Time Forecasting Models for Arion

Introduction:

Boaz Allen Harris was invited by Arion, an international retailer, to conduct an analysis of sales in their furniture division. Furniture sales take significant resources to manufacture and warehouse. Increasing the accuracy in forecasting future demand would have a significant improvement in Arion's bottom line.

Business Understanding:

Arion is concerned about the underlying trend and seasonality in their furniture sales division. The ability to project future sales will help them bring efficiencies to their manufacturing schedule and empower their marketing/sales team. Since some of their furniture lines are manufactured domestically, Arion has to manage the following challenges:

- Skilled labor shortages – near full employment is creating issues
- Increasing costs – many states are phasing in increases in the minimum wage
- Fluctuating customer demand – furniture sales are highly correlated to new home sales and new business creation. Any downturn in the economy slows furniture sales.
- Supply chain issues – disruptions in material availability, trade wars, etc.

Boaz Allen undertook a detailed time series analysis to help Arion understand their past sales and give projections of future demand. Good forecasting will ensure that Arion can supply the right product at the right time and location, minimizing out-of-stock issues and reducing inventory that is unsaleable. We are providing a traditional forecast methodology, but we do recommend that judgmental forecasting from Arion's existing team or consultants from Boaz Allen also used as an adjunct to our initial observations.

Data Understanding:

Arion supplied us with a dataset of all their retail activity from 2014-2017. Our data science team is working in both Python and R to conduct the time series analysis. We will share the Python coding first and follow with the R code.

Data Exploration/Preparation:

We imported the following Python libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
%matplotlib inline
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 10, 6
import warnings
import itertools
warnings.filterwarnings("ignore")
plt.style.use('fivethirtyeight')
import statsmodels.api as sm
import matplotlib

matplotlib.rcParams['axes.labelsize'] = 14
matplotlib.rcParams['xtick.labelsize'] = 12
matplotlib.rcParams['ytick.labelsize'] = 12
matplotlib.rcParams['text.color'] = 'k'
```

Here's our first look at Arion's dataset.

```
1 dataset.head()
```

	Row ID	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	State	Postal Code	Region	Product ID	Category	Sub-Category	
0	1	CA-2016-152156	2016-11-08	2016-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	Kentucky	42420	South	FUR-BO-10001798	Furniture	Bookcases	S C B
1	2	CA-2016-152156	2016-11-08	2016-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	Kentucky	42420	South	FUR-CH-10000454	Furniture	Chairs	Hoı Uph
2	3	CA-2016-138688	2016-06-12	2016-06-16	Second Class	DV-13045	Darrin Van Huff	Corporate	United States	Los Angeles	California	90036	West	OFF-LA-10000240	Office Supplies	Labels	A L Typ
3	4	US-2015-108966	2015-10-11	2015-10-18	Standard Class	SO-20335	Sean O'Donnell	Consumer	United States	Fort Lauderdale	Florida	33311	South	FUR-TA-10000577	Furniture	Tables	Se Rec
4	5	US-2015-108966	2015-10-11	2015-10-18	Standard Class	SO-20335	Sean O'Donnell	Consumer	United States	Fort Lauderdale	Florida	33311	South	OFF-ST-10000760	Office Supplies	Storage	El 'N

Since we are conducting research on only the furniture sales, we had to prune the dataset.

We chose the Order Date, Category and Sales columns.

```
1 dataset = dataset[["Order Date","Category","Sales"]]
```

```
1 dataset.head()
```

	Order Date	Category	Sales
0	2016-11-08	Furniture	261.9600
1	2016-11-08	Furniture	731.9400
2	2016-06-12	Office Supplies	14.6200
3	2015-10-11	Furniture	957.5775
4	2015-10-11	Office Supplies	22.3680

Further data cleaning in the Category column was necessary to select only the "Furniture" sales.

```
1 dataset = dataset[dataset['Category'] == 'Furniture']
2 dataset.head()
3 # use.loc or.iloc when you have to change the value o
```

	Order Date	Category	Sales
0	2016-11-08	Furniture	261.9600
1	2016-11-08	Furniture	731.9400
3	2015-10-11	Furniture	957.5775
5	2014-06-09	Furniture	48.8600
10	2014-06-09	Furniture	1706.1840

Converted "Order Date" to string and date time object.

```
dataset['Order Date'] = dataset['Order Date'].apply(lambda x:x.date().strftime('%y-%m'))
dataset['Order Date']= pd.to_datetime(dataset['Order Date'],format='%y-%m')
dataset
```

	Order Date	Sales
7474	2014-01-01	2573.8200
7660	2014-01-01	76.7280
866	2014-01-01	51.9400
716	2014-01-01	9.9400
2978	2014-01-01	545.9400

Finally, we summed each day to a monthly figure.

```
dataset = dataset.groupby('Order Date')['Sales'].sum().reset_index()
dataset.head()
```

	Order Date	Sales
0	2014-01-01	6242.525
1	2014-02-01	1839.658
2	2014-03-01	14573.956
3	2014-04-01	7944.837
4	2014-05-01	6912.787

We ended up with a dataset with 48 rows and 2 columns beginning with January 2014 and ending December 2017.

```
dataset.shape
```

```
(48, 2)
```

```
dataset['Order Date'].min()
```

```
Timestamp('2014-01-01 00:00:00')
```

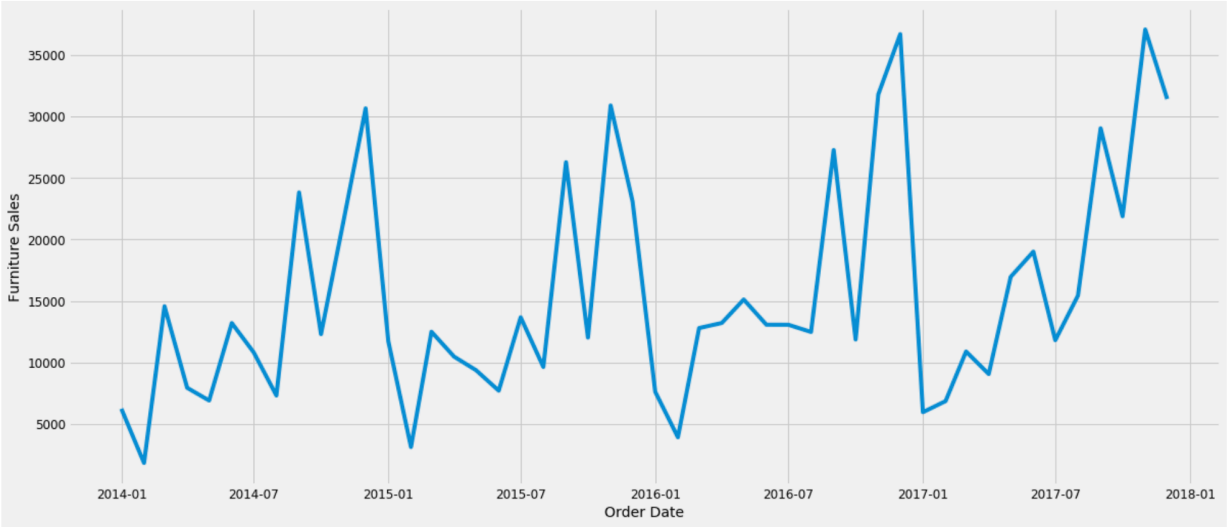
```
dataset['Order Date'].max()
```

```
Timestamp('2017-12-01 00:00:00')
```

Our initial plot of the time series:

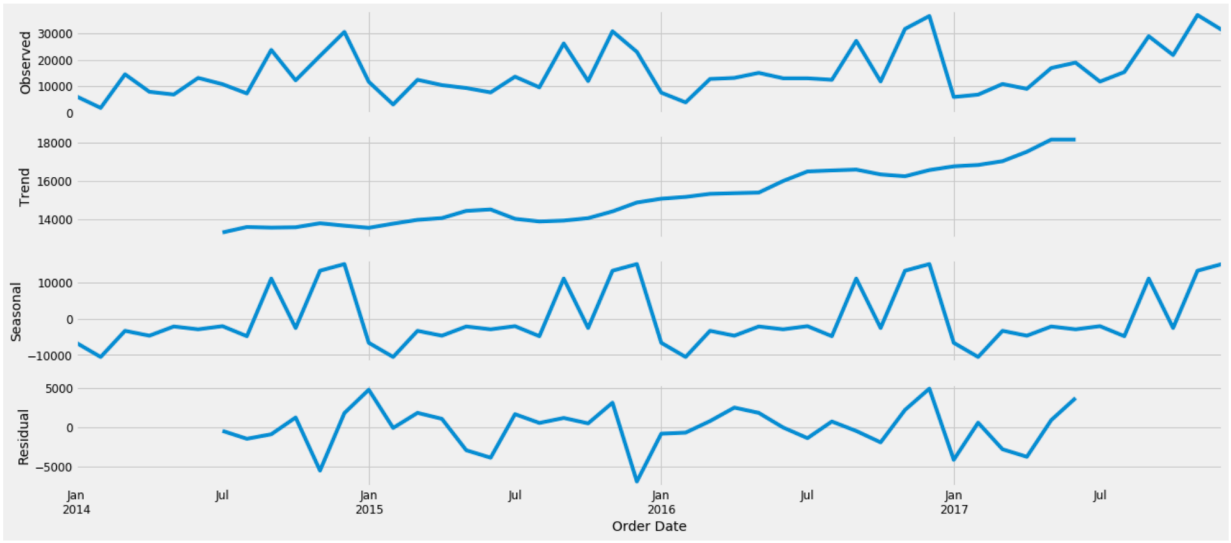
```
plt.xlabel("Order Date")
plt.ylabel("Furniture Sales")
plt.plot(indexeddataset)
```

```
[<matplotlib.lines.Line2D at 0x7ff894c86400>]
```



There is clear seasonality to this time series and also a general upward trend line. The peaks get higher over time as do the valleys. There is a steep decline in sales in the early months of the year, relatively slow growth over the summer months and an increase in the fall through December.
We also charted a seasonal decomposition with an additive model:

```
from pylab import rcParams
rcParams['figure.figsize'] = 18, 8

decomposition = sm.tsa.seasonal_decompose(indexeddataset, model='additive')
fig = decomposition.plot()
plt.show()
```



This further illustrates the trend line and the seasonality.

Model building and Model evaluation:

We calculated the rolling mean and rolling standard deviation:

```
rolmean = indexeddataset.rolling(window=4).mean()
```

```
1
2 rolmean
```

Order Date	Sales
2014-01-01	NaN
2014-02-01	NaN
2014-03-01	NaN
2014-04-01	7650.244000
2014-05-01	7817.809500
2014-06-01	10659.426400
2014-07-01	9721.200150
2014-08-01	9565.077525
2014-09-01	13791.000975
2014-10-01	13565.531325

This time series does not have stationarity. The average mean is increasing over time.

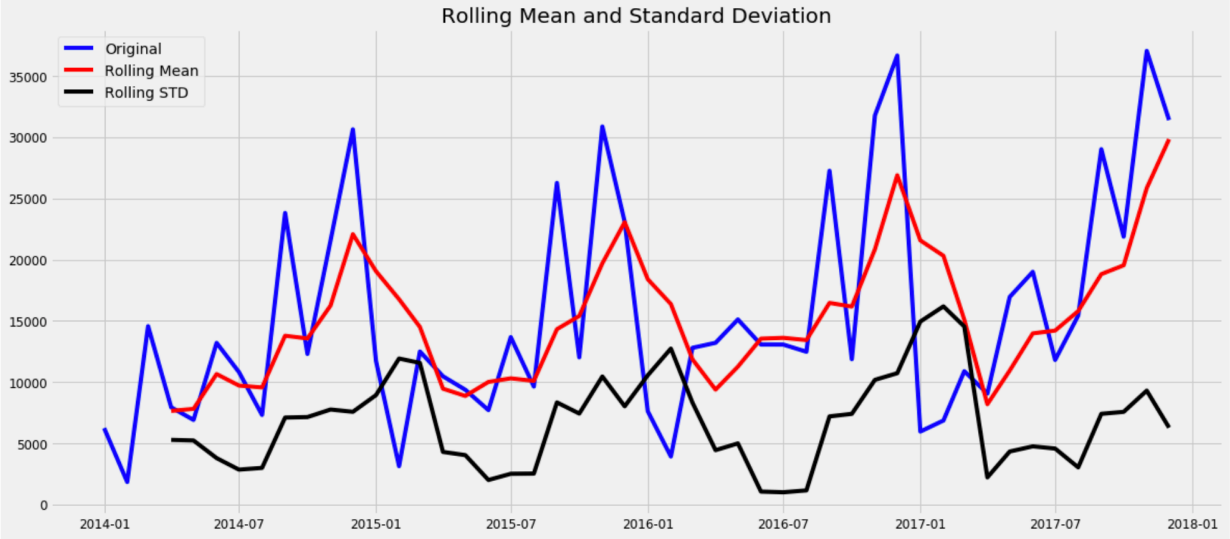
```
1 rolstd = indexeddataset.rolling(window=4).std()
```

```
1 rolstd
```

Order Date	Sales
2014-01-01	NaN
2014-02-01	NaN
2014-03-01	NaN
2014-04-01	5284.226697
2014-05-01	5235.106258
2014-06-01	3795.414831
2014-07-01	2851.726246
2014-08-01	2994.893591
2014-09-01	7107.327137
2014-10-01	7146.264969
2014-11-01	7763.314864

The Rolling Mean (red) and Rolling Standard Deviation (black) are not stationary. We will have to bring this time series into stationarity before we can conduct any time series analysis.

