avnuzwucn

April 2, 2024

1 Final Project Part 2

1.1 Profit Projections - Forecasting in Grocery Stores

1.1.1 Team

- Ingrid Arreola
- Yash Pandey
- Rahul Palivela

1.1.2 Dataset 101

This project is using data gathered by Ann Mary Paattarn, this is available via Kaggle as a competition entry. The dataset was released in 2023 and the competition focused on forecast sales for several product categories offered at the grocery stores Corporación Favorita. This dataset has a monthly time interval.

1.1.3 Description

This notebook contains the following:

1. Time series decomposition 10pts – Look at HW 2 Q2

- classical_decomposition: Rahul
- additive decomposition: Ingrid

2. Time Series Visualization 10pts (at least 4 plots) – gg_plot here refers to seasonality

- Autoplot: Ingrid
- GG_Plot Year: Rahul
- GG_Plot Quarter: Yash
- GG_Plot Month: Yash

3. Description of Time series (is it stationary, are there any trends, seasonality etc) 10pts

- Everyone writes a description on their visualization

4. TS models (transformations if needed). Make sure to explain your decision. Compare and select the best model. 50pts — things from homework (NAIVE, SNAIVE, Random Walk, Moving Average, Drift, ARIMA)

- ARIMA: Ingrid
- SNAIVE & LSTM:Yash
- NAIVE:Rahul

5. Predictions (make some predictions and plot them). 10pts
6. Team contributions. For each team member, describe the contribution. 10pts. Graders reserve the right to apply a different individual grade based on individual contribution.

Ingrid Arreola Contributions: - Setup Notebooks, Data Exploration for 2 data frames, Question 1: Additive Decomposition, Question 2: Autoplot, Question 3: Autoplot Description, Question 4: ARIMA Model, Question 5: ARIMA Predictions, Asking Group Questions During Office Hours & Communicating to Group, Cleaning up Notebook with Headers and Structure as example for the rest of group

Yash Pandey Contributions: - Did the time series vizualization part which includes Quarterly and Monthly time series plot. - Implemented SNAIVE and LSTM model

Rahul Palivela Contributions:- - Did the Classical Decomposition and time series vizualization part which includes Yearly time series plot. - Implemented NAIVE model

1.2 0. Load In Data

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
train_df = pd.read_csv('train.csv')
```

1.3 1. Time Series Decomposition

[ ]:
```
train_df.head()
```

[ ]:
```
   id        date  store_nbr      family  sales  onpromotion
0   0  2013-01-01          1  AUTOMOTIVE    0.0            0
1   1  2013-01-01          1   BABY CARE    0.0            0
2   2  2013-01-01          1      BEAUTY    0.0            0
3   3  2013-01-01          1   BEVERAGES    0.0            0
4   4  2013-01-01          1       BOOKS    0.0            0
```

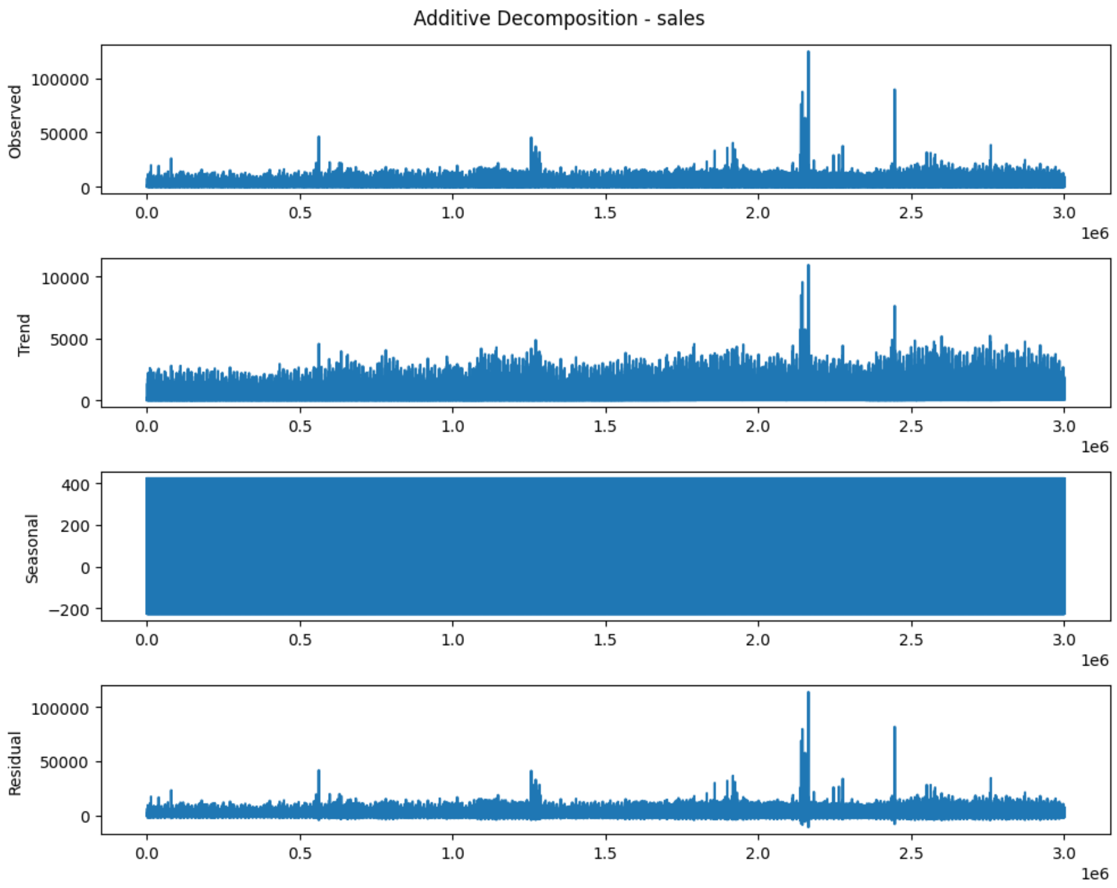
[ ]:
```
# Formatting the train_df so that we can do the decomp on it
df = pd.melt(train_df, id_vars=['date'], value_vars=['sales', 'onpromotion'],
                    var_name='metric', value_name='value')

# Display the data frame that will be used for decomp
df.head()
```

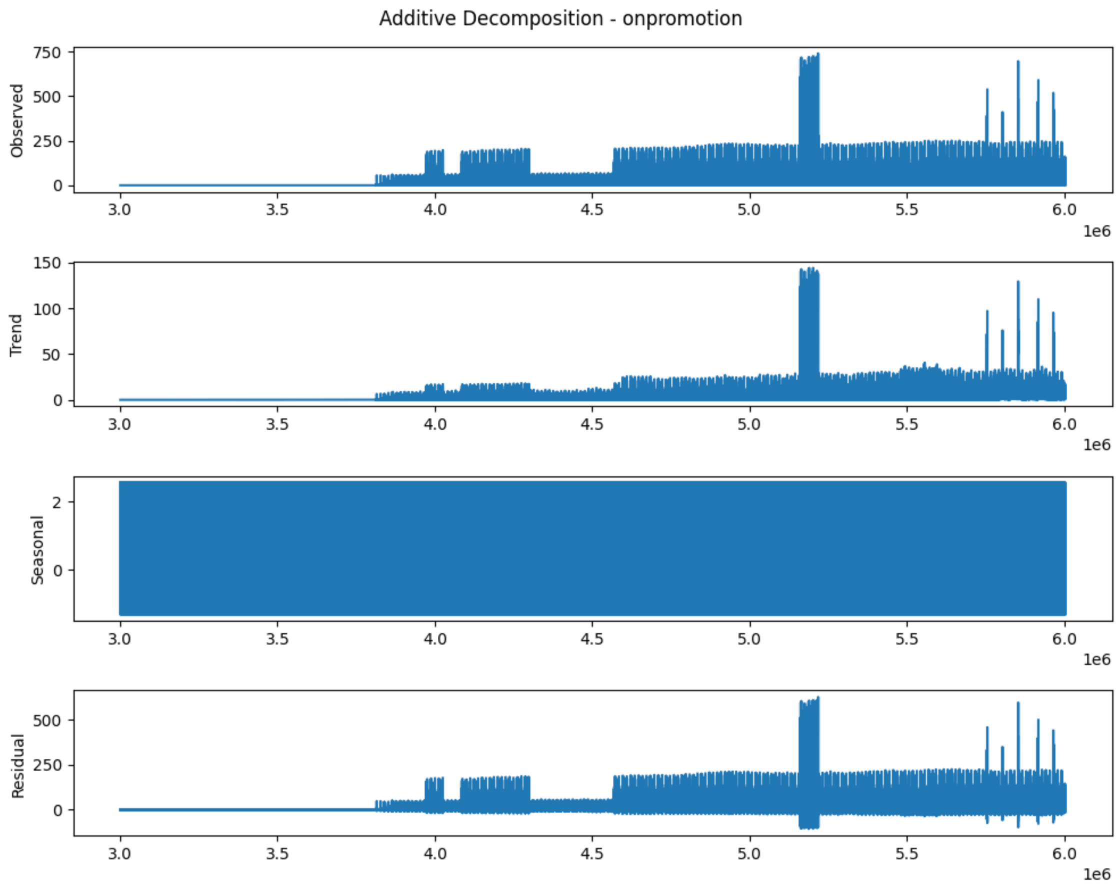
[ ]:
```
         date metric  value
0  2013-01-01  sales    0.0
1  2013-01-01  sales    0.0
2  2013-01-01  sales    0.0
3  2013-01-01  sales    0.0
4  2013-01-01  sales    0.0
```

1.3.1 Additive Decomposition

Using additive decomposition because there are values for zero present in sales in the data frame. Additive is used for time series that has these zero values.

[ ]:
```
from statsmodels.tsa.seasonal import seasonal_decompose

metrics = df['metric'].unique()

# Seasonal decomposition on sale and onpromotion
for metric in metrics:
    subset_df = df[df['metric'] == metric]
    decomposition = seasonal_decompose(subset_df['value'], model='additive',␣
  ↪period=12)

