Future sales prediction

Introduction

**1. Feature Engineering:** Feature engineering is a crucial step in building predictive models. It involves selecting, creating, or transforming features that are most relevant to your prediction task. For IMDb score prediction, you can consider features like the director's past work, lead actors' popularity, genre, movie budget, and more.

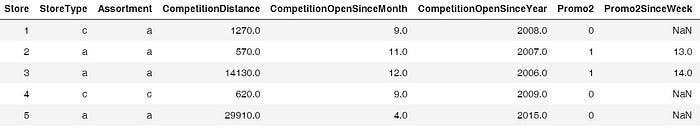
This post is divided into two parts: EDA & Forecasting.

Program and outputs:

# **Part A) Exploratory Data Analysis (EDA)**

Let’s start by first importing the required libraries followed by data exploration.

# Importing required libraries  
import numpy as np  
import pandas as pd, datetime  
import seaborn as sns  
from statsmodels.tsa.stattools import adfuller  
import matplotlib.pyplot as plt  
get\_ipython().run\_line\_magic('matplotlib', 'inline')  
from time import time  
import os  
from math import sqrt  
from statsmodels.tsa.seasonal import seasonal\_decompose  
from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf  
import itertools  
import statsmodels.api as sm  
from statsmodels.tsa.stattools import acf,pacf  
from statsmodels.tsa.arima\_model import ARIMA  
from sklearn import model\_selection  
from sklearn.metrics import mean\_squared\_error, r2\_score  
from pandas import DataFrame  
import xgboost as xgb  
from fbprophet import Prophet  
import warnings  
warnings.filterwarnings('ignore')# Importing store data  
store = pd.read\_csv('./data/store.csv')  
store.head()



The above table gives us information about 1115 stores owned.  
train = pd.read\_csv('./data/train.csv', index\_col='Date', parse\_dates = True)  
train.head()

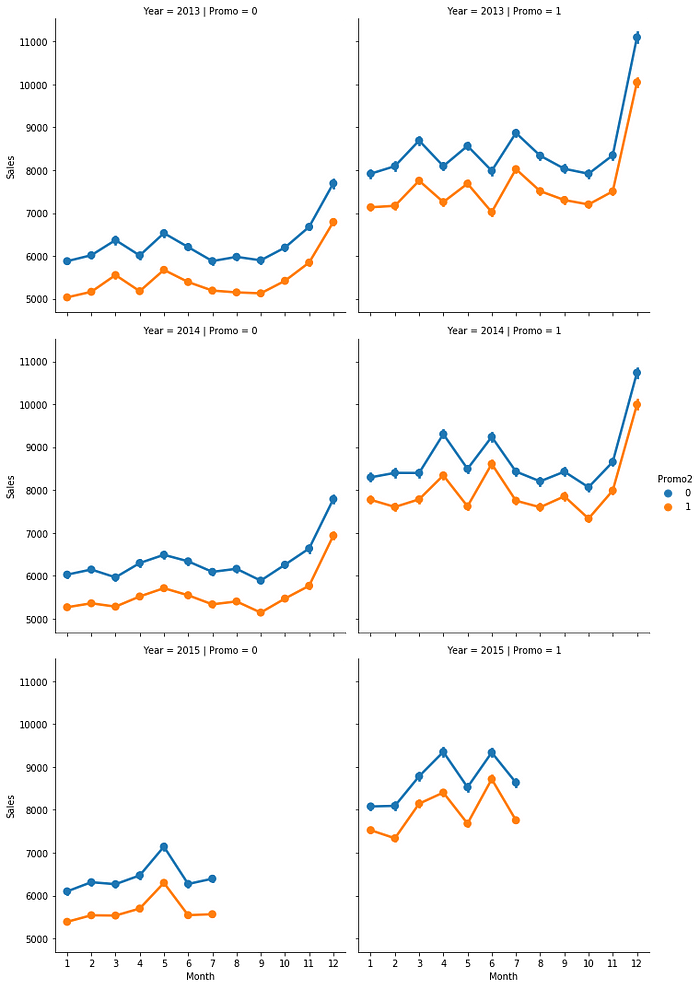


We can see from the above table that the date is one of the columns. this is a time-series data.

