UNIVERSITY OF ECONOMICS AND LAW
VIETNAM NATIONAL UNIVERSITY HCM

FINAL SEMESTER PROJECT:
ROSSMAN STORE SALES FORECAST

Lecture: Phan Huy Tâm
Course: Data Mining
Name: Phạm Duy Tùng
Student ID: K214140961

Ho Chi Minh City, January 7th, 2023

TABLE OF CONTENT

Topic: Rossman store sales forecast

Question:

1. Describe the dataset and the problem statement, what is the nature of this case?
2. Does data have any defects or issues? State the solution if any!
3. What kind of model could be used in this case? Explain!
4. Perform data exploratory analysis (you could use descriptive analysis or charts)!
5. Is there any special point or potential issue that the analyst must pay attention to?
6. Bonus: perform the model to solve the problem, discuss the result, make conclusions or recommendations (if any), this part must be done by code (either R or python).

1. Describe the dataset and the problem statement, what is the nature of this case?

Describe the dataset:

With a quick glimpse at the dataset, we could observe there are 2 main csv files included in, namely rossman_train and rossman_store. In the rossman_train file, there are 1,017,210 observations with 9 recorded attributes.

Data – Historical sales (rossman_train.csv file):

- Store: a unique id for each store
- DayOfWeek: the date of week in which the store sale information record (1 = Mon, 2 = Tues, 3 = Wed, 4 = Thur, 5 = Fri, 6 = Sat, 7 = Sun)
- Date: that date of recording store sale information
- Sales: the turnover for any given day (this is what you are predicting)
- Customers: the number of customers on a given day
- Open: an indicator for whether the store was open: 0 = closed, 1 = opened
- Promo: indicates whether a store is running a promo on that day
- StateHoliday: indicates 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. 0 = none, 1 = holiday.
- SchoolHoliday: indicates if the (Store, Date) was affected by the closure of public schools.

In the rossman_store file, there are 1115 observations with 10 attributes.

Data – Stores (rossman_store.csv file):

- Store: a 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: Promo2 is a continuting and consecutive promotion for some stores: 0 = store is not participating, 1 = store is participating
- Promo2Since[Week/Year]: describes the calendar week and year when the store started participating in Promo2
- PromoInterval – describes the consecutive intervals Promo2 is started, naming the months the promotion is started anew. E.g., "Feb,May,Aug,Nov" means each round starts in February, May, August, November of any given year for that store

The data ranges in which company provided constraint from 01/01/2013 to 31/07/2015 and is collected at a daily frequency, considered all the sale of all the stores available.

The problem statement:

Rossman operates over 3,000 drug stores in 7 European countries. Currently, Rossman store managers are tasked with predicting their daily sales for up to 6 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. Historical sales data for 1,115 Rossmann stores are provided.

With all this information, the task for us is to predict the sales of the Rossman stores for the next 6 weeks.

Natural of the case:

This dataset and problem represent a type of data, modeling and analysis which is called "Time series". The dataset provided is a time series data, which is a collection of quantities that are assembled over even intervals in time and ordered chronologically, in this case, it's the information about Rossman store sale situation over the given time period. And because data points in time series are collected at adjacent time periods, there is potential for correlation between observations. If the correlation between these observations exists, it could reveal hidden insights and enable deeper understanding of the situation for those who are responsible. The use of time series model and analysis could be for a variety of reasons: predicting future outcomes, understanding past outcomes, making policy suggestions, and much more. In this case to forecast Rossman sales for 6 weeks in advance.

2. Does data have any defects or issues? State the solution if any!

Inspect the data:

On first glance, as we query through the first and the last 5 observations to get a closer look, the rossman_training dataset appears to be as a whole and show no sign of missing or impaired value.

```
In [4]: #how does the data looks like:
        pd.concat([train_df.head(), train_df.tail()]) #show the first and last 5 rows.
```

Out[4]:

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
0	1	5	7/31/2015	5263	555	1	1	0	1
1	2	5	7/31/2015	6064	625	1	1	0	1
2	3	5	7/31/2015	8314	821	1	1	0	1
3	4	5	7/31/2015	13995	1498	1	1	0	1
4	5	5	7/31/2015	4822	559	1	1	0	1
1017204	1111	2	1/1/2013	0	0	0	0	a	1
1017205	1112	2	1/1/2013	0	0	0	0	a	1
1017206	1113	2	1/1/2013	0	0	0	0	a	1
1017207	1114	2	1/1/2013	0	0	0	0	a	1
1017208	1115	2	1/1/2013	0	0	0	0	a	1

Then we check for null value of all items in the dataset, the result shows there is no null value persist.

```
In [5]: # no missing values.
        train_df.isnull().all()
```

```
Out[5]: Store            False
        DayOfWeek        False
        Date             False
        Sales            False
        Customers        False
        Open             False
        Promo            False
        StateHoliday     False
        SchoolHoliday    False
        dtype: bool
```

We also look at the first and last 10 observations of the rossman_store dataset, which review there are some missing values.

```
In [19]: pd.concat([store_df.head(),store_df.tail()])
```

Out[19]:

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Promo2SinceWeek	Promo2SinceYea
0	1	c	a	1270.0	9.0	2008.0	0	NaN	Nal
1	2	a	a	570.0	11.0	2007.0	1	13.0	2010.
2	3	a	a	14130.0	12.0	2006.0	1	14.0	2011.
3	4	c	c	620.0	9.0	2009.0	0	NaN	Nal
4	5	a	a	29910.0	4.0	2015.0	0	NaN	Nal
1110	1111	a	a	1900.0	6.0	2014.0	1	31.0	2013.
1111	1112	c	c	1880.0	4.0	2006.0	0	NaN	Nal
1112	1113	a	c	9260.0	NaN	NaN	0	NaN	Nal
1113	1114	a	c	870.0	NaN	NaN	0	NaN	Nal
1114	1115	d	c	5350.0	NaN	NaN	1	22.0	2012.

Calculate how many missing data do we have in % for each attribute.

```
In [20]: #how may missing data do we have in %:
         100- (store_df.count() / store_df.shape[0] * 100)
```

```
Out[20]: Store                      0.000000
         StoreType                  0.000000
         Assortment                 0.000000
         CompetitionDistance        0.269058
         CompetitionOpenSinceMonth 31.748879
         CompetitionOpenSinceYear  31.748879
         Promo2                     0.000000
         Promo2SinceWeek           48.789238
         Promo2SinceYear           48.789238
         PromoInterval             48.789238
         dtype: float64
```

View how many non-null items in listing by each attribute.

```
In [21]: store_df.info()
```

```
<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
```

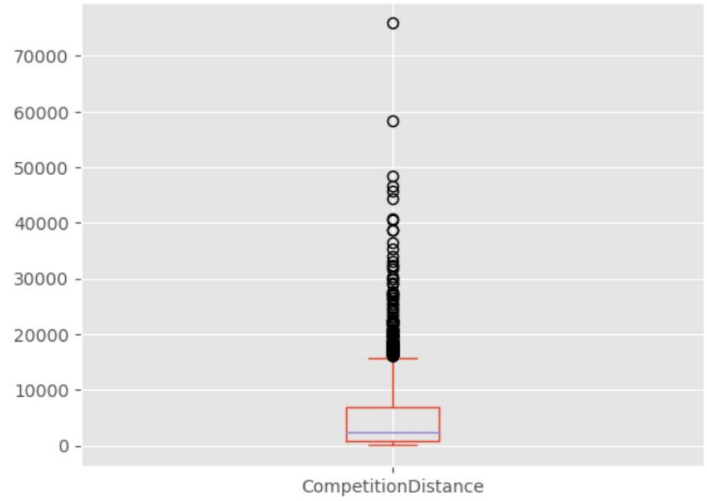
State the solution for missing data:

CompetitionDistance

First, let's take a look at the outliers, so we can decide whether to choose mean or median to fill the NaN of the missing values. This is the most common method of imputing missing values of numeric columns. But if there are outliers then the mean will not be appropriate. Instead, it would be better to use the median value for imputation in the case of outliers.

```
In [22]: store_df.CompetitionDistance.plot.box()
         print("the median is", store_df.CompetitionDistance.median(), "and mean is", store_df.CompetitionDistance.mean())
```

```
the median is 2325.0 and mean is 5404.901079136691
```



Since we have here some outliers, it's better to input the median value to those few missing values.

```
In [23]: print("Since we have here some outlier, its better to input the median value to those few missing values.")
         store_df["CompetitionDistance"].fillna(store_df["CompetitionDistance"].median(), inplace = True)
```

```
Since we have here some outlier, its better to input the median value to those few missing values.
```

CompetitionOpenSinceMonth and CompetitionOpenSinceYear

Because the stores had no competition, the missing values are not there. So, we will fill the missing values with zeros.

```
In [24]: store_df["CompetitionOpenSinceMonth"].fillna(0, inplace = True)
         store_df["CompetitionOpenSinceYear"].fillna(0, inplace = True)
```

Promo2SinceWeek, Promo2SinceYear and PromoInterval

Because there are no promotions that had been made, we could replace the NaN from Promo2SinceWeek/Year and PromoInterval with zero.

```
In [26]: store_df["Promo2SinceWeek"].fillna(0, inplace = True)
         store_df["Promo2SinceYear"].fillna(0, inplace = True)
         store_df["PromoInterval"].fillna(0, inplace = True)
```

Let’s check back to our dataset to see whether the data has been added or not.

```
store_df.info()
```

```
<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        1115 non-null   float64
 4   CompetitionOpenSinceMonth  1115 non-null   float64
 5   CompetitionOpenSinceYear   1115 non-null   float64
 6   Promo2                     1115 non-null   int64
 7   Promo2SinceWeek            1115 non-null   float64
 8   Promo2SinceYear            1115 non-null   float64
 9   PromoInterval              1115 non-null   object
dtypes: float64(5), int64(2), object(3)
memory usage: 87.2+ KB
```

3. What kind of model could be used in this case? Explain!

Because the type of data we have is time series, the following predictive models would be the most appropriate to use in this scenario:

- SARIMAX
- Random Forest Regression
- XGBoost

ARIMA and SARIMAX:

What is ARIMA and SARIMAX Model?

ARIMA is a method for forecasting or predicting future outcomes based on a historical time series. It is based on the statistical concept of serial correlation, where past data points influence future data points. SARIMAX is an updated version of the ARIMA model. ARIMA includes an autoregressive integrated moving average, while SARIMAX includes seasonal effects and eXogenous factors with the autoregressive and moving average component in the model. Therefore, we can say SARIMAX is a seasonal equivalent model.