    # The seasonal components here, plotting them individually
    fig, axes = plt.subplots(4, 1, figsize=(10, 8))
    decomposition.observed.plot(ax=axes[0], legend=False)
    axes[0].set_ylabel('Observed')
    decomposition.trend.plot(ax=axes[1], legend=False)
    axes[1].set_ylabel('Trend')
    decomposition.seasonal.plot(ax=axes[2], legend=False)
    axes[2].set_ylabel('Seasonal')
    decomposition.resid.plot(ax=axes[3], legend=False)
    axes[3].set_ylabel('Residual')
    fig.suptitle(f'Additive Decomposition - {metric}')
    plt.tight_layout()
    plt.show()
```


Additive Decomposition - sales
Observed
Trend
Seasonal
Residual
1e6

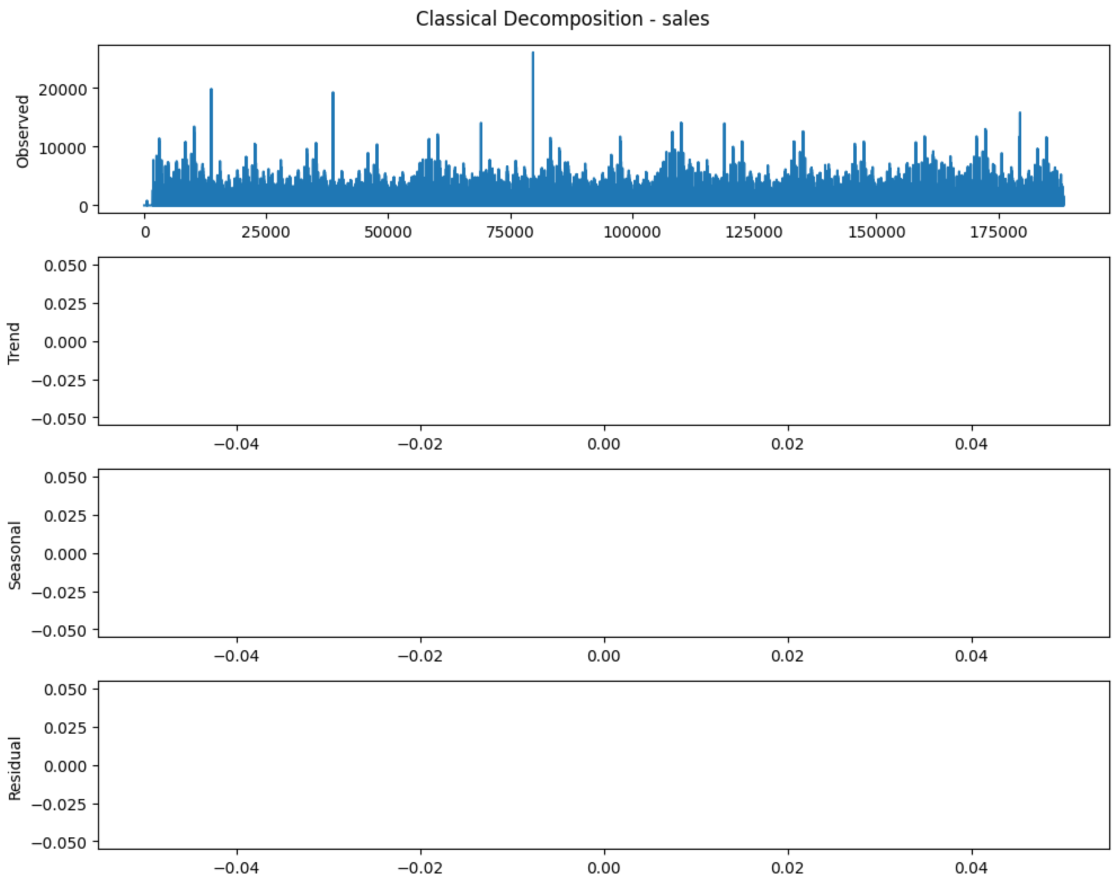


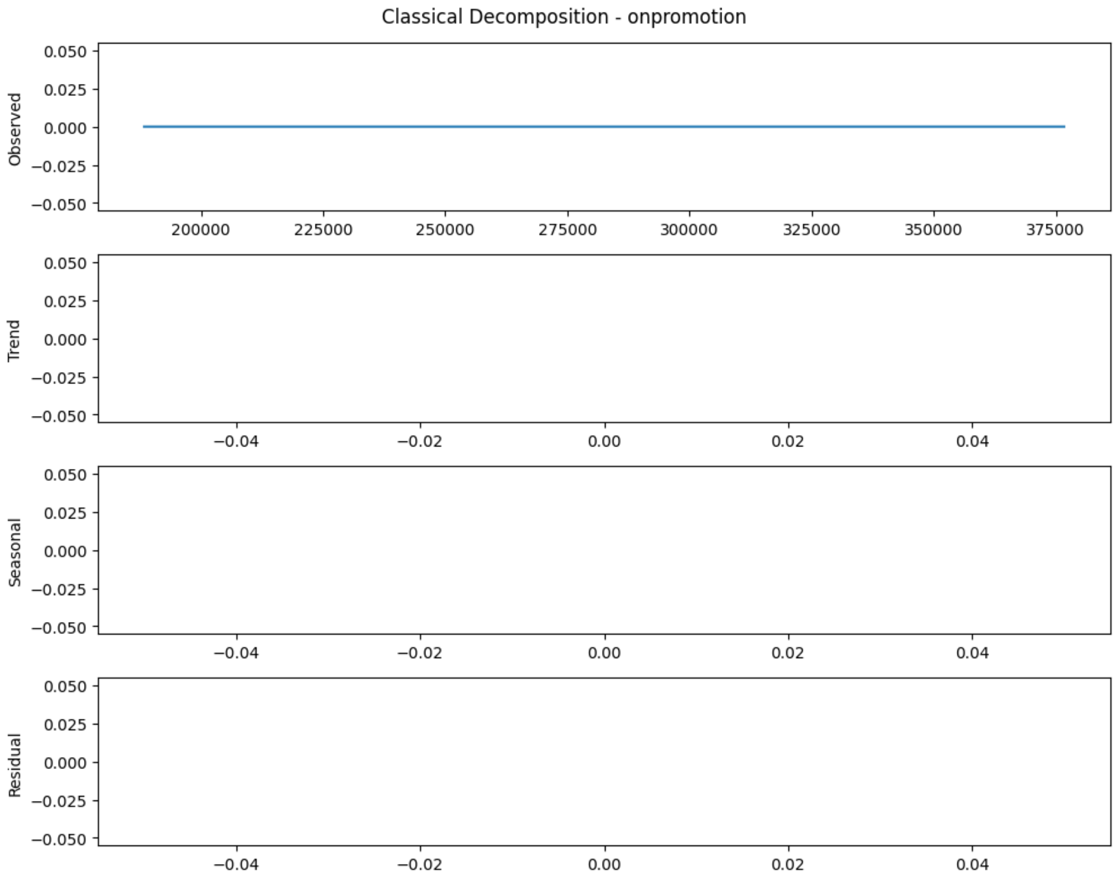
```
[ ]: from google.colab import drive
     drive.mount('/content/drive')
```

1.3.2 Classical Decompostion

```
[ ]: from statsmodels.tsa.filters.hp_filter import hpfilter
     import matplotlib.pyplot as plt

     # Converting the 'date' column to datetime
     df['date'] = pd.to_datetime(df['date'])

     for metric in df['metric'].unique():
         subset_df = df[df['metric'] == metric].copy()

         # Extracting the trend component using the Hodrick-Prescott filter
         cycle, trend = hpfilter(subset_df['value'], lamb=1600)
         subset_df['trend'] = trend

         # Detrending the data
```

```
subset_df['detrended'] = subset_df['value'] - subset_df['trend']

# Calculating seasonal component as mean detrended value for each period
seasonal = subset_df.groupby(subset_df['date'].dt.month)['detrended'].mean()
subset_df['seasonal'] = subset_df['date'].dt.month.map(seasonal)

# Calculating residuals
subset_df['resid'] = subset_df['detrended'] - subset_df['seasonal']

# Plotting
fig, axes = plt.subplots(4, 1, figsize=(10, 8))
subset_df['value'].plot(ax=axes[0], legend=False)
axes[0].set_ylabel('Observed')
subset_df['trend'].plot(ax=axes[1], legend=False)
axes[1].set_ylabel('Trend')
subset_df['seasonal'].plot(ax=axes[2], legend=False)
axes[2].set_ylabel('Seasonal')
subset_df['resid'].plot(ax=axes[3], legend=False)
axes[3].set_ylabel('Residual')
fig.suptitle(f'Classical Decomposition - {metric}')
plt.tight_layout()
plt.show()
```





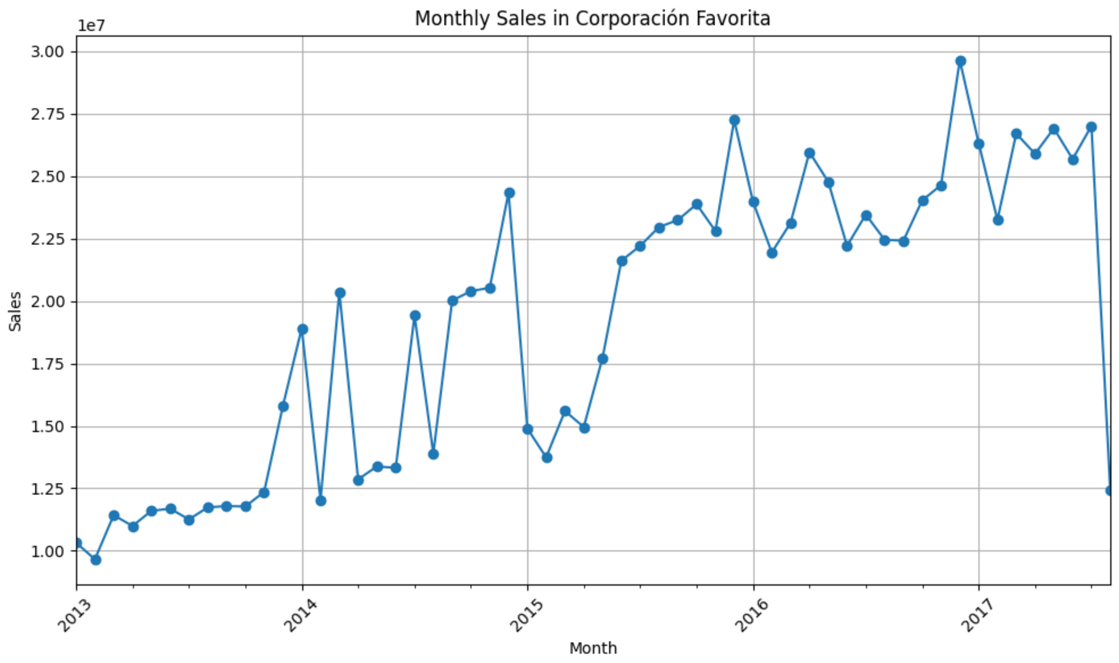
1.4 2. Time Series Visualization & 3. Description

1.4.1 Plot 1 Visualization & Plot Description

This first plot is the similar to `autoplot()` in R. This is using the data frame `train_df` as we are interested in projecting sales for our project and this data frame has `sales`

- The plot below displays how the data present here is from 2013 - 2017 and the chart shows how the data is missing months from 2017 since the x-axis tick marks aren't full like they are in the rest of the years.
- The plot displays trends, such as that sales are increasing overall from 2013 - 2017.
- The plot also shows seasonality, as the x-axis nears the end of a year there appears to be a peak. It looks like December appears to be the peak sales for the entire year. This is true for each year present in the data frame.

[ ]:
```
# Making sure that the date is a date time
train_df['date'] = pd.to_datetime(train_df['date'])