1. **Trends & Seasonality**

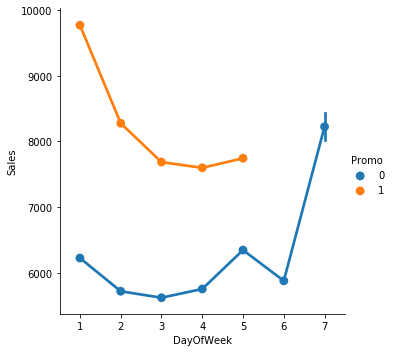
Let’s see how the sales vary with month, promo, promo2 and year.

# Sales trend over the months and year  
sns.factorplot(data = train\_store\_joined, x ="Month", y = "Sales",   
 col = 'Promo', # per store type in cols  
 hue = 'Promo2',  
 row = "Year")



The above graph tells us that sales tend to spike in December, which makes sense because of the Christmas and holiday season. So, this confirms that the sales vary with the ‘Date’ (time) and there is a seasonality factor present in our data.

# Sales trend over days  
sns.factorplot(data = train\_store\_joined, x = "DayOfWeek", y = "Sales", hue = "Promo")



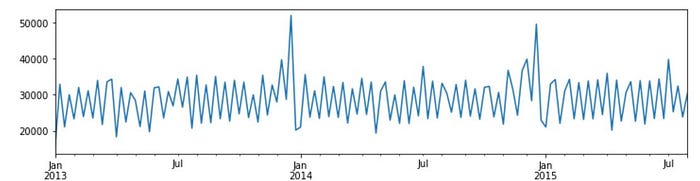
The sales tend to increase on Sunday because people shop during the weekend.

**2. Stationarity of Time Series**

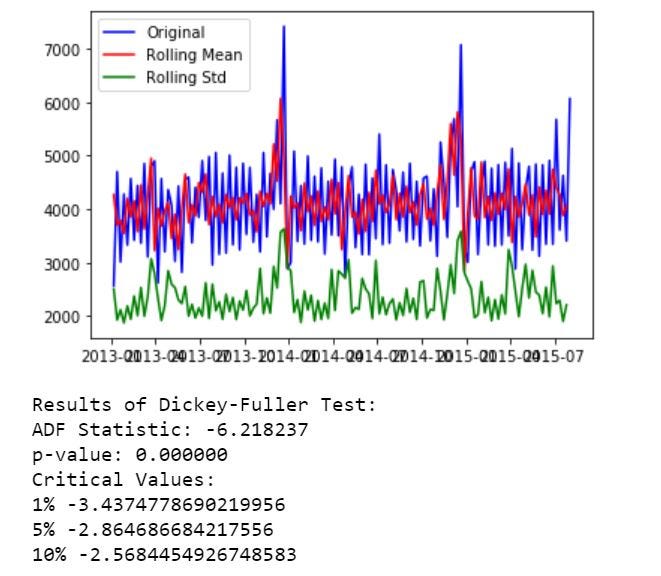
In order to use time series forecasting models, we need to ensure that our time series data is stationary.

Let’s check the stationarity of a store of type ‘a’.

# Data Preparation: input should be float type  
train['Sales'] = train['Sales'] \* 1.0# Assigning one store from each category  
sales\_a = train[train.Store == 2]['Sales']# Trend  
sales\_a.resample('W').sum().plot(ax = ax1)

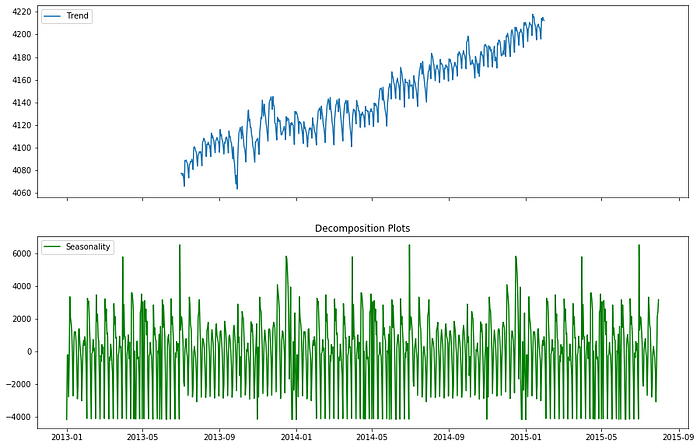


The above graph tells us that sales tend to peak at the end of the year.  
 # Determing rolling statistics  
 roll\_mean = timeseries.rolling(window=7).mean()  
 roll\_std = timeseries.rolling(window=7).std()# Plotting rolling statistics:  
 orig = plt.plot(timeseries.resample('W').mean(), color='blue',label='Original')  
 mean = plt.plot(roll\_mean.resample('W').mean(), color='red', label='Rolling Mean')  
 std = plt.plot(roll\_std.resample('W').mean(), color='green', label = 'Rolling Std')  
 plt.legend(loc='best')  
 plt.show(block=False)  
   