```
# Plot rolling statistics
orig = plt.plot(indexeddataset,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') # will decide on best location for legend
plt.title("Rolling Mean and Standard Deviation")
plt.show(block=False)
```



We conduct the Dickey Fuller test to confirm that this is not a stationary time series.

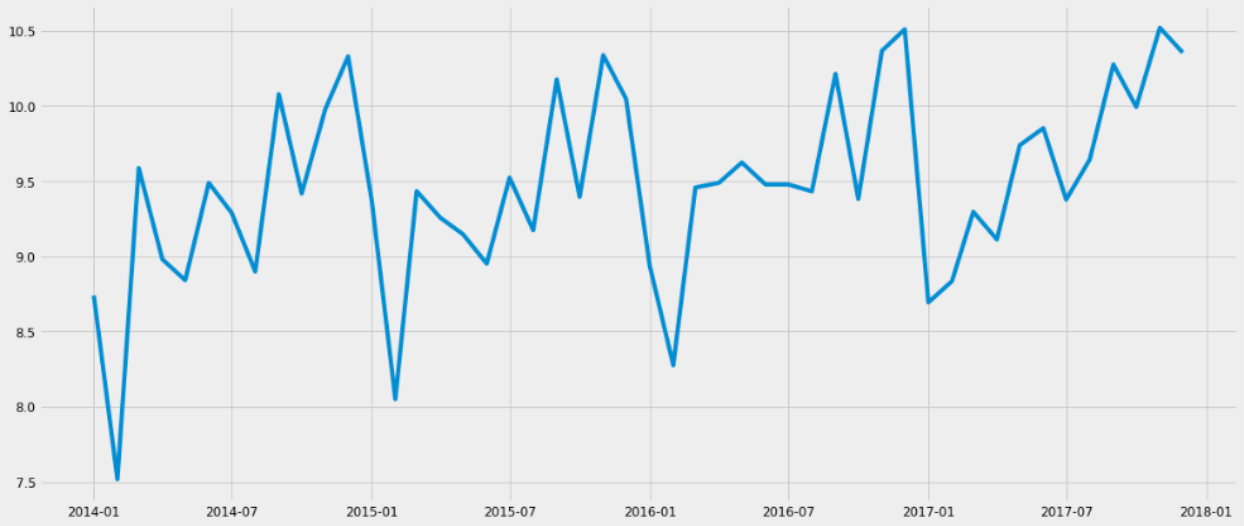
```
# perform Dickey fuller test
from statsmodels.tsa.stattools import adfuller
print("Results from Dickey Fuller Tests")

dftest = adfuller(indexeddataset['Sales'], autolag='AIC') # exact values and the estimated values metric
#dftest = adfuller(indexeddataset.iloc[:,1], autolag='AIC')

dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value','#Lags Used', 'Number of Observations Used'])
for key,value in dftest[4].items():
  dfoutput['Critical Value %s'%key] = value

print(dfoutput)
```

The p-value is very small the Test Statistic is larger than the Critical Values. We must reject the null hypothesis that this time series is stationary.

```
Results from Dickey Fuller Tests
Test Statistic                  -4.699026
p-value                          0.000085
#Lags Used                       0.000000
Number of Observations Used     47.000000
Critical Value 1%               -3.577848
Critical Value 5%               -2.925338
Critical Value 10%              -2.600774
dtype: float64
```

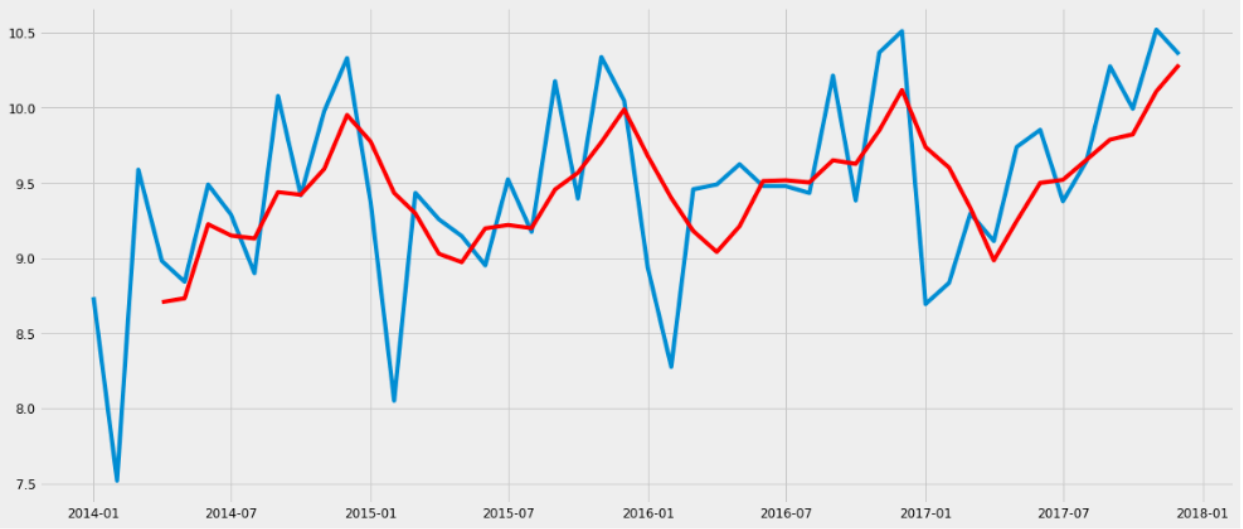
Our next step was to conduct data transformations of the time series:

```
# Estimating trend
indexeddataset_logscale = np.log(indexeddataset)
plt.plot(indexeddataset_logscale)
#normalized by using log_scale
```



Using the log_scale does not bring in stationarity.

```
movingAverage = indexeddataset_logscale.rolling(window=4).mean()
movingstd = indexeddataset_logscale.rolling(window=4).std()
plt.plot(indexeddataset_logscale)
plt.plot(movingAverage,color='red')
# Confirms that the mean is not stationary, graph plotted with logscale, still has an upward trend
```



The log-scale rolling mean (red line) is not stationary; it still has an upward trend.

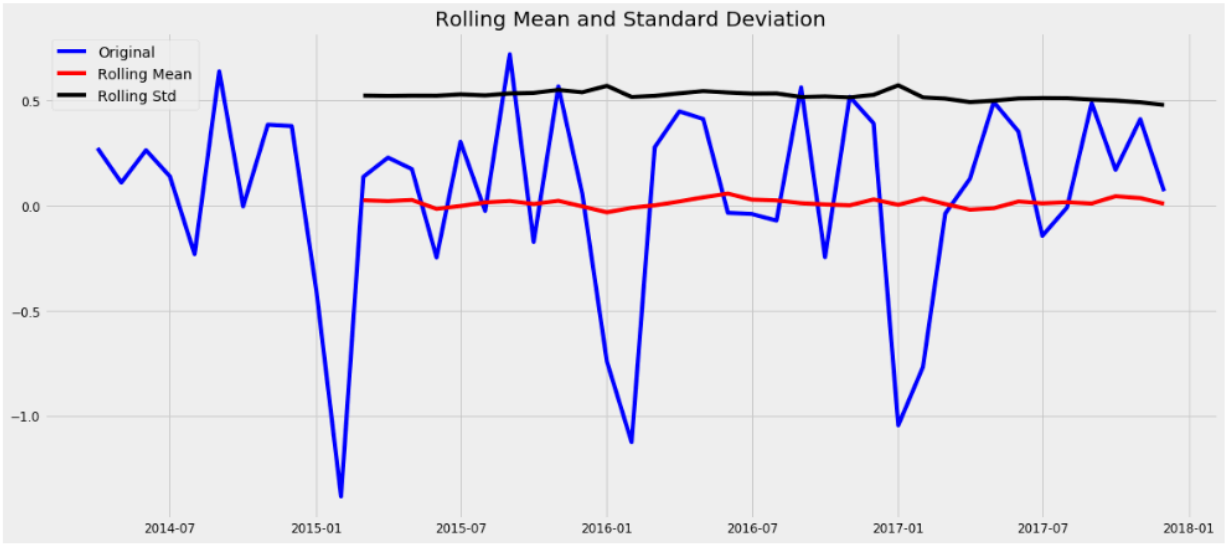
```
datasetLogScaleMinusMovingAverage = indexeddataset_logscale - movingAverage
datasetLogScaleMinusMovingAverage.head(10)
```

We subtract the moving average from the log_scale dataset.

```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):
    #Determining 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 and Standard Deviation')
    plt.show(block=False)
    #Perform Dickey Fuller Test
    print ("Results of Dickey Fuller Test: ")
    dftest = adfuller(timeseries['Sales'],autolag='AIC')
    dfoutput = pd.Series(dftest[0:4],index=['Test Statistic','p-value','#Lags Used','Number of Observations Used']
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key]= value
    print (dfoutput)

```

Set up a function test_stationarity to plot the rolling statistics and perform a Dickey Fuller Test.