# grabbing the date by month for plotting
monthly_data = train_df.groupby(train_df['date'].dt.to_period('M'))['sales'].
  ↪sum()



# Plotting the train df below
plt.figure(figsize=(10, 6))
monthly_data.plot(marker='o')

# Focusing on the transactions present in the data frame
plt.title('Monthly Sales in Corporación Favorita')
plt.xlabel('Month')
plt.ylabel('Sales')

# Equivalent to autoplot in R
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()

plt.show()
```

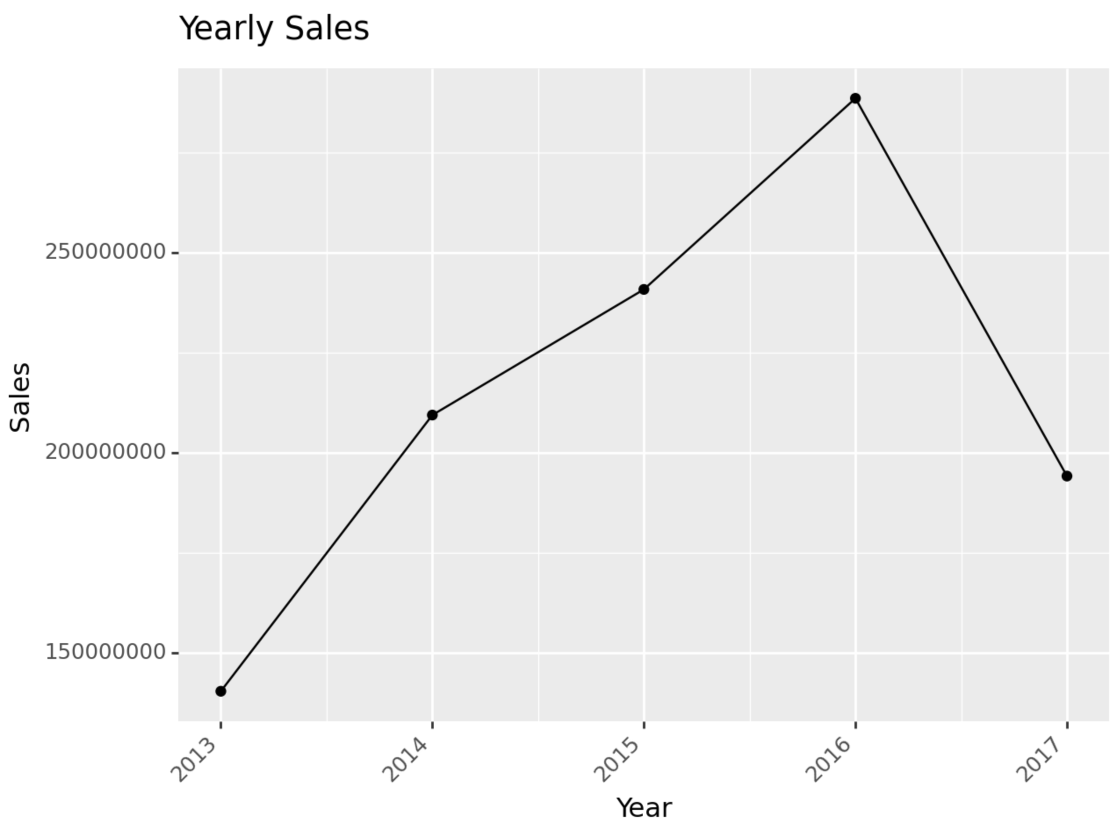


1.4.2 Plot 2 Visualization & Plot Description

These plots appear to be time series visualizations of sales data, broken down by year, quarter, and month.

Yearly Sales Plot: - This plot shows the annual total sales from 2013 to 2017. - There is a clear upward trend in sales from 2013 to 2016, suggesting growth year over year. - However, there's a significant drop in 2017. This could indicate a data issue, such as incomplete data for 2017, or a real decrease in sales. It would be important to understand the reason behind this decrease.

```
[ ]: from plotnine import ggplot, aes, geom_line, geom_point, labs, theme,␣
     ↪element_text, scale_x_discrete
     import calendar


     train_df['date'] = pd.to_datetime(train_df['date'])
     train_df['year'] = train_df['date'].dt.year
     train_df['quarter'] = train_df['date'].dt.quarter
     train_df['month'] = train_df['date'].dt.month

     # Aggregating sales data
     yearly_sales = train_df.groupby('year')['sales'].sum().reset_index()
     quarterly_sales = train_df.groupby(['year', 'quarter'])['sales'].sum().
     ↪reset_index()
     monthly_sales = train_df.groupby(['year', 'month'])['sales'].sum().reset_index()

     # Plotting Yearly Sales
     ggplot(yearly_sales, aes(x='year', y='sales')) + geom_line() + geom_point() + \
         labs(title='Yearly Sales', x='Year', y='Sales') + \
         theme(axis_text_x=element_text(rotation=45, hjust=1))
```

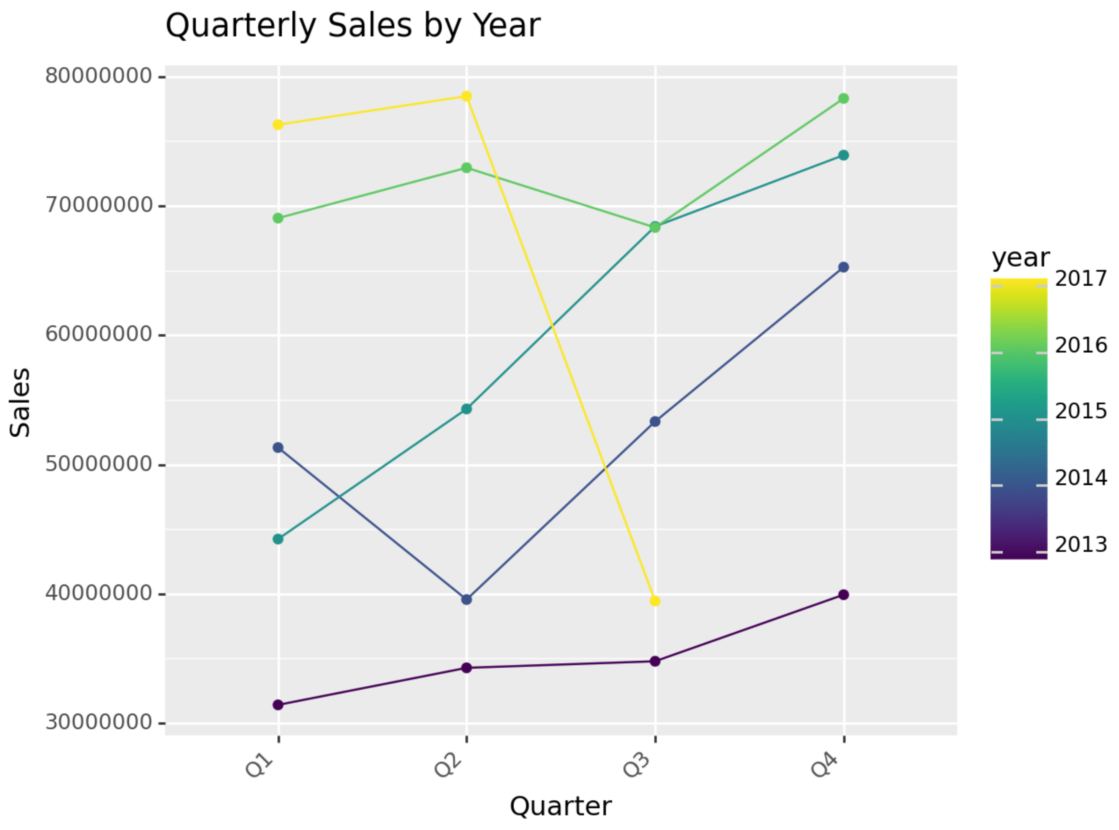


```
[ ]: <Figure Size: (640 x 480)>
```

1.4.3 Plot 3 Visualization & Plot Description

Quarterly Sales by Year: - Each line represents a year's worth of sales data, divided into quarters. - There's a clear seasonality with sales peaking in Q4 for most years, which is common in retail due to the holiday season. - There is an exception in one year where Q3 sales are the highest; this could be due to specific events or promotions during that period. - Year-on-year, there's variability in sales performance per quarter, indicating that while seasonality is a factor, other variables also significantly impact sales. - The sales trajectory for each year varies, suggesting differing yearly trends. For example, the year represented by the yellow line shows a sharp increase in sales from Q2 to Q3 and remains high in Q4, while the year represented by the blue line shows more consistent growth across quarters.