How does it work?

$$Y_t = \beta_2 + \omega_1 \varepsilon_{t-1} + \omega_2 \varepsilon_{t-2} + \ldots + \omega_q \varepsilon_{t-q} + \varepsilon_t$$

ARIMA Formula

ARIMA model is a class of linear models that utilizes historical values to forecast future values. ARIMA stands for Autoregressive Integrated Moving Average, each of which technique contributes to the final forecast:

- Autoregressive (AR): In an autoregression model, we forecast the variable of interest using a linear combination of past values of that variable. The term autoregression indicates that it is a regression of the variable against itself. That is, we use lagged values of the target variable as our input variables to forecast values for the future. (p - Number of autoregressive lags)
- Integrated (I): Integrated represents any differencing that has to be applied in order to make the data stationary. (d: Number of times differencing pre-processing step is applied to make the time series stationary)
- Moving Average (MA) – q: Moving average models uses past forecast errors rather than past values in a regression-like model to forecast future values. (q: Number of moving average lags)

ARIMA means Auto Regressive Integrated Moving Average. It is a combination of two models: AR (Auto Regressive) model which uses lagged values of the time series to forecast and MA (Moving Average) model that uses lagged values of residual errors to forecast. In other words, this model uses dependencies both between data values and errors values from the past to optimize the predictions.

$$d_t = c + \sum_{n=1}^{p} \alpha_n d_{t-n} + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \sum_{n=1}^{r} \beta_n x_{n_t} + \sum_{n=1}^{P} \phi_n d_{t-sn} + \sum_{n=1}^{Q} \eta_n \epsilon_{t-sn} + \epsilon_t$$

SARIMAX Formula

SARIMAX model inheritance the same characteristic as ARIMA model, but the difference between ARIMA and SARIMAX is the seasonality and exogenous factors ("S" stand for seasonality and "X" stand for exogenous factors). SARIMAX requires not only the p, d, and q arguments that ARIMA requires, but it also requires another set of p, d, and q arguments for the seasonality aspect as well as an argument called "s" which is the periodicity of the data's seasonal cycle.

Apply to this case?

The SARIMAX would be appropriate to use in this case as there are a lot of external factors that affect the store sales (promotions, competition, school and state holidays…) and because the interval of each data point is daily, running through the

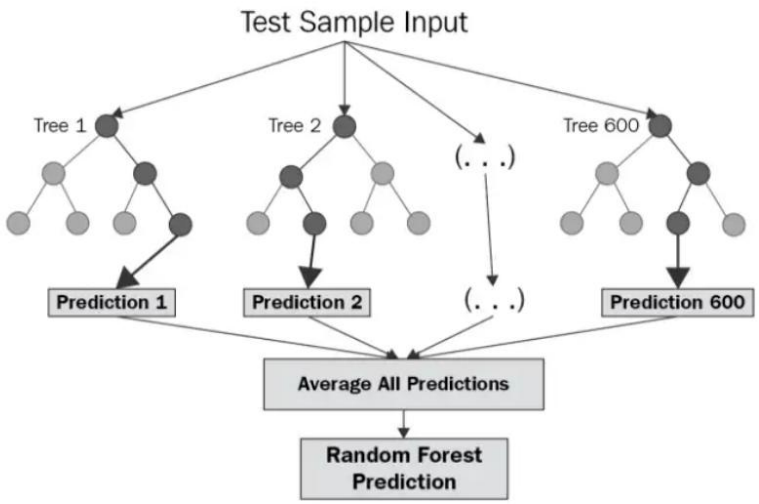
period of two and a half years, then it would be likely to exist a seasonal frequency that influence the model forecast.

Random Forest Regression:

What is Random Forest Regression Model?

Random Forest Regression is a supervised learning algorithm that uses ensemble learning method for regression. Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

How does it work?



The diagram above shows the structure of a Random Forest. The algorithm creates each tree from a different sample of input data. At each node, a different sample of features is selected for splitting and the trees run in parallel without any interaction amongst them. Because all the trees are running in parallel, it would create an independent state and present no cross influence on each other. A Random Forest operates by constructing several decision trees during training time and outputting the mean of the classes as the prediction of all the trees.

Apply to this case?

Random forest regression is used to solve a variety of business problems where the company needs to predict a continuous value. Like predicting future prices/costs, you can use random forest regression to predict what the prices of these products and services will be in the future. Or predict future revenue, using random forest regression to model your operations, discover the connection between the input and output. This connection

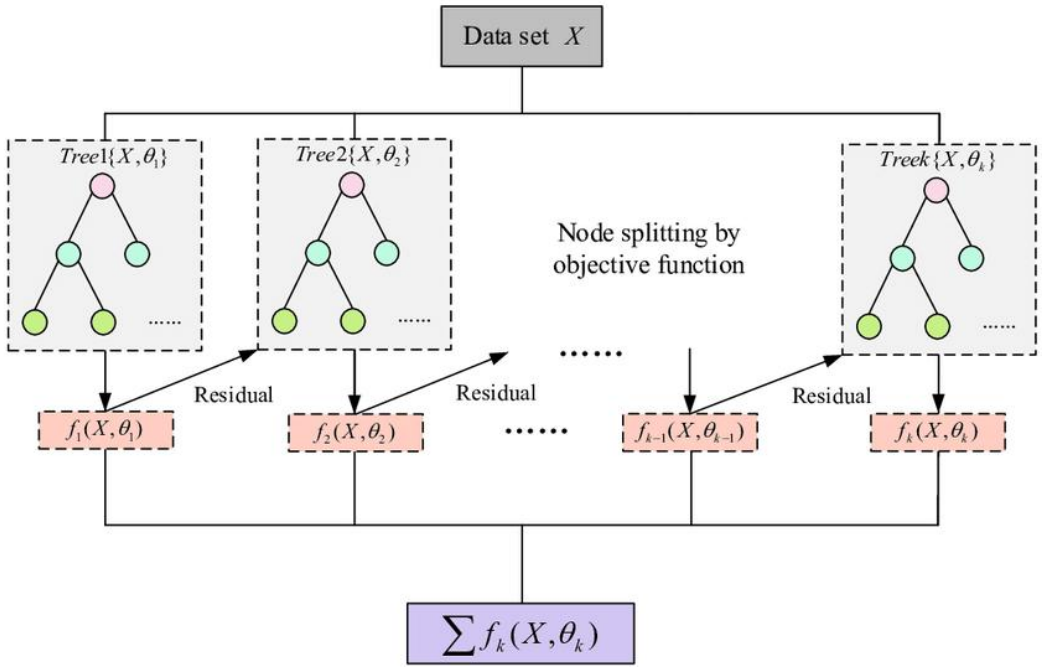
can be used to predict how much revenue you will generate based on the growth activity. In terms of this case, the model would be an appropriate choice to forecast the sale of Rossman stores considering the utility of the model when applying for this type of business problem.

XGBoost:

What is XGBoost Model?

XGBoost stands for "Extreme Gradient Boosting". XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. It implements Machine Learning algorithms under the Gradient Boosting framework. It provides a parallel tree boosting to solve many data science problems in a fast and accurate way.

How does it work?



Flow chart of XGBoost

In this algorithm, decision trees are created in sequential form. Weights play an important role in XGBoost. Weights are assigned to all the independent variables which are then fed into the decision tree which predicts results. The weight of variables predicted wrong by the tree is increased and these variables are then fed to the second decision tree. These individual classifiers/predictors then ensemble to give a strong and more precise model. It can work on regression, classification, ranking, and user-defined prediction problems.

Apply to this case?

XGBoost is a fast implementation of a gradient boosted tree. It has obtained good results in many domains including time series forecasting.

XGBoost could be considered for any supervised machine learning task when satisfies the following criteria:

- When you have large number of observations in training data. (Recommended over 1000 training samples, condition satisfied)
- Number features < number of observations in training data. (Dataset have over 1 million observations and 9 attributes, condition satisfied)
- It performs well when data has mixture numerical and categorical features or just numeric features or just numeric features (Condition satisfied)
- When the model performance metrics are to be considered. (We want to have the best performance for our prediction so of course a satisfied condition)

4. Perform data exploratory analysis (you could use descriptive analysis or charts)!

Exploratory Data Analysis (EDA)

a) Univariate Analysis (understanding each field in the dataset)

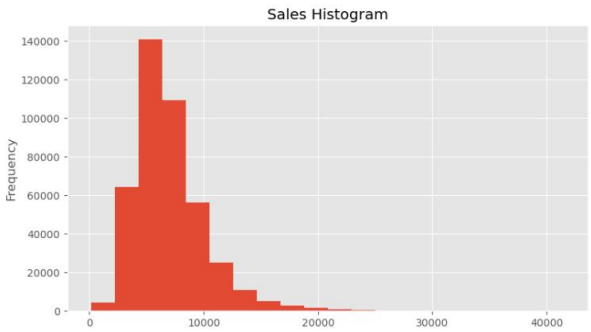
Variable Sales

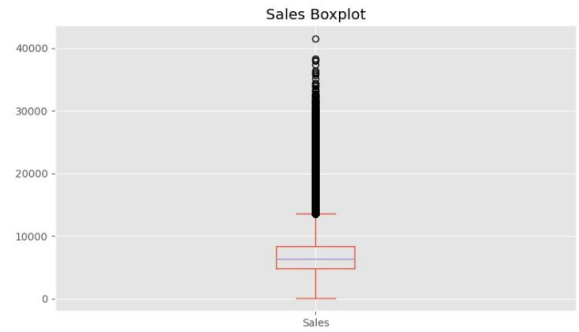
There are a lot of stores which have sales at 0 due to not being open for business on the given day. As such information is not useful for summarizing the sales variable, it would be best to exclude them temporarily for now.

```
In [6]: opened_sales = (train_df[(train_df.Open == 1) & (train_df.Sales)]) #if the stores are opend
        opened_sales.Sales.describe()

Out[6]: count    422307.000000
        mean       6951.782199
        std        3101.768685
        min         133.000000
        25%        4853.000000
        50%        6367.000000
        75%        8355.000000
        max       41551.000000
        Name: Sales, dtype: float64
```

The total number of open for business days at each store is 422,307 days. The mean for total sales at around 6,952. Median is at 6,367. Standard deviation is 3,101.768685. The minimum sales in a day is 133 and the maximum is 41,551. Q1 is

4,853 and Q3 is equal to 8,355. Below is the Histogram of variable Sales and Boxplot diagram.





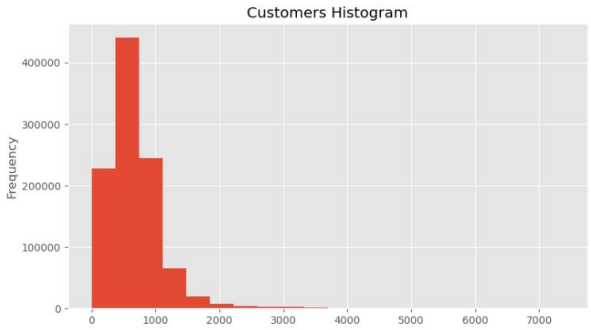
Rossmann has 13.45 % of the time big sales, over 10.000 Euros and has 0.0277 % of the time low sales, under 1000 Euros.