```
1 test_stationarity(datasetLogScaleMinusMovingAverage)
```

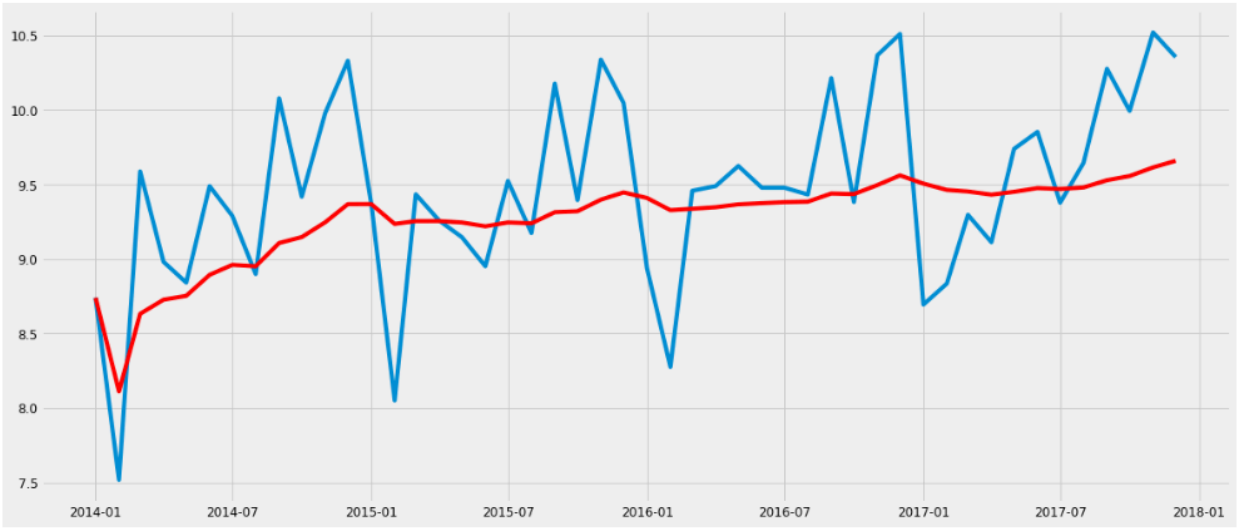



```
Results of Dickey Fuller Test:
Test Statistic                -6.777470e+00
p-value                        2.549062e-09
#Lags Used                     1.000000e+01
Number of Observations Used    3.400000e+01
Critical Value (1%)           -3.639224e+00
Critical Value (5%)           -2.951230e+00
Critical Value (10%)          -2.614447e+00
dtype: float64
```

We are getting closer to stationarity. We continue to perform further experiments.

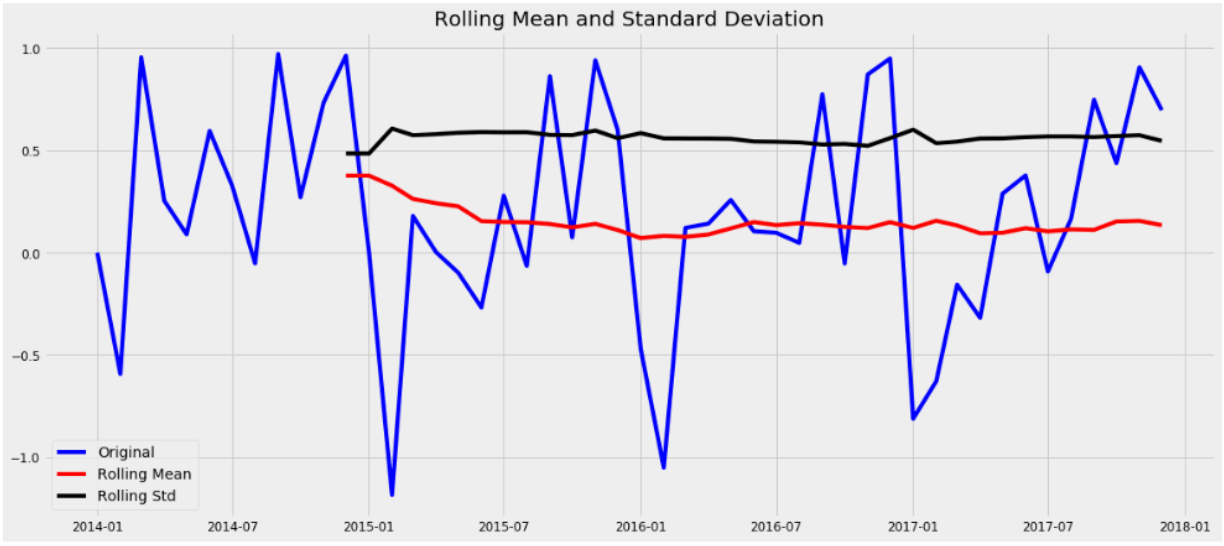
```
1 exponentialDecayWeightedAverage = indexeddataset_logscale.ewm(halflife=12,min_periods=0, adjust=True).mean()
2 plt.plot(indexeddataset_logscale)
3 plt.plot(exponentialDecayWeightedAverage, color = 'red') # trend is upwards
```

```
[<matplotlib.lines.Line2D at 0x7ff894955978>]
```



Further transformed the log_scale by adding an exponential decay with pandas.DataFrame.ewm. This provides exponential weighted functions. The trend is still upwards.

```
datasetLogScaleMinusMovingExponentialDecayAverage = indexeddataset_logscale - exponentialDecayWeightedAverage
test_stationarity(datasetLogScaleMinusMovingExponentialDecayAverage)
```

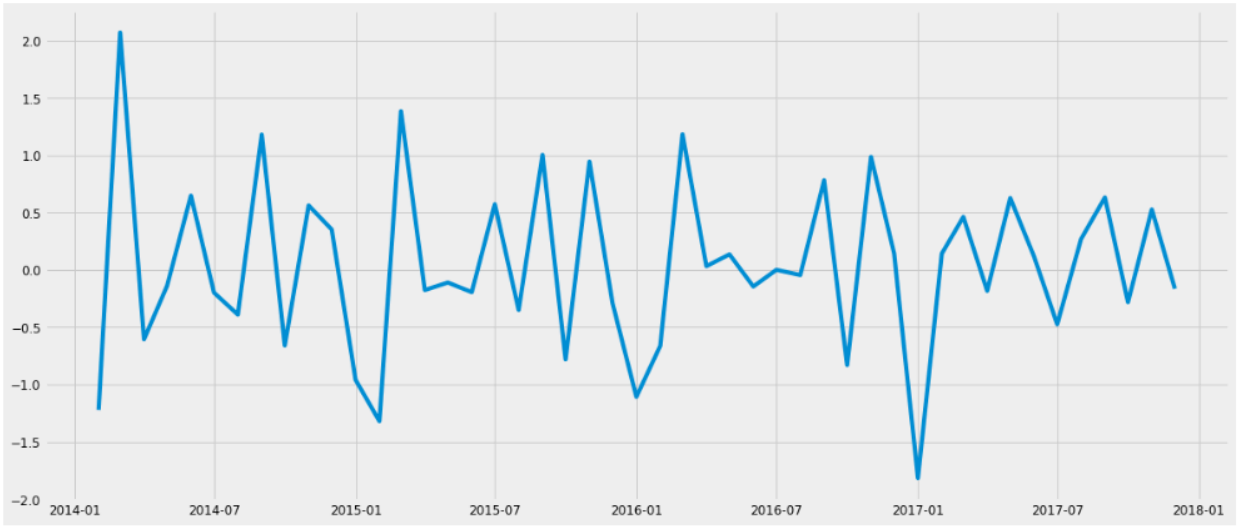



```
Results of Dickey Fuller Test:
Test Statistic                -4.227750
p-value                        0.000591
#Lags Used                     9.000000
Number of Observations Used   38.000000
Critical Value (1%)           -3.615509
Critical Value (5%)           -2.941262
Critical Value (10%)          -2.609200
dtype: float64
```

We subtract the exponential decay weighted average from the log-scale, plot and perform the Dickey Fuller Test.

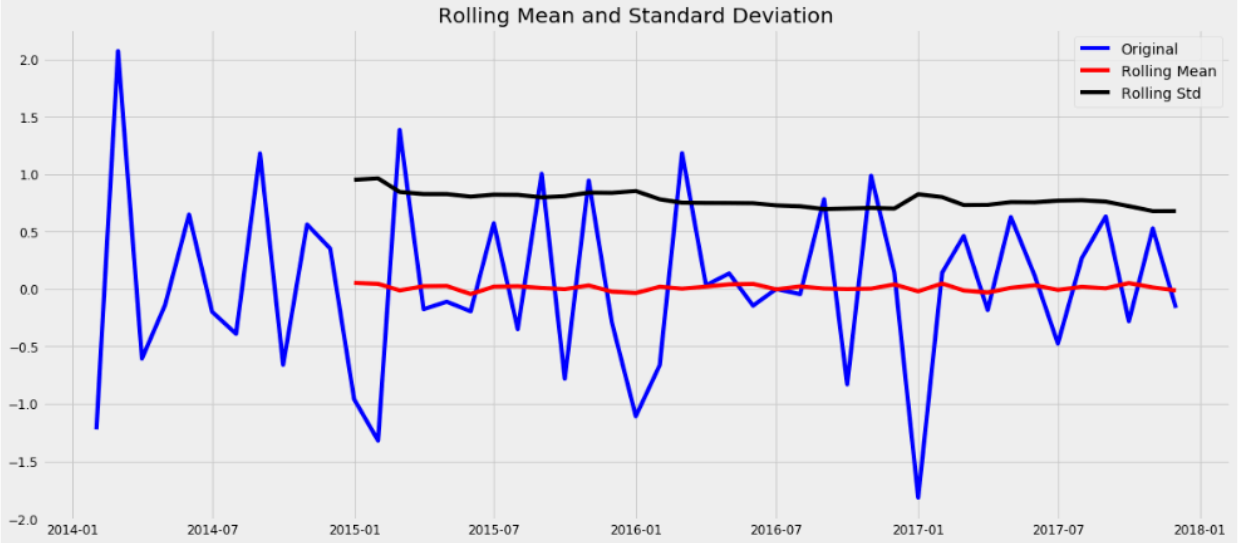
```
datasetLogDiffShifting = indexeddataset_logscale - indexeddataset_logscale.shift()
plt.plot(datasetLogDiffShifting) # differentiating portion of ARIMA 'I' 1 is usually enough
```

```
[<matplotlib.lines.Line2D at 0x7ff894f44c88>]
```



Determining the I portion of ARIMA: usually 1 is enough Differencing.

```
datasetLogDiffShifting.dropna(inplace=True)
test_stationarity(datasetLogDiffShifting)
```

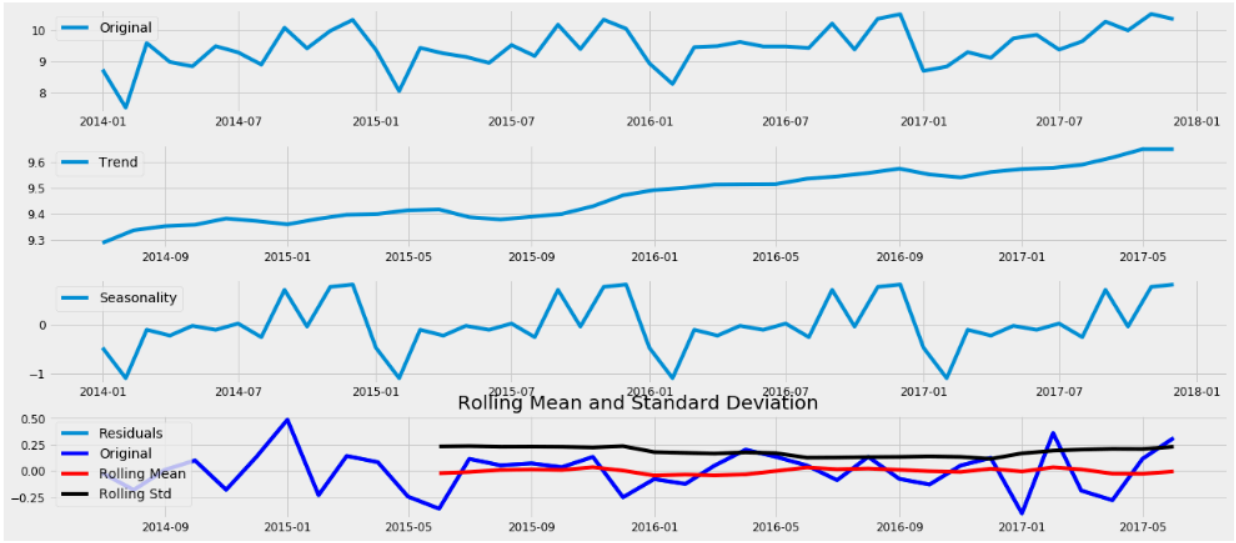



