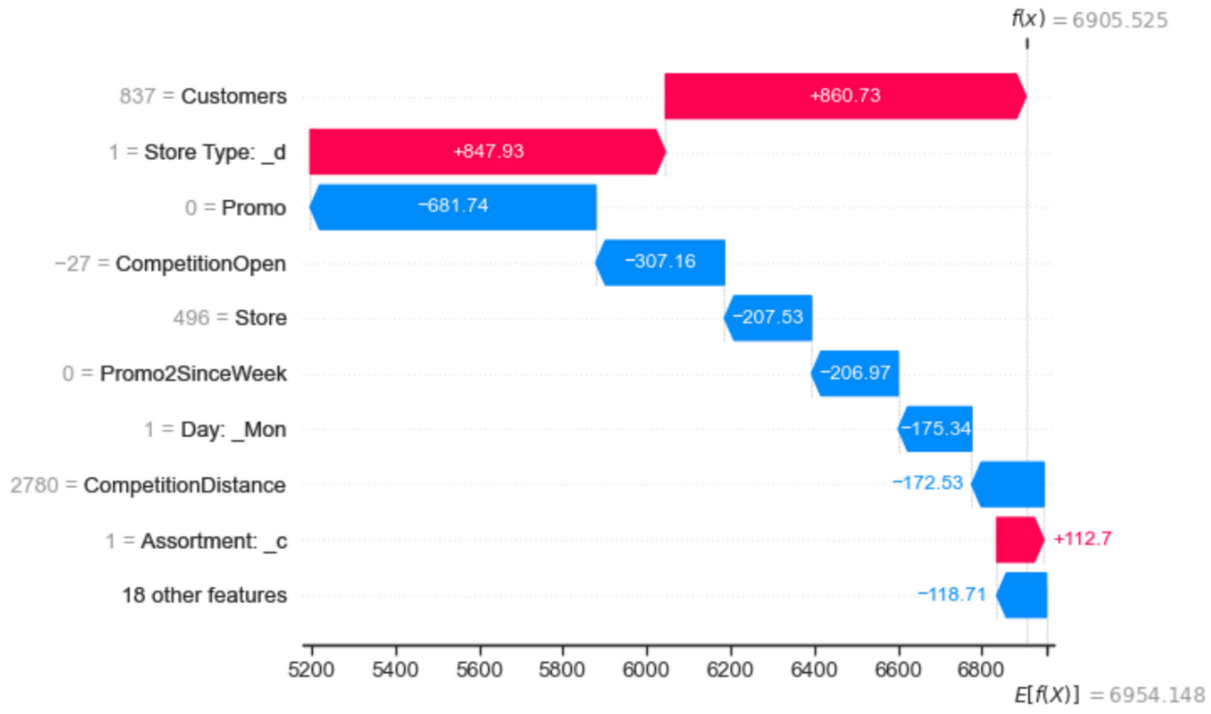
- SHAP stands for **SH**apley **A**dditive ex**P**planations, and is used to interpret the impact features have on the predictability of the model.

Summary Plot



- **Feature importance**: From top to bottom, features are ranked in order of importance.
- **Impact**: The horizontal position shows whether the effect of the value is associated with a higher or lower prediction.
- **Original value**: Color shows whether that variable is high (in red) or low (in blue) for that observation.
- **Correlation**: A high level of 'Customers' has a high positive impact on the amount of sales predicted. The “high” comes from the red color, and the “positive” impact is shown on the x-axis. Similarly, we can say that just about all of the other features have positive impacts, whether indicated by low negative impacts, or high positive ones.

Waterfall Plot



- Waterfall plots are used to display explanations for individual predictions, so they expect a single row of an explanation object as input. The bottom of a waterfall plot starts as the expected value of the model output, and then each row as you go up shows how the positive (red) or negative (blue) contribution of each feature moves the value from the expected model output over the background dataset to the model output for this prediction.

7. Conclusion

- Of the 4 models tested, the XG Boost regressor does an outstanding job of predicting store sales. It would be in the best interest of Rossmann to consider implementing such a model as to make the predictions they need to.
- The number of customers a store experiences is the main driver of sales, which is probably not a surprise to the company.
- The next couple of features ranked by predictability power are as follows: **StoreType_d**, **Promo**, **CompetitionOpen**, **Store**, **Promo2SinceWeek**, **Day_Mon**, **CompetitionDistance**, and **Assortment_c**. The bottom 18 features do not constitute enough impact to be listed. Almost all of these features listed have a negative correlation with sales, except for **StoreType_d** and **Assortment_c**, so this would be important for the company to take note of.

- Using this information about the feature importances, clearly store type d is doing something correct, that possibly the other store types could adapt. Promotions are surprisingly not great indicators of sales, so the company needs to shift their focus from using promotions to observing their competition more.
- For future improvements, it would be very useful if there was more information on the competition, seeing as how it has such a great impact on sales. Knowing how the competition conducts their sales and promotions, as well as how similar they are to the nearby Rossmann stores are all things to consider.