```
[ ]: (
         ggplot(quarterly_sales, aes(x='quarter', y='sales', color='year',␣
      ↪group='year')) +
         geom_line() +
         geom_point() +
         labs(title='Quarterly Sales by Year', x='Quarter', y='Sales') +
         scale_x_discrete(limits=range(1, 5), labels=['Q1', 'Q2', 'Q3', 'Q4']) +
         theme(axis_text_x=element_text(rotation=45, hjust=1))
     )
```



[ ]:
```
<Figure Size: (640 x 480)>
```

1.4.4 Plot 4 Visualization & Plot Description

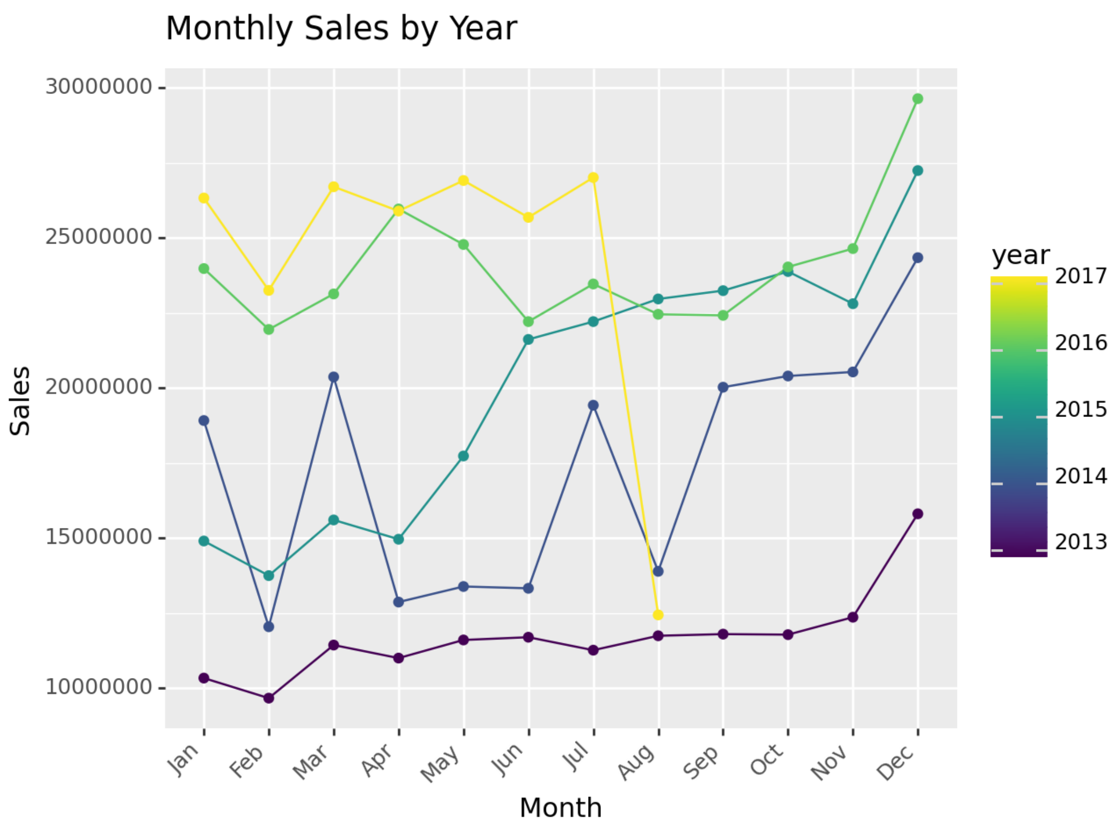
2. **Monthly Sales by Year**:
    - Sales trends are broken down month by month for each year, offering a more detailed view of seasonality.
    - As with the quarterly data, sales generally peak towards the end of the year, particularly in December.
    - There are some notable dips, for instance, the year represented by the blue line shows a significant decrease in sales during mid-year, which recovers by the end of the year.
    - The chart highlights the volatility of sales on a month-to-month basis. For example, the yellow line indicates a year where there was a sharp fall followed by a sharp rise from one month to the next.
    - Some years show more stability from month to month, whereas others show more pronounced fluctuations.

Overall, both charts illustrate the intricacies of sales patterns, with clear seasonal peaks towards the end of the year. The monthly chart, in particular, displays finer variations that could be tied to specific monthly events or promotions. It is also evident that the sales performance is not uniform across years, and each year has its unique profile of sales. Such insights could inform strategic business decisions, such as inventory planning, marketing strategies, and resource allocation to capitalize on peak sales periods and bolster sales during the lower-performing months.

For each plot, the team working on this project should investigate the following: - **Data Completeness**: Confirm whether the 2017 sales data is complete. The sudden drop in yearly sales is unusual and warrants a closer look. - **External Factors**: Identify any external factors that may have influenced sales patterns, such as economic conditions, changes in consumer behavior, or company initiatives like marketing campaigns.

- **Data Quality**: Check for errors in the data that may affect the visualizations, such as missing values or incorrect aggregation.
- **Statistical Testing**: Consider performing statistical tests for seasonality and trend to quantify the observed patterns.
- **Model Relevance**: Based on these visualizations, consider the appropriateness of different time series models that can account for the observed trends and seasonality.

[ ]:
```python
import pandas as pd

monthly_sales['month_factor'] = pd.Categorical(monthly_sales['month'])

(
    ggplot(monthly_sales, aes(x='month', y='sales', color='year',␣
 ↪group='year')) +
    geom_line() +
    geom_point() +
    labs(title='Monthly Sales by Year', x='Month', y='Sales') +
    scale_x_discrete(limits=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],␣
 ↪labels=calendar.month_abbr[1:13]) +
    theme(axis_text_x=element_text(rotation=45, hjust=1))
)
```



```
[ ]: <Figure Size: (640 x 480)>
```

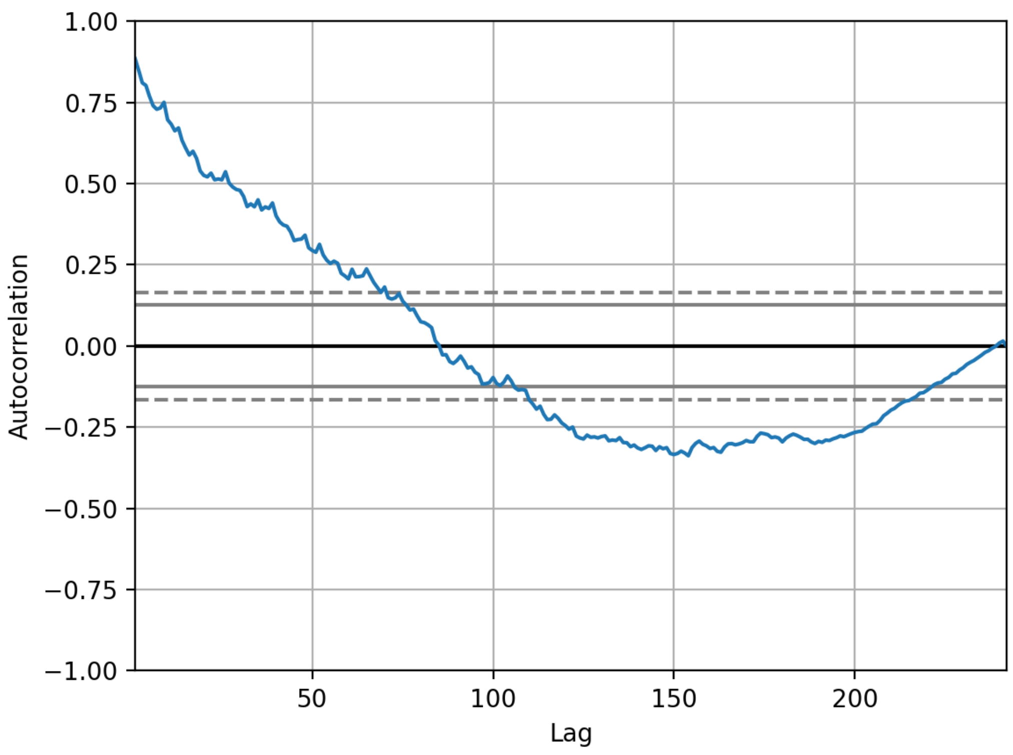
1.4.5 Plot 5 Visualization & Plot Description

The autocorrelation plot below shows the trend that we saw during the initial EDA. We see repeating trends at certain intervals, and positive and negative autocorrelations observed at different lags.

- There is a strong positive autocorrelation up until lag 50.
- Next we can see how there's a slow decline from lag 50 until lag 150
- From lag 100 until lag 150 we can see how there's a negative autocorrelation and that pattern stays consistant after
- Around lag 150 and lag 200 we can see how the trend is consistant here, with the negative autocorrelation
- After 200 there's a return to 0 for the autocorrelation

```
[ ]: from pandas.plotting import autocorrelation_plot
     import matplotlib.pyplot as plt

     # autocorrelation for the 'sales' column since it's what we'll perform the␣
      ↪model on
```

```
autocorrelation_plot(train_df_weekly['sales'])
plt.show()
```



1.5 4. TS Models

1.6 ARIMA Model

1.6.1 ARIMA Model Pre-Work

First, a new data frame is created `train_df_weekly` which is looking at weekly data, because it'll be interested to see what the ARIMA result is for weekly data

[ ]:
```
# This is creating a new data frame so that we can run the model
train_df['date'] = pd.to_datetime(train_df['date'])

# setting the date as index
train_df.set_index('date', inplace=True)

# Interested in a model that looks at weekly data
train_df_weekly = train_df.resample('W').sum()
```

```
# Adjusting 'sales' column to numeric, replacing non-numeric values with 0
train_df_weekly['sales'] = pd.to_numeric(train_df_weekly['sales'],␣
 ↪errors='coerce').fillna(0)

# Adjusting 'sales' again, astype because just the to_numeric wasn't actually␣
 ↪making this a numeric field
train_df_weekly['sales'] = train_df_weekly['sales'].astype(int)
```

```
<ipython-input-16-a5ef046cef1e>:8: FutureWarning: The default value of
numeric_only in DataFrameGroupBy.sum is deprecated. In a future version,
numeric_only will default to False. Either specify numeric_only or select only
columns which should be valid for the function.
```

[ ]:
```
# Verifying that int64 is the data type for sales so that we can run␣
 ↪autocorrelation on it
print(train_df_weekly.dtypes)
```

```
id               int64
store_nbr        int64
sales            int64
onpromotion      int64
dtype: object
```

1.6.2 ARIMA Model Grid Search

We're interested in seeing the results of an ARIMA model and are using a grid search to find the best parameters for p,d,q

[ ]:
```
from math import sqrt
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima.model import ARIMA

# changing data type to float
train_df_weekly['sales'] = train_df_weekly['sales'].astype(float)

# Defining the function to evaluate an ARIMA model for a given order (p, d, q)
def evaluate_arima_model(train_data, test_data, arima_order):
    history = [x for x in train_data]
    predictions = list()
    for t in range(len(test_data)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit()
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test_data[t])
    rmse = sqrt(mean_squared_error(test_data, predictions))
    return rmse
```

```
# Defining the function to evaluate combinations of p, d, and q values for an␣
↪ARIMA model
def evaluate_models(train_data, test_data, p_values, d_values, q_values):
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p, d, q)
                try:
                    rmse = evaluate_arima_model(train_data, test_data, order)
                    if rmse < best_score:
                        best_score, best_cfg = rmse, order
                    print('ARIMA%s RMSE=%.3f' % (order, rmse))
                except:
                    continue
    print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')

p_values = [0, 1, 2, 4, 6, 8, 10]
d_values = range(0, 3)
q_values = range(0, 3)

split_ratio = 0.66
split_index = int(len(train_df_weekly['sales']) * split_ratio)
train_data = train_df_weekly['sales'][:split_index]
test_data = train_df_weekly['sales'][split_index:]
evaluate_models(train_data, test_data, p_values, d_values, q_values)
```

