Matthew Blessing
*Student ID: **000507424***
NLM2 - Task 1, D213 - Advanced Data Analytics

NLM 2 - Performance Assessment Task 1: Time Series Modeling

Part I: Research Question

A1.
For this analysis, we will explore if we can create a time series ARIMA model that can accurately forecast revenue based on historical data from the company's first two years.

A2.
The goal of this analysis is to be able to build an ARIMA model that can be trained on historical revenue data and to determine the accuracy of the model's ability or make a predicted forecast based on this training data.

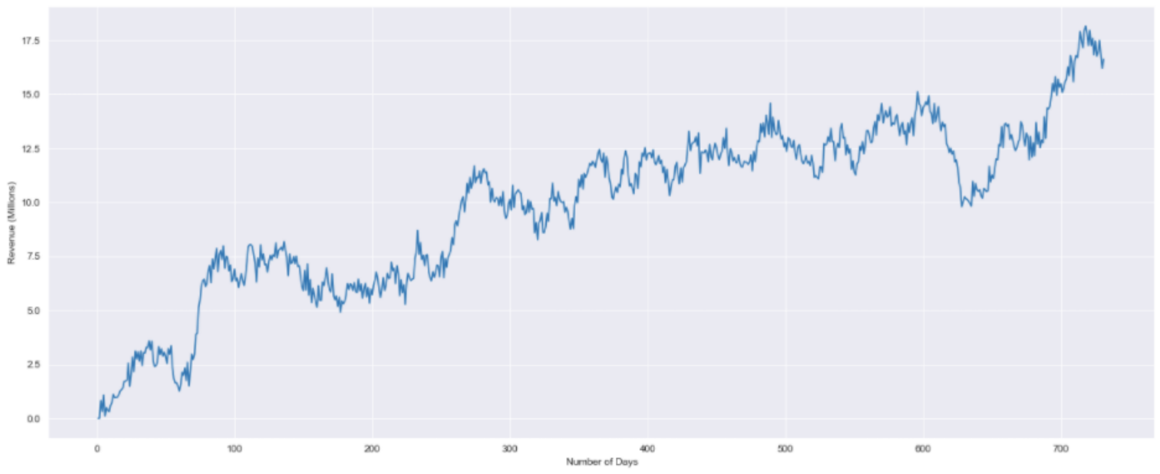
Part II: Method Justification

B1.
In general, time series modeling makes the assumption that trends from the past will continue into the future, in other words, future trends will hold similar to historical trends (Time Series Forecasting Methods, Techniques & Models, 2021). Time series modeling also assumes that the data are stationary, or that the mean, variance, and autocorrelation structure of the data does not change over time, and lacks periodic fluctuations, or seasonality (6.4.4.2. Stationarity, n.d.).

Part III: Data Preparation

C1.
We begin our data preparation by first visualizing the realization of the time series with a simple line graph:

```
In [61]: #plots a line graph using the two variables within our dataset
         plt.figure(figsize=(20,8))
         sns.lineplot(data=df, x='Day', y='Revenue')
         plt.xlabel(' Number of Days')
         plt.ylabel('Revenue (Millions)');
```



C2.

We know from our data dictionary that the dataset is split into two columns, revenue and number of days. We should ensure that both variables have the same number of observations:

```
In [12]: #gets the data type, count, and data type of all variables in the data set
         df.info()

         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 731 entries, 0 to 730
         Data columns (total 2 columns):
         Day         731 non-null int64
         Revenue     731 non-null float64
         dtypes: float64(1), int64(1)
         memory usage: 11.5 KB
```

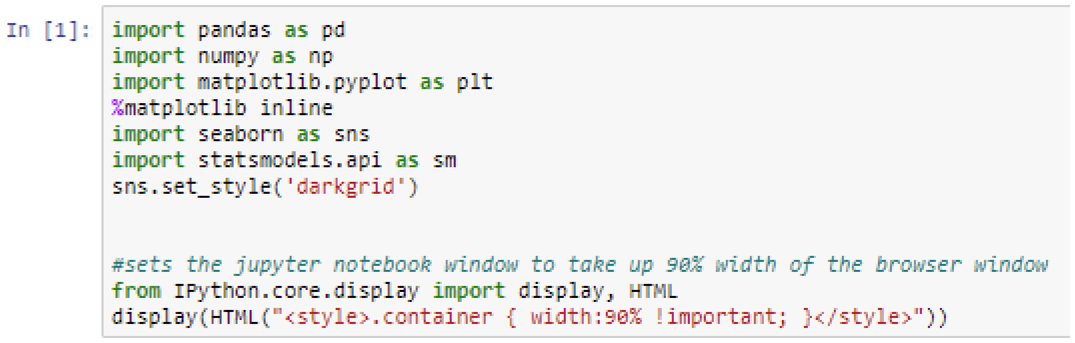
As we can see, 'Day' is an Int64 data type, with 731 observations and 'Revenue' is a float64 data type with 731 observations. The length of our sequence appears to be 731 days, or 2 years. We will also verify there are no duplicate or null values in the dataset:

```
In [44]: #identifies & sums any null values
         df.isnull().sum()

Out[44]: Day        0
         Revenue    0
         dtype: int64

In [43]: #identifies & sums any duplicate rows
         df.duplicated().sum()

Out[43]: 0
```

As we can see, there are no null or duplicate values in the dataset, and since these observations are continuous for 731 days, there appears to be no gaps in our measurements.

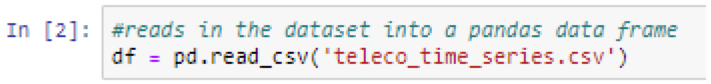
C3.
As we can see from our line graph we plotted above, our data is nonstationary with several fluctuations in revenue, while generally trending upwards over the course of 731 days.

C4.
To prepare the data for our analysis, we began by importing our initial Python libraries for data analysis and visualization:



```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
        import statsmodels.api as sm
        sns.set_style('darkgrid')


        #sets the jupyter notebook window to take up 90% width of the browser window
        from IPython.core.display import display, HTML
        display(HTML("<style>.container { width:90% !important; }</style>"))
```

We then read in our dataset into a dataframe utilizing Pandas .read_csv() function:

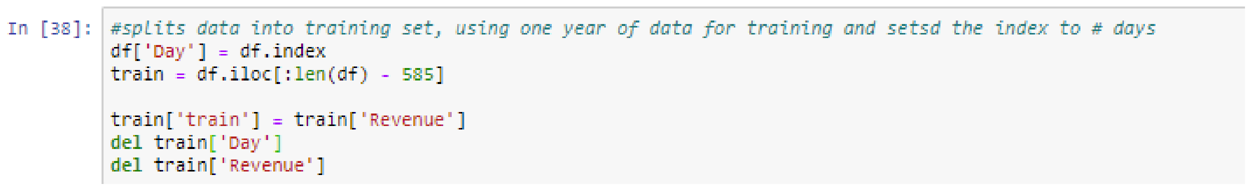


```
In [2]: #reads in the dataset into a pandas data frame
        df = pd.read_csv('teleco_time_series.csv')
```

We also checked the number of variables and observations, as well as checked for duplicate and null values, as seen above in section C2.

Lastly, we will want to split our data into training and testing sets. Generally, we could utilize Sci-Kit Learn's train/test split function, however, since observations in a time series are not independent, we will need to manually split our data into training and testing sets.

We will do this by creating a separate data frame called 'train' and 'test'. We will define the index of the data frame based on the 'Day' column of our dataset, and select a number of 'Revenue' observations for both the train and the test sets. As we have 731 days of data, we have decided to split this data using the 80/20 rule, using 585 days for the training set and 146 days for the testing set, we then remove the remainder of the . We define both sets using the following code:



```
In [38]: #splits data into training set, using one year of data for training and setsd the index to # days
         df['Day'] = df.index
         train = df.iloc[:len(df) - 585]

         train['train'] = train['Revenue']
         del train['Day']
         del train['Revenue']
```

```
In [39]: #splits data into test set, using one year of data for test and setsd the index to # days
         test = df.iloc[len(df) - 586:]

         test['test'] = test['Revenue']
         del test['Day']
         del test['Revenue']
```

We then export our cleaned dataset to a .csv file:

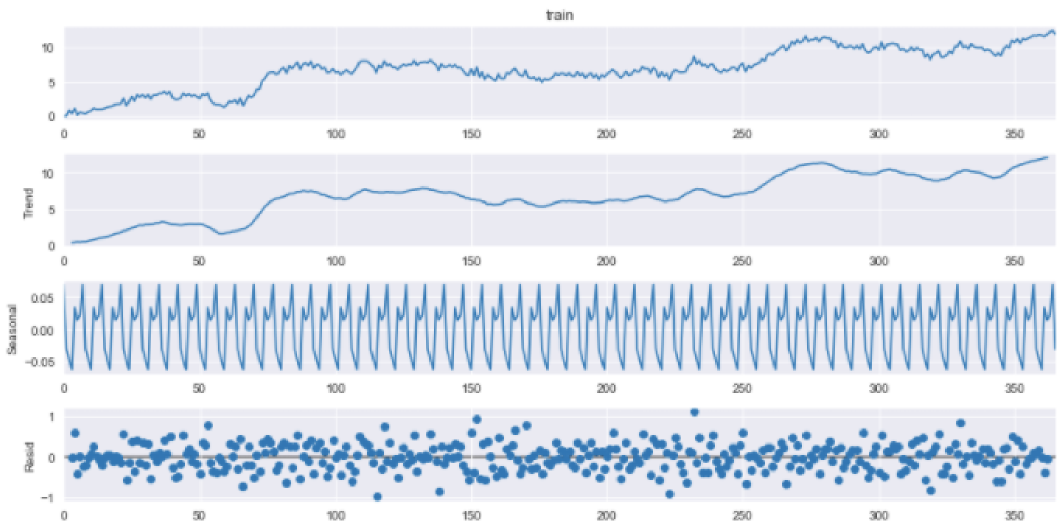
```
In [61]: #exports our cleaned data
         df.to_csv('D213_dataset.csv')
```

C5.
Please see the attached 'D213_dataset.csv'.