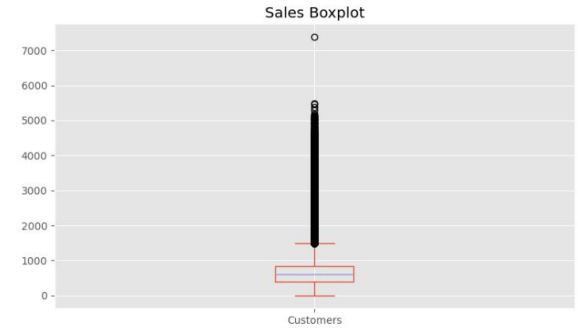
Variable Customers

The following is a brief description contains information of the Customers variable:

```
In [9]: train_df.Customers.describe()

Out[9]: count    1.017209e+06
        mean     6.331459e+02
        std      4.644117e+02
        min      0.000000e+00
        25%      4.050000e+02
        50%      6.090000e+02
        75%      8.370000e+02
        max      7.388000e+03
        Name: Customers, dtype: float64
```

The total number of customers who have paid a visit to Rossman store for the last two and a half years is 1,017,209 customers. The mean for total customer at around 633. Median is at 609. Standard deviation is 464.4411. The minimum customer in a day is 0 and the maximum is 7,388. Q1 is 405 and Q3 is equal to 837. Below is the Histogram of variable Customers and Boxplot diagram.





It seems like they had a great march out on 22nd of January 2013. They hit the record of customers at 7,388.

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
993496	817	2	1/22/2013	27190	7388	1	1	0	0

Variable Open

```
In [12]: print("In 3 years, different stores where", train_df[(train_df.Open == 0)].count()[0], "times closed")
         print("From this days,", train_df[(train_df.Open == 0) &
                 ((train_df.StateHoliday == "a") |
                 (train_df.StateHoliday == "b") |
                 (train_df.StateHoliday == "c"))].count()[0], "times the stores were closed because of holidays")
         print(train_df[(train_df.Open == 0) & (train_df.SchoolHoliday == 1)].count()[0], "times, some stores were closed because of schoo
         print("The stores were in some sundays opend ->", train_df[(train_df.Open == 1) & (train_df.DayOfWeek == 7)].count()[0], "times")
         print("However,", train_df[(train_df.Open == 0) & ((train_df.StateHoliday == "0") | (train_df.StateHoliday == 0)) & (train_df.Sch
               "times, the stores were closed for no reason (No Holidays or Sunday)")
```

In 3 years, different stores have 172,817 times closed. From this days, 30,140 times the stores were closed because of holidays. 18,264 times, some stores were closed because of school holiday. The stores were in some Sundays opened 3593 times. However, 139,610 times, the stores were closed for no reason (No Holidays or Sunday).

Rossman described clearly, that they were undergoing refurbishments sometimes and had to close. Most probably those were the times when this event happened. However, we don't want to have those observations in our dataset, when predicting. So, let's delete those days to make our dataset neater and the data won't affect when we train model.

Delete the times when the stores were open with no sales because of days in inventory.

```
In [18]: train_df = train_df.drop(train_df[(train_df.Open == 0) & (train_df.Sales == 0)].index)
         train_df = train_df.reset_index(drop = True) # to ge the indexes back to 0, 1, 2,etc.

         train_df.isnull().all() #to check for NaNs
```

```
Out[18]: Store              False
         DayOfWeek          False
         Date               False
         Sales              False
         Customers          False
         Open               False
         Promo              False
         SchoolHoliday      False
         StateHoliday_cat   False
         dtype: bool
```

Variable Promotion

```
In [14]: print(round((train_df.Promo[train_df.Promo == 1].count() / train_df.shape[0] * 100), 2)
```

For 38.15% of the time, there has been at least 1 promotion that had been running.

Variable StateHoliday

```
In [15]: # StateHoliday is not a continous number.
         train_df.StateHoliday.value_counts()
```

```
Out[15]: 0    855087
         0    131072
         a     20260
         b      6690
         c      4100
         Name: StateHoliday, dtype: int64
```

StateHoliday is a string and it not important to know what kind of holiday (a, b or c). So we will convert it into 0 and 1, by creating a new variable.

```
In [16]: train_df["StateHoliday_cat"] = train_df["StateHoliday"].map({0:0, "0": 0, "a": 1, "b": 1, "c": 1})
         train_df.StateHoliday_cat.count()
```

```
Out[16]: 1017209
```

Remove the StateHoliday column and use the new one.

In [17]:
```
train_df = train_df.drop("StateHoliday", axis = 1)
train_df.tail()
```

Out[17]:

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	SchoolHoliday	StateHoliday_cat
1017204	1111	2	1/1/2013	0	0	0	0	1	1
1017205	1112	2	1/1/2013	0	0	0	0	1	1
1017206	1113	2	1/1/2013	0	0	0	0	1	1
1017207	1114	2	1/1/2013	0	0	0	0	1	1
1017208	1115	2	1/1/2013	0	0	0	0	1	1

After the univariate analysis, we head to the multivariate analysis to evaluate multiple variables (more than two) to identify any possible association among them. Multivariate analysis offers a more complete examination of data by looking at all possible independent variables and their relationships to one another.

b) Multivariate Analysis (understanding the interactions between different fields and target)

Merge the files stores and training to accommodate the process of multivariate analysis.

In [27]:
```
train_store_df = pd.merge(train_df, store_df, how = "left", on = "Store")
train_store_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 844392 entries, 0 to 844391
Data columns (total 18 columns):
 #   Column                     Non-Null Count   Dtype
---  ------                     --------------   -----
 0   Store                      844392 non-null  int64
 1   DayOfWeek                  844392 non-null  int64
 2   Date                       844392 non-null  object
 3   Sales                      844392 non-null  int64
 4   Customers                  844392 non-null  int64
 5   Open                       844392 non-null  int64
 6   Promo                      844392 non-null  int64
 7   SchoolHoliday              844392 non-null  int64
 8   StateHoliday_cat           844392 non-null  int64
 9   StoreType                  844392 non-null  object
 10  Assortment                 844392 non-null  object
 11  CompetitionDistance        844392 non-null  float64
 12  CompetitionOpenSinceMonth  844392 non-null  float64
 13  CompetitionOpenSinceYear   844392 non-null  float64
 14  Promo2                     844392 non-null  int64
 15  Promo2SinceWeek            844392 non-null  float64
 16  Promo2SinceYear            844392 non-null  float64
 17  PromoInterval              844392 non-null  object
dtypes: float64(5), int64(9), object(4)
memory usage: 122.4+ MB
```

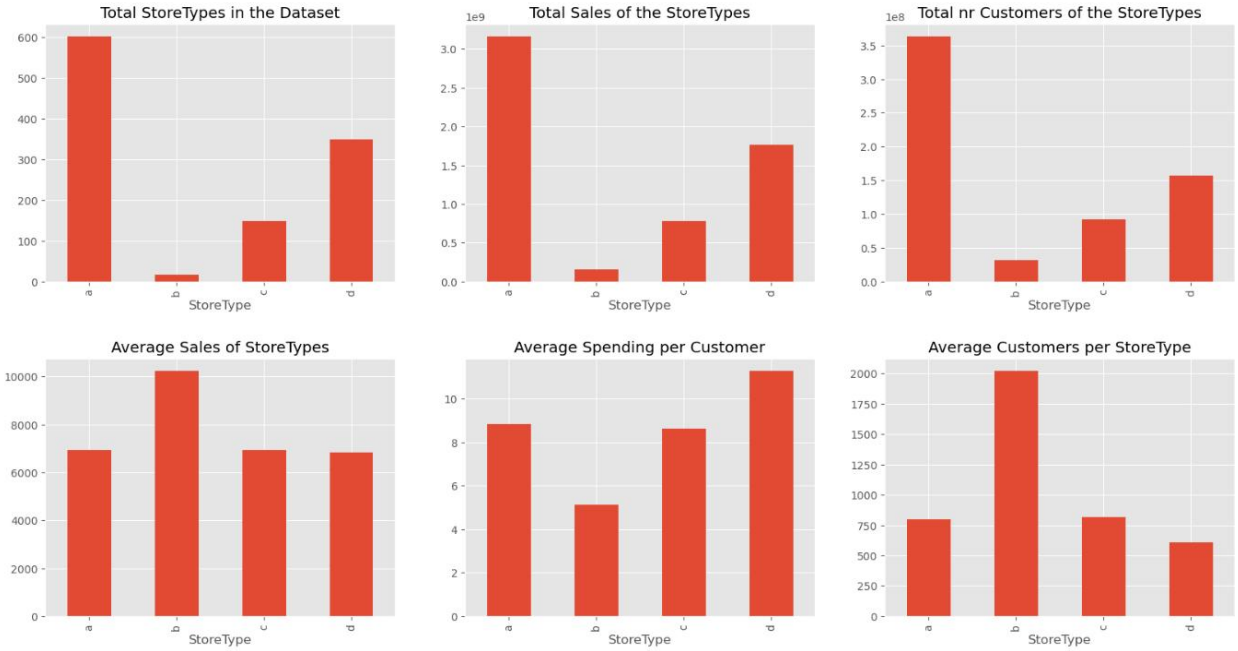
Compare the Stores type by Sales, Customers, Average Sales, Average Spending and Average Customers.