```
Results of Dickey Fuller Test:
Test Statistic                -9.673713e+00
p-value                        1.256989e-16
#Lags Used                     1.000000e+01
Number of Observations Used    3.600000e+01
Critical Value (1%)           -3.626652e+00
Critical Value (5%)           -2.945951e+00
Critical Value (10%)          -2.611671e+00
dtype: float64
```

Plotting the Differenced data.

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(indexeddataset_logscale)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(indexeddataset_logscale, label = "Original")
plt.legend(loc = 'best')
plt.subplot(412)
plt.plot(trend, label = "Trend")
plt.legend(loc = 'best')
plt.subplot(413)
plt.plot(seasonal, label = "Seasonality")
plt.legend(loc = 'best')
plt.subplot(414)
plt.plot(residual, label = "Residuals")
plt.legend(loc = 'best')
plt.tight_layout()

decomposedLogData = residual
decomposedLogData.dropna(inplace=True)
test_stationarity(decomposedLogData)
```

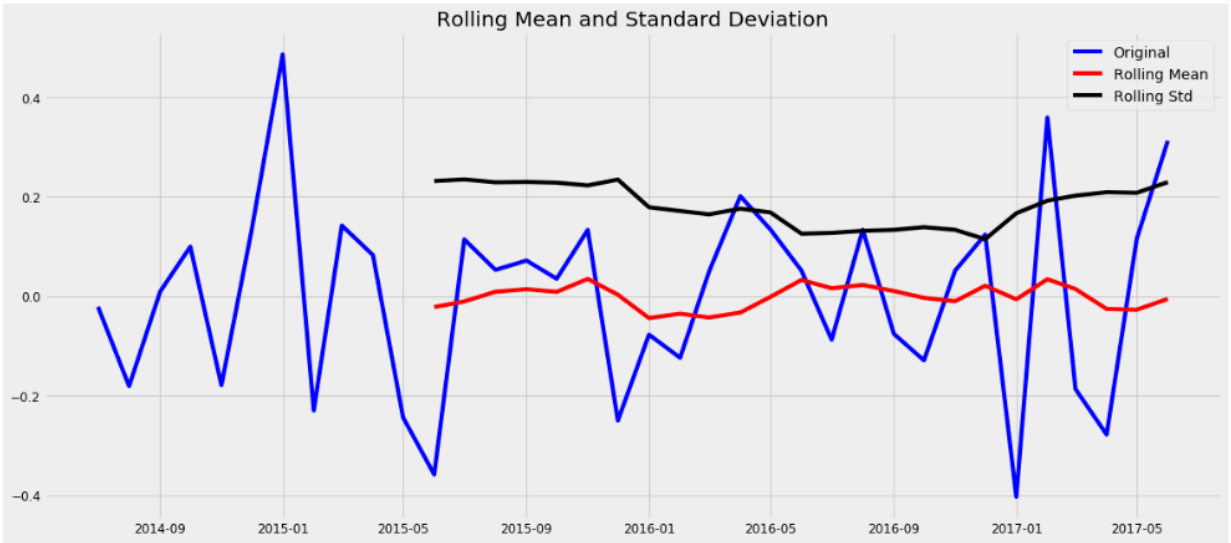
Code to plot the seasonal decomposition on the log_scaled dataset.




```
Results of Dickey Fuller Test:
Test Statistic                  -5.499676
p-value                          0.000002
#Lags Used                       4.000000
Number of Observations Used     31.000000
Critical Value (1%)             -3.661429
Critical Value (5%)             -2.960525
Critical Value (10%)            -2.619319
dtype: float64
```

Testing the residuals:

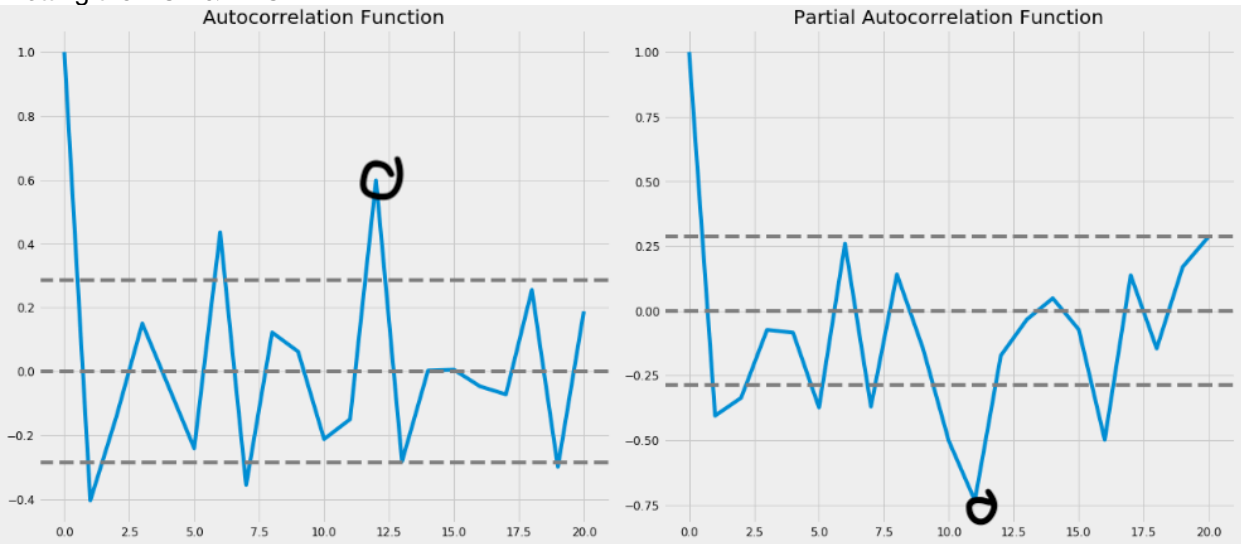
```
1 decomposedLogData = residual
2 decomposedLogData.dropna(inplace=True)
3 test_stationarity(decomposedLogData)
```




```
Results of Dickey Fuller Test:
Test Statistic                  -5.499676
p-value                          0.000002
#Lags Used                       4.000000
Number of Observations Used     31.000000
Critical Value (1%)             -3.661429
Critical Value (5%)             -2.960525
Critical Value (10%)            -2.619319
dtype: float64
```

```
#ACF & PACF plots:
from statsmodels.tsa.stattools import acf, pacf

lag_acf = acf(datasetLogDiffShifting, nlags=20)
lag_pacf = pacf(datasetLogDiffShifting, nlags=20, method = 'ols')

#plot ACF
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(datasetLogDiffShifting)),linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(datasetLogDiffShifting)),linestyle='--', color='gray')
plt.title("Autocorrelation Function")

#plot PACF
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(datasetLogDiffShifting)),linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(datasetLogDiffShifting)),linestyle='--', color='gray')
plt.title("Partial Autocorrelation Function")
plt.tight_layout()
```

Plotting the ACF & PACF:



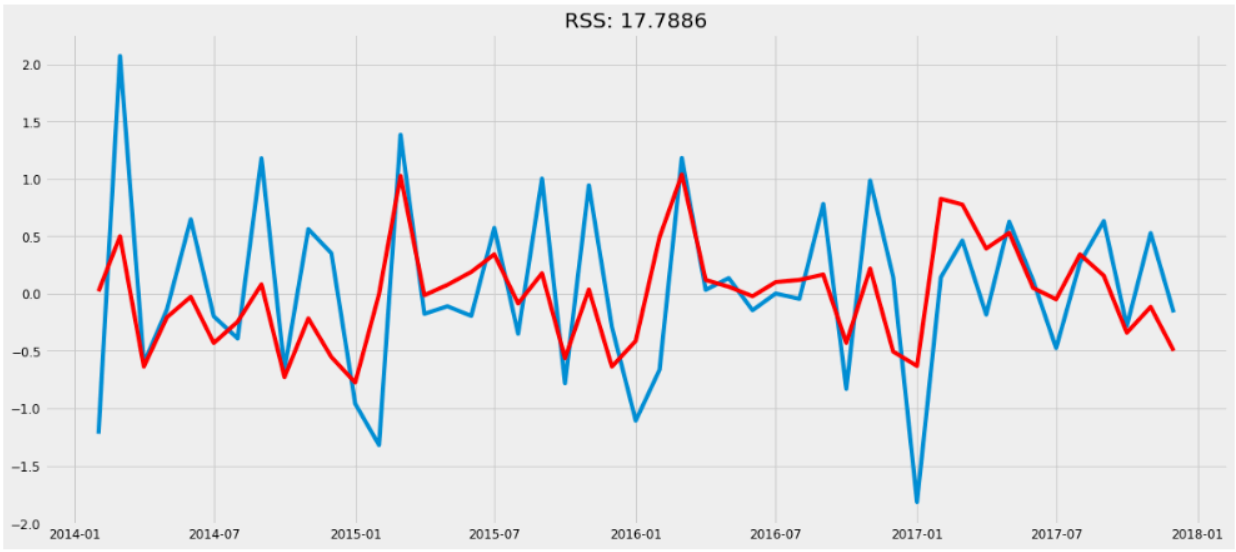
The ACF and the PACF show activity at the 12th lag.

```
from statsmodels.tsa.arima_model import ARIMA
```

```
model = ARIMA(indexeddataset_logscale,order=(0,1,2))
results_ARIMA = model.fit(disp=-1)
plt.plot(datasetLogDiffShifting)
plt.plot(results_ARIMA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues - datasetLogDiffShifting['Sales'])**2))
print("Plotting ARIMA Model")
#17.8043 (1,1,1)
#17.8406 (2,1,1)
# 17.7886 (0,1,2)***
# 18.3606 (0,1,1)
```

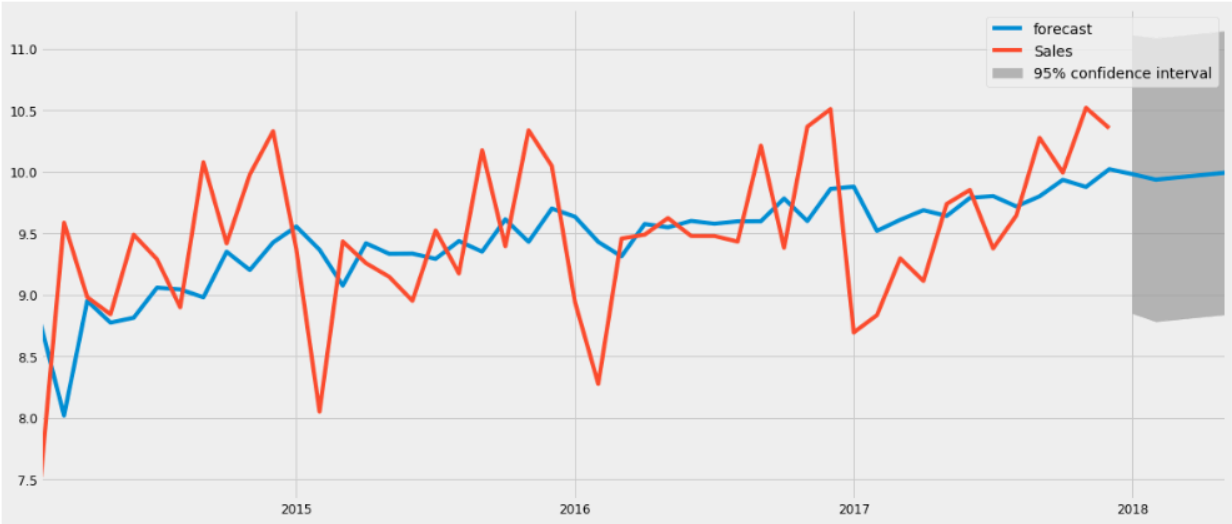
Import ARIMA from statsmodels and begin testing to discover the best ARIMA model. The results of our testing point to the (0,1,2) ARIMA model off of the log_scale dataset. We determine this because it has the lowest Residual Sum of Squares number: 17.7886.

```
Plotting ARIMA Model
```