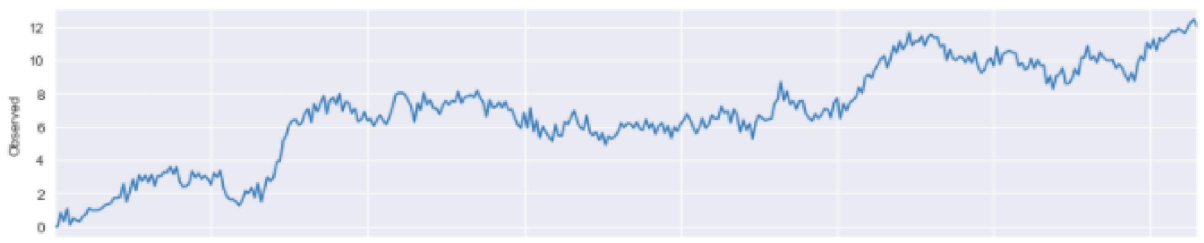
Part IV: Model Identification and Analysis

D1.
We begin to analyze our time series dataset by utilizing the seasonal_decompose module from Statsmodel to determine the trends, seasonality, and noise of our time series data:

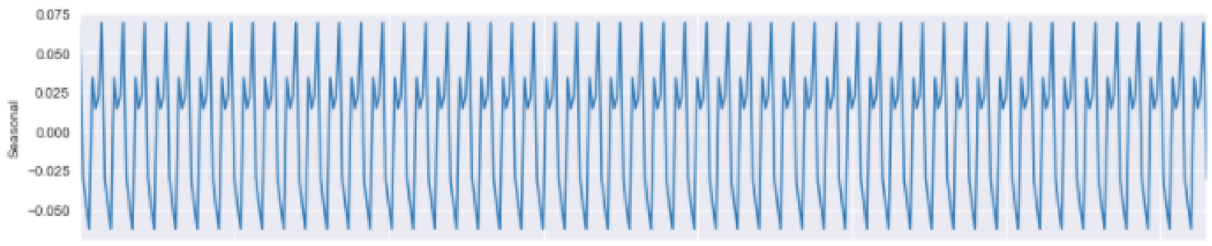
```
In [23]: #decomposes our training set using the seasonal_decompose module from statsmodel
         from statsmodels.tsa.seasonal import seasonal_decompose
         decompose = seasonal_decompose(train['train'], model='additive', period=7)
         decompose.plot()
         plt.rcParams['figure.figsize'] = [12,6]
         plt.show();
```



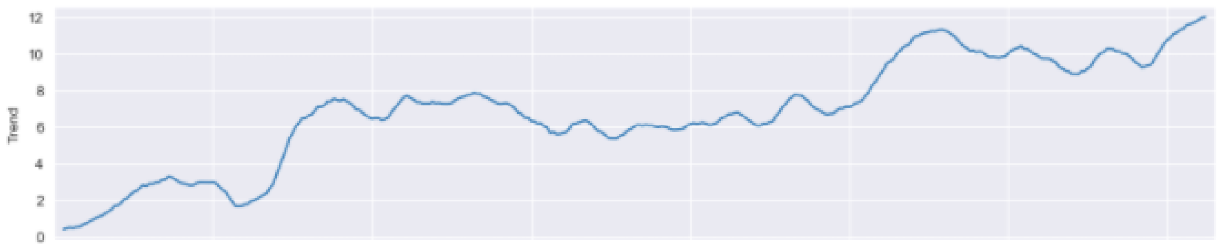
We see our observed decomposed time series:



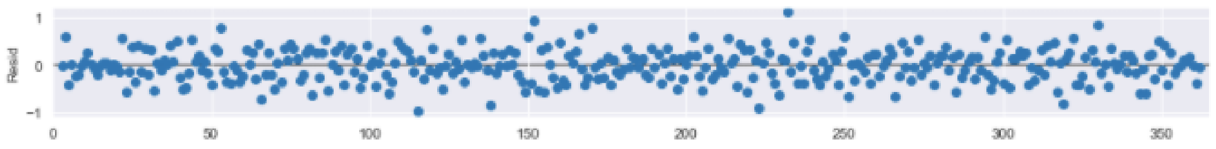
When looking for seasonality, we can see in our decomposition that there is a very strong seasonality presence to our time series data:



We can see in our decomposed model that there is a clear, generally upwards, trend:



Looking at our residuals, we can identify no clear trends:



We can also calculate the autocorrelation by first looking at varying lag ranges. Autocorrelation is the degree of correlation for the same variables between two successive time intervals. Autocorrelation, similarly to correlation, can be positive or negative, ranging from -1 to 1. A positive autocorrelation implies that increases that are observed within a time interval leads to a proportionate increase in the lagged time interval (Corporate Finance Institute, 2021). For example, the next day's temperature will tend to be higher when temperatures in the previous days have been also trending upwards.

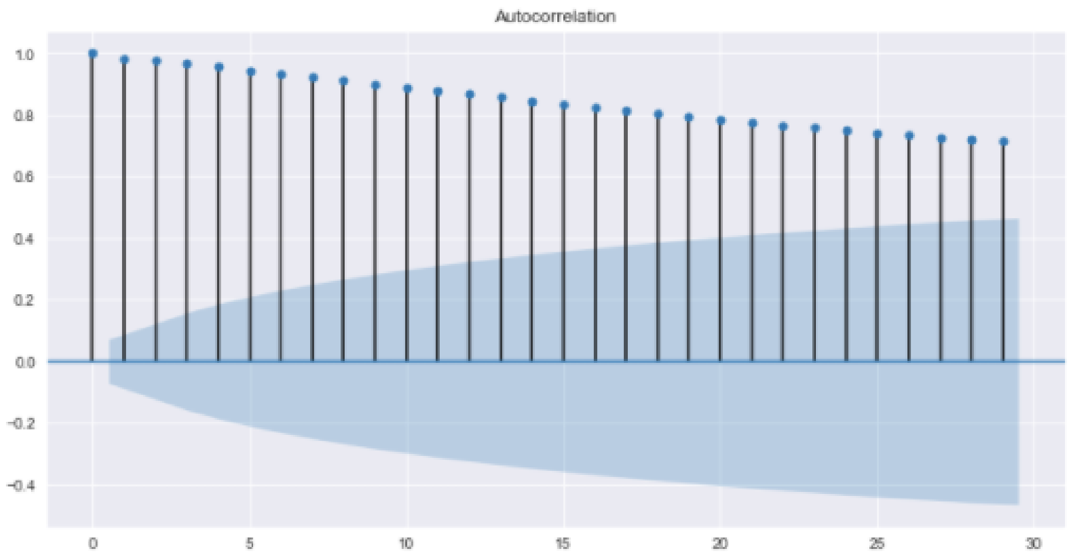
To explore the autocorrelation values, we have selected to look at autocorrelation values with lags in the order of 1 day, 7 days, 30 days, 90 days, and 180 days:

```
In [77]: #calculates the autocorrelation of varying time range lags - 1 day, 7 days, 30 days, 90 days, and 180 days
         ac1 = df['Revenue'].autocorr(1)
         ac2 = df['Revenue'].autocorr(7)
         ac3 = df['Revenue'].autocorr(30)
         ac4 = df['Revenue'].autocorr(90)
         ac5 = df['Revenue'].autocorr(180)

         print("Autocorrelation - 1 day lag: ", ac1)
         print("Autocorrelation - 7 days lag: ", ac2)
         print("Autocorrelation - 30 days lag: ", ac3)
         print("Autocorrelation - 90 days lag: ", ac4)
         print("Autocorrelation - 180 days lag: ", ac5)
```

```
Autocorrelation - 1 day lag:  0.9902917999411421
Autocorrelation - 7 days lag:  0.9679529235243273
Autocorrelation - 30 days lag:  0.8713864374795335
Autocorrelation - 90 days lag:  0.7694050384107837
Autocorrelation - 180 days lag:  0.815306964933998
```

As we can see, our autocorrelation values are very high, particularly when looking at lags of 1 day and 7 days. This implies that future revenue will be increasing if it has been trending upwards in the previous days, weeks, and even months, as well as the opposite; future revenues will likely decrease as they trend downward in the past.
We can also plot an autocorrelation plot to graph the autocorrelation values and lags:

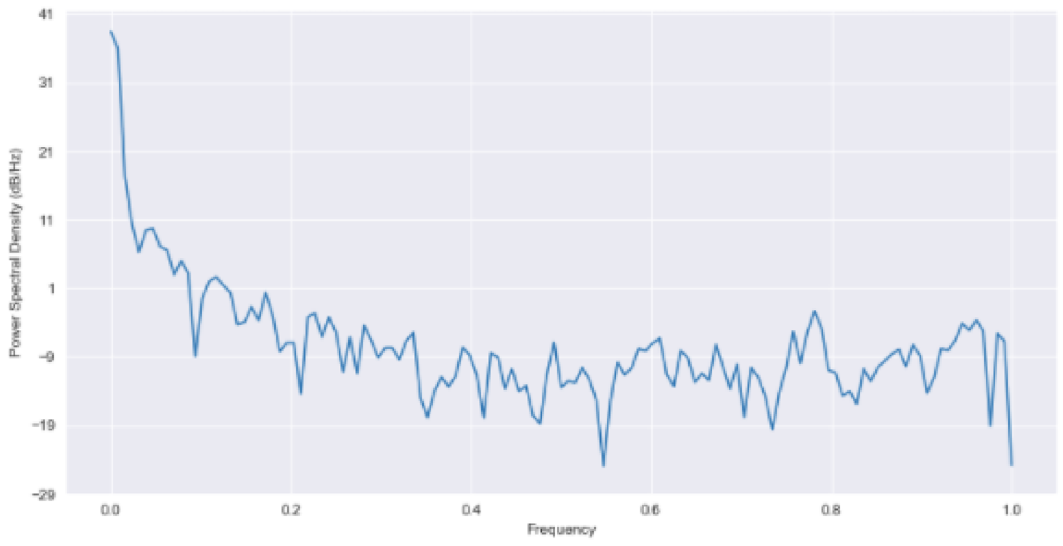
```
In [61]: #plots an autocorrelation graph
         from statsmodels.graphics.tsaplots import plot_acf
         plot_acf(df['Revenue'])
         plt.show()
```



We can see in this autocorrelation plot, that there are extremely high autocorrelation values up to a 30 day lag, confirming our values above.