```
ARIMA(0, 0, 0) RMSE=810842.824
ARIMA(0, 0, 1) RMSE=804003.298
ARIMA(0, 0, 2) RMSE=824084.699
ARIMA(0, 1, 0) RMSE=841421.228
ARIMA(0, 1, 1) RMSE=837225.507
ARIMA(0, 1, 2) RMSE=836828.784
ARIMA(0, 2, 0) RMSE=743907.486
ARIMA(0, 2, 1) RMSE=751798.793
ARIMA(0, 2, 2) RMSE=788402.605
ARIMA(1, 0, 0) RMSE=817309.993
ARIMA(1, 0, 1) RMSE=839650.821
ARIMA(1, 0, 2) RMSE=837394.193
ARIMA(1, 1, 0) RMSE=837538.846
ARIMA(1, 1, 1) RMSE=837143.636
ARIMA(1, 1, 2) RMSE=842897.417
ARIMA(1, 2, 0) RMSE=750819.099
ARIMA(1, 2, 1) RMSE=809864.480
ARIMA(1, 2, 2) RMSE=973330.306
ARIMA(2, 0, 0) RMSE=830758.836
ARIMA(2, 0, 1) RMSE=841698.858
ARIMA(2, 0, 2) RMSE=801566.771
ARIMA(2, 1, 0) RMSE=835877.878
ARIMA(2, 1, 1) RMSE=838937.443
ARIMA(2, 1, 2) RMSE=812959.373
ARIMA(2, 2, 0) RMSE=756854.879
ARIMA(2, 2, 1) RMSE=816370.476
ARIMA(4, 0, 0) RMSE=810957.162
ARIMA(4, 0, 1) RMSE=810768.784
ARIMA(4, 1, 0) RMSE=831294.946
ARIMA(4, 1, 1) RMSE=923165.362
ARIMA(4, 1, 2) RMSE=829512.724
ARIMA(4, 2, 2) RMSE=786586.907
ARIMA(6, 0, 0) RMSE=835771.742
ARIMA(6, 0, 1) RMSE=817437.953
ARIMA(6, 0, 2) RMSE=825592.596
ARIMA(6, 1, 0) RMSE=856389.934
ARIMA(6, 1, 1) RMSE=850194.588
ARIMA(6, 1, 2) RMSE=810877.068
ARIMA(6, 2, 1) RMSE=1010559.895
ARIMA(8, 0, 0) RMSE=837499.486
ARIMA(8, 0, 2) RMSE=813383.989
ARIMA(8, 1, 1) RMSE=930195.145
ARIMA(8, 1, 2) RMSE=925115.773
ARIMA(8, 2, 0) RMSE=954696.818
ARIMA(8, 2, 2) RMSE=7142913.759
ARIMA(10, 0, 0) RMSE=886890.631
ARIMA(10, 0, 1) RMSE=881233.346
ARIMA(10, 0, 2) RMSE=873391.853
ARIMA(10, 1, 0) RMSE=872901.734
ARIMA(10, 1, 1) RMSE=862020.136
ARIMA(10, 1, 2) RMSE=860371.977
ARIMA(10, 2, 0) RMSE=724300.454
ARIMA(10, 2, 1) RMSE=877143.380
ARIMA(10, 2, 2) RMSE=890053.176
Best ARIMA(10, 2, 0) RMSE=724300.454
```

After determining the best parameters to use, decided to go with `Best ARIMA(8, 0, 2) RMSE=669833.771`

[ ]:
```
split_ratio = 0.66
split_index = int(len(train_df_weekly['sales']) * split_ratio)
train_data = train_df_weekly['sales'][:split_index]
test_data = train_df_weekly['sales'][split_index:]
```

We're looking at the summary for the arima model using the `order=(8,0,2)` as this was the best given the grid search that we did.

```
[ ]: # Fitting ARIMA model
     model = ARIMA(train_data, order=(8, 0, 2))
     model_fit = model.fit()

     # Summary of the fitted model
     print(model_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  sales   No. Observations:                  159
Model:                 ARIMA(8, 0, 2)   Log Likelihood               -2289.188
Date:                Sun, 31 Mar 2024   AIC                           4602.377
Time:                        15:46:09   BIC                           4639.203
Sample:                    01-06-2013   HQIC                          4617.332
                         - 01-17-2016
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const       3.801e+06   5.16e-10   7.37e+15      0.000     3.8e+06     3.8e+06
ar.L1          0.9037      0.110      8.252      0.000       0.689       1.118
ar.L2         -0.6701      0.154     -4.338      0.000      -0.973      -0.367
ar.L3          0.4056      0.217      1.866      0.062      -0.020       0.832
ar.L4          0.2671      0.201      1.331      0.183      -0.126       0.660
ar.L5         -0.1285      0.166     -0.774      0.439      -0.454       0.197
ar.L6          0.0710      0.173      0.411      0.681      -0.268       0.410
ar.L7         -0.1380      0.150     -0.922      0.356      -0.431       0.155
ar.L8          0.2874      0.092      3.141      0.002       0.108       0.467
ma.L1         -0.1663      0.081     -2.043      0.041      -0.326      -0.007
ma.L2          0.9485      0.071     13.294      0.000       0.809       1.088
sigma2      2.196e+11   7.97e-13   2.76e+23      0.000     2.2e+11     2.2e+11
================================================================================
===
Ljung-Box (L1) (Q):                   0.79   Jarque-Bera (JB):
73.55
Prob(Q):                              0.37   Prob(JB):
0.00
Heteroskedasticity (H):               3.17   Skew:
0.78
Prob(H) (two-sided):                  0.00   Kurtosis:
5.94
================================================================================
===

Warnings:
```

```
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
[2] Covariance matrix is singular or near-singular, with condition number
3.99e+39. Standard errors may be unstable.

/usr/local/lib/python3.10/dist-packages/statsmodels/base/model.py:607:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

1.7 5. Predictions

1.8 ARIMA Predictions

The two plots that follow are the predictions for the ARIMA model. The difference between these is what date range is being compared against the forecast.

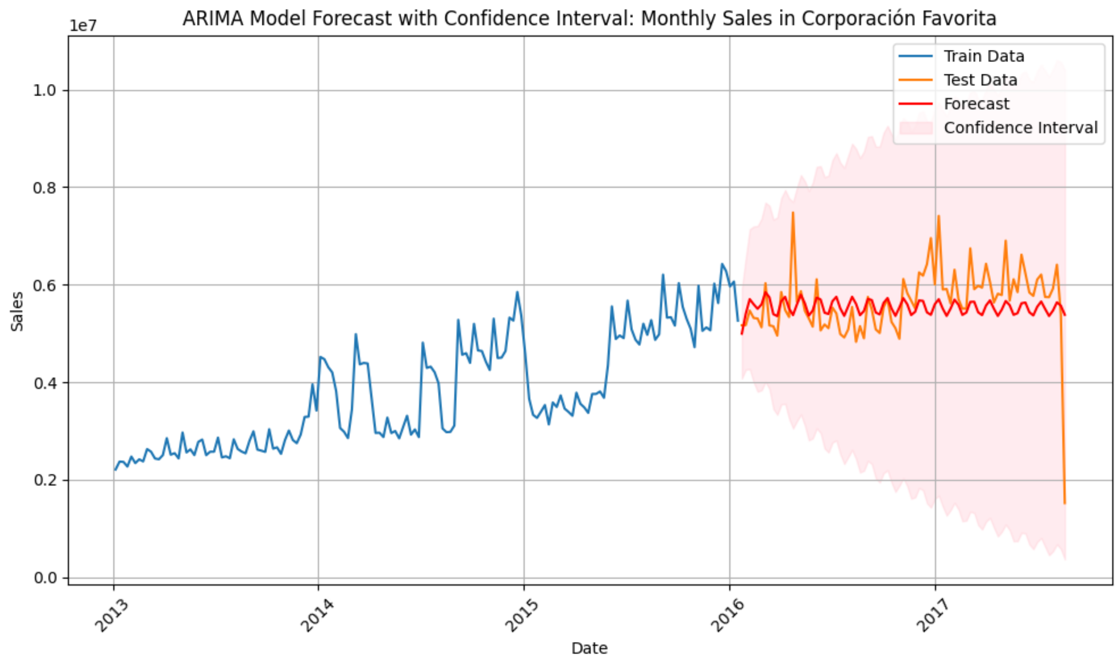
The prediction plot below demonstrates the predictions during the same time period as the test data, meaning that is the prediction for data that is inside of the dataset.

- The graph shows the train data which covers weekly data from 2013 - 2016, the test data is from 2016 until the end of the data range at 2017, and this is different than the prior plot because the forecast is for the test data, we are comparing in a date range available inside the data set
- The forecast shows a comparable value, compared to the test data set, it looks like it's in between the test data. Meaning that it isn't on the upper bound or the lower bound of the test data, it's hanging out in the middle.
- The forecast shading also shows how the bounds are broad, which matches the data that we see here inside of the test data.

[ ]:
```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

train_data = train_df_weekly['sales'][:split_index]
test_data = train_df_weekly['sales'][split_index:]

# Fitting an ARIMA model to the training data
model = ARIMA(train_data, order=(8, 0, 2))
model_fit = model.fit()

# Get the forecasted values and confidence intervals
forecast_results = model_fit.get_forecast(steps=len(test_data), alpha=0.05)  #␣
 ↪Adjust alpha as needed
forecast_values = forecast_results.predicted_mean
forecast_conf_int = forecast_results.conf_int()

# Converting Pandas Series to NumPy arrays
```

```
train_dates = train_data.index.values
train_values = train_data.values
test_dates = test_data.index.values
test_values = test_data.values

# Get the forecasted dates starting from the last date in the test set
forecast_dates = pd.date_range(start=test_dates[0], periods=len(test_data),␣
  ↪freq='W')

# Ploting original data and predictions
plt.figure(figsize=(10, 6))
plt.plot(train_dates, train_values, label='Train Data')
plt.plot(test_dates, test_values, label='Test Data')
plt.plot(forecast_dates, forecast_values, label='Forecast', color='red')

# Filling area between upper and lower confidence intervals
plt.fill_between(forecast_dates, forecast_conf_int.iloc[:, 0],␣
  ↪forecast_conf_int.iloc[:, 1], color='pink', alpha=0.3, label='Confidence␣
  ↪Interval')

plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('ARIMA Model Forecast with Confidence Interval: Monthly Sales in␣
  ↪Corporación Favorita')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/base/model.py:607:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/usr/local/lib/python3.10/dist-
packages/statsmodels/tsa/statespace/representation.py:374: FutureWarning:
Unknown keyword arguments: dict_keys(['alpha']).Passing unknown keyword
arguments will raise a TypeError beginning in version 0.15.
  warnings.warn(msg, FutureWarning)
```