First, we calculate the Average Customer Sales with the available data from Customer and Sales attributes. We will name it “Avg_Customer_Sales”

```
In [28]: train_store_df["Avg_Customer_Sales"] = train_store_df.Sales / train_store_df.Customers
```

Then, we will create some bar chart that is associated with StoreTypes, in which StoreTypes is the vertical column and all other data type occupied for horizontal column.

```
In [29]: f, ax = plt.subplots(2, 3, figsize = (20,10))

store_df.groupby("StoreType")["Store"].count().plot(kind = "bar", ax = ax[0, 0], title = "Total StoreTypes in the Dataset")
train_store_df.groupby("StoreType")["Sales"].sum().plot(kind = "bar", ax = ax[0,1], title = "Total Sales of the StoreTypes")
train_store_df.groupby("StoreType")["Customers"].sum().plot(kind = "bar", ax = ax[0,2], title = "Total nr Customers of the StoreT
train_store_df.groupby("StoreType")["Sales"].mean().plot(kind = "bar", ax = ax[1,0], title = "Average Sales of StoreTypes")
train_store_df.groupby("StoreType")["Avg_Customer_Sales"].mean().plot(kind = "bar", ax = ax[1,1], title = "Average Spending per C
train_store_df.groupby("StoreType")["Customers"].mean().plot(kind = "bar", ax = ax[1,2], title = "Average Customers per StoreType

plt.subplots_adjust(hspace = 0.3)
plt.show()
```



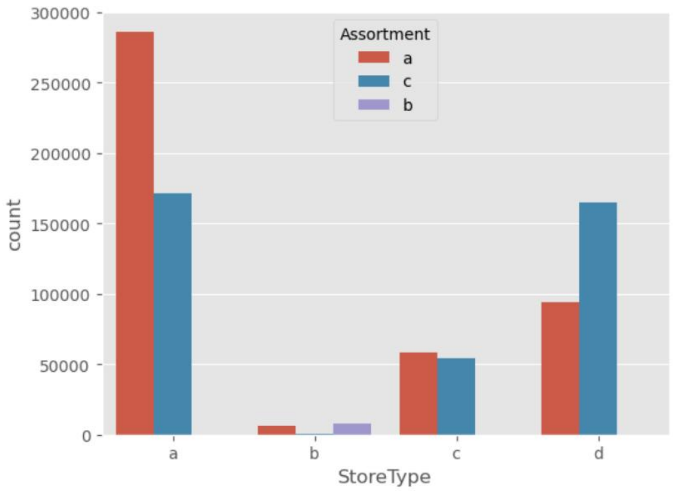
The charts above illustrate each attribute (Sales, Customers, Average Sales, Average Spending and Average Customers) according to each StoreTypes. As we can observe, here are some conclusions that could be extracted from the graphs:

- Store type A has the most stores, sales and customers.
- Store type D has the best averages spendings per customer.
- Store type B, with only 17 stores (is the least number type of store - b assortment is extremely low compared to other assortments), has the most average customers and most average sales per store.

- The majority of the stores, almost 54%, are type A stores. The same goes with the total sales and customers, type A store dominant the number, occupied with more over than half of the sales and customers.
- The number of average sales and customers of store type A and C are approximately coordinated, which could give an insight indicated the similarity of these two types of store.

Variable Assortments

```
In [30]: sns.countplot(data = train_store_df, x = "StoreType", hue = "Assortment", order=["a","b","c","d"])
         print("""So only the StoreType B has all assortments. I think thats why they are performing so good. Maybe this StoreType has mor
               The assortment C is a good one, because the StoreType D has the best average customer spending.""")

         plt.show()
```

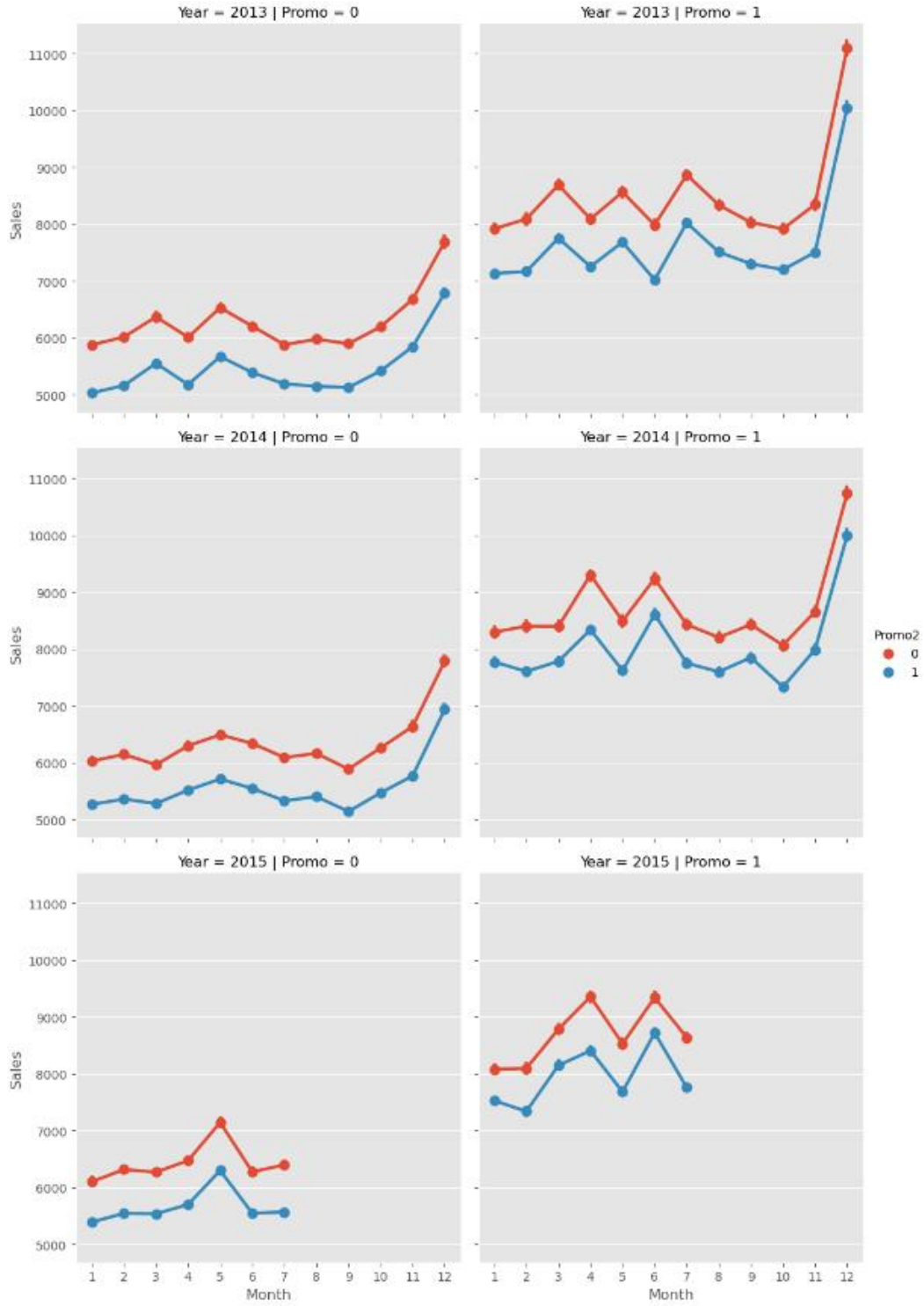


Only StoreType B has all assortments. That could be a suggestion to figure out why they are performing so good (could it be this StoreType has more sales area as related to the name “extend”). Assortment C is a good one, because StoreType D has the best average customer spending. The majority of the stores have type A assortments, other types of assortments are low (except for StoreType D).

Promotions – Promo and Promo2

First, we should exam whether promotions have influenced over the customers behavior by increasing their purchase. We could observe the effect by separate sales on days that have promotion and days that didn’t. Which type of promotions maintain a higher impact on customers (Promo or Promo2). And split the data into three separate periods, each period contains a year of information, to exam whether or not the data have seasonality frequency.

In [32]:
```
sns.factorplot(data = train_store_df, x ="Month", y = "Sales",
               col = 'Promo', # per store type in cols
               hue = 'Promo2',
               row = "Year"
              )
```

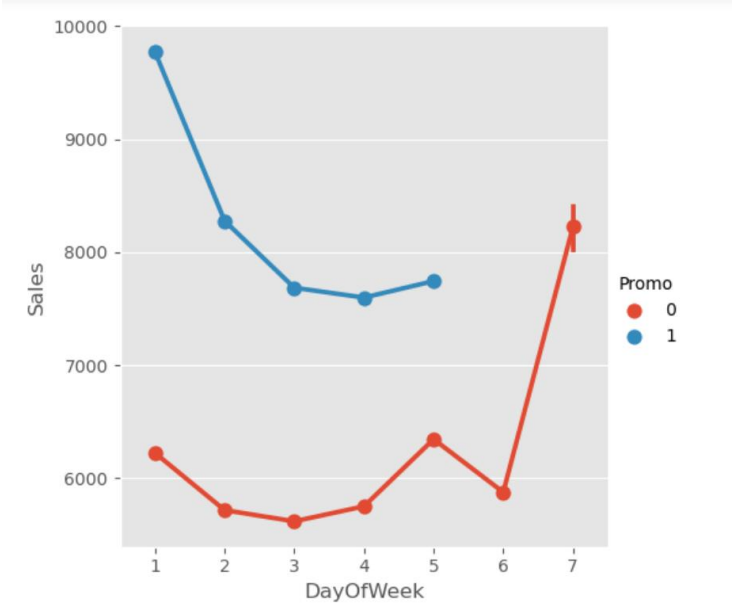


Needless to say, when the stores were having promotions, the sales were higher. Overall, the store promotions sales (Promo) are also higher than the seasonality promotions (Promo2). There are some patterns of a yearly trend that occurred, such as store sales spike up in March and May. And the sales built up significantly at the end of the year. However, those trends are not consistent and fluctuate differently on year. Because of that, it could not actually be considered to be a distinct annual trend.

DayOfWeek, Sales and Promo

We want to explore the correlation between three attributes: Day of Week, Sales, and Promo.

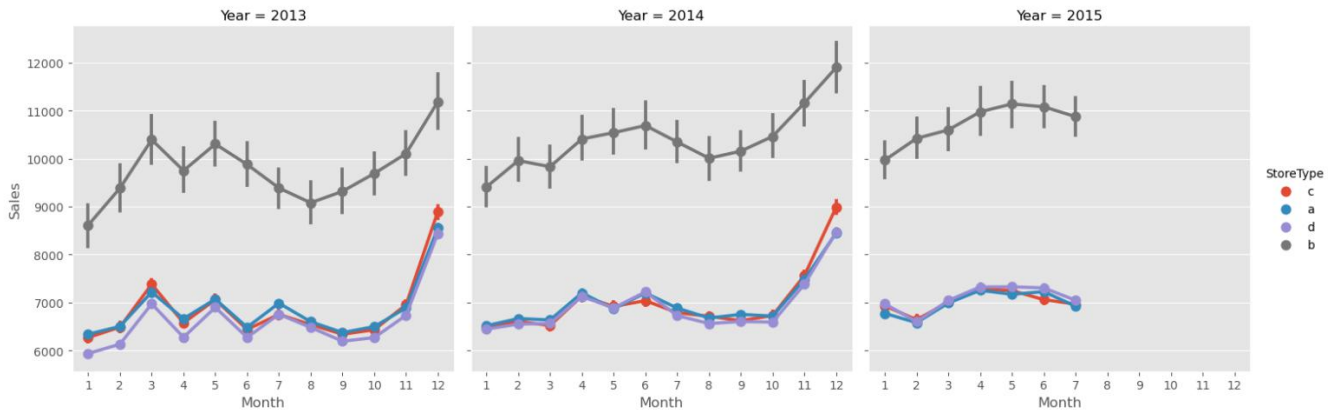
```
In [33]: sns.factorplot(data = train_store_df, x = "DayOfWeek", y = "Sales", hue = "Promo")
```



On the weekend there are no promotions, despite the fact that Sunday has the most sales over 62% of the time there are no promo campaign running. But on the other hand, the promotion runs on 38% of weekdays, which have Monday prove to be the most effective and gradually decline though out days later.