 # Performing Dickey-Fuller test:  
 print('Results of Dickey-Fuller Test:')  
 result = adfuller(timeseries, autolag='AIC')  
 print('ADF Statistic: %f' % result[0])  
 print('p-value: %f' % result[1])  
 print('Critical Values:')  
 for key, value in result[4].items():  
 print(key, value)# Testing stationarity of store type a  
test\_stationarity(sales\_a)



Now, let’s see the seasonality and trend using decomposition plots.

# Plotting seasonality and trend  
def plot\_timeseries(sales,StoreType):fig, axes = plt.subplots(2, 1, sharex=True, sharey=False)  
 fig.set\_figheight(10)  
 fig.set\_figwidth(15)decomposition= seasonal\_decompose(sales, model = 'additive',freq=365)estimated\_trend = decomposition.trend  
 estimated\_seasonal = decomposition.seasonal  
 estimated\_residual = decomposition.resid  
   
 axes[1].plot(estimated\_seasonal, 'g', label='Seasonality')  
 axes[1].legend(loc='upper left');  
   
 axes[0].plot(estimated\_trend, label='Trend')  
 axes[0].legend(loc='upper left');plt.title('Decomposition Plots')



From the above plots, we can see that there are seasonality and trend present in our data. So, we’ll use the forecasting models that take both of these factors into consideration.

# **Part B) Forecasting: Predictive Modelling**

**Evaluation Metrics**

There are two popular metrics used in measuring the performance of regression models i.e. MAE & RMSE.

**Mean Absolute Error (MAE)**: It is the average of the absolute difference between the predicted values and observed values.

**Root Mean Square Error (RMSE)**: It is the square root of the average of squared differences between the predicted values and observed values.

**Predictive Modelling**

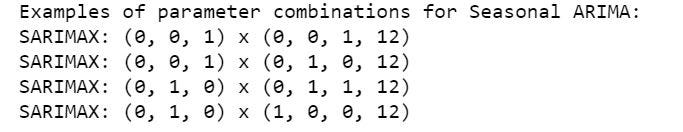
**Model 1: Autoregressive Integrated Moving Average (ARIMA)**

We will use one of the most commonly used methods for time-series forecasting, known as ARIMA.

ARIMA models are denoted by ARIMA(p, d, q).

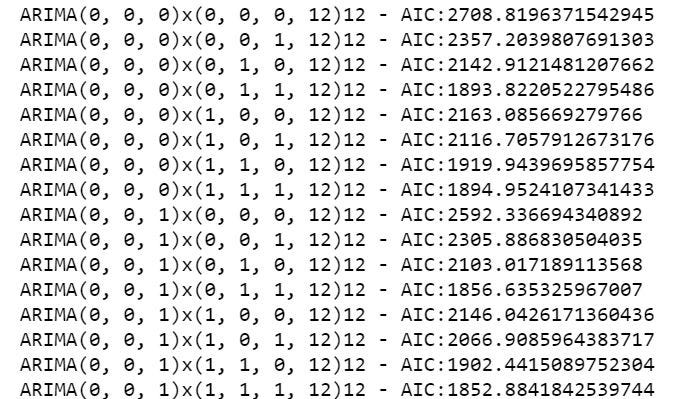
p, d, and q represent seasonality, trend, and noise in data respectively. We’ll first create all possible combinations of p, d, and q as follows:

# Define the p, d and q parameters to take any value between 0 and 3  
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]))



**Hyperparameter tuning for ARIMA**

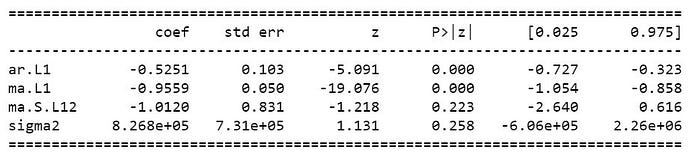
# Determing p,d,q combinations with AIC scores.  
for param in pdq:  
 for param\_seasonal in seasonal\_pdq:  
 try:  
 mod = sm.tsa.statespace.SARIMAX(train\_arima,  
 order=param,  
 seasonal\_order=param\_seasonal,  
 enforce\_stationarity=False,  
 enforce\_invertibility=False)results = mod.fit()print('ARIMA{}x{}12 - AIC:{}'.format(param, param\_seasonal, results.aic))  
 except:  
 continue