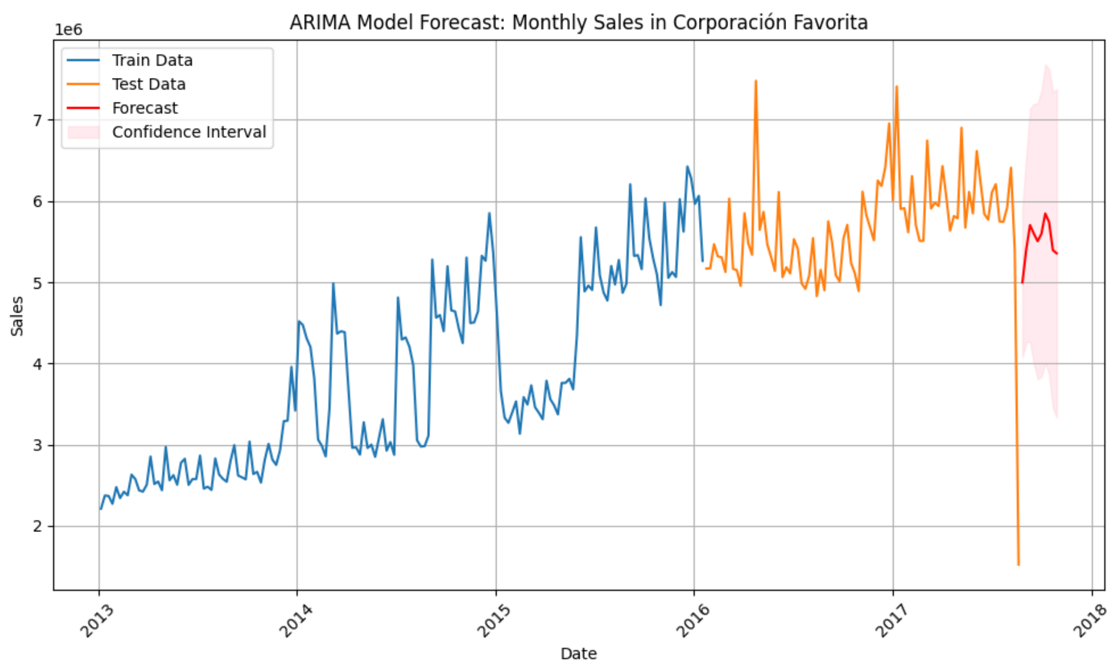


This also an ARIMA prediction.

- The graph shows the train data which covers weekly data from 2013 - 2016, the test data is from 2016 until the end of the data range at 2017, while the forecast is for data after the date range available in this data set
- The forecast shows a decrease and trends more similar to the end of the test data, the confidence interval also shows a wider range here

```
[ ]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Converting to NumPy arrays
train_dates = train_data.index.values
train_values = train_data.values
test_dates = test_data.index.values
test_values = test_data.values

test_dates = np.array(test_dates)
test_values = np.array(test_values)

# Looking at a few steps into the future, this is after the date range␣
↪available in our train_df
forecast_steps = 10

# The predictions
```

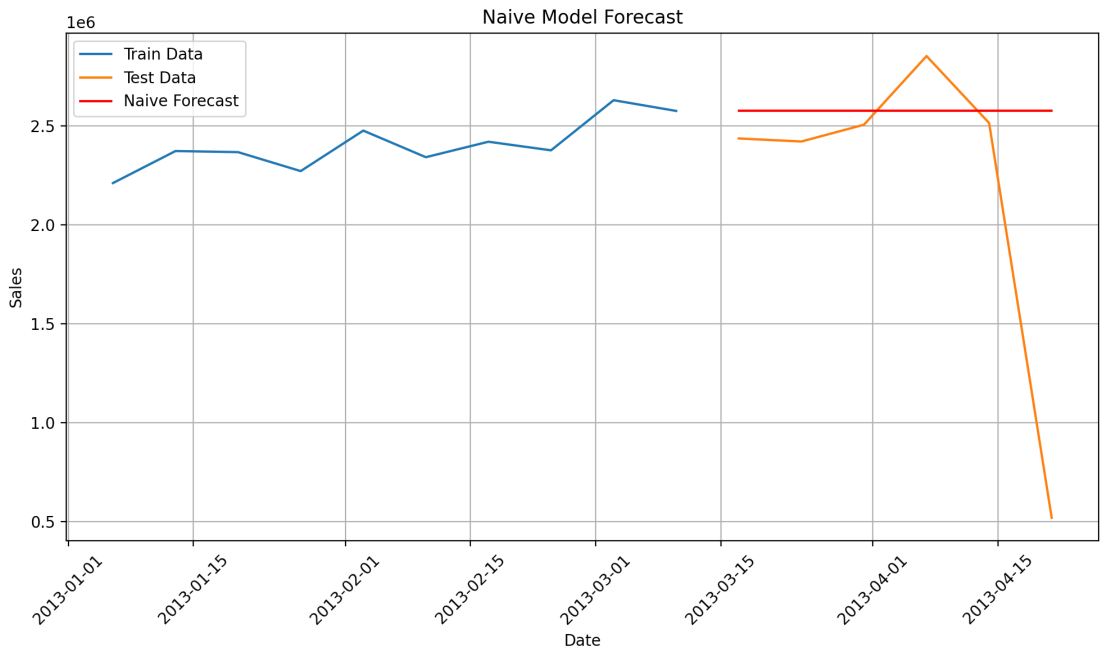
```python
forecast_values = model_fit.forecast(steps=forecast_steps)

# Forecast starts from the last date in the test set
forecast_dates = pd.date_range(start=test_dates[-1], periods=forecast_steps +␣
↪1, freq='W')[1:]

# Converting forecast_dates and forecast_values to numpy arrays
forecast_dates = np.array(forecast_dates)
forecast_values = np.array(forecast_values)

# Predictions plot
plt.figure(figsize=(10, 6))
plt.plot(train_dates, train_values, label='Train Data')
plt.plot(test_dates, test_values, label='Test Data')
plt.plot(forecast_dates, forecast_values, label='Forecast', color='red')

# Showing the upper and lower confidence intervals
forecast_conf_int = model_fit.get_forecast(steps=forecast_steps).conf_int()
plt.fill_between(forecast_dates, forecast_conf_int.iloc[:, 0].values,␣
↪forecast_conf_int.iloc[:, 1].values, color='pink', alpha=0.3,␣
↪label='Confidence Interval')

# The final prediction plot
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('ARIMA Model Forecast: Monthly Sales in Corporación Favorita')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



```
[ ]: import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.metrics import mean_squared_error
     from math import sqrt


     # The training and test data split has already been defined
     train_data = train_df_weekly['sales'][:split_index]
     test_data = train_df_weekly['sales'][split_index:]

     # The naive forecast will be the last observed value from the training set
     last_observed = train_data.iloc[-1]
     naive_forecast = [last_observed] * len(test_data)

     # Calculating RMSE for the naive forecast
     rmse = sqrt(mean_squared_error(test_data, naive_forecast))
     print('Naive forecast RMSE: %.3f' % rmse)

     # Visualizing the train data, test data, and the naive forecast
     plt.figure(figsize=(10, 6))
     train_dates = train_data.index.values
     test_dates = test_data.index.values

     plt.plot(train_dates, train_data.values, label='Train Data')
     plt.plot(test_dates, test_data.values, label='Test Data')
```

```python
plt.plot(test_dates, naive_forecast, label='Naive Forecast', color='red')

plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Naive Model Forecast')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
Naive forecast RMSE: 852680.945
```



1.9 NAIVE

Naive model is the simplest form of forecasting which assumes that the future will exactly replicate the most recent observed data point. It doesn't account for any trends, seasonality, or patterns in the data.

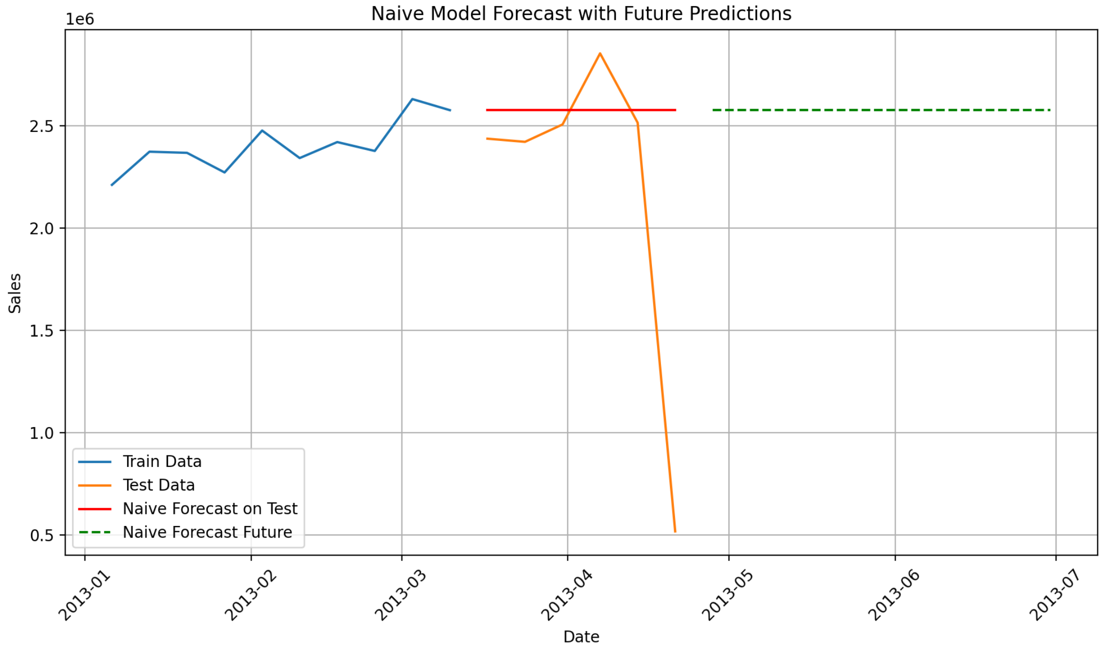
So for each predicted time point, the Naive model simply says "the future sales will be exactly what the last sales figure was." If the last observed sales were 100 units, then all future sales, according to the Naive model, are forecasted to be 100 units as well. This is why the forecasted values are a flat line — the model does not change its prediction regardless of how far into the future it's forecasting.

This model is helpful as a baseline; more sophisticated models should ideally perform better than this. If a more complex model does not perform better than the Naive model, it suggests that the model is not capturing the underlying patterns in the data effectively.