Trends on a yearly basis

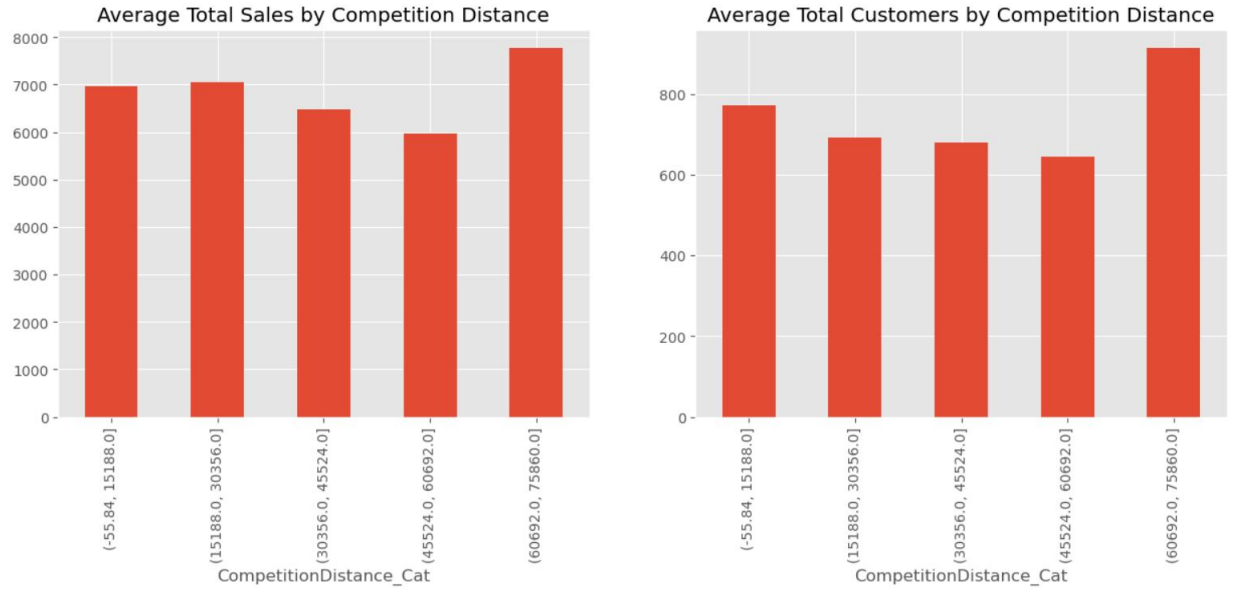
```
In [35]: sns.factorplot(data = train_store_df, x = "Month", y = "Sales", col = "Year", hue = "StoreType")
```



We can observe a seasonality effect, but not trends. The sales stay constant yearly. The graph also illustrates the significant sales for StoreType B, which explain why StoreType B have the highest number on average sales and average customers.

Competition Distance

```
In [37]: f, ax = plt.subplots(1,2, figsize = (15,5))

         train_store_df.groupby(by = "CompetitionDistance_Cat").Sales.mean().plot(kind = "bar", title = "Average Total Sales by Com
         train_store_df.groupby(by = "CompetitionDistance_Cat").Customers.mean().plot(kind = "bar", title = "Average Total Customer
```



Naturally, if the competition is far away, the stores are performing better (sales and customers). But that just explains the last column of the chart. The first four columns are slightly counterintuitive when the farer the competitor, the less sales and customers the store has obtained. This could be interpreted by Game theory. According to the Nash equilibrium, the closer the competition is, the more attractive that location would bring to the customer, raise up the value for the whole area, which increase the value of self and unintended lift the value of your competition. When the distance is too high, the marginal

surplus value for customers drops off due to travelling costs. Therefore, column five is rise up back to highest cause the region has been explicitly owned by the store.

Variable transformation/creation and correlation graph

First, we must convert the variables to categories before we convert them to codes.

In [38]:
```
# train_store_df["Promo"] = train_store_df["Promo"].astype("category") # it's already numerica
# train_store_df["SchoolHoliday"] = train_store_df["SchoolHoliday"].astype("category") # it's already numerica
train_store_df["StoreType"] = train_store_df["StoreType"].astype("category")
train_store_df["Assortment"] = train_store_df["Assortment"].astype("category")
# train_store_df["Promo2"] = train_store_df["Promo2"].astype("category") # it's already numerica
train_store_df["PromoInterval"] = train_store_df["PromoInterval"].astype("category")

train_store_df["StoreType_cat"] = train_store_df["StoreType"].cat.codes
train_store_df["Assortment_cat"] = train_store_df["Assortment"].cat.codes
train_store_df["PromoInterval_cat"] = train_store_df["Assortment"].cat.codes

train_store_df["StateHoliday_cat"] = train_store_df["StateHoliday_cat"].astype("float")
train_store_df["StoreType_cat"] = train_store_df["StoreType_cat"].astype("float")
train_store_df["Assortment_cat"] = train_store_df["Assortment_cat"].astype("float")
train_store_df["PromoInterval_cat"] = train_store_df["PromoInterval_cat"].astype("float")
```

In [39]:
```
train_store_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 844392 entries, 0 to 844391
Data columns (total 26 columns):
 #   Column                     Non-Null Count   Dtype
---  ------                     --------------   -----
 0   Store                      844392 non-null  int64
 1   DayOfWeek                  844392 non-null  int64
 2   Date                       844392 non-null  datetime64[ns]
 3   Sales                      844392 non-null  int64
 4   Customers                  844392 non-null  int64
 5   Open                       844392 non-null  int64
 6   Promo                      844392 non-null  int64
 7   SchoolHoliday              844392 non-null  int64
 8   StateHoliday_cat           844392 non-null  float64
 9   StoreType                  844392 non-null  category
 10  Assortment                 844392 non-null  category
 11  CompetitionDistance        844392 non-null  float64
 12  CompetitionOpenSinceMonth  844392 non-null  float64
 13  CompetitionOpenSinceYear   844392 non-null  float64
 14  Promo2                     844392 non-null  int64
 15  Promo2SinceWeek            844392 non-null  float64
 16  Promo2SinceYear            844392 non-null  float64
 17  PromoInterval              844392 non-null  category
 18  Avg_Customer_Sales         844340 non-null  float64
 19  Month                      844392 non-null  int64
 20  Year                       844392 non-null  int64
 21  Day                        844392 non-null  int64
 22  CompetitionDistance_Cat    844392 non-null  category
 23  StoreType_cat              844392 non-null  float64
 24  Assortment_cat             844392 non-null  float64
 25  PromoInterval_cat          844392 non-null  float64
dtypes: category(4), datetime64[ns](1), float64(10), int64(11)
```

After converting the variables, we could transform the data into correlation graph. In this case we embedded a heatmap to compare all attributes of the dataset.

```
In [40]: df_correlation = train_store_df[["Store", "DayOfWeek", "Sales", "Customers", "Promo", "SchoolHoliday", "Competit
                                         "CompetitionOpenSinceMonth", "CompetitionOpenSinceYear", "Promo2", "Promo2Since
                                         "Month", "Year", "Day", "StateHoliday_cat", "Assortment_cat", "StoreType_cat",


         f, ax = plt.subplots(figsize = (15, 10))
         sns.heatmap(df_correlation.corr(),ax = ax, annot=True, cmap=sns.diverging_palette(10, 133, as_cmap=True), linewi
```

The following correlations we could confirm from the graph above:

- Customer vs Sales (0.82) – the most significant observed correlation.
- Promo vs Sales (0,37).
- Avg_Customer_Sales vs Promo (0,28).
- Avg_Customer_Sales vs Promo2 (0,22).
- StoreType vs Avg_Customer_Sales (0,44).

Conclusion:

After executing an Exploratory Data Analysis on the dataset, we come to some conclusion:

- StoreType A has the most sales and customers.

- StoreType D had the highest buyer shopping cart.
- Promo runs only on weekdays.
- Promo2 doesn’t seem to be correlated to any significant change in the sales amount.
- Customers tend to buy more on Monday, when there's promotion running (Promo) and on Sundays, when there is no promotion at all.
- Sales and customers have a very strong co-relation with a factor of 0.82. StoreType and Average Customer Sales also have a strong correlation at 0.44, which explains the exceptional sales of store type B compared to others. For other variables, it is insignificant.
- Competition does not seem to have any correlations against sales. School and State holiday do not seem to impact sales
- At this point, no distinct yearly trends have been observed. Only seasonal patterns.

5. Is there any special point or potential issue that the analyst must pay attention to?

After executing an Exploratory Data Analysis on the dataset, we come to some discoveries that could be an attention-attract point or potential insight which could lead to further exploration by the analyst:

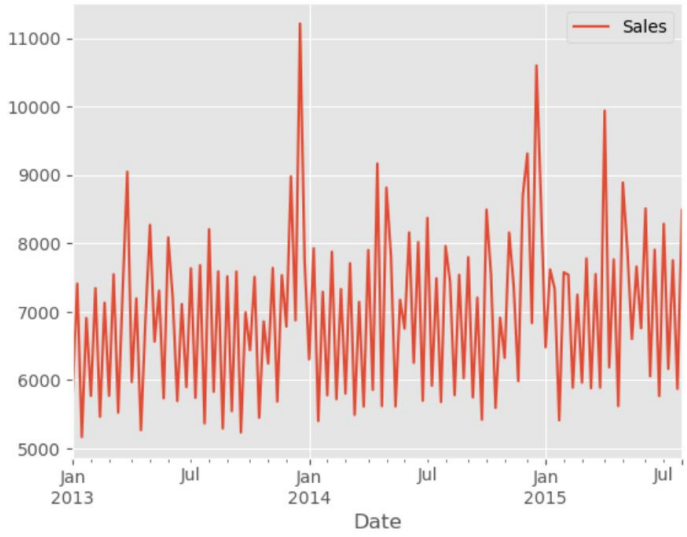
- StoreType B has the lowest Average Spending per Customer and the highest Average Sale per Customer and Customer per Store. So, we suspect customers visit this type only for small things, while the store maintains a substantial traffic of customers. The location and special characteristic of StoreType B could be infer to some interesting insights make way for analysis to dig in, helping business to optimize the procedure for other type of store.
- The number of customers who bought b assortment is extremely low compared to other assortments. Hence, b assortment could be a totally different type of product while a and c might be related to each other. Based on the sales trend of b assortment, there are 2 assumptions considered. First, although having a low number of customers, b assortment could be a product type that could be bought with a large amount. Second, the price of b assortment is much higher than those of a and c. Inferences on the fact that only StoreType B have assortment b display as purchase items. These are interesting insights that can be referred to later in the exploration process.