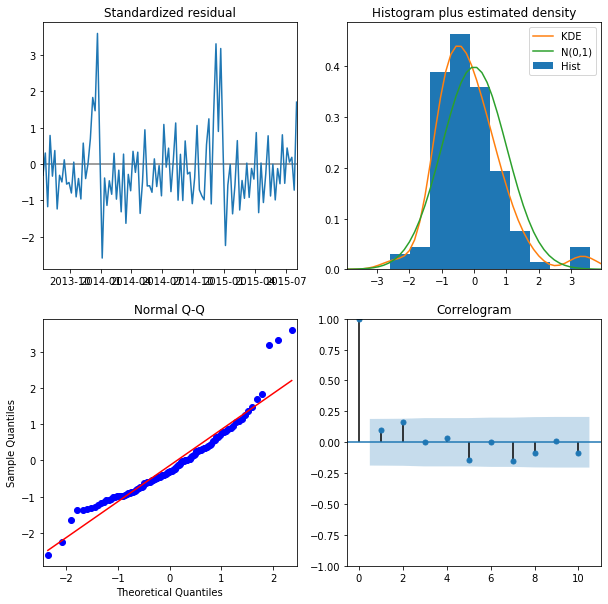
I’ve only included a snapshot of how the grid search looks. The above iteration suggested that **SARIMAX(1, 1, 1)x(0, 1, 1, 12)12** is the best parameter combination with the lowest **AIC: 1806.29.**

**Fitting the ARIMA model**

# Fitting the data to ARIMA model   
model\_sarima = sm.tsa.statespace.SARIMAX(train\_arima,  
 order=(1, 1, 1),  
 seasonal\_order=(0, 1, 1, 12),  
 enforce\_stationarity=False,  
 enforce\_invertibility=False)results\_sarima = model\_sarima.fit()print(results\_sarima.summary().tables[1])



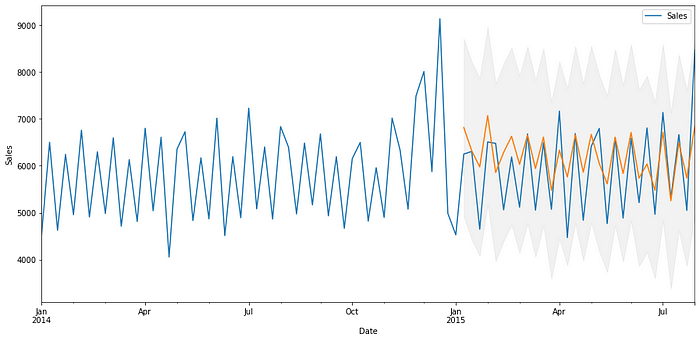
Let’s check diagnostic plots to visualize the performance of our model.# Checking diagnostic plotsresults\_sarima.plot\_diagnostics(figsize=(10, 10))plt.show()



The Normal Q-Q plot shows that the ordered distribution of residuals follows the distribution similar to normal distribution. Thus, our model seems to be pretty good.

**Model Prediction**

# Model Prediction  
# Predictions are performed for the 11th Jan' 2015 onwards of the train data.pred = results\_sarima.get\_prediction(start=pd.to\_datetime('2015-01-11'), dynamic = False)# Get confidence intervals of forecasts  
pred\_ci = pred.conf\_int()ax = train\_arima["2014":].plot(label = "observed", figsize=(15, 7))  
pred.predicted\_mean.plot(ax = ax, label = "One-step ahead Forecast", 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()train\_arima\_forecasted = pred.predicted\_mean  
train\_arima\_truth = train\_arima["2015-01-11":]# Calculating the error  
rms\_arima = sqrt(mean\_squared\_error(train\_arima\_truth, train\_arima\_forecasted))  
print("Root Mean Squared Error: ", rms\_arima)





The above plot shows that our predicted values catch up to the observed values in the dataset. Our forecasts seem to align with the ground truth very well and show a spike in December as expected. RMSE is also reasonably low in our case.