Plotting the resulting forecast is not very satisfying. It is extremely vague with a large confidence interval that doesn't capture previous activity or forecast well.

```
1 y = results_ARIMA.plot_predict(1,52)
2 #x=results_ARIMA.forecast(steps=120)
3
```



Our next step was to work with Facebook's Prophet library.

```
1 from fbprophet import Prophet
```

According to their website, https://facebook.github.io/prophet/, "Prophet is a procedure 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. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well." Prophet is available in both R and Python. It also has advanced abilities to model holiday effects on a time series. Their model is a decomposable model with three main elements: trend, seasonality and holidays. Time is used as a regressor and the seasonality modeling is an additive element which mimics the exponential smoothing in the Holt-Winters method.
This time series does have strong seasonal pattern and we have 4 years of historical data.
Next, we rename our columns and instantiate a Prophet model. We set the uncertainty interval to 95% (the Prophet default is 80%). We call the fit method on our dataset.

```
indexeddataset = indexeddataset.rename(columns={'Order Date': 'ds', 'Sales': 'y'})
furniture_model = Prophet(interval_width=0.95)
furniture_model.fit(indexeddataset)
```

This is the output:

```
<fbprophet.forecaster.Prophet at 0x7ff88c3f89b0>
```

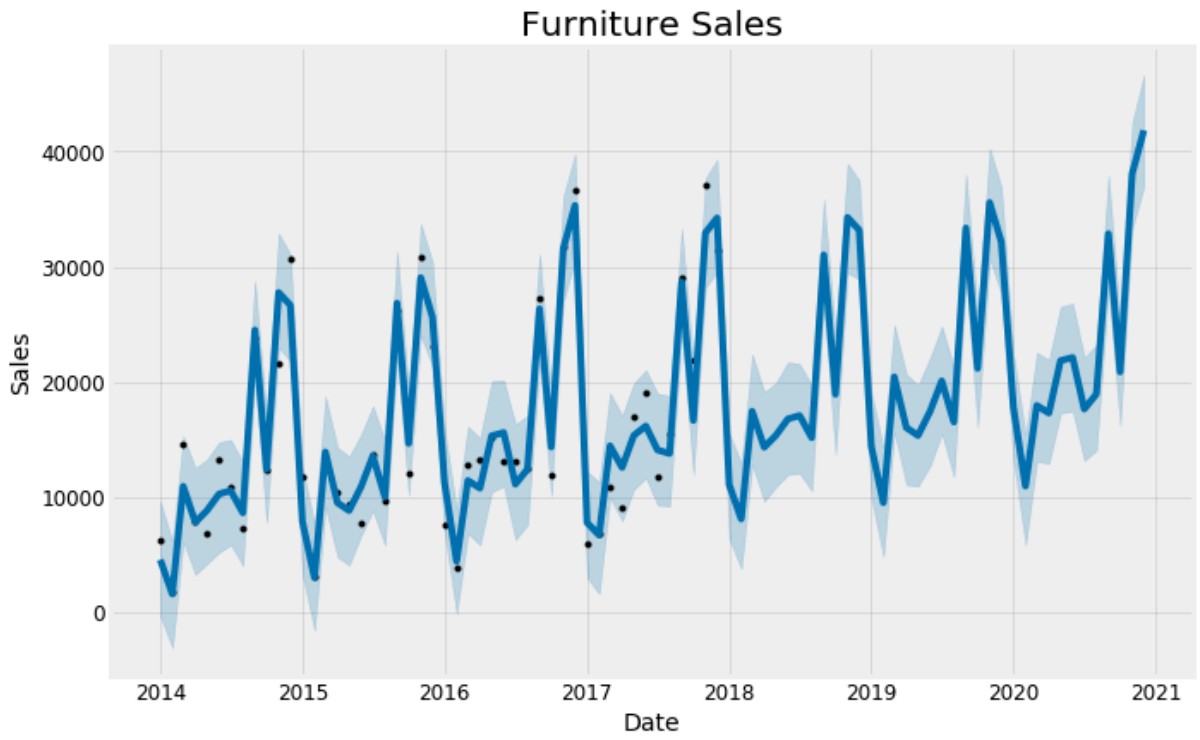
We use the helper function make_future_dataframe to set up our dataframe for our forecasts.

```
furniture_forecast = furniture_model.make_future_dataframe(periods=36, freq='MS')
furniture_forecast = furniture_model.predict(furniture_forecast)
```

We are working with monthly data. We are asking for 36 months of forecasts indicating with “MS” that it is monthly. Our dataframe “furniture_forecast” is then used as input to the predict method of our fitted model. The predict function produces the forecasts.

```
plt.figure(figsize=(18, 6))
furniture_model.plot(furniture_forecast, xlabel = 'Date', ylabel = 'Sales')
plt.title('Furniture Sales');
```

Plotted Furniture Sales:

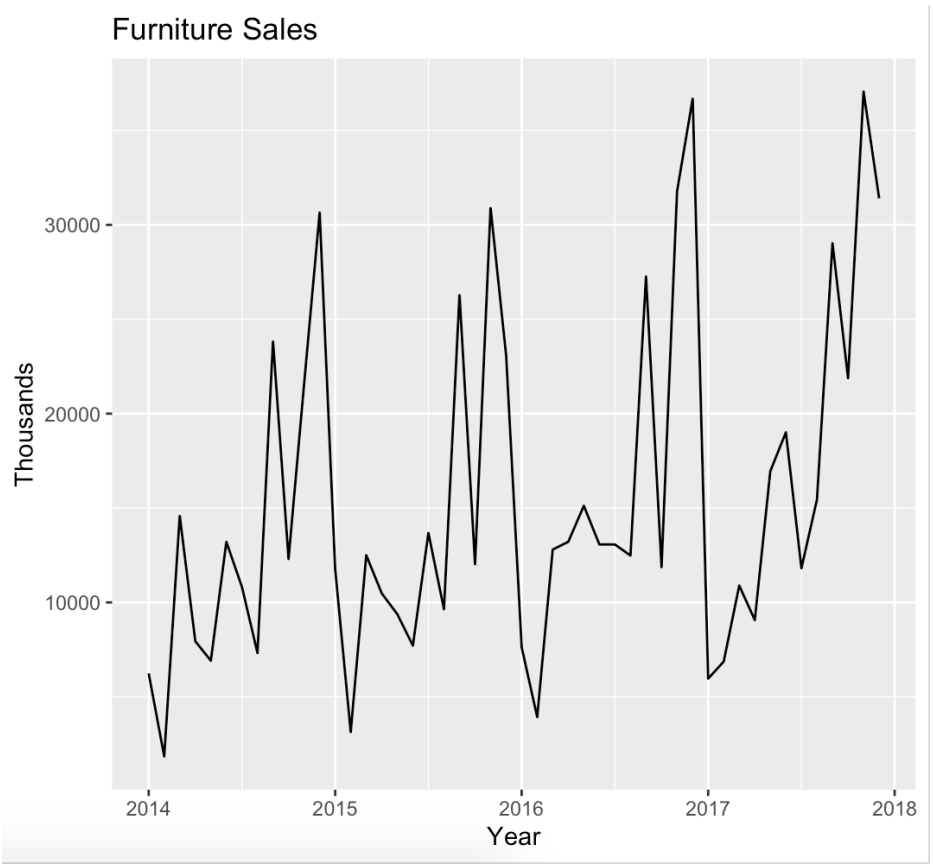


This forecast captures the seasonality and trend. The confidence bands are much tighter than the previous forecast produced by the previous Python ARIMA model and it captures the historical data.

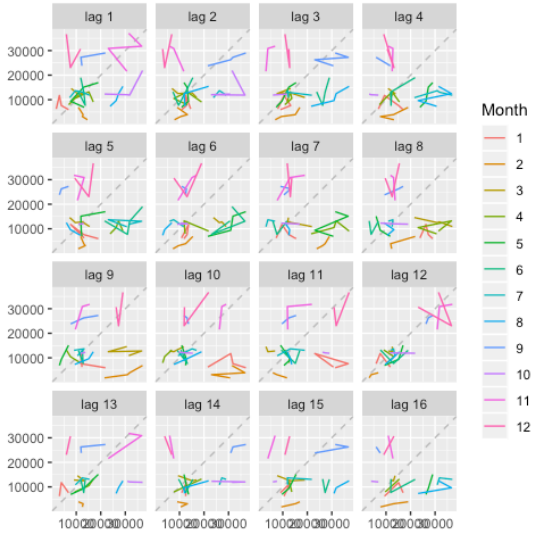
Our final exploration was using R with the fpp2 forecasting library. We again charted the dataset. This dataset has clear seasonality and a trend.
The following interesting features:

- Clear spike in demand at the end of the year with a sharp drop at the end of the year.
- Each year the peaks and the bottoms are rising higher indicating an overall rising trend in sales

autoplot(y[,"Sales"]) + ggtitle("Furniture Sales") + xlab("Year") + ylab("Thousands")

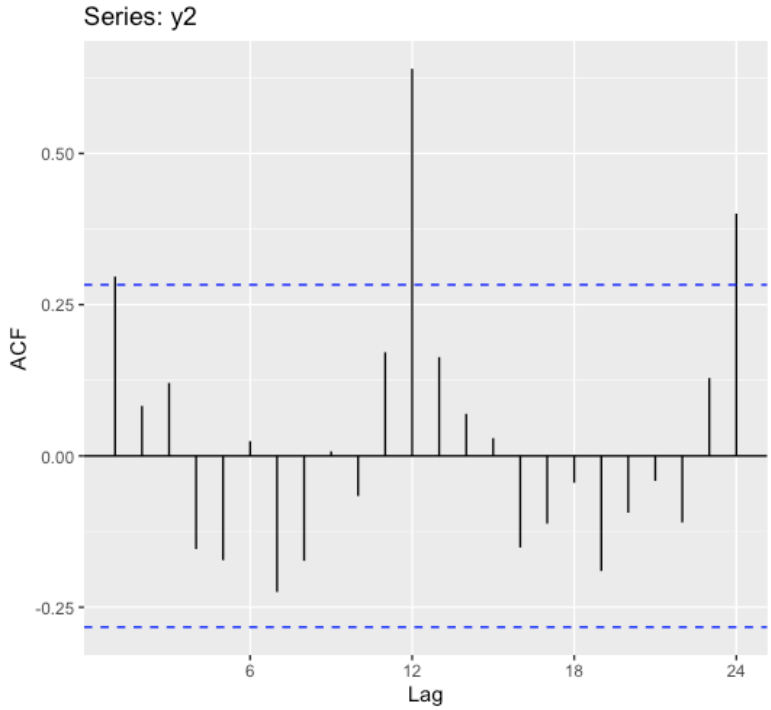



```
y2 <- window(y[,"Sales"], start=2014)
gglagplot(y2)
```



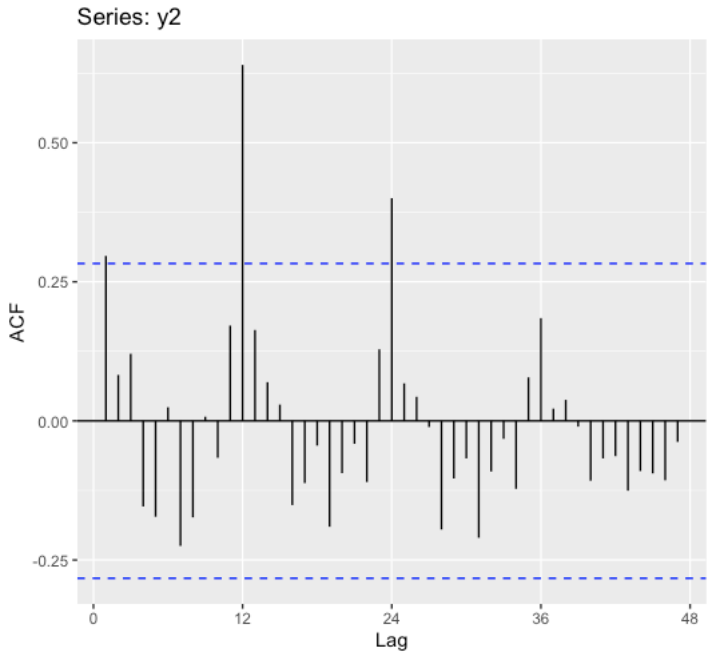
Notice that the strongly positive lag 12. This confirms the strong seasonality of this time series.
We check for the Autocorrelation using the ggAcf function:

```
> ggAcf(y2)
```