6. Bonus: perform the model to solve the problem, discuss the result, make conclusions or recommendations (if any), this part must be done by code (either R or python).

ARIMA Forecasting

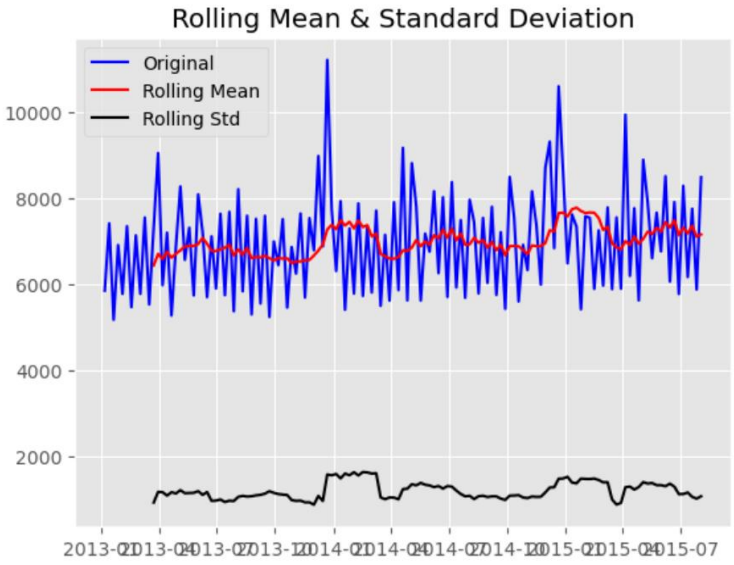
First, we set the index to date and resample it by summing to monthly values

In [41]:
```
ts_arima = train_store_df.set_index("Date").resample("W").mean()
ts_arima = ts_arima[["Sales"]]
ts_arima.plot()
```



Check whether we have stationary or non-stationary time series

In [42]:
```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = timeseries.rolling(window=12).mean()
    rolstd = timeseries.rolling(window=12).std()

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

test_stationarity(ts_arima)
```



The script doesn't perform well, so we cannot check the critical value and test statistic. If the test statistic is greater than the critical value, then we cannot reject the null hypothesis, the series is stationary. That said it is still non-stationary. If we increase the i value in ARIMA model, perhaps the above condition may meet, and we may get the good forecast values.

As we can see from the graph, we are dealing with a seasonal time series data. Therefore the right parameters need to be found for the ARIMA Model as ARIMA(p,d,q)(P,D,Q)s. Here (p,d,q) are the non-seasonal parameter, while (P,D,Q) follow the same definition, but are for the seasonal component of the series. The term "s" is the periodicity (4 for the quarterly periods, 12 for yearly periods).

We will proceed by generating the various combinations of parameters that we wish to assess. Define the p, d and q parameters to take any value between 0 and 2. Generate all different combinations of p, q and q triplets. Generate all different combinations of seasonal p, q and q triplets.

In [43]:
```
import warnings
import itertools
import statsmodels.api as sm
from sklearn.metrics import mean_squared_error
from math import sqrt

# Let's begin by generating the various combination of parameters that we wish to assess:

# Define the p, d and q parameters to take any value between 0 and 2
p = d = q = range(0, 2)

# Generate all different combinations of p, q and q triplets
pdq = list(itertools.product(p, d, q))

# Generate all different combinations of seasonal p, q and q triplets
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]

print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

The return examples of parameter combinations for Seasonal ARIMAX:

- SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
- SARIMAX: (0, 0, 1) x (0, 1, 0, 12)
- SARIMAX: (0, 1, 0) x (0, 1, 1, 12)
- SARIMAX: (0, 1, 0) x (1, 0, 0, 12)

Now we will iterate through some combinations of parameters and use the SARIMAX function to get the AIC Score. The lowest AIC value is the optimal option for our model.

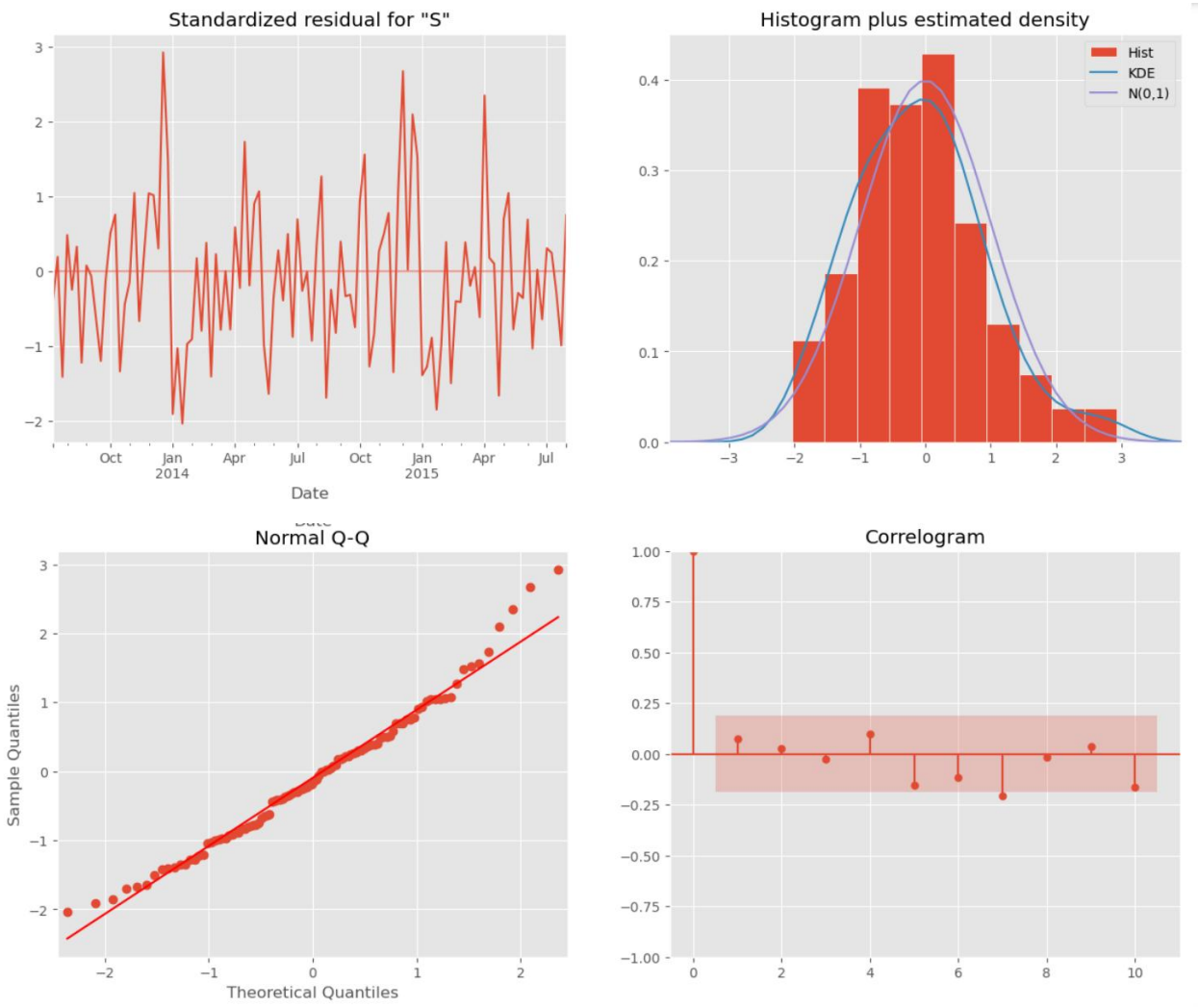
In [44]:
```
warnings.filterwarnings("ignore") # specify to ignore warning messages

for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(ts_arima,
                                            order=param,
                                            seasonal_order=param_seasonal,
                                            enforce_stationarity=False,
                                            enforce_invertibility=False)

            results = mod.fit()

            print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal, results.aic))
        except:
            continue
```

After the function ran, it returned an abundance of AIC Score and the lowest one, which is the optimal parameter for our model: ARIMA(1, 1, 1)x(1, 1, 1, 12)12 - AIC:1847.5087433770632

In [45]:
```
# this is the optimal paramater for our model: ARIMA(1, 1, 1)x(1, 1, 1, 12)12 - AIC:1847.5087433770632

mod = sm.tsa.statespace.SARIMAX(ts_arima,
                                order=(1, 1, 1),
                                seasonal_order=(1, 1, 1, 12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)

results = mod.fit()

print(results.summary().tables[1])
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.5577      0.090     -6.229      0.000      -0.733      -0.382
ma.L1         -0.7300      0.060    -12.221      0.000      -0.847      -0.613
ar.S.L12      -0.1579      0.133     -1.186      0.236      -0.419       0.103
ma.S.L12      -1.0328      0.663     -1.559      0.119      -2.331       0.266
sigma2      1.199e+06   8.76e+05      1.370      0.171   -5.17e+05    2.92e+06
==============================================================================
```

The column “coef” is at attention. This column shows the importance (weight) of each feature and how each one impacts the time series. The P>|z| column informs us of the significance of each feature. Here, each weight has a p-value higher than 0.



Our primary concern is to ensure that the residuals of our model are uncorrelated and normally distributed with zero-mean. If the seasonal ARIMA model does not satisfy these properties, it is a good indication that it can be further improved.

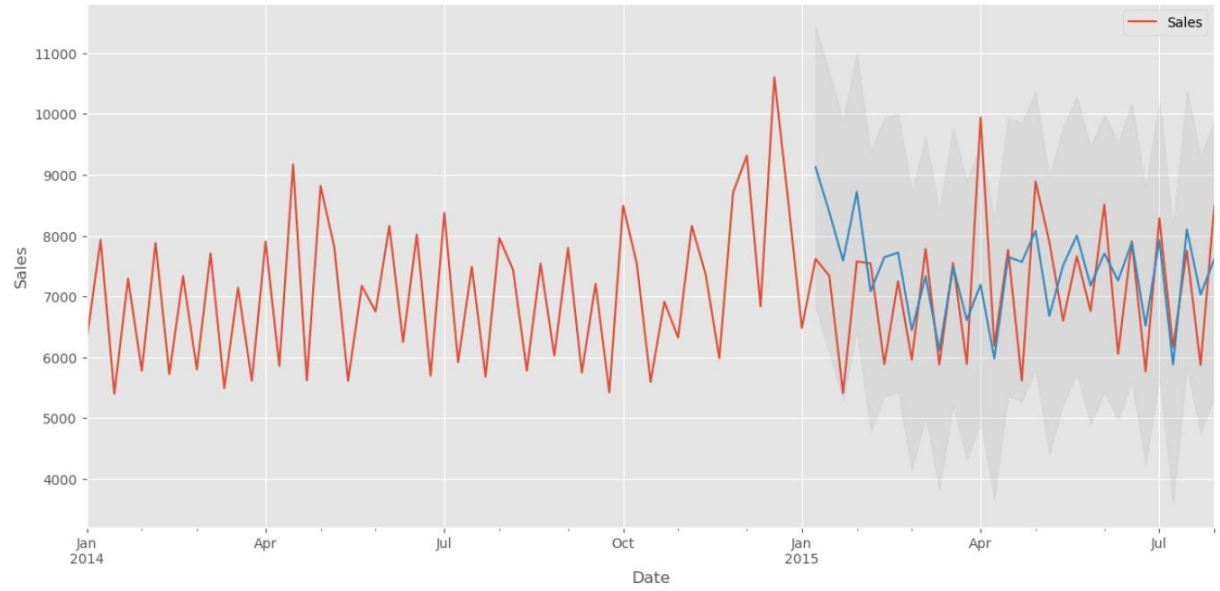
In this case, our model diagnostics suggests that the model residuals are normally distributed based on the following:

- In the top right plot, we see that the red KDE line follows closely with the N(0,1) line (where N(0,1)) is the standard notation for a normal distribution with mean 0 and standard deviation of 1). This is a good indication that the residuals are normally distributed.
- The qq-plot on the bottom left shows that the ordered distribution of residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution with N(0, 1). Again, this is a strong indication that the residuals are normally distributed.
- The residuals over time (top left plot) don't display any obvious seasonality and appear to be white noise. This is confirmed by the autocorrelation (i.e. correlogram) plot on the bottom right, which shows that the time series residuals have low correlation with lagged versions of itself.

In [47]:
```
# Lets go ahead with validating forecasts