Lastly, we can also determine the spectral density of our time series by utilizing a periodogram plot within matplotlib for plotting the power spectral density (note that we have suppressed the array for brevity):

```
In [79]: #plots a periodogram for the power spectral density graph
         plt.psd(df['Revenue']);
```



D2.

While we have determined that our time series data is nonstationary, in order to use it for accurate forecasting, we need to convert our data to be white noise, which is a series with mean that is constant with time, a variance that is also constant with time, and zero autocorrelation at all lags (Some Simple Time Series - Autocorrelation Function, n.d.).

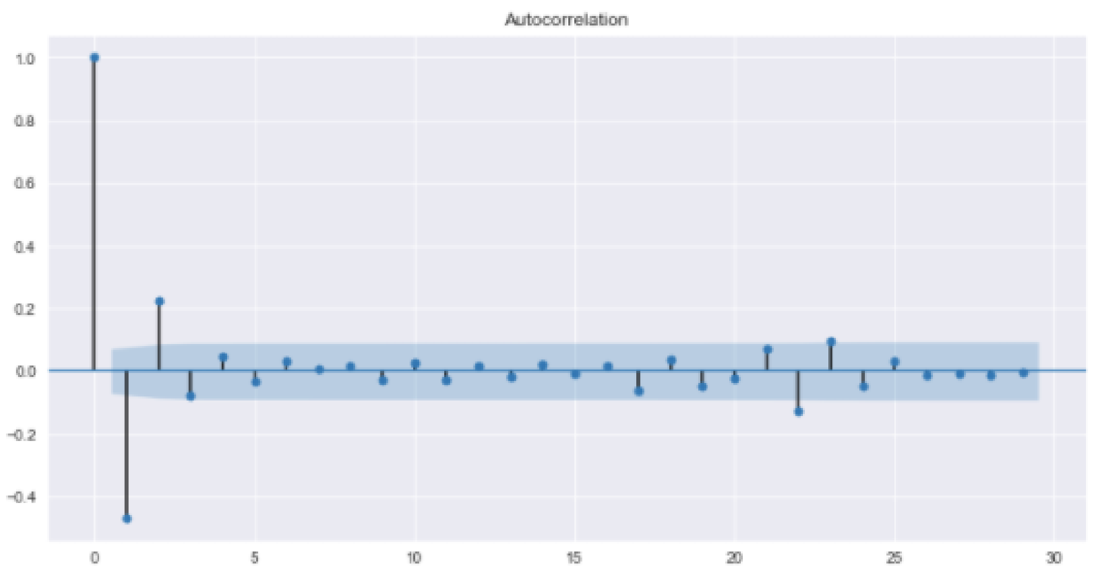
We can convert our time series from nonstationary to stationary by simply taking the first differences, which should result in the data becoming stationary. We will create a new data frame for our stationary time series data and utilize first differences, as well as dropping any null values:

```
In [20]: #creates a stationary time series by utilizing first differences and dropping null values
         diff = df.diff().dropna()

In [23]: diff.head()

Out[23]:
```

	Day	Revenue
1	1.0	0.000793
2	1.0	0.824749
3	1.0	-0.505210
4	1.0	0.762222
5	1.0	-0.974900

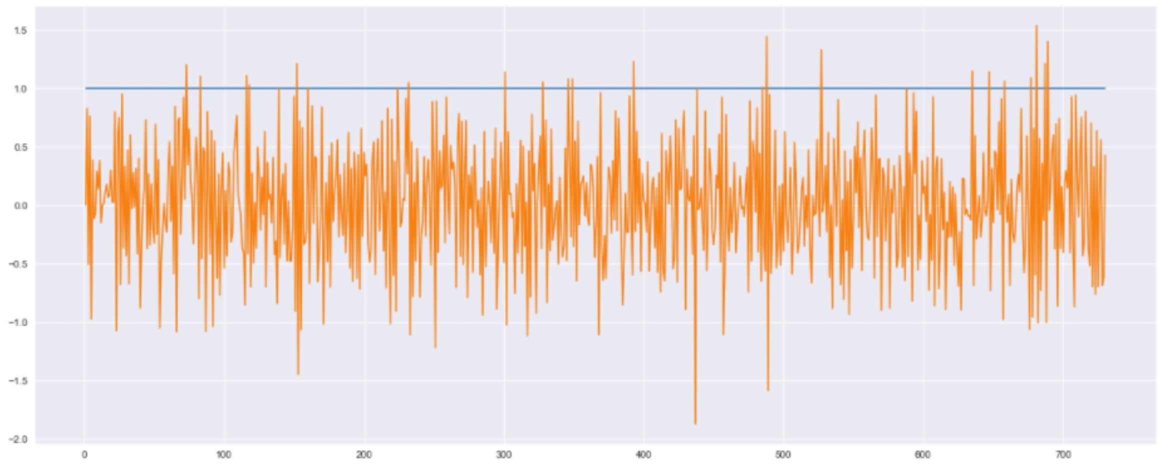
Now that we have utilized first differences, let's plot an autocorrelation chart to ensure that we have little to no autocorrelation at all lags:

```
In [25]: #plots an autocorrelation graph for our non-stationary time series data
         from statsmodels.graphics.tsaplots import plot_acf
         plot_acf(diff['Revenue'])
         plt.show()
```



As we can see, beyond 0 lag, we have practically zero autocorrelation. We can then plot our stationary data to ensure it has no visible trends:

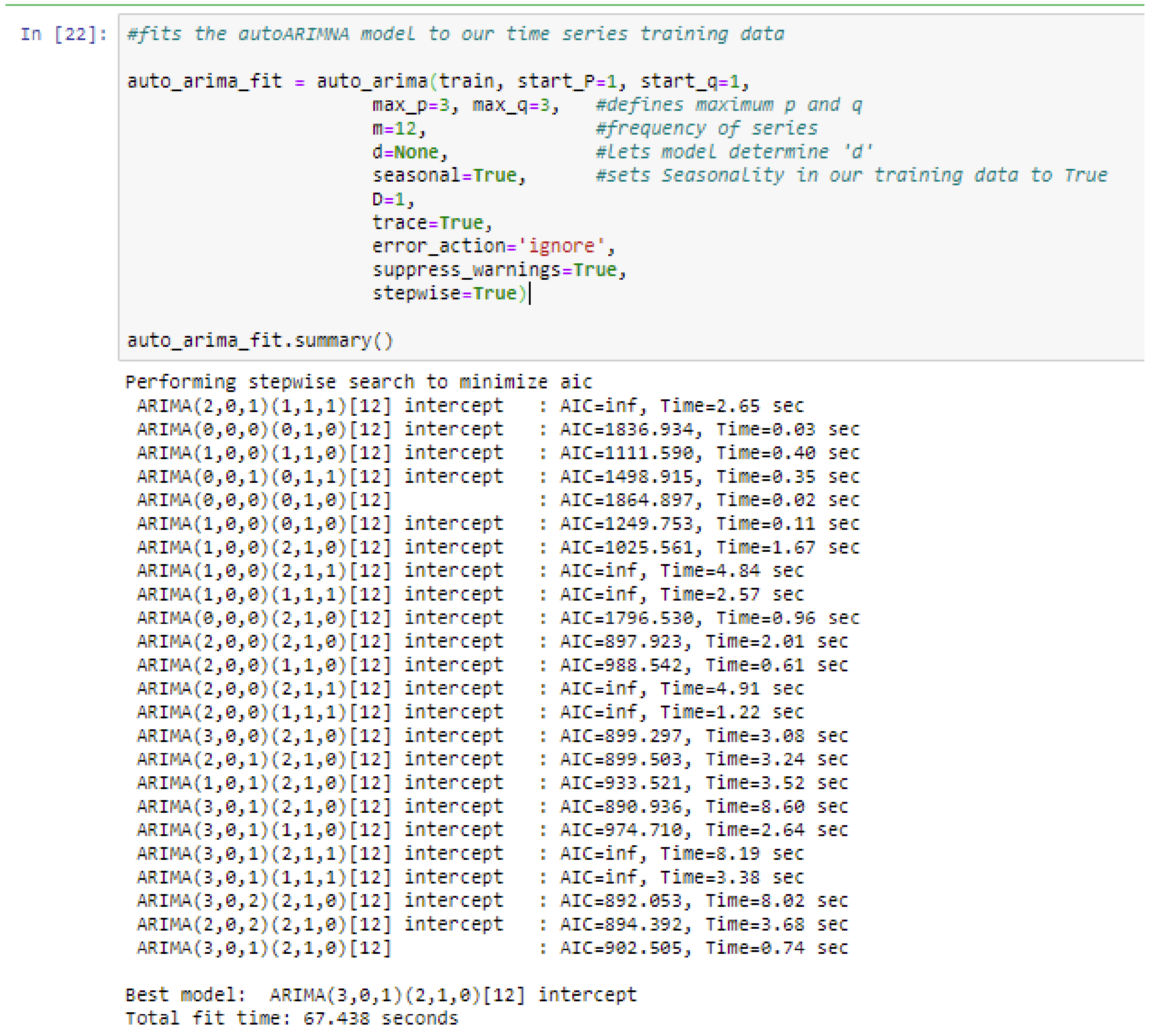
```
In [26]: #computes first differences to transform the data to become stationary
         plt.figure(figsize=(20,8))
         plt.plot(diff);
```



With our time series data now being stationary, we can proceed by identifying an autoregressive integrated moving average (ARIMA) model that will take into account the observed trends and seasonality of our time series data.