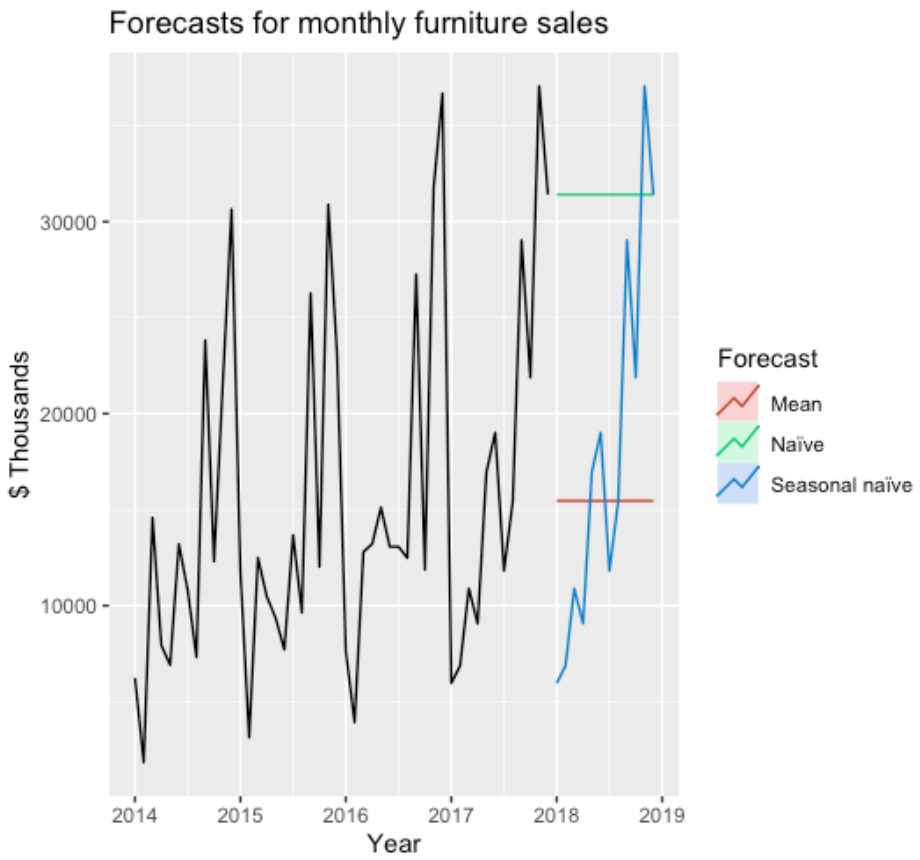
The correlogram shows a strong lag at 12. R12 is higher than for the other lags. This is due to the seasonal pattern in the data: the peaks tend to be 12 months apart and the troughs tend to be 12 months apart. When time series are seasonal, the autocorrelations will be larger for the seasonal lags (at multiples of the seasonal frequency) than for other lags. This is additional confirmation of the strong seasonality.

ggAcf(y2, lag = 48) We check 48 months out to see the correlogram over a greater time period.



The overall decrease in the ACF as the lags increase is due to the trend, and the seasonality is expressed in the wavy shape. This is not an example of white noise because autocorrelation is present in this time series.

Next we tried some simple forecasting methods: naïve, seasonal naïve, and mean.



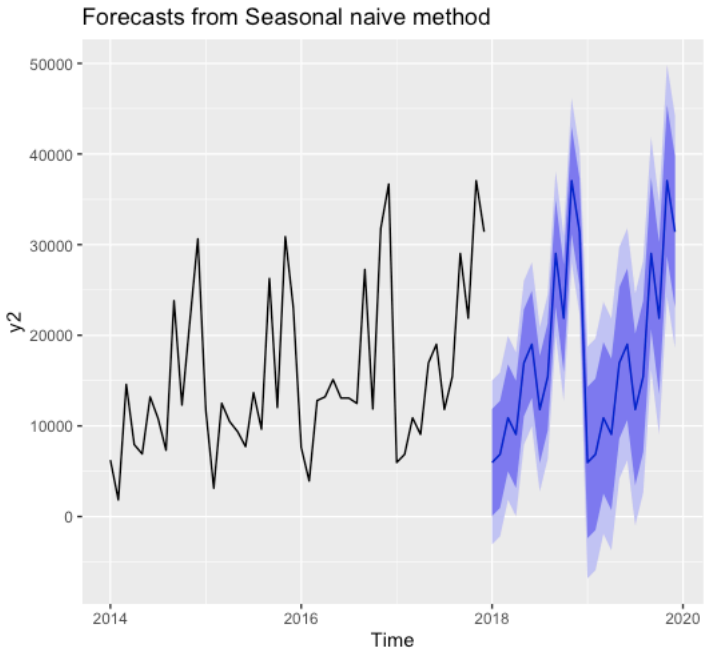
Mean is just an average of the previous activity, Naïve continues the last point in the time series and Seasonal naïve simply duplicates the previous year's pattern. None of these methods provide a sophisticated analysis of the furniture time series.

We evaluated the forecast accuracy of these three methods:

```
> accuracy(yfit1,y4)
                     ME      RMSE      MAE       MPE     MAPE     MASE
Training set 6.564625e-13  8612.664 6712.822 -52.71474 76.06437 1.958265
Test set     3.320813e+03 10238.807 8077.119 -11.58464 51.37871 2.356258
                  ACF1 Theil's U
Training set 0.2315520        NA
Test set     0.5672242  1.294449
> accuracy(yfit2,y4)
                     ME     RMSE       MAE        MPE      MAPE    MASE
Training set    869.6054 10067.76  8013.815  -25.58397  67.58489 2.33779
Test set     -18729.7759 21085.77 18792.776 -179.78848 179.95849 5.48223
                   ACF1 Theil's U
Training set -0.4548424        NA
Test set      0.5672242  4.550047
> accuracy(yfit3,y4)
                    ME     RMSE      MAE      MPE     MAPE     MASE        ACF1
Training set 1737.858 4673.929 3427.944 9.529880 24.76228 1.000000 -0.09349851
Test set     1373.819 4473.384 3747.025 4.304622 24.04933 1.093082  0.01554347
             Theil's U
Training set        NA
Test set     0.5747407
>
```

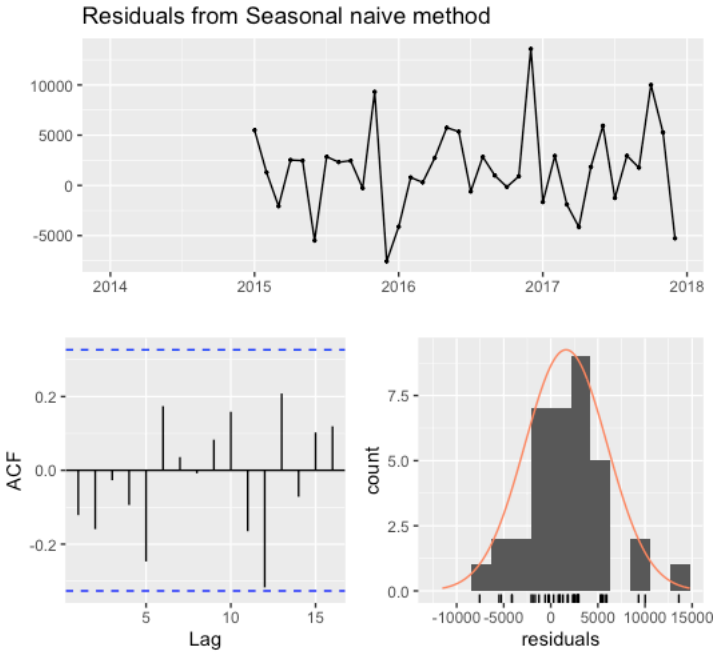
The seasonal naïve forecast model had the best RMSE of the three forecast methods.

autoplot(snaive(y2))

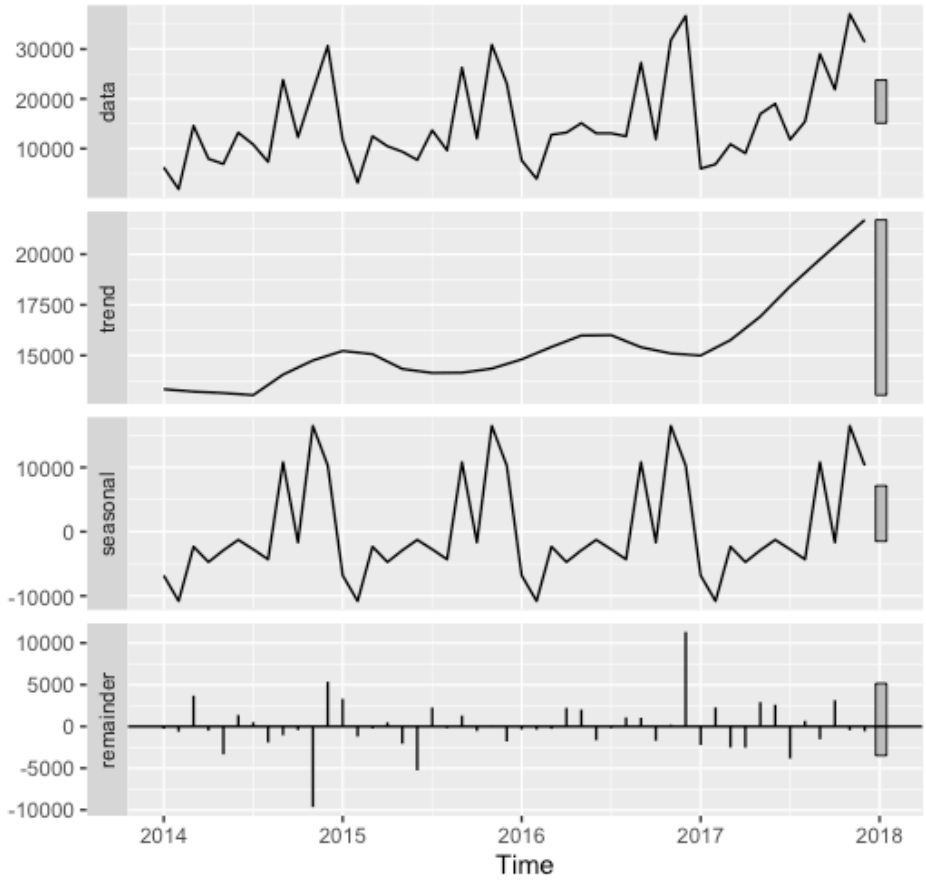


This graph only mirrors the previous year's seasonal pattern with more variability in the second year's forecast.