```
[ ]: import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.metrics import mean_squared_error
     from math import sqrt
     import numpy as np

     # `train_df_weekly` and `split_index` are already defined

     # The training and test data split has already been defined
     train_data = train_df_weekly['sales'][:split_index]
     test_data = train_df_weekly['sales'][split_index:]

     # The naive forecast will be the last observed value from the training set
     last_observed = train_data.iloc[-1]
     naive_forecast_test = [last_observed] * len(test_data)

     # Calculating RMSE for the naive forecast on the test set
     rmse = sqrt(mean_squared_error(test_data, naive_forecast_test))
     print('Naive forecast RMSE on the test set: %.3f' % rmse)

     # Future predictions beyond the test set
     forecast_horizon = 10  # Number of future periods to forecast
     naive_forecast_future = [last_observed] * forecast_horizon
     future_dates = pd.date_range(start=test_data.index[-1],␣
      ↪periods=forecast_horizon + 1, freq='W')[1:]

     # Converting to numpy arrays
     future_dates = np.array(future_dates)
     naive_forecast_future = np.array(naive_forecast_future)

     # Visualizing the train data, test data, and the naive forecast
     plt.figure(figsize=(10, 6))
     train_dates = train_data.index.values
     test_dates = test_data.index.values

     plt.plot(train_dates, train_data.values, label='Train Data')
     plt.plot(test_dates, test_data.values, label='Test Data')
     plt.plot(test_dates, naive_forecast_test, label='Naive Forecast on Test',␣
      ↪color='red')
     plt.plot(future_dates, naive_forecast_future, label='Naive Forecast Future',␣
      ↪color='green', linestyle='--')

     plt.xlabel('Date')
     plt.ylabel('Sales')
     plt.title('Naive Model Forecast with Future Predictions')
```

```python
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
Naive forecast RMSE on the test set: 852680.945
```



1.9.1 Naive Model Forecast with Future Predictions

- **Blue Line (Train Data):** Represents the historical sales data used to train the model, showing actual sales up to the split between training and testing data.
- **Orange Line (Test Data):** Indicates the actual sales data reserved for testing the model, following the last point in the training data.
- **Red Line (Naive Forecast on Test):** Shows the Naive model predictions for the test set. The forecast is a horizontal line, suggesting that future values are predicted to be constant, equal to the last observed value in the training data.
- **Green Dashed Line (Naive Forecast Future):** Depicts the Naive model's forecasts for future time periods beyond the test set. The prediction is presented as a dashed line, extending the last observed sales value into the future, under the assumption that sales levels will persist indefinitely without change.

2 LSTM

LSTM (Long Short-Term Memory)

LSTM is a type of recurrent neural network capable of learning long-term dependencies in data. It's particularly well-suited for sequences and time series data because of its ability to maintain information in its memory for extended periods. The LSTM model displayed here captures complex patterns in the historical sales data, which may include trends and seasonality.

[ ]:
```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from math import sqrt
import matplotlib.pyplot as plt



# Resampling the data to weekly frequency, summing up the sales
train_df_weekly = train_df['sales'].resample('W').sum().values.reshape(-1, 1)

# Normalizing the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
train_df_weekly_scaled = scaler.fit_transform(train_df_weekly)

# Splitting into train and test sets
train_size = int(len(train_df_weekly_scaled) * 0.67)
test_size = len(train_df_weekly_scaled) - train_size
train, test = train_df_weekly_scaled[0:train_size,:],␣
 ↪train_df_weekly_scaled[train_size:len(train_df_weekly_scaled),:]

# Converting an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

# Reshaping into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)

# Reshaping input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
```

```python
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

# Creating and fitting the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)

# Makeing predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)

# Inverting predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])



# Adjust the plot indices for training predictions
trainPredictPlot = np.empty_like(train_df_weekly_scaled)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict) + look_back, :] = trainPredict

# Adjust the plot indices for test predictions
testPredictPlot = np.empty_like(train_df_weekly_scaled)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:
 ↪len(train_df_weekly_scaled)-1, :] = testPredict

# Plot baseline and predictions
plt.figure(figsize=(15, 6))
plt.plot(scaler.inverse_transform(train_df_weekly_scaled), label='Actual Data')
plt.plot(trainPredictPlot, label='Train Prediction')
plt.plot(testPredictPlot, label='Test Prediction')
plt.legend()
plt.show()


# # Calculate root mean squared error
trainScore = sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

```
Epoch 1/100
```

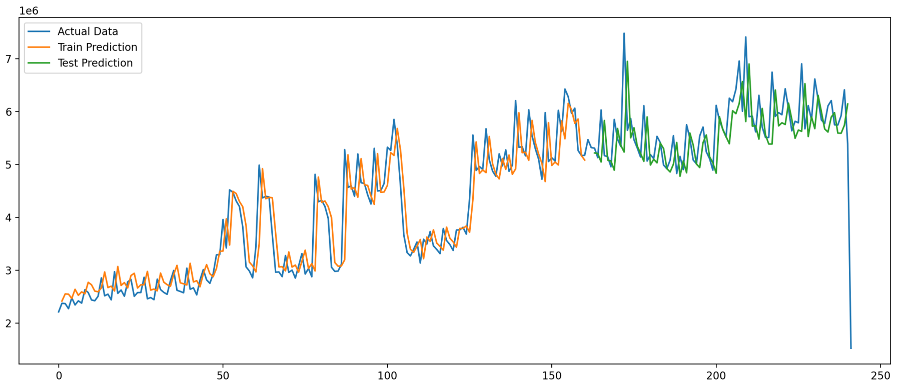
```
160/160 - 5s - loss: 0.1575 - 5s/epoch - 29ms/step
Epoch 2/100
160/160 - 0s - loss: 0.0535 - 357ms/epoch - 2ms/step
Epoch 3/100
160/160 - 0s - loss: 0.0308 - 397ms/epoch - 2ms/step
Epoch 4/100
160/160 - 1s - loss: 0.0271 - 503ms/epoch - 3ms/step
Epoch 5/100
160/160 - 1s - loss: 0.0244 - 804ms/epoch - 5ms/step
Epoch 6/100
160/160 - 1s - loss: 0.0220 - 844ms/epoch - 5ms/step
Epoch 7/100
160/160 - 1s - loss: 0.0195 - 698ms/epoch - 4ms/step
Epoch 8/100
160/160 - 1s - loss: 0.0173 - 1s/epoch - 7ms/step
Epoch 9/100
160/160 - 1s - loss: 0.0152 - 826ms/epoch - 5ms/step
Epoch 10/100
160/160 - 1s - loss: 0.0134 - 514ms/epoch - 3ms/step
Epoch 11/100
160/160 - 1s - loss: 0.0117 - 558ms/epoch - 3ms/step
Epoch 12/100
160/160 - 1s - loss: 0.0106 - 516ms/epoch - 3ms/step
Epoch 13/100
160/160 - 0s - loss: 0.0094 - 485ms/epoch - 3ms/step
Epoch 14/100
160/160 - 0s - loss: 0.0085 - 452ms/epoch - 3ms/step
Epoch 15/100
160/160 - 0s - loss: 0.0080 - 287ms/epoch - 2ms/step
Epoch 16/100
160/160 - 0s - loss: 0.0076 - 348ms/epoch - 2ms/step
Epoch 17/100
160/160 - 0s - loss: 0.0072 - 396ms/epoch - 2ms/step
Epoch 18/100
160/160 - 0s - loss: 0.0071 - 264ms/epoch - 2ms/step
Epoch 19/100
160/160 - 0s - loss: 0.0069 - 443ms/epoch - 3ms/step
Epoch 20/100
160/160 - 0s - loss: 0.0068 - 406ms/epoch - 3ms/step
Epoch 21/100
160/160 - 0s - loss: 0.0068 - 331ms/epoch - 2ms/step
Epoch 22/100
160/160 - 0s - loss: 0.0067 - 482ms/epoch - 3ms/step
Epoch 23/100
160/160 - 0s - loss: 0.0069 - 479ms/epoch - 3ms/step
Epoch 24/100
160/160 - 0s - loss: 0.0068 - 439ms/epoch - 3ms/step
Epoch 25/100
```

```
160/160 - 0s - loss: 0.0067 - 375ms/epoch - 2ms/step
Epoch 26/100
160/160 - 0s - loss: 0.0067 - 368ms/epoch - 2ms/step
Epoch 27/100
160/160 - 0s - loss: 0.0067 - 416ms/epoch - 3ms/step
Epoch 28/100
160/160 - 0s - loss: 0.0067 - 386ms/epoch - 2ms/step
Epoch 29/100
160/160 - 0s - loss: 0.0068 - 297ms/epoch - 2ms/step
Epoch 30/100
160/160 - 0s - loss: 0.0066 - 443ms/epoch - 3ms/step
Epoch 31/100
160/160 - 0s - loss: 0.0068 - 263ms/epoch - 2ms/step
Epoch 32/100
160/160 - 0s - loss: 0.0069 - 433ms/epoch - 3ms/step
Epoch 33/100
160/160 - 1s - loss: 0.0068 - 509ms/epoch - 3ms/step
Epoch 34/100
160/160 - 1s - loss: 0.0068 - 706ms/epoch - 4ms/step
Epoch 35/100
160/160 - 1s - loss: 0.0068 - 648ms/epoch - 4ms/step
Epoch 36/100
160/160 - 1s - loss: 0.0068 - 658ms/epoch - 4ms/step
Epoch 37/100
160/160 - 1s - loss: 0.0067 - 546ms/epoch - 3ms/step
Epoch 38/100
160/160 - 0s - loss: 0.0068 - 477ms/epoch - 3ms/step
Epoch 39/100
160/160 - 0s - loss: 0.0067 - 288ms/epoch - 2ms/step
Epoch 40/100
160/160 - 0s - loss: 0.0067 - 454ms/epoch - 3ms/step
Epoch 41/100
160/160 - 0s - loss: 0.0067 - 440ms/epoch - 3ms/step
Epoch 42/100
160/160 - 0s - loss: 0.0068 - 448ms/epoch - 3ms/step
Epoch 43/100
160/160 - 1s - loss: 0.0068 - 745ms/epoch - 5ms/step
Epoch 44/100
160/160 - 1s - loss: 0.0067 - 669ms/epoch - 4ms/step
Epoch 45/100
160/160 - 1s - loss: 0.0067 - 702ms/epoch - 4ms/step
Epoch 46/100
160/160 - 1s - loss: 0.0069 - 736ms/epoch - 5ms/step
Epoch 47/100
160/160 - 1s - loss: 0.0068 - 692ms/epoch - 4ms/step
Epoch 48/100
160/160 - 0s - loss: 0.0066 - 469ms/epoch - 3ms/step
Epoch 49/100
```

```
160/160 - 0s - loss: 0.0067 - 435ms/epoch - 3ms/step
Epoch 50/100
160/160 - 0s - loss: 0.0067 - 446ms/epoch - 3ms/step
Epoch 51/100
160/160 - 0s - loss: 0.0066 - 436ms/epoch - 3ms/step
Epoch 52/100
160/160 - 0s - loss: 0.0068 - 369ms/epoch - 2ms/step
Epoch 53/100
160/160 - 0s - loss: 0.0067 - 390ms/epoch - 2ms/step
Epoch 54/100
160/160 - 0s - loss: 0.0067 - 378ms/epoch - 2ms/step
Epoch 55/100
160/160 - 0s - loss: 0.0067 - 357ms/epoch - 2ms/step
Epoch 56/100
160/160 - 0s - loss: 0.0068 - 316ms/epoch - 2ms/step
Epoch 57/100
160/160 - 0s - loss: 0.0067 - 293ms/epoch - 2ms/step
Epoch 58/100
160/160 - 0s - loss: 0.0068 - 387ms/epoch - 2ms/step
Epoch 59/100
160/160 - 0s - loss: 0.0066 - 490ms/epoch - 3ms/step
Epoch 60/100
160/160 - 1s - loss: 0.0067 - 513ms/epoch - 3ms/step
Epoch 61/100
160/160 - 1s - loss: 0.0066 - 524ms/epoch - 3ms/step
Epoch 62/100
160/160 - 1s - loss: 0.0067 - 647ms/epoch - 4ms/step
Epoch 63/100
160/160 - 1s - loss: 0.0066 - 622ms/epoch - 4ms/step
Epoch 64/100
160/160 - 1s - loss: 0.0067 - 705ms/epoch - 4ms/step
Epoch 65/100
160/160 - 1s - loss: 0.0067 - 562ms/epoch - 4ms/step
Epoch 66/100
160/160 - 0s - loss: 0.0067 - 433ms/epoch - 3ms/step
Epoch 67/100
160/160 - 0s - loss: 0.0067 - 397ms/epoch - 2ms/step
Epoch 68/100
160/160 - 0s - loss: 0.0067 - 298ms/epoch - 2ms/step
Epoch 69/100
160/160 - 0s - loss: 0.0068 - 292ms/epoch - 2ms/step
Epoch 70/100
160/160 - 0s - loss: 0.0067 - 433ms/epoch - 3ms/step
Epoch 71/100
160/160 - 0s - loss: 0.0067 - 433ms/epoch - 3ms/step
Epoch 72/100
160/160 - 0s - loss: 0.0067 - 384ms/epoch - 2ms/step
Epoch 73/100
```

```
160/160 - 0s - loss: 0.0068 - 392ms/epoch - 2ms/step
Epoch 74/100
160/160 - 0s - loss: 0.0068 - 399ms/epoch - 2ms/step
Epoch 75/100
160/160 - 0s - loss: 0.0066 - 461ms/epoch - 3ms/step
Epoch 76/100
160/160 - 0s - loss: 0.0068 - 406ms/epoch - 3ms/step
Epoch 77/100
160/160 - 0s - loss: 0.0067 - 379ms/epoch - 2ms/step
Epoch 78/100
160/160 - 0s - loss: 0.0066 - 332ms/epoch - 2ms/step
Epoch 79/100
160/160 - 0s - loss: 0.0066 - 298ms/epoch - 2ms/step
Epoch 80/100
160/160 - 0s - loss: 0.0066 - 283ms/epoch - 2ms/step
Epoch 81/100
160/160 - 0s - loss: 0.0068 - 401ms/epoch - 3ms/step
Epoch 82/100
160/160 - 0s - loss: 0.0065 - 366ms/epoch - 2ms/step
Epoch 83/100
160/160 - 0s - loss: 0.0067 - 388ms/epoch - 2ms/step
Epoch 84/100
160/160 - 0s - loss: 0.0067 - 452ms/epoch - 3ms/step
Epoch 85/100
160/160 - 1s - loss: 0.0066 - 658ms/epoch - 4ms/step
Epoch 86/100
160/160 - 0s - loss: 0.0067 - 499ms/epoch - 3ms/step
Epoch 87/100
160/160 - 1s - loss: 0.0066 - 579ms/epoch - 4ms/step
Epoch 88/100
160/160 - 1s - loss: 0.0068 - 589ms/epoch - 4ms/step
Epoch 89/100
160/160 - 1s - loss: 0.0066 - 844ms/epoch - 5ms/step
Epoch 90/100
160/160 - 1s - loss: 0.0068 - 1s/epoch - 6ms/step
Epoch 91/100
160/160 - 1s - loss: 0.0067 - 896ms/epoch - 6ms/step
Epoch 92/100
160/160 - 1s - loss: 0.0067 - 1s/epoch - 7ms/step
Epoch 93/100
160/160 - 1s - loss: 0.0068 - 838ms/epoch - 5ms/step
Epoch 94/100
160/160 - 1s - loss: 0.0065 - 1s/epoch - 6ms/step
Epoch 95/100
160/160 - 1s - loss: 0.0068 - 761ms/epoch - 5ms/step
Epoch 96/100
160/160 - 1s - loss: 0.0068 - 600ms/epoch - 4ms/step
Epoch 97/100
```

```
160/160 - 1s - loss: 0.0067 - 606ms/epoch - 4ms/step
Epoch 98/100
160/160 - 1s - loss: 0.0067 - 650ms/epoch - 4ms/step
Epoch 99/100
160/160 - 0s - loss: 0.0067 - 499ms/epoch - 3ms/step
Epoch 100/100
160/160 - 0s - loss: 0.0067 - 427ms/epoch - 3ms/step
5/5 [==============================] - 1s 4ms/step
3/3 [==============================] - 1s 4ms/step
```



```
Train Score: 479514.84 RMSE
Test Score: 619110.86 RMSE
```