To generate our ideal values for our ARIMA model, we will utilize AutoARIMA, a package for Python. The AutoARIMA model will automatically iterate different models on our time series training data and determine the optimal model, along with the optimal number of autoregressive terms (*p*), number of nonseasonal differences needed for stationarity (*d*), or integrated, and number of lagged forecast errors in the prediction equation (*q*), or moving average (Pulagam, 2020).

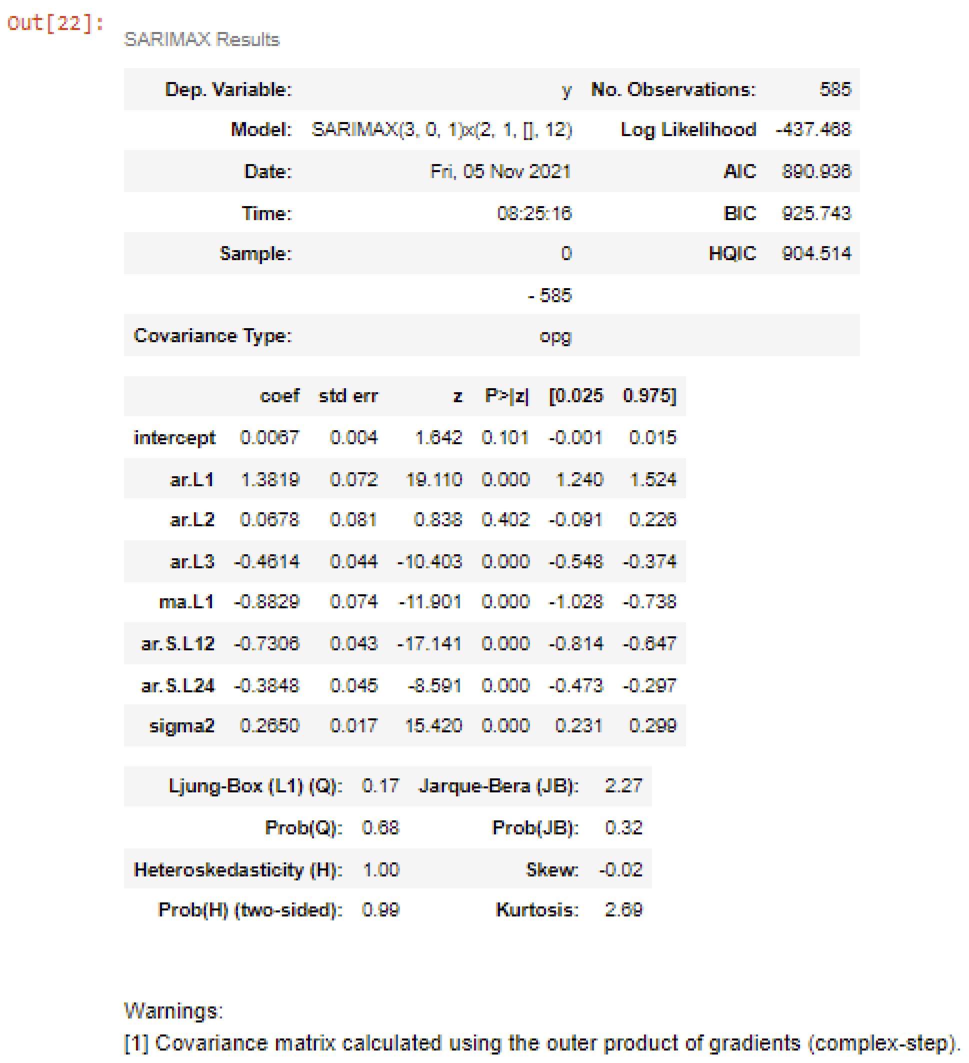
We begin by importing the AutoARIMA model and defining the parameters before running AutoARIMA on our test set we defined above, where the model will try values for *p*, *q*, and *d* from 0-3 to get the most optimal model, by allowing the model to determine the best model (Kim, 2021):



```
In [22]: #fits the autoARIMNA model to our time series training data

         auto_arima_fit = auto_arima(train, start_P=1, start_q=1,
                                     max_p=3, max_q=3,    #defines maximum p and q
                                     m=12,                #frequency of series
                                     d=None,              #Lets model determine 'd'
                                     seasonal=True,       #sets Seasonality in our training data to True
                                     D=1,
                                     trace=True,
                                     error_action='ignore',
                                     suppress_warnings=True,
                                     stepwise=True)

         auto_arima_fit.summary()
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,1)(1,1,1)[12] intercept   : AIC=inf, Time=2.65 sec
 ARIMA(0,0,0)(0,1,0)[12] intercept   : AIC=1836.934, Time=0.03 sec
 ARIMA(1,0,0)(1,1,0)[12] intercept   : AIC=1111.590, Time=0.40 sec
 ARIMA(0,0,1)(0,1,1)[12] intercept   : AIC=1498.915, Time=0.35 sec
 ARIMA(0,0,0)(0,1,0)[12]             : AIC=1864.897, Time=0.02 sec
 ARIMA(1,0,0)(0,1,0)[12] intercept   : AIC=1249.753, Time=0.11 sec
 ARIMA(1,0,0)(2,1,0)[12] intercept   : AIC=1025.561, Time=1.67 sec
 ARIMA(1,0,0)(2,1,1)[12] intercept   : AIC=inf, Time=4.84 sec
 ARIMA(1,0,0)(1,1,1)[12] intercept   : AIC=inf, Time=2.57 sec
 ARIMA(0,0,0)(2,1,0)[12] intercept   : AIC=1796.530, Time=0.96 sec
 ARIMA(2,0,0)(2,1,0)[12] intercept   : AIC=897.923, Time=2.01 sec
 ARIMA(2,0,0)(1,1,0)[12] intercept   : AIC=988.542, Time=0.61 sec
 ARIMA(2,0,0)(2,1,1)[12] intercept   : AIC=inf, Time=4.91 sec
 ARIMA(2,0,0)(1,1,1)[12] intercept   : AIC=inf, Time=1.22 sec
 ARIMA(3,0,0)(2,1,0)[12] intercept   : AIC=899.297, Time=3.08 sec
 ARIMA(2,0,1)(2,1,0)[12] intercept   : AIC=899.503, Time=3.24 sec
 ARIMA(1,0,1)(2,1,0)[12] intercept   : AIC=933.521, Time=3.52 sec
 ARIMA(3,0,1)(2,1,0)[12] intercept   : AIC=890.936, Time=8.60 sec
 ARIMA(3,0,1)(1,1,0)[12] intercept   : AIC=974.710, Time=2.64 sec
 ARIMA(3,0,1)(2,1,1)[12] intercept   : AIC=inf, Time=8.19 sec
 ARIMA(3,0,1)(1,1,1)[12] intercept   : AIC=inf, Time=3.38 sec
 ARIMA(3,0,2)(2,1,0)[12] intercept   : AIC=892.053, Time=8.02 sec
 ARIMA(2,0,2)(2,1,0)[12] intercept   : AIC=894.392, Time=3.68 sec
 ARIMA(3,0,1)(2,1,0)[12]             : AIC=902.505, Time=0.74 sec

Best model:  ARIMA(3,0,1)(2,1,0)[12] intercept
Total fit time: 67.438 seconds
```