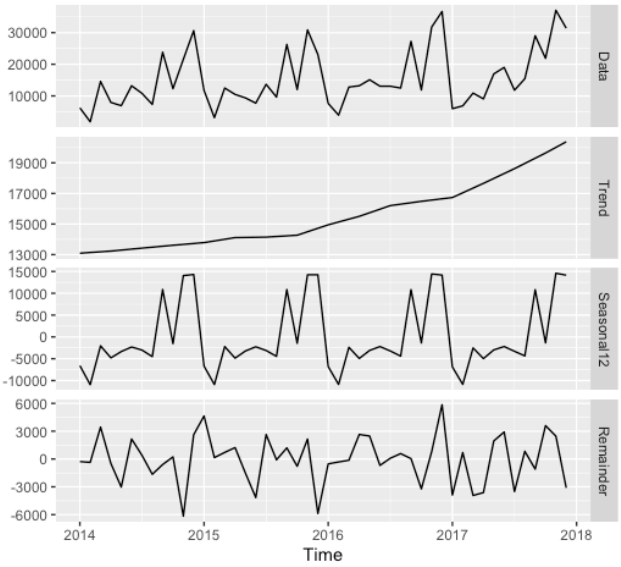
```
> checkresiduals(snaive(y2))

        Ljung-Box test

data:  Residuals from Seasonal naive method
Q* = 7.8197, df = 10, p-value = 0.6464

Model df: 0.   Total lags used: 10
```



We used STL to decompose the time series.
> y2 %>% stl(t.window=13, s.window="periodic", robust=TRUE) %>% autoplot()



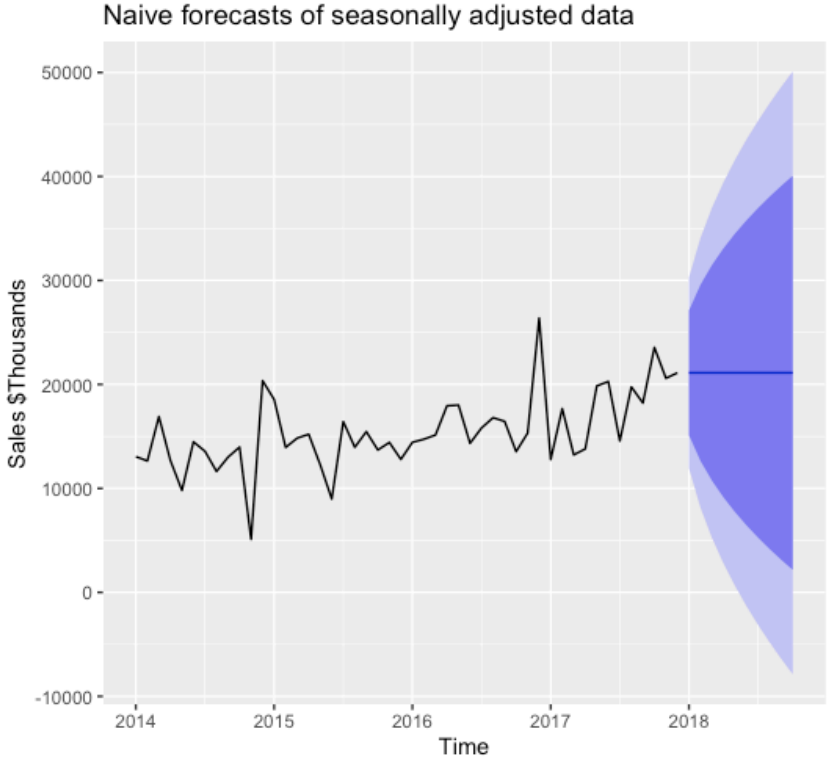
There's some activity in the remainders at 3rd quarter 2015 and 2017.
> y2 %>% mstl() %>% autoplot()
We also plotted the automated STL decomposition for comparison:



The remainders are different in this plot compared to the previous decomposition.

We forecasted the seasonally adjusted time series using naïve since a non-seasonal forecasting method must be used.

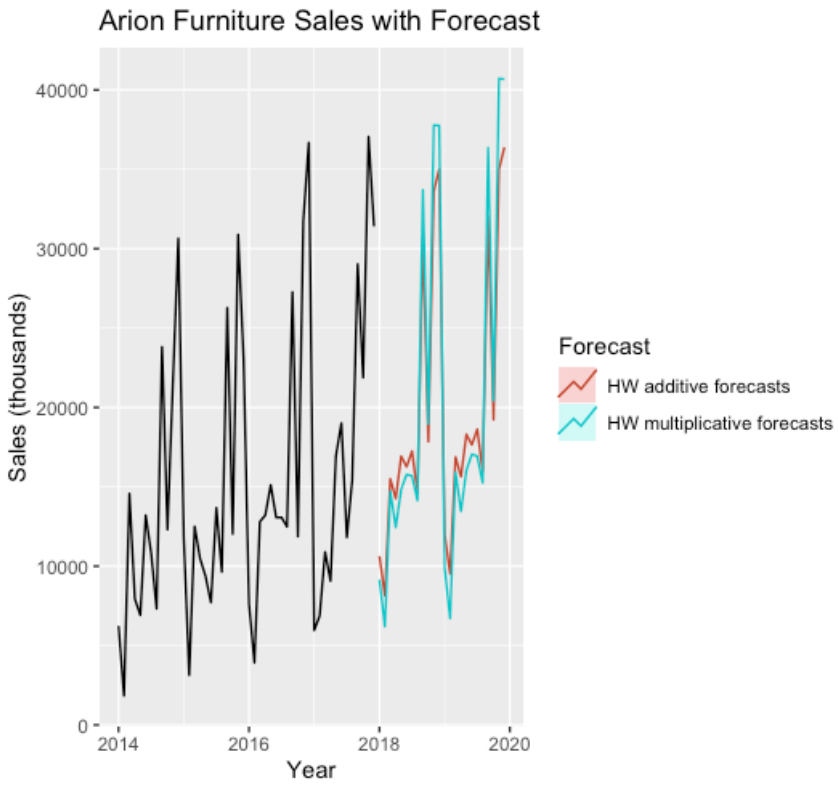
```
> fit <- stl(y2, t.window=13, s.window="periodic",robust=TRUE)
> fit %>% seasadj() %>% naive() %>% autoplot() + ylab("Sales $Thousands") + ggtitle("Naive forecasts of seasonally adjusted data")
```



This chart doesn't provide much insight as the naïve method is the last point extended into time, but it is interesting to see the data seasonally adjusted.

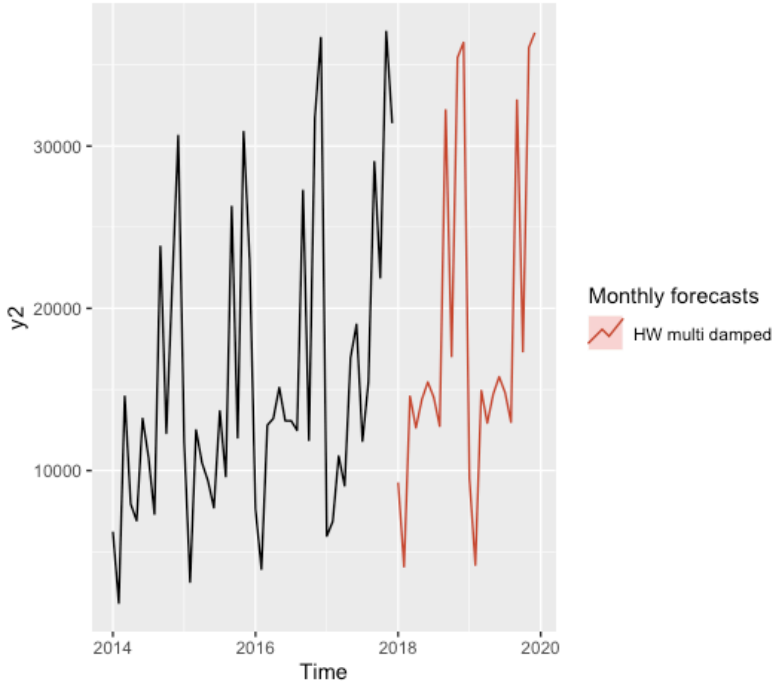
Next we did a comparison of the Holt-Winters' seasonal method with additive and multiplicative variations.

```
> fit1 <- hw(y2,seasonal="additive")
> fit2 <- hw(y2,seasonal="multiplicative")
> autoplot(y2) + autolayer(fit1, series="HW additive forecasts", PI=FALSE) + autolayer(fit2, series="HW multiplicative forecasts",PI=FALSE) + xlab("Year") + ylab("Sales (thousands)") + ggtitle("Arion Furniture Sales with Forecast") + guides(colour=guide_legend(title="Forecast"))
```

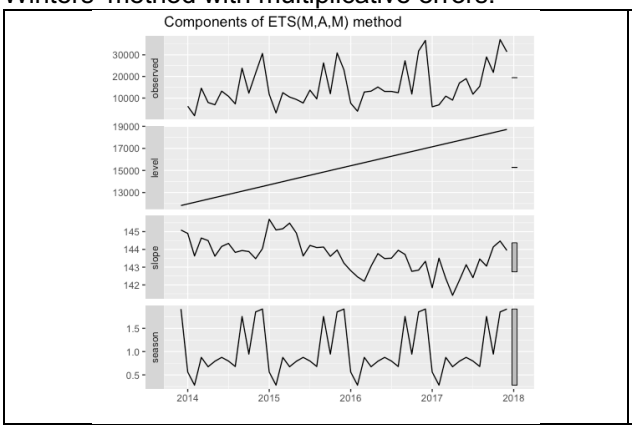


Damped Holt-Winters multiplicative method typically is very accurate in forecasting for seasonal time series. This is part of the family of Exponential Smoothing models.

```
> fc <- hw(y2, damped=TRUE, seasonal="multiplicative")
> autoplot(y2) + autolayer(fc, series="HW multi damped", PI=FALSE)+
guides(colour=guide_legend(title="Monthly forecasts"))
```



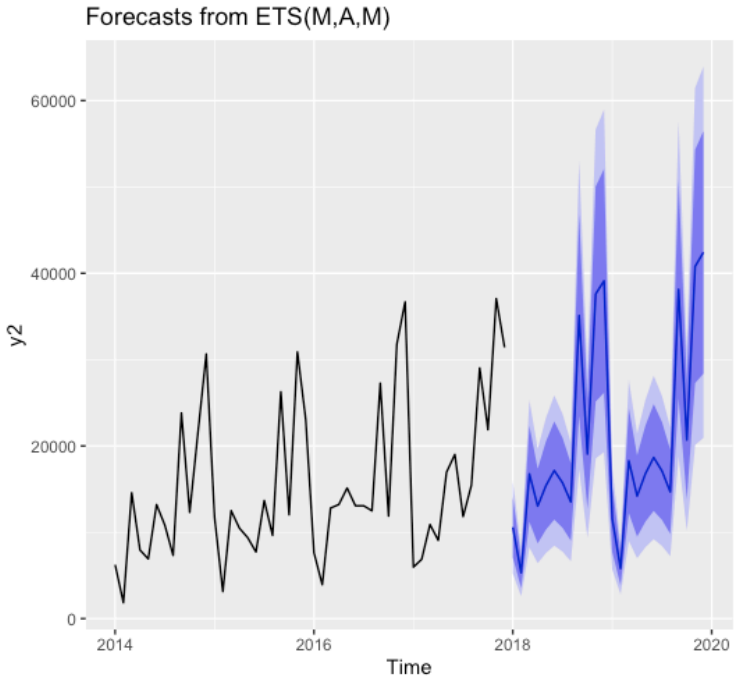
Use ets() function choose a model for this time series which was ETS(M,A,M) the multiplicative Holt-Winters' method with multiplicative errors.




```
> ets(y2)
ETS(M,A,M)

Call:
 ets(y = y2)

  Smoothing parameters:
    alpha = 3e-04
    beta  = 2e-04
    gamma = 4e-04

  Initial states:
    l = 11822.6336
    b = 145.0904
    s = 1.913 1.8516 0.9457 1.7534 0.6797 0.7971
          0.8757 0.7929 0.6751 0.8751 0.2798 0.5609

  sigma:  0.2584

     AIC      AICc       BIC
982.2692 1002.6692 1014.0797
```

The forecast () function can return forecasts when applied to an ets object or many other time series models.