pred = results.get_prediction(start = pd.to_datetime("2015-01-11"), dynamic = False)
# lets start the fc to start from 1.11.2015.
# The dynamic=False argument ensures that we produce one-step ahead forecasts, meaning that forecasts at
pred_ci = pred.conf_int() # Get confidence intervals of forecasts

ax = ts_arima["2014":].plot(label = "observed", figsize=(15, 7))
pred.predicted_mean.plot(ax = ax, label = "One-step ahead FC", alpha = 1)
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1],
                color = "k", alpha = 0.05)

ax.set_xlabel("Date")
ax.set_ylabel("Sales")

plt.legend
plt.show()

# -----------
#  extract the predicated and true values of our time series
ts_forecasted = pred.predicted_mean
ts_truth = ts_arima["2015-01-11":]
# to use, in my case, the mean squared error:
rms_arima = sqrt(mean_squared_error(ts_truth, ts_forecasted))
print("RMS:", rms_arima)
```

Let’s go ahead with validating forecasts. We will start the fc to start from 1.11.2015. The dynamic=False argument ensures that we produce one-step ahead forecasts, meaning that forecasts at each point are generated using the full history up to that point. After showing the plot, we will exam it accuracy by comparing it with the data

begin from 2015-01-01 and finally calculate the Root mean square for the final score comparation.

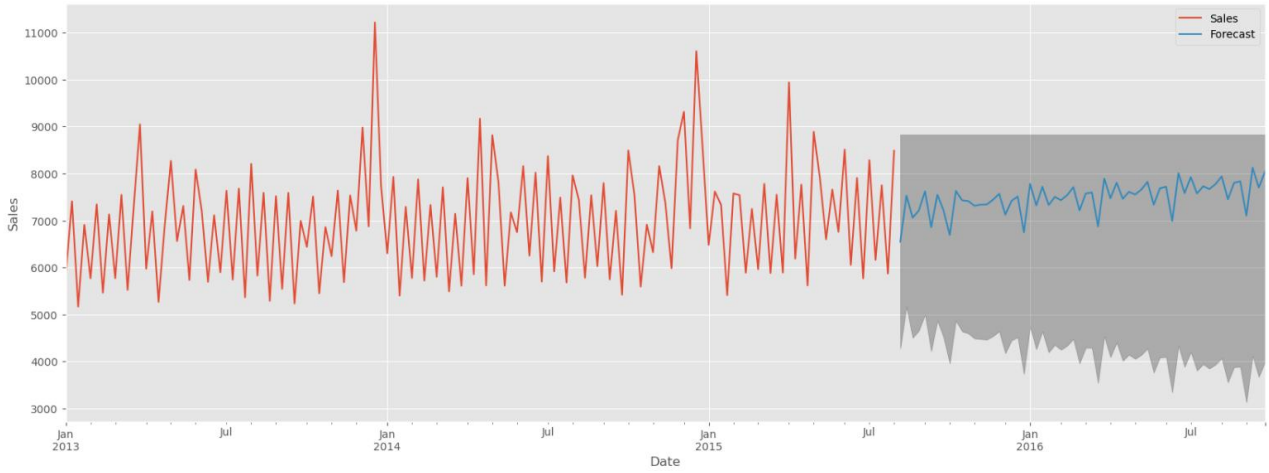



```
RMS: 1060.2381220085788
```

The Root mean square appear to be 1,060.2381220085788

Now, we will forecast for the next few periods using the SARIMAX model that we have trained.

```
In [48]: pred_uc = results.get_forecast(steps = 60) # lets get a forecast for the next few periods
         pred_ci = pred_uc.conf_int() # Get confidence intervals of forecasts

         ax = ts_arima.plot(label = "observed", figsize = (20,7))
         pred_uc.predicted_mean.plot(ax = ax, label = "Forecast")
         ax.fill_between(pred_ci.index,
                         pred_ci.iloc[:, 0],
                         pred_ci.iloc[0, 1], color = "k", alpha = 0.25)
         ax.set_xlabel("Date")
         ax.set_ylabel("Sales")

         plt.legend()
         plt.show()
```



Random Forest Regression

Before we can start with fit and train our model, we need to do some feature engineering. The CompetitionOpenSinceMonth/Year variables have the same meaning. So let's convert them into one variable that we call CompetitionOpenSince. It makes it easier for the algorithm to understand the pattern and creates less branches and thus complex trees.

```
In [49]: train_store_df["CompetitionOpenSince"] = np.where((train_store_df["CompetitionOpenSinceMonth"] == 0) & (tr
                                                          0,(train_store_df.Month - train_store_df.CompetitionOpenSi

         # lets drop the variables
         train_store_df = train_store_df.drop(["CompetitionOpenSinceMonth", "CompetitionOpenSinceYear"], axis = 1)
```

We will drop a few variables, that either or not numeric or we don't need them anymore. A new data frame will be created explicitly for this model.

```
In [50]: # lets drop few variables, that either or not numeric or we dont need them anymore
         # lets create a new data frame for this model
         ts_rfr = train_store_df.copy()
         ts_rfr = train_store_df.drop(["Date","StoreType", "Assortment", "PromoInterval", "CompetitionDistanc
```

Develop the Model

```
In [51]: from sklearn import model_selection
         from sklearn import metrics

         features = ts_rfr.drop(["Customers", "Sales", "Avg_Customer_Sales"], axis = 1)
         target = ts_rfr["Sales"]

         X_train, X_train_test, y_train, y_train_test = model_selection.train_test_split(features, target, test_size =
         # We call here train_test_set which is  divided 80% and 20% validation
         print(X_train.shape, X_train_test.shape, y_train.shape, y_train_test.shape)
```

```
(675513, 17) (168879, 17) (675513,) (168879,)
```

We will call this the train_test_set which is divided into 80% of training and 20% of validation.

Now, we implement RandomForestRegressor. The model will calculate the Root mean square as we are using it to measure the efficiency of models.

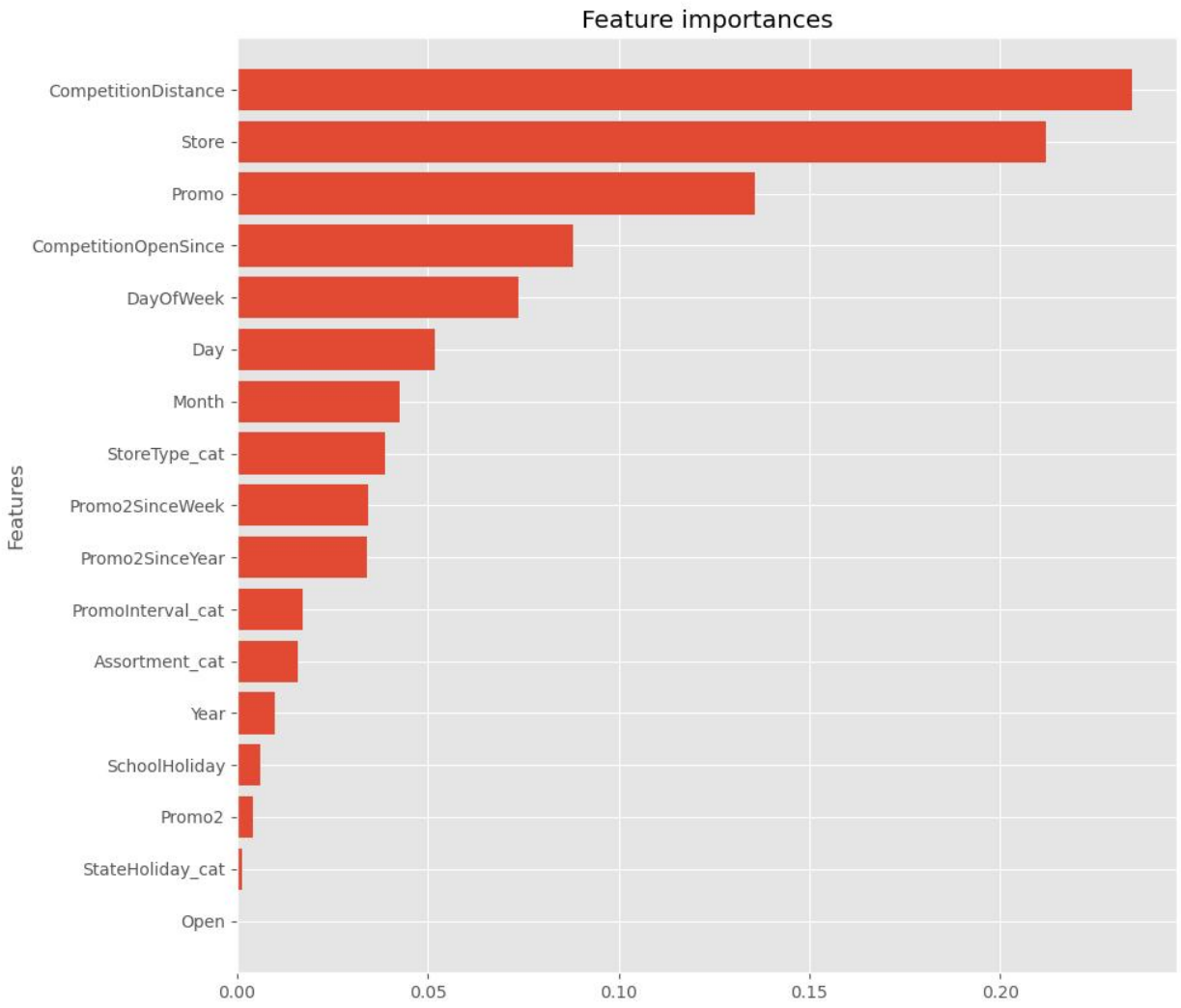
In [52]:
```
from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(n_estimators=10)
rfr.fit(X_train, y_train)
yhat = rfr.predict(X_train_test)
rms_rfr = sqrt(mean_squared_error(y_train_test, yhat))
print("RMS:", rms_rfr)
```