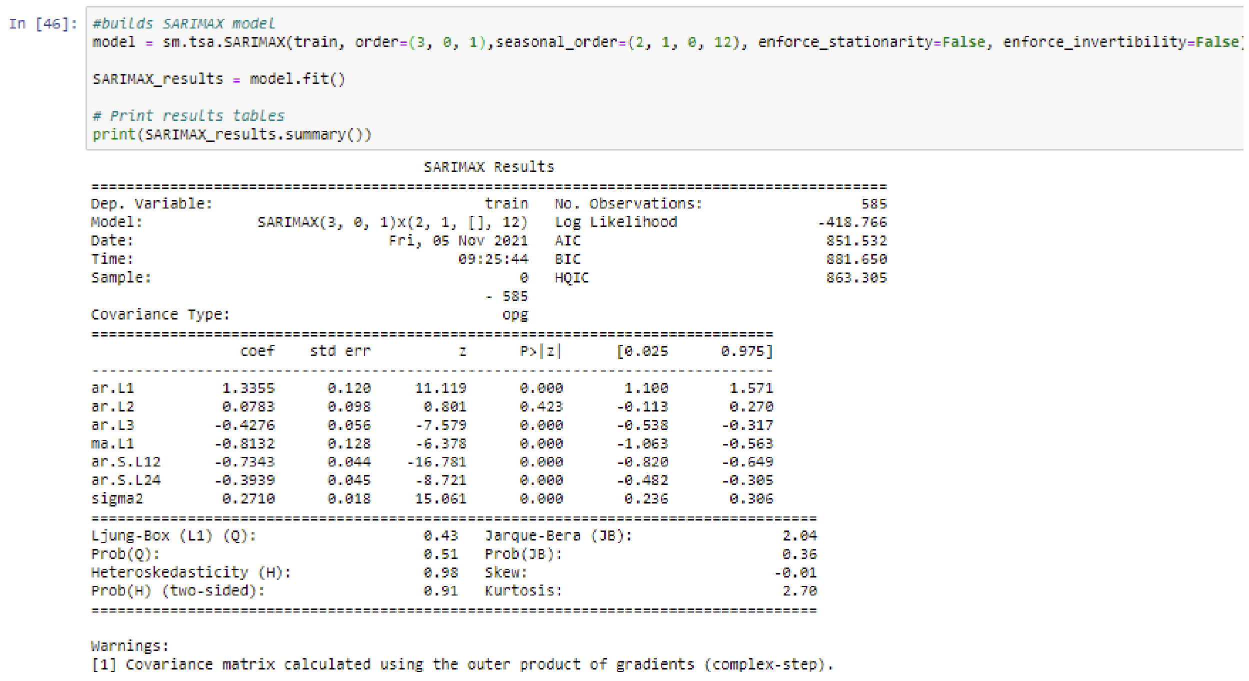
After iterating through assorted parameters and models, our model's summary is as follows:



```
Out[22]:
SARIMAX Results

        Dep. Variable:                                y   No. Observations:       585
               Model:    SARIMAX(3, 0, 1)x(2, 1, [], 12)   Log Likelihood  -437.468
                Date:                  Fri, 05 Nov 2021              AIC   890.936
                Time:                          08:25:16              BIC   925.743
              Sample:                                 0             HQIC   904.514
                                                  - 585
Covariance Type:                                    opg

             coef  std err        z  P>|z|  [0.025  0.975]
intercept  0.0067    0.004    1.642  0.101  -0.001   0.015
    ar.L1  1.3819    0.072   19.110  0.000   1.240   1.524
    ar.L2  0.0678    0.081    0.838  0.402  -0.091   0.226
    ar.L3 -0.4614    0.044  -10.403  0.000  -0.548  -0.374
    ma.L1 -0.8829    0.074  -11.901  0.000  -1.028  -0.738
 ar.S.L12 -0.7306    0.043  -17.141  0.000  -0.814  -0.647
 ar.S.L24 -0.3848    0.045   -8.591  0.000  -0.473  -0.297
   sigma2  0.2650    0.017   15.420  0.000   0.231   0.299

    Ljung-Box (L1) (Q):  0.17   Jarque-Bera (JB):   2.27
              Prob(Q):  0.68           Prob(JB):   0.32
Heteroskedasticity (H):  1.00               Skew:  -0.02
  Prob(H) (two-sided):  0.99           Kurtosis:   2.69


Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

AutoARIMA has determined the best ARIMA model with the best combinations of *p*, *d*, and *q* that provide us with the lowest Akaike Information Criterion (AIC) number, as well as the best seasonal order.

We can then create our model:



```
In [46]: #builds SARIMAX model
         model = sm.tsa.SARIMAX(train, order=(3, 0, 1),seasonal_order=(2, 1, 0, 12), enforce_stationarity=False, enforce_invertibility=False

         SARIMAX_results = model.fit()

         # Print results tables
         print(SARIMAX_results.summary())
```

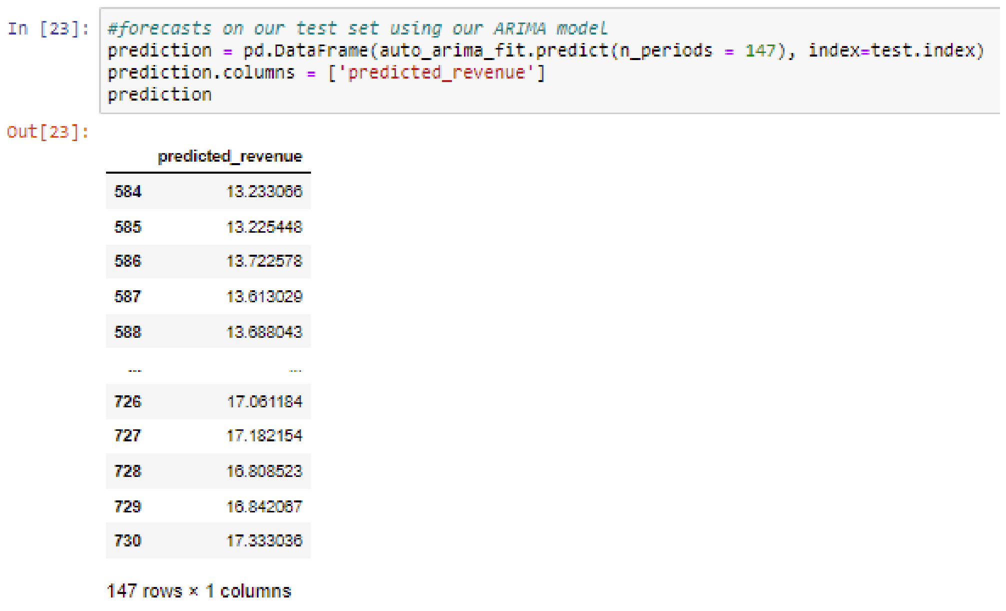
```
                                      SARIMAX Results
==========================================================================================
Dep. Variable:                                train   No. Observations:                585
Model:             SARIMAX(3, 0, 1)x(2, 1, [], 12)   Log Likelihood                -418.766
Date:                              Fri, 05 Nov 2021   AIC                            851.532
Time:                                      09:25:44   BIC                            881.650
Sample:                                           0   HQIC                           863.305
                                              - 585
Covariance Type:                                opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.3355      0.120     11.119      0.000       1.100       1.571
ar.L2          0.0783      0.098      0.801      0.423      -0.113       0.270
ar.L3         -0.4276      0.056     -7.579      0.000      -0.538      -0.317
ma.L1         -0.8132      0.128     -6.378      0.000      -1.063      -0.563
ar.S.L12      -0.7343      0.044    -16.781      0.000      -0.820      -0.649
ar.S.L24      -0.3939      0.045     -8.721      0.000      -0.482      -0.305
sigma2         0.2710      0.018     15.061      0.000       0.236       0.306
===================================================================================
Ljung-Box (L1) (Q):                   0.43   Jarque-Bera (JB):                 2.04
Prob(Q):                              0.51   Prob(JB):                         0.36
Heteroskedasticity (H):               0.98   Skew:                            -0.01
Prob(H) (two-sided):                  0.91   Kurtosis:                         2.70
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

The ARIMA model is now created with our *p*, *q*, and *r* values selected by AutoARIMA. Our final model has an AIC of 851.532, with *p*, *q*, and *r* values of 3, 0, and 1, respectively, and a seasonal order of (2, 1, 0, 12).

D3.

We can now use this model to create a forecast and compare that to our test data in order to determine how accurate our model is at prediction.

We create our forecast by using the best model selected by AutoARIMA and placing those predicted revenue values into our test data set:

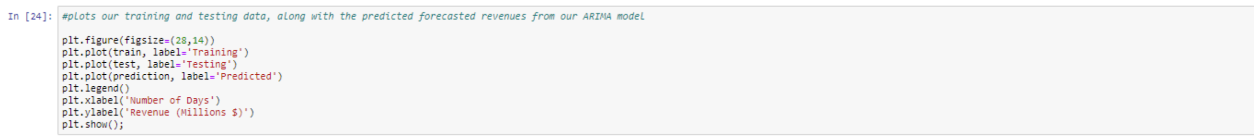


```
In [23]: #forecasts on our test set using our ARIMA model
         prediction = pd.DataFrame(auto_arima_fit.predict(n_periods = 147), index=test.index)
         prediction.columns = ['predicted_revenue']
         prediction

Out[23]:
```

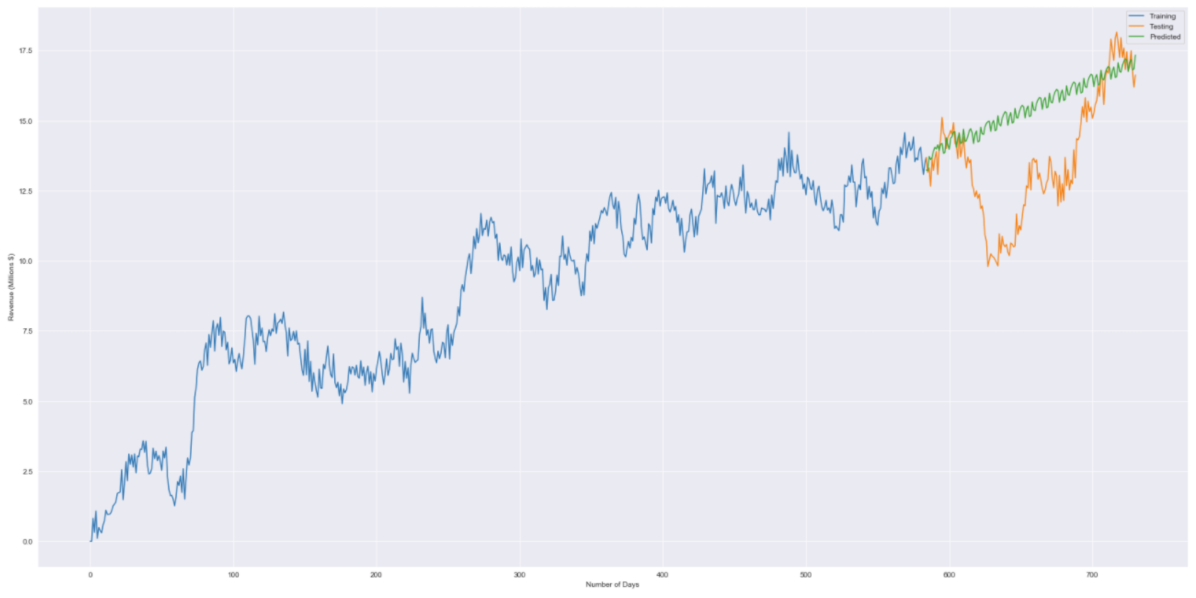
	predicted_revenue
584	13.233066
585	13.225448
586	13.722578
587	13.613029
588	13.688043
...	...
726	17.061184
727	17.182154
728	16.808523
729	16.842067
730	17.333036