In the provided results, the LSTM achieved a lower RMSE on the training set and a reasonably good RMSE on the test set. This suggests that the LSTM has learned the underlying patterns in the training data without overfitting, as evidenced by its performance on unseen test data. Its predictions seem to follow the actual sales closely, making it a strong candidate for this time series forecasting task.

However, LSTMs are computationally expensive to train and can be difficult to interpret. Despite these challenges, if the priority is forecast accuracy, and if we have the computational resources, LSTMs can be a good choice.

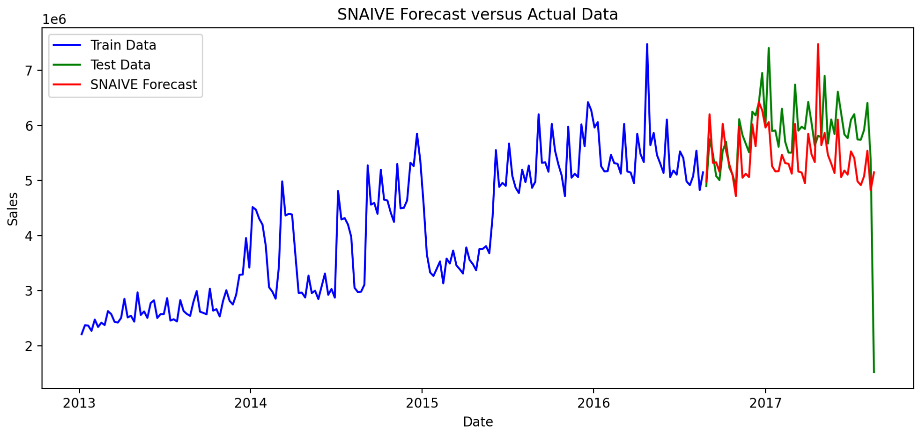
3 SNAIVE

SNAIVE (Seasonal Naive)

The SNAIVE method is a variation of the naive forecast that accounts for seasonality. It assumes that the seasonal pattern will repeat itself from the last observed season. In this case, it forecasts future sales based on the sales from the corresponding season in the previous cycle, typically the same day or week from the last observed year.

```
[ ]: import pandas as pd
     import numpy as np
     from sklearn.metrics import mean_squared_error
     from math import sqrt
     import matplotlib.pyplot as plt

     # Resampling the data to weekly frequency, summing up the sales
     train_df_weekly = train_df['sales'].resample('W').sum()

     # Train-test split (keeping the last year as test data in this case)
     n_test = 52
     train = train_df_weekly.iloc[:-n_test]
     test = train_df_weekly.iloc[-n_test:]

     # SNAIVE forecast
     def snaive_forecast(train_series, test_series):
         forecast = train_series[-n_test:].values
         return forecast

     # Generating forecast
     snaive_pred = snaive_forecast(train, test)

     # Calculating RMSE
     rmse = sqrt(mean_squared_error(test.values, snaive_pred))
     print('Test RMSE: %.3f' % rmse)

     # Plotting the forecasts
     plt.figure(figsize=(12, 5))
     plt.plot(train.index, train.values, label='Train Data', color='blue')
     plt.plot(test.index, test.values, label='Test Data', color='green')
     plt.plot(test.index, snaive_pred, label='SNAIVE Forecast', color='red')
     plt.legend()
     plt.title('SNAIVE Forecast versus Actual Data')
     plt.xlabel('Date')
     plt.ylabel('Sales')
     plt.show()
```

```
Test RMSE: 817502.034
```



The SNAIVE method's RMSE on the test set was higher compared to ARIMA and LSTM, indicating that while it incorporates seasonality, it may not fully capture other underlying patterns or trends in the sales data. This model is often used as a baseline in seasonal time series forecasting due to its simplicity and the intuitive appeal of carrying forward seasonal patterns.

The Seasonal Naive forecast is straightforward and computationally inexpensive, but it is less sophisticated than models like ARIMA and LSTM. Its strength lies in its simplicity and the direct incorporation of seasonality, but it lacks the flexibility of more complex models to adapt to changes in trends or relationships over time.

4 MODEL COMPARISON

To compare the models, we'll use the root mean square error (RMSE) as a measure of accuracy. The RMSE is a widely used measure of the differences between a model's predicted values and the observed values. The lower the RMSE, the better the model's performance.

Here are the RMSE values from all the models:

1. ARIMA: The best ARIMA model had an RMSE of 669,833.771, which indicates the model's average prediction error over the test set.
2. Naive Forecast: The naive forecast, which uses the last observed value as the prediction for all future points, had an RMSE of 852,680.945 on the test set.
3. LSTM: The LSTM model achieved a train score RMSE of 479,514.84 and a test score RMSE of 619,110.86.
4. Seasonal Naive (SNAIVE): The SNAIVE approach had an RMSE of 817,502.034 on the test set.

Now, if we compare these RMSE values:

- **LSTM Model**: This model has the lowest RMSE on the training set and performs reasonably well on the test set too, suggesting a good fit without severe overfitting.
- **ARIMA Model**: This model has a lower RMSE on the test set compared to the LSTM model, which might suggest it's the best performing model for this specific test set.

- **Naive Forecast**: The naive model's performance is weaker, as expected, since it doesn't account for any patterns in the data.
- **SNAIVE Forecast**: Performs better than the naive forecast but not as well as ARIMA or LSTM on the test set.

Considering these results, the **ARIMA model** is the best model based on the test RMSE among the models tested.

```
[ ]:
```