```
RMS: 983.0790972500499
```

The Model return the value of Root mean square is 983.0790972500499

In order to understand better what happened when we ran our RandomForestRegressor, here is a chart that represents, the importance and role that each variable that we decided to include played in this learning process:

In [53]:
```
importances = rfr.feature_importances_
std = np.std([rfr.feature_importances_ for tree in rfr.estimators_],
             axis=0)
indices = np.argsort(importances)
palette1 = itertools.cycle(sns.color_palette())
# Store the feature ranking
features_ranked=[]
for f in range(X_train.shape[1]):
    features_ranked.append(X_train.columns[indices[f]])
# Plot the feature importances of the forest

plt.figure(figsize=(10,10))
plt.title("Feature importances")
plt.barh(range(X_train.shape[1]), importances[indices],
           color=[next(palette1)], align="center")
plt.yticks(range(X_train.shape[1]), features_ranked)
plt.ylabel('Features')
plt.ylim([-1, X_train.shape[1]])
plt.show()
```



XGBoost

First, we have to define two functions: Root mean squared percentage error and Root mean squared percentage error extreme. With two parameters for each function: y and yhat.

In [54]:
```
def rmspe(y, yhat):
    return np.sqrt(np.mean((yhat/y-1) ** 2))

def rmspe_xg(yhat, y):
    y = np.expm1(y.get_label())
    yhat = np.expm1(yhat)
    return "rmspe", rmspe(y,yhat)
```

To use the XGBoost model, we first must install the module xgboost.

In [55]:
```
import sys
!{sys.executable} -m pip install xgboost
```

We define the parameters for the model with the instruction as follows. The maximum depth for a tree here is 10. Second parameter command model to use tree based models. Set the learning rate at 1. Activate the silent mode. Mode for linear regression. The number of boosting rounds is required at 100.

```
In [56]: import xgboost as xgb

param = {'max_depth':10, # maximum depth of a tree
         "booster": "gbtree",   # use tree based models
         'eta':1, # learning rate
         'silent':1, # silent mode
         'objective':'reg:linear', # for linear regression
        }

num_round = 100 #how many boosting rounds

dtrain = xgb.DMatrix(X_train, y_train)
dtest = xgb.DMatrix(X_train_test, y_train_test)
watchlist = [(dtrain, 'train'), (dtest, 'eval')]

xgboost = xgb.train(param, dtrain, num_round, evals=watchlist, \
  early_stopping_rounds= 100, feval=rmspe_xg, verbose_eval=True)

# make prediction
preds = xgboost.predict(dtest)
```

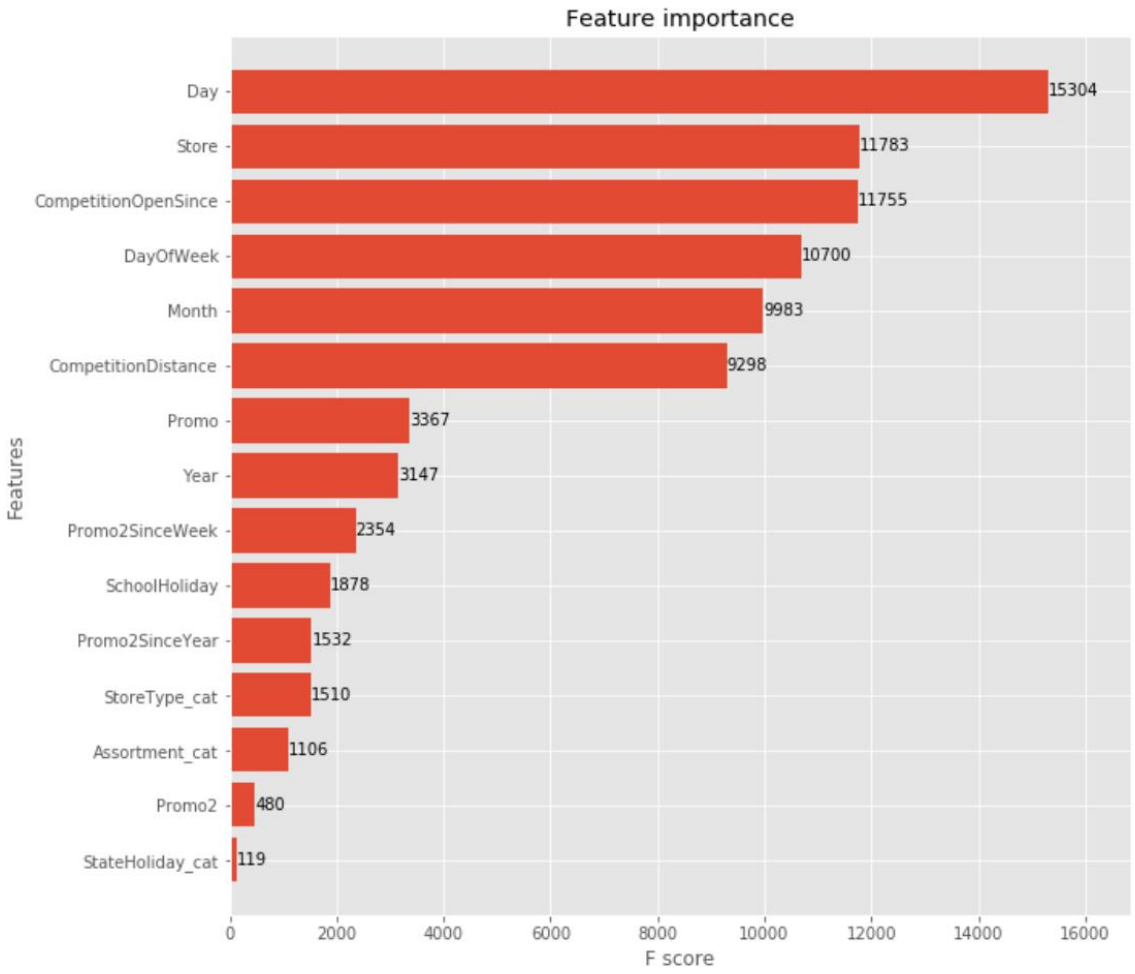
After running for a while, we got the result back. With the optimize prediction the model give us: train-rmse:578.39888, train-rmspe:nan eval-rmse:841.92073.

```
[81]   train-rmse:611.81273   train-rmspe:nan eval-rmse:846.55334   eval-rmspe:nan
[82]   train-rmse:609.85728   train-rmspe:nan eval-rmse:845.94311   eval-rmspe:nan
[83]   train-rmse:608.09155   train-rmspe:nan eval-rmse:845.27606   eval-rmspe:nan
[84]   train-rmse:606.25869   train-rmspe:nan eval-rmse:845.24424   eval-rmspe:nan
[85]   train-rmse:603.74840   train-rmspe:nan eval-rmse:844.20730   eval-rmspe:nan
[86]   train-rmse:601.84006   train-rmspe:nan eval-rmse:844.03770   eval-rmspe:nan
[87]   train-rmse:600.16195   train-rmspe:nan eval-rmse:843.63750   eval-rmspe:nan
[88]   train-rmse:598.36481   train-rmspe:nan eval-rmse:843.81607   eval-rmspe:nan
[89]   train-rmse:597.35696   train-rmspe:nan eval-rmse:843.66111   eval-rmspe:nan
[90]   train-rmse:595.68247   train-rmspe:nan eval-rmse:843.01566   eval-rmspe:nan
[91]   train-rmse:593.18431   train-rmspe:nan eval-rmse:842.65250   eval-rmspe:nan
[92]   train-rmse:590.25907   train-rmspe:nan eval-rmse:842.21020   eval-rmspe:nan
[93]   train-rmse:588.55760   train-rmspe:nan eval-rmse:842.42477   eval-rmspe:nan
[94]   train-rmse:587.32625   train-rmspe:nan eval-rmse:842.41873   eval-rmspe:nan
[95]   train-rmse:585.49250   train-rmspe:nan eval-rmse:842.10242   eval-rmspe:nan
[96]   train-rmse:583.21185   train-rmspe:nan eval-rmse:841.68443   eval-rmspe:nan
[97]   train-rmse:581.76636   train-rmspe:nan eval-rmse:842.10414   eval-rmspe:nan
[98]   train-rmse:580.38415   train-rmspe:nan eval-rmse:841.82416   eval-rmspe:nan
[99]   train-rmse:578.39888   train-rmspe:nan eval-rmse:841.92073   eval-rmspe:nan
```

```
In [57]: rms_xgboost = sqrt(mean_squared_error(y_train_test, preds))
         print("RMS:", rms_xgboost)

         RMS: 841.9207317999857
```

The Root mean square for XGBoost model is 841.9207317999857.

Now, let's have a look at the feature importance of the model.



As we are going through applying all different types of models to test the performance on our dataset, we have been using the Root mean square score as the standard for the efficiency of the model. The lower the score, the more accurate the model performance. Now, let's us see the summarize table score of all the models:

	Model	Score
2	XGBoost	841.920732
1	Random Forest Regression	983.079097
0	SARIMAX	1060.238122

The XGBoost Model has come out on top with an significant different of 141.158365 point compared to the last model. Followed by Random Forest Regression and SARIMAX.

As mentioned before, XGBoost is a faster algorithm when compared to other algorithms because of its parallel and distributed computing. XGBoost is developed with both deep considerations in terms of systems optimization and principles in machine learning. The goal of this library is to push the extreme of the computation limits of machines to provide a scalable, portable, and accurate library. Nowadays XGBoost is dominate structured or tabular datasets on classification and regression predictive modelling problems, which bring no surprise when it is obtain the best performance applying into the Time series of Rossman Store Sales dataset.