147 rows × 1 columns

We can then plot our time series data and segment the data into training, testing, and predicted sets:



```
In [24]: #plots our training and testing data, along with the predicted forecasted revenues from our ARIMA model

         plt.figure(figsize=(28,14))
         plt.plot(train, label='Training')
         plt.plot(test, label='Testing')
         plt.plot(prediction, label='Predicted')
         plt.legend()
         plt.xlabel('Number of Days')
         plt.ylabel('Revenue (Millions $)')
         plt.show();
```



D4-D5.

Please see the code & output screenshots above, as well as the attached 'D213 - Task 1 -Time Series Modeling.py' and attached Jupyter Notebook.

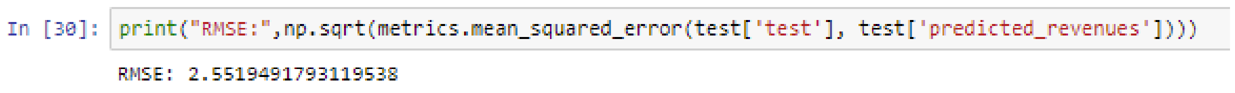
Part V: Data Summary and Implications

E1.

As noted above, we utilized AutoARIMA to determine the best ARIMA model, as well as the best values for *p*, *q*, and *r*. The model selected has selected a basic ARIMA model, with the *p*, *q*, and *r* values of 3, 0, and 1, respectively, and a seasonal order of (2, 1, 0, 12). Our model has an AIC of 851.532, which was the lowest AIC of all of the models and parameters attempted by AutoARIMA.

The model has returned predictions, by day, based on the training set, and we have forecasted on the test set with the same length as the test data, 147 days, in order to compare the accuracy of the predictions from the ARIMA model to the actual test data.

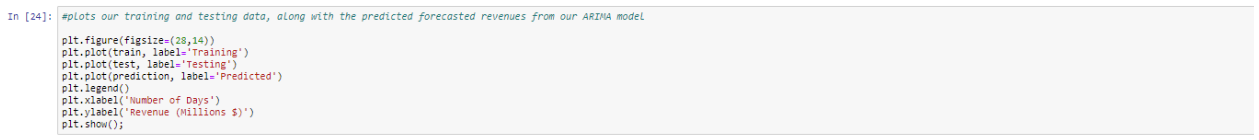
We also utilize Sci-Kit Learn's metrics to give us the Root Mean Squared Error (RMSE) of our predicted revenues vs our test revenues:



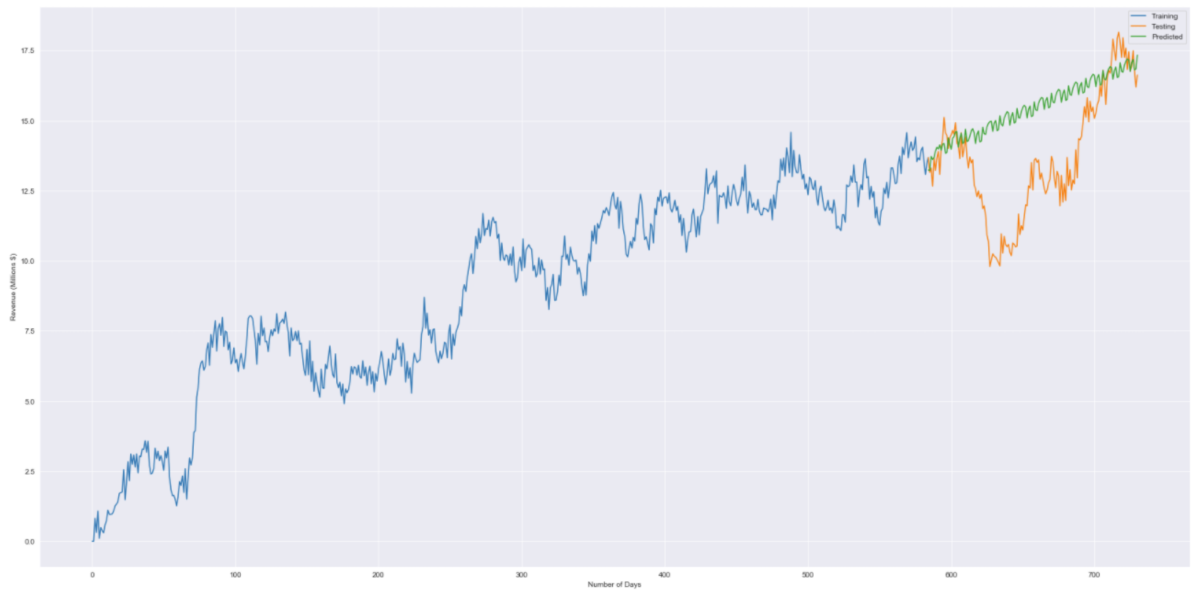
```
In [30]: print("RMSE:",np.sqrt(metrics.mean_squared_error(test['test'], test['predicted_revenues'])))

         RMSE: 2.5519491793119538
```

As we can see, our RMSE is 2.55.

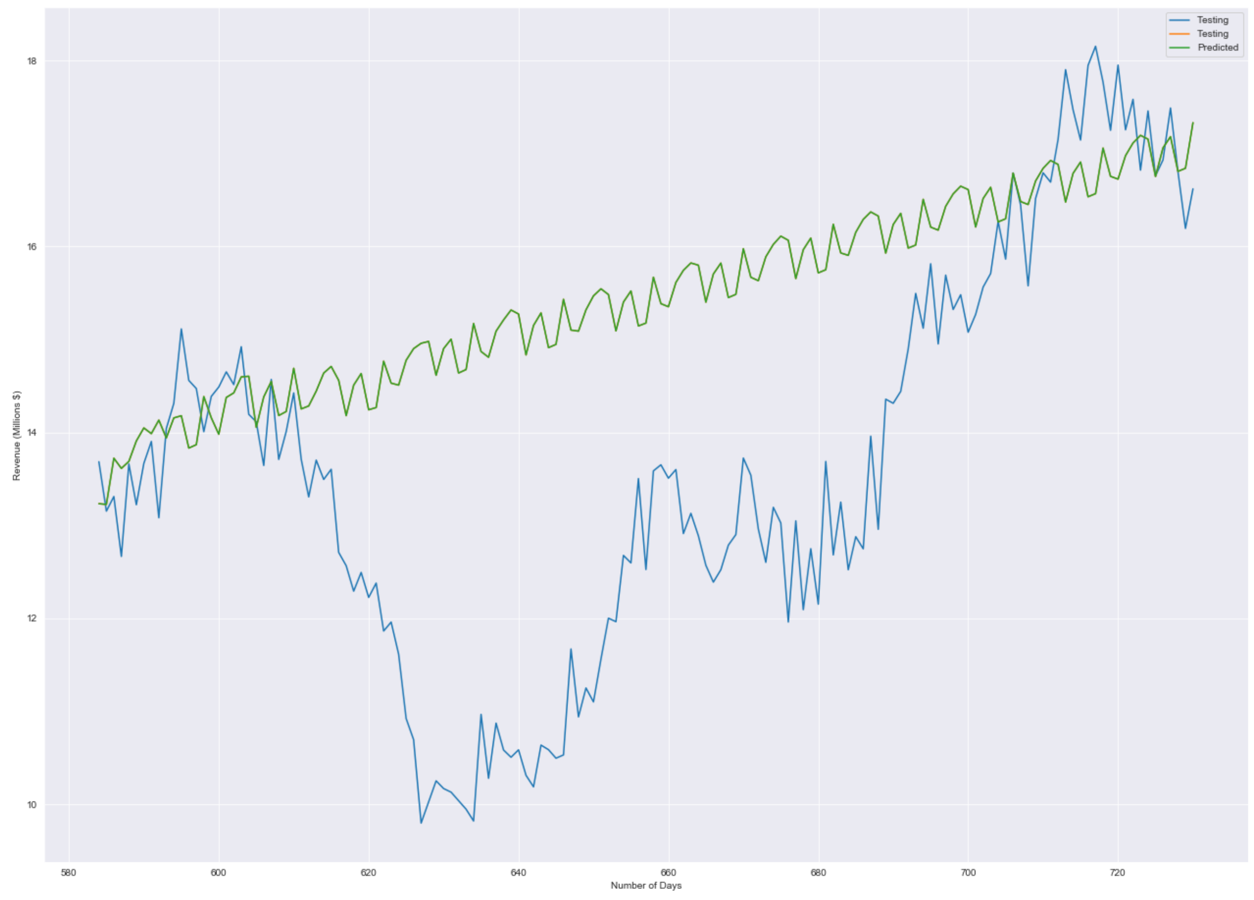
We can now plot our forecasted revenue along with our training and test data to determine the accuracy of our model's ability to predict:



```
In [24]: #plots our training and testing data, along with the predicted forecasted revenues from our ARIMA model

         plt.figure(figsize=(28,14))
         plt.plot(train, label='Training')
         plt.plot(test, label='Testing')
         plt.plot(prediction, label='Predicted')
         plt.legend()
         plt.xlabel('Number of Days')
         plt.ylabel('Revenue (Millions $)')
         plt.show();
```