> autoplot(forecast(fit))



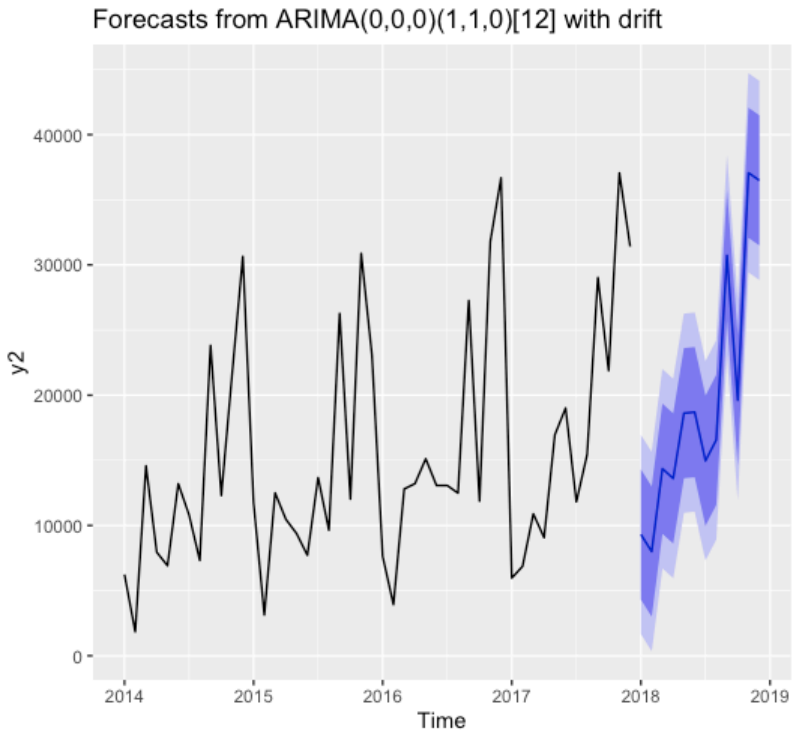
Auto.arima() automatically selects the ARIMA model with the smallest AIC.

```
> auto.arima(y2)
Series: y2
ARIMA(0,0,0)(1,1,0)[12] with drift

Coefficients:
         sar1     drift
      -0.4810  143.3318
s.e.   0.1644   40.2019

sigma^2 estimated as 15184501:  log likelihood=-349.28
AIC=704.55   AICc=705.3   BIC=709.3
```

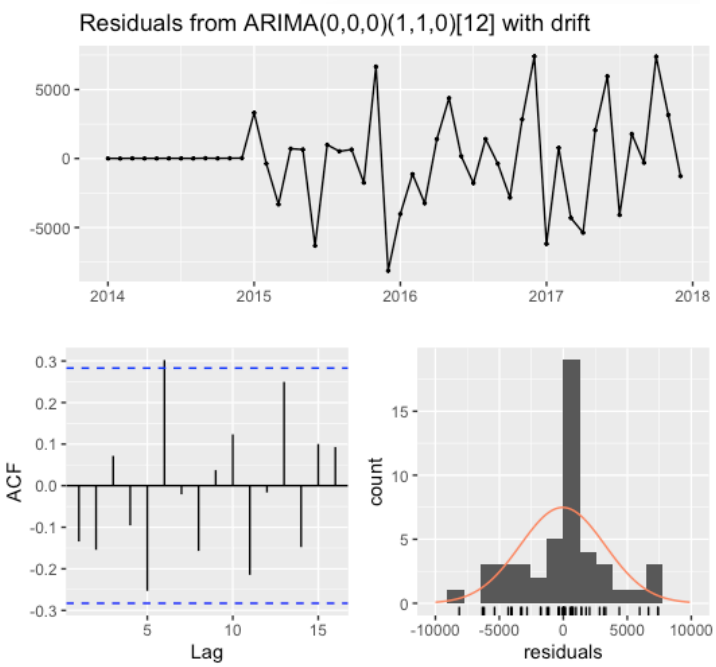
> fit2 <- auto.arima(y2)

> autoplot(fit2)

> fit2 %>% forecast(h=12) %>% autoplot()




```
> checkresiduals(fit2)

        Ljung-Box test

data:  Residuals from ARIMA(0,0,0)(1,1,0)[12] with drift
Q* = 14.286, df = 8, p-value = 0.07461

Model df: 2.   Total lags used: 10
```

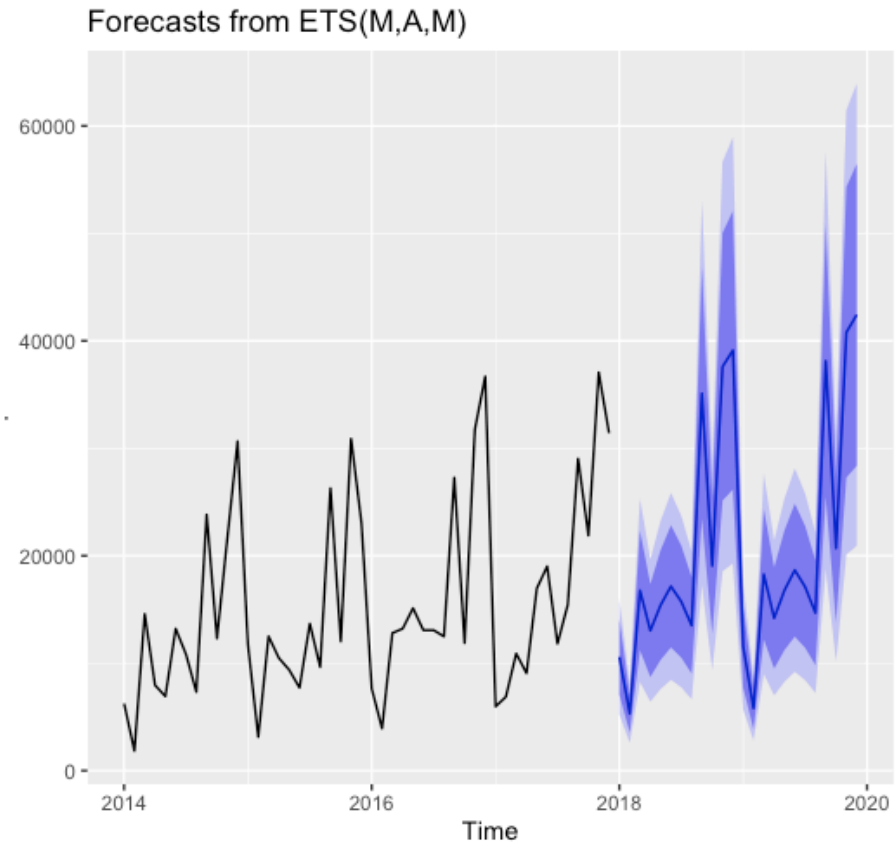


We used the auto.arima() function which automated the process that we used earlier in Python to determine the best ARIMA model. Note that a different model was selected, and it is a much more robust and different forecast than we received with the Python modeling. It predicted a (0,1,2) as the best ARIMA model and didn't include the P,D,Q seasonal portion in its model. Remember that the ACF and PACF graphs had significant spikes at Lag 12.
Finally, we compared an ETS to an ARIMA model:

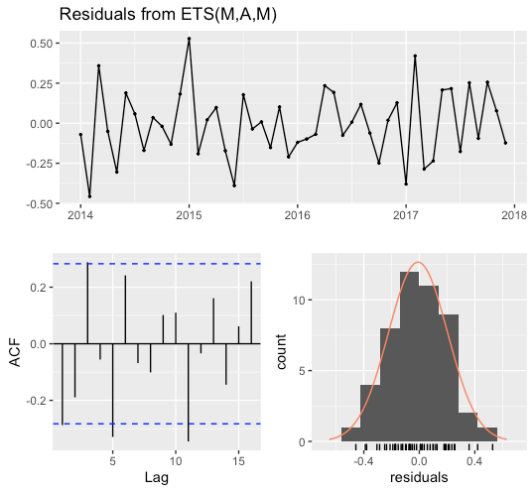
```
> fets <- function(x, h) {
+     forecast(ets(x), h = h)
+ }
> farima <- function(x, h) {
+     forecast(auto.arima(x), h=h)
+ }
```

```
> e1 <- tsCV(y2, fets, h=12)
> e2 <- tsCV(y2, farima, h=12)
> mean(e1^2, na.rm=TRUE)
[1] 63112255
> mean(e2^2, na.rm=TRUE)
[1] 66350522
```

In this case the ETS model has a lower tsCV statistic based on MSEs.
We make a final plot of the ETS(M,A,M) model as the winner based on the results from the MSE statistic.
> y2 %>% ets() %>% forecast() %>% autoplot()




```
> checkresiduals(fit3)

        Ljung-Box test

data:  Residuals from ETS(M,A,M)
Q* = 39.191, df = 3, p-value = 1.581e-08

Model df: 16.   Total lags used: 19
```



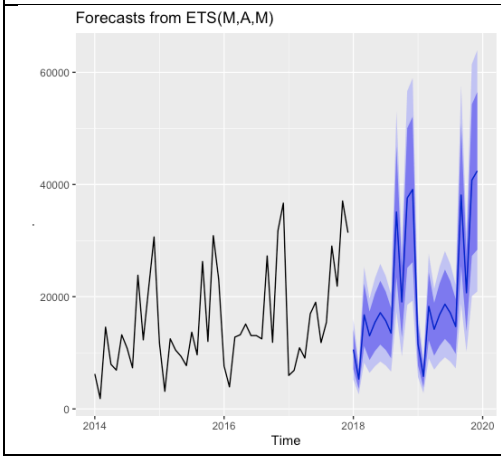
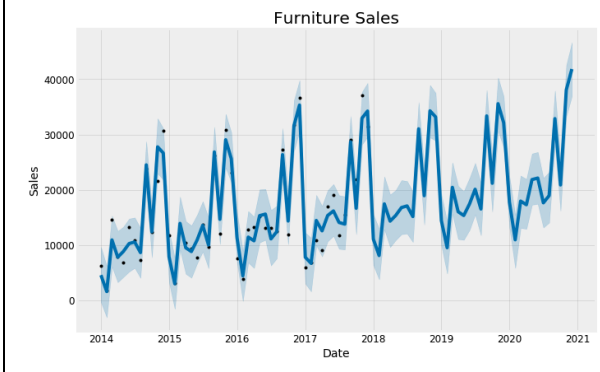
Observations:

The Python ARIMA modeling was disappointing as it didn't capture the trend or seasonality. For future research we would investigate the statsmodels SARIMAX library and conduct a grid-search for the best ARIMA model.

The Facebook Prophet model does seem to capture the seasonality and trend, but the confidence bands are very tight. This could be due to over-fitting. Their website https://facebook.github.io/prophet/, states "Get a reasonable forecast on messy data with no manual effort. Prophet is robust to outliers, missing data, and dramatic changes in your time series." For future research we would explore the multiple parameters that Prophet has i.e. growth, changepoints, yearly-seasonality, holidays, etc. This would help us fine-tune the forecast to Arion's unique situation.

The fpp2 package by Rob J Hyndman and George Athanasopoulos was extremely powerful in conducting the forecasting in R. We were able to conduct many experiments. The winner in all our experiments (by a whisker) was the ETS(M,A,M) the multiplicative Holt-Winters' method with multiplicative errors.

Conclusion:

R Forecast	Facebook Prophet Forecast
	

There are interesting differences between the ETS(M,A,M) and the Facebook Prophet forecasts. The ETS model forecasts much higher spending, well above the 40,000 mark. The Prophet model is more

conservative in its projections. The Prophet model was compared to the historical data (Dark blue line is forecasting spend numbers, black dots are the actual spend numbers. The light blue shade is 95% confidence interval around the forecast.). It captured all of the historical data points within its confidence intervals. Our team would recommend continuing to tune the Prophet forecast with additional information that is unique to Arion's business model.

Boaz Allen has provided a traditional forecast methodology utilizing the latest technologies available. But we do recommend that judgmental forecasting from Arion's existing team or consultants from Boaz Allen be added as an adjunct to these concluding observations. There is expert knowledge that can be applied to further increase the accuracy of these forecasting models.