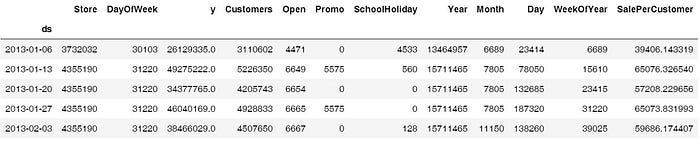
So, final ARIMA model can be represented as **SARIMAX(1, 1, 1)x(0, 1, 1, 12)12.** This is the best we can do with ARIMA, so let’s try another model to see whether we can decrease the RMSE.

**Model 2: Prophet**

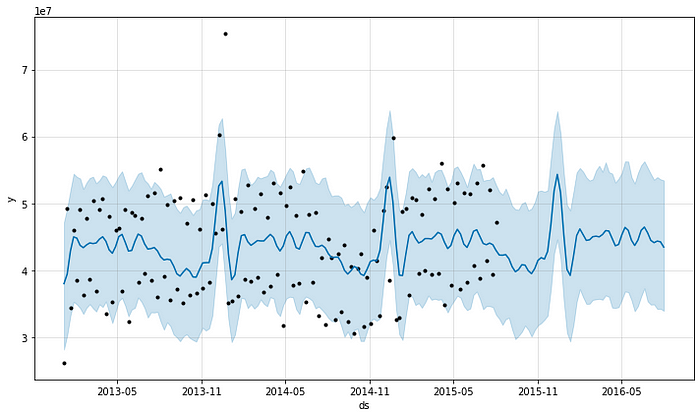
Prophet is an open-source tool by Facebook. This procedure is used for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects.

**Baseline model**

Our baseline (initial) model will use the default parameters. Let’s see how it performs.names to specific names as required by Prophet library  
train\_prophet = train\_prophet.rename(columns = {'Date': 'ds',  
 'Sales': 'y'})# Downsampling to week because modelling on daily basis takes a lot of time  
ts\_week\_prophet = train\_prophet.set\_index("ds").resample("W").sum()



**Visualizing predictions forecast**  
prophet.plot(forecast\_1);

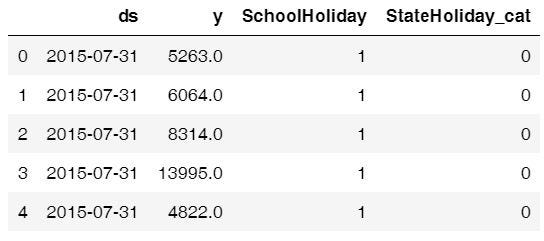


We can see from the above plot that the predictions are decent enough but let’s look at the RMSE to get a better idea.

# Checking the RMSE of Prophet model  
metric\_prophet\_1 = forecast\_1.set\_index('ds')[['yhat']].join(ts\_week\_prophet\_train.set\_index('ds').y).reset\_index()  
metric\_prophet\_1.dropna(inplace=True)  
rms\_prophet\_1 = mean\_squared\_error(metric\_prophet\_1.y, metric\_prophet\_1.yhat)  
rms\_prophet\_1



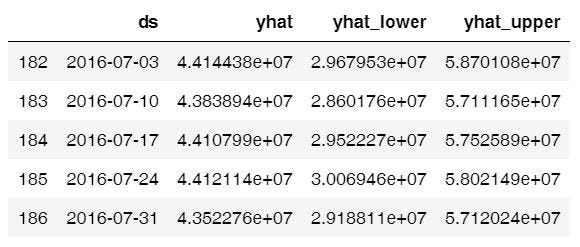
# Encoding state holiday categorical variable  
train\_prophet["StateHoliday\_cat"] = train\_prophet["StateHoliday"].map({0:0, "0": 0, "a": 1, "b": 1, "c": 1})# Choosing only required cols  
train\_prophet = train\_prophet[['ds', 'y', 'SchoolHoliday', 'StateHoliday\_cat']]  
train\_prophet.head()