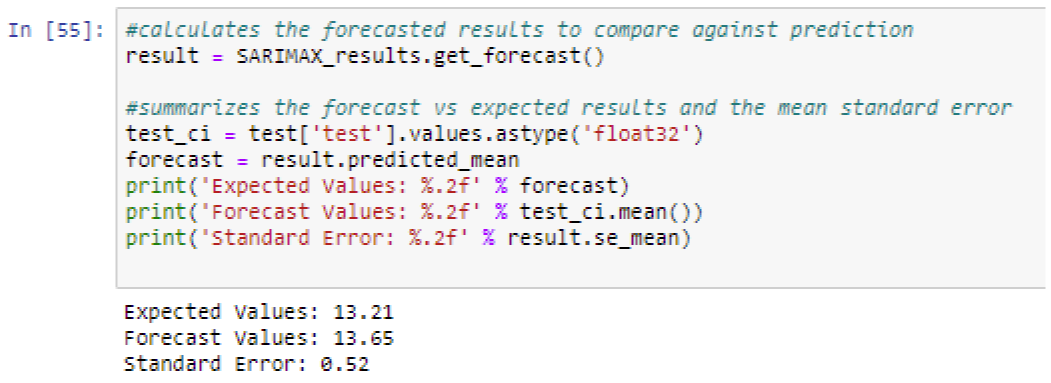


Our model predicts the general trend with surprising accuracy. Let's take a closer look at the forecasted area:



As we can see, the forecasted revenues compared to the test set are fairly accurate in not only predicting the overall trend, but also the revenue itself. However, we notice the model was not accurate where the test set dips. We can see in the line graph with the test set data that there is a sharp drop in revenue around the 630 day mark of the data. Nowhere else in the data is this large of a revenue drop seen, which highlights one potential issue with ARIMA models; ARIMA models do not account for irregularity (Kim, 2021). The model can only learn from what it is trained on from the past. With no stark drops in revenue like this seen in the rest of the testing data, the model cannot be expected to predict this event. Outlier events that could lead to a revenue drop that we see in this data cannot be predicted accurately, but our forecasted revenue does accurately predict where the revenue would be regardless of this temporary drop.

We can also look at the summary of the mean of the predicted forecast values vs. the mean of our expected results:

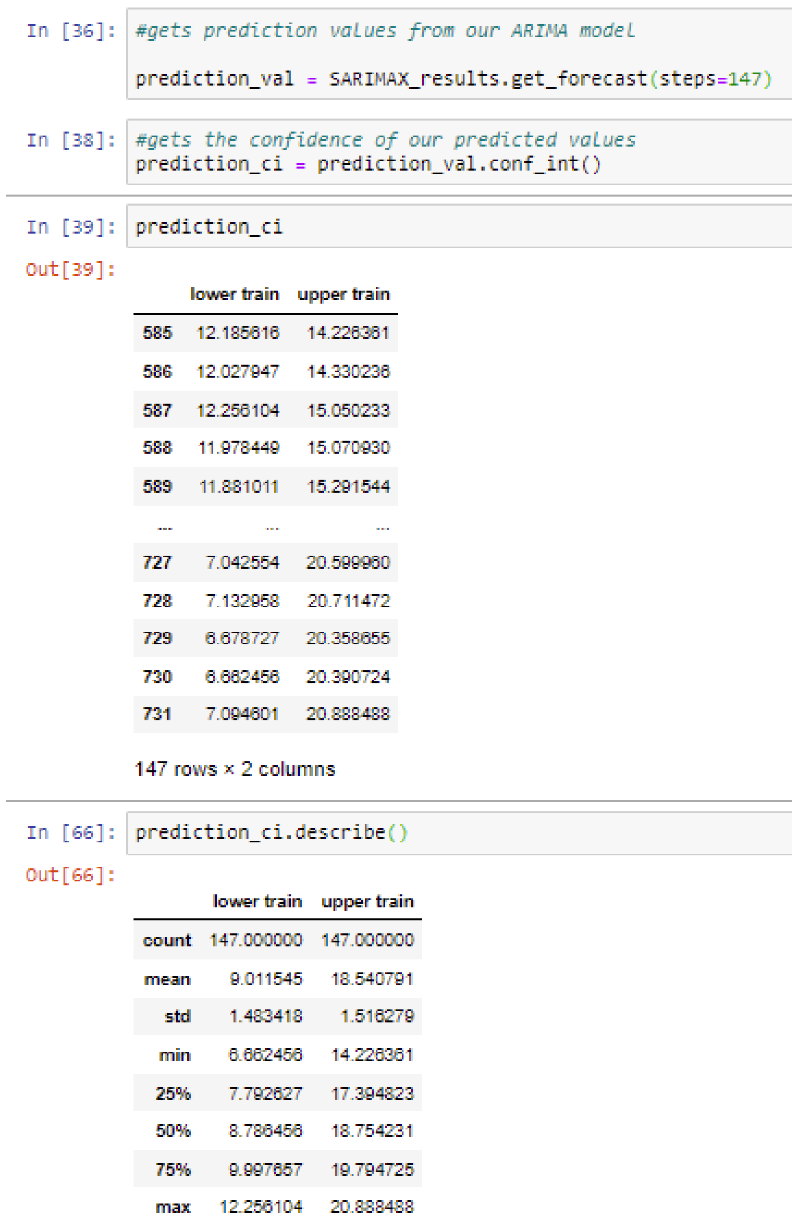


```
In [55]: #calculates the forecasted results to compare against prediction
         result = SARIMAX_results.get_forecast()

         #summarizes the forecast vs expected results and the mean standard error
         test_ci = test['test'].values.astype('float32')
         forecast = result.predicted_mean
         print('Expected Values: %.2f' % forecast)
         print('Forecast Values: %.2f' % test_ci.mean())
         print('Standard Error: %.2f' % result.se_mean)
```

```
Expected Values: 13.21
Forecast Values: 13.65
Standard Error: 0.52
```

As we can see here, our expected mean and forecast mean are fairly close, with a standard error mean of only 0.52.

We can also create our prediction intervals:



```
In [36]: #gets prediction values from our ARIMA model

         prediction_val = SARIMAX_results.get_forecast(steps=147)
```

```
In [38]: #gets the confidence of our predicted values
         prediction_ci = prediction_val.conf_int()
```

```
In [39]: prediction_ci
```

Out[39]:

	lower train	upper train
585	12.185616	14.226361
586	12.027947	14.330236
587	12.256104	15.050233
588	11.978449	15.070930
589	11.881011	15.291544
...	...	...
727	7.042554	20.599960
728	7.132958	20.711472
729	6.678727	20.358655
730	6.662456	20.390724
731	7.094601	20.888488

147 rows × 2 columns

```
In [66]: prediction_ci.describe()
```

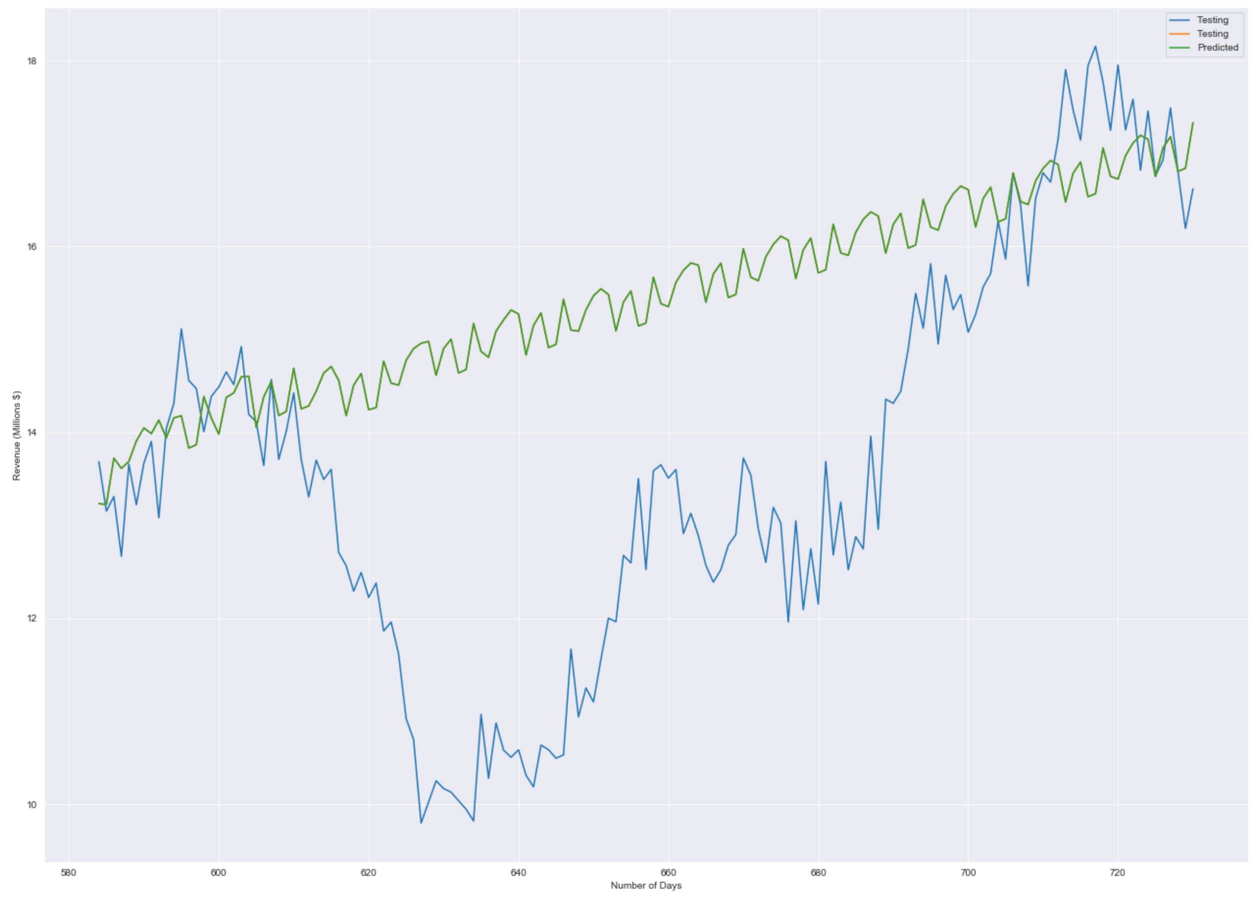
Out[66]:

	lower train	upper train
count	147.000000	147.000000
mean	9.011545	18.540791
std	1.483418	1.516279
min	6.662456	14.226361
25%	7.792627	17.394823
50%	8.786456	18.754231
75%	9.997657	19.794725
max	12.256104	20.888488

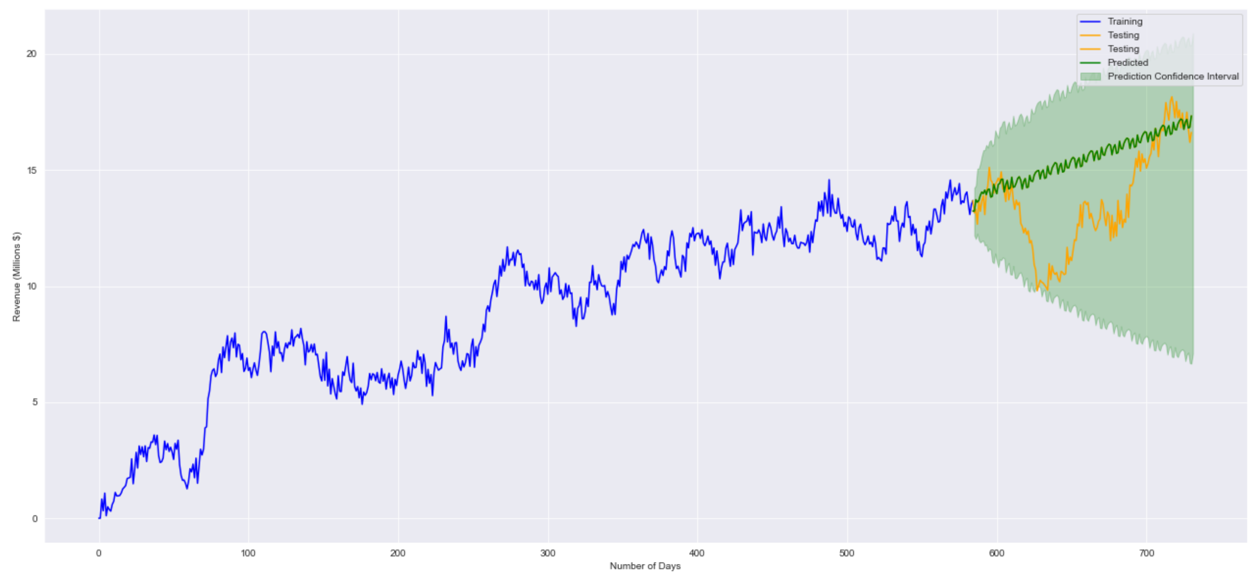
In the above, we've forecasted using the .get_forecast() function and specified the number of steps, which is the same number of days in our test set, and then retrieved the confidence intervals for the forecast using the .conf_int() function. We can then look at our prediction intervals and use Pandas .describe() function to see the min, mean, and max of our prediction intervals. We will plot these prediction intervals below in order to see the range of the intervals compared to our forecast.

E2.

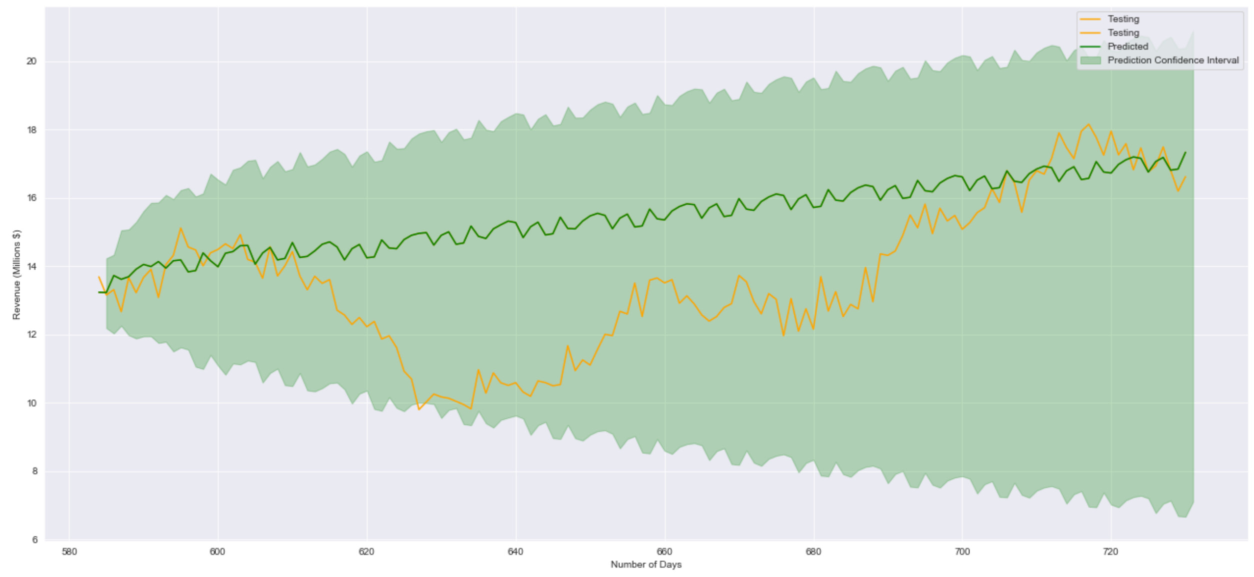
We take a closer look at just the test data set, along with the ARIMA model's predicted revenue forecast:



We can also plot our prediction intervals, with our intervals in green vs our forecasted values in dark green and test set in orange:



And a closer look at just our test set vs forecasted values with our confidence intervals:



E3.
While no model is perfect immediately, it does appear that our simple ARIMA model does a somewhat accurate job of predicting revenue. Of course, we saw with our RMSE score of 2.55, that there is certainly room for improvement in the tuning of the model. Our model, while able to predict the ups and downs of day-to-day revenue changes, cannot predict large drops in revenue like we see above. If this drop in revenue is the result of an outlier event, it is likely that it will not be able to be predicted. However, if it is a recurring event, the further training of this model should be able to predict these recurring events with some accuracy.

Overall, the model seems to relatively accurately predict revenue on our test set, and we would recommend to the stakeholders within the business to iterate on this model, potentially training the model on more data, in an attempt to not only lower the AIC but also the RMSE. We can see the model has room for improvement by looking at the prediction interval ranges on the plot above, where there is quite a wide range for the intervals. Ideally, we would like the interval range to be significantly tighter. However, once our model is more accurately able to predict on the test set, it should prove valuable to the business as a way to accurately forecast future revenue.

References

6.4.4.2. Stationarity. (n.d.). National Institute of Standards and Technology. Retrieved November 1, 2021, from https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc442.htm

12.1 Estimating the Spectral Density | STAT 510. (n.d.). PennState: Statistics Online Courses. Retrieved November 1, 2021, from https://online.stat.psu.edu/stat510/lesson/12/12.1

Brownlee, J. (2020, December 9). *Understand Time Series Forecast Uncertainty Using Prediction Intervals with Python*. Machine Learning Mastery. Retrieved November 2, 2021, from https://machinelearningmastery.com/time-series-forecast-uncertainty-using-confidence-intervals-python/

Corporate Finance Institute. (2021, April 11). *Autocorrelation*. Retrieved November 1, 2021, from https://corporatefinanceinstitute.com/resources/knowledge/other/autocorrelation/

Kim, H. (2021, April 20). *ARIMA for dummies - Analytics Vidhya*. Medium. Retrieved November 1, 2021, from https://medium.com/analytics-vidhya/arima-for-dummies-ba761d59a051

Pulagam, S. (2020, June 27). *Time Series forecasting using Auto ARIMA in python - Towards Data Science*. Medium. Retrieved November 1, 2021, from https://towardsdatascience.com/time-series-forecasting-using-auto-arima-in-python

Some Simple Time Series - Autocorrelation Function. (n.d.). DataCamp. Retrieved November 1, 2021, from https://campus.datacamp.com/courses/time-series-analysis-in-python/some-simple-time-series

Time Series Forecasting Methods, Techniques & Models. (2021, August 30). InfluxData. Retrieved November 1, 2021, from https://www.influxdata.com/time-series-forecasting-methods/

https://machinelearningmastery.com/time-series-forecast-uncertainty-using-confidence-intervals-python/