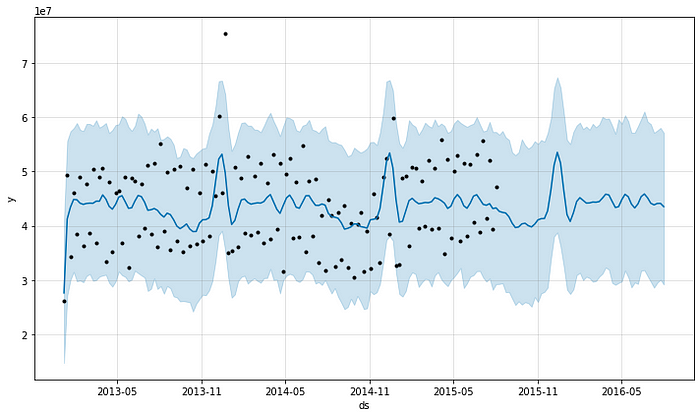
# Modelling holidays - creating holidays dataframe  
state\_dates = train\_prophet[(train\_prophet.StateHoliday\_cat == 1)].loc[:, "ds"].values  
school\_dates = train\_prophet[(train\_prophet.SchoolHoliday == 1)].loc[:, "ds"].valuesstate = pd.DataFrame({"holiday": "state\_holiday", "ds": pd.to\_datetime(state\_dates)})  
school = pd.DataFrame({"holiday": "school\_holiday", "ds": pd.to\_datetime(school\_dates)})holidays = pd.concat((state, school))  
holidays.head()# Dropping holiday columns because not needed any more  
train\_prophet\_clean = train\_prophet.drop(["SchoolHoliday", "StateHoliday\_cat"], axis = 1)# Downsampling to week because modelling on daily basis takes a lot of time  
ts\_week\_prophet = train\_prophet\_clean.set\_index("ds").resample("W").sum()# Resetting the index  
ts\_week\_prophet\_train = ts\_week\_prophet.reset\_index()

**Model Prediction**

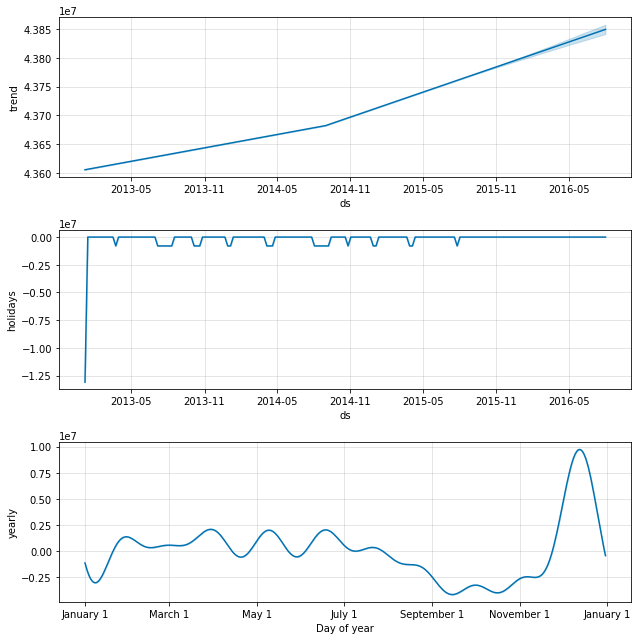
future\_2 = prophet\_2.make\_future\_dataframe(periods = 52, freq = "W")   
forecast\_2 = prophet\_2.predict(future)  
forecast\_2[["ds", "yhat", "yhat\_lower", "yhat\_upper"]].tail() # We have a new dataframe, which includes, the forecast and the uncertainity invervals.



# Visualizing predicions of forecast  
prophet.plot(forecast\_2);



# Visualizing trend and seasonality components  
prophet.plot\_components(forecast\_2);



The first plot shows that the total sales on a weekly basis are increasing. The second plot shows the holiday gaps in the dataset and the third plot shows that the store sees very high sales in the last week of December (because of the Christmas holidays).

# Checking the RMSE of Prophet model  
metric\_prophet\_2 = forecast\_2.set\_index('ds')[['yhat']].join(ts\_week\_prophet\_train.set\_index('ds').y).reset\_index()  
metric\_prophet\_2.dropna(inplace=True)  
rms\_prophet\_2 = mean\_squared\_error(metric\_prophet\_2.y, metric\_prophet\_2.yhat)  
rms\_prophet\_2

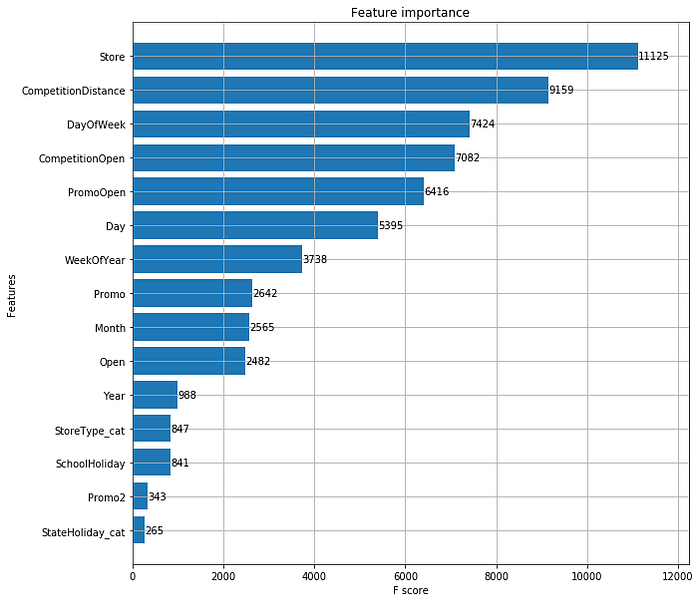


**Model 3: XGBoost**

XGBoost is an optimized distributed gradient boosting library designed to be highly **efficient**, **flexible** and **portable**. Although it is not specifically designed for time-series data, it is known to perform extremely well in all kinds of regression problems.

# Dropping Customers and Sale per customer  
ts\_xgboost = train\_store\_joined.copy()  
ts\_xgboost = ts\_xgboost.drop(['Customers', 'SalePerCustomer', 'PromoInterval'], axis=1)# Combining similar columns into one column and dropping old columns  
ts\_xgboost['CompetitionOpen'] = 12 \* (ts\_xgboost.Year - ts\_xgboost.CompetitionOpenSinceYear) + (ts\_xgboost.Month - ts\_xgboost.CompetitionOpenSinceMonth)  
ts\_xgboost['PromoOpen'] = 12 \* (ts\_xgboost.Year - ts\_xgboost.Promo2SinceYear) + (ts\_xgboost.WeekOfYear - ts\_xgboost.Promo2SinceWeek) / 4.0  
ts\_xgboost = ts\_xgboost.drop(["CompetitionOpenSinceMonth", "CompetitionOpenSinceYear"], axis = 1)  
ts\_xgboost = ts\_xgboost.drop(["Promo2SinceWeek", "Promo2SinceYear"], axis = 1)# Converting categorical cols to numerical cols and removing old cols  
mappings = {0:0, "0": 0, "a": 1, "b": 1, "c": 1}  
ts\_xgboost["StateHoliday\_cat"] = ts\_xgboost["StateHoliday"].map(mappings)  
ts\_xgboost["StoreType\_cat"] = ts\_xgboost["StoreType"].map(mappings)  
ts\_xgboost["Assortment\_cat"] = ts\_xgboost["Assortment"].map(mappings)  
ts\_xgboost = ts\_xgboost.drop(["StateHoliday", "StoreType", "Assortment"], axis = 1)# Splitting the data  
features = ts\_xgboost.drop(["Sales"], axis = 1)  
target = ts\_xgboost["Sales"]X\_train, X\_test, y\_train, y\_test = model\_selection.train\_test\_split(features, target, test\_size = 0.

Let’s see what features impact the sales of a store.



As expected, there are five major reasons affecting the sales of a store viz. the **number of stores, competition distance, day of the week, is the competition open, and promotions**.

Our final XGBoost model after hyper tuning is the one with **‘max\_depth’:10, ‘eta’:0.1, ‘gamma’: 2 and RMSE score of 1191.90**, which is great! Now, let's compare the performance of all models

**3. Results**

We used the Root Mean Squared Error (RMSE) to evaluate and validate the performance of various models. Let’s see which model performed better and why/why not.

# Comparing performance of above three models - through RMSE  
rms\_arima = format(float(rms\_arima))  
rms\_prophet\_2 = format(float(rms\_prophet\_2))  
rms\_xgboost\_2 = format(float(rms\_xgboost\_2))model\_errors = pd.DataFrame({  
 "Model": ["SARIMA", "Prophet", "XGBoost"],  
 "RMSE": [rms\_arima, rms\_prophet\_2, rms\_xgboost\_2]  
})model\_errors.sort\_values(by = "RMSE")

**5. Conclusion**

* The most interesting thing about the data was that the category of stores having the highest sales didn’t have the highest sale per customer. It might be because those stores sell small items, which are needed on a daily basis.
* Another interesting thing is probably because customers already purchased whatever they wanted during the first promotional sale.