Using Regression Models For First Hypothesis

In our first hypothesis, we assumed that the average rating decreases as the delivery time increases. This seems to be a linear relation. So, we perform a linear regression to determine the relationship.

Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Reading the dataset

```
df = pd.read_csv('../food_delivery_dataset/train.csv')

df = df.replace('NaN', np.nan, regex=True)

df['Delivery_person_Ratings'] =
pd.to_numeric(df['Delivery_person_Ratings'], errors='coerce')

Average_rating = df['Delivery_person_Ratings'].mean()
df['Delivery_person_Ratings'] =
df['Delivery_person_Ratings'].fillna(round(Average_rating, 1))

delivery_time = np.array(df['Time_taken(min)'].str.split("
").str[1]).astype(int)
ratings = np.array(df['Delivery_person_Ratings'])

data = {
    'delivery_time': delivery_time,
    'ratings': ratings
}

model_input = pd.DataFrame(data)

model_input
```

	delivery_time	ratings
0	24	4.9
1	33	4.5
2	26	4.4
3	21	4.7
4	30	4.6
...	...	...

```
45588              32      4.8
45589              36      4.6
45590              16      4.9
45591              26      4.7
45592              36      4.9

[45593 rows x 2 columns]
```

```
mean_input = model_input.groupby(delivery_time)['ratings'].mean()

mean_input = mean_input.to_frame().reset_index()
```

Linear Regression

```
delivery_time = mean_input['index'].values.reshape(-1, 1)  # Reshape
to a 2D array
ratings = mean_input['ratings'].values

# Initialize and fit the linear regression model
model = LinearRegression()
model.fit(delivery_time, ratings)

# Predictions for the model
ratings_pred = model.predict(delivery_time)

# Print the coefficients of the linear regression model
print(f"Intercept (beta_0): {model.intercept_}")
print(f"Slope (beta_1): {model.coef_[0]}")

# Model evaluation
mse = mean_squared_error(ratings, ratings_pred)
r2 = r2_score(ratings, ratings_pred)
print(f"Mean Squared Error: {mse}")
print(f"R-squared: {r2}")

# Plot the data and the regression line
plt.scatter(delivery_time, ratings, color='blue', label="Actual
Ratings")
plt.plot(delivery_time, ratings_pred, color='red', label="Regression
Line")
plt.title('Delivery Time vs. Ratings')
plt.xlabel('Delivery Time (minutes)')
plt.ylabel('Average Rating')
plt.legend()
plt.show()
```

```
Intercept (beta_0): 4.836381593584211
Slope (beta_1): -0.007887255576593012
```

```
Mean Squared Error: 0.009727460975453768
R-squared: 0.5189191294105682
```

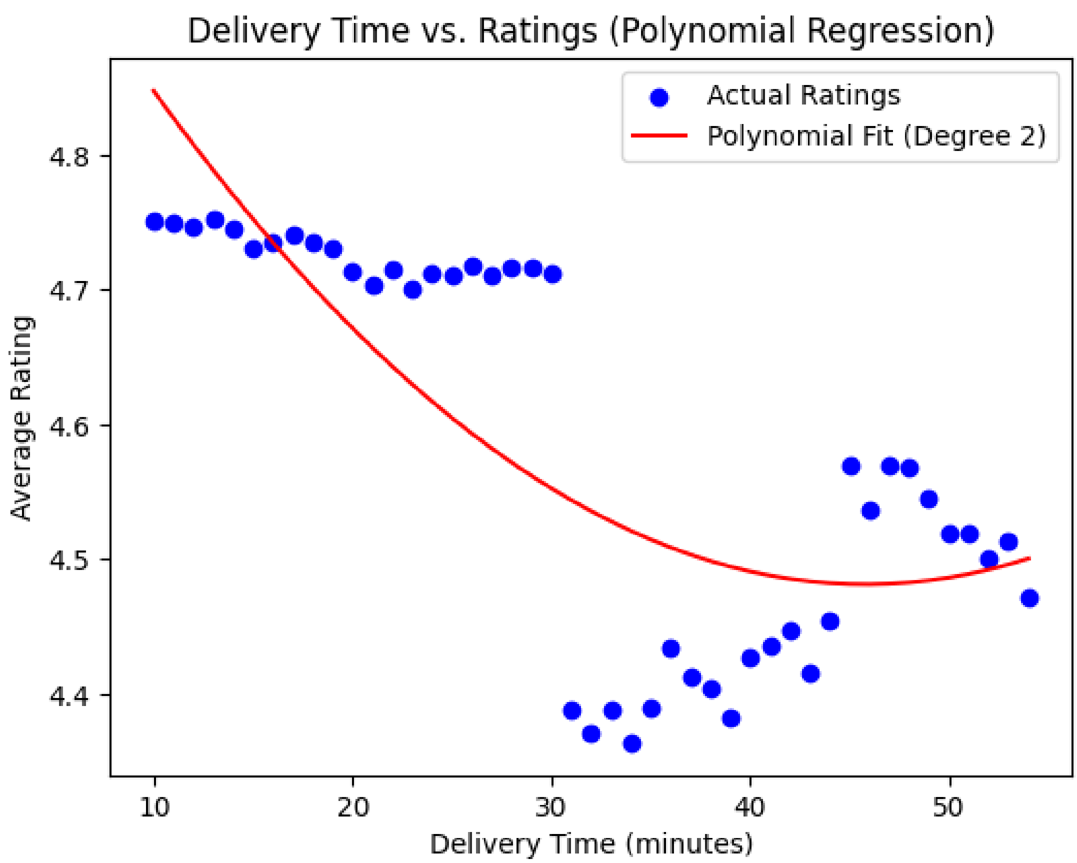
Using Linear regression, as we can see there are huge errors in the predicted model. The graph doesn't align with the data.

Since, we used a linear regression, we could only draw a straight line. If we try to use a polynomial regression, we can have a graph that aligns to the data better with a curve.

Polynomial Regression of degree 2

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression


poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(delivery_time.reshape(-1, 1))

# Fit the polynomial regression model
```

```
model = LinearRegression()
model.fit(X_poly, ratings)

# Predict values
y_pred = model.predict(X_poly)

# Plotting the data
plt.scatter(delivery_time, ratings, color='blue', label='Actual
Ratings')
plt.plot(delivery_time, y_pred, color='red', label='Polynomial Fit
(Degree 2)')
plt.title('Delivery Time vs. Ratings (Polynomial Regression)')
plt.xlabel('Delivery Time (minutes)')
plt.ylabel('Average Rating')
plt.legend()
plt.show()
```



We were able to get a better graph using Polynimial Regression. Now, we want to explore polynomial regression of higher degree. We can use accuracy as a parameter to determine if the model is performing better.

If we increase the degree of polynomial, we can get a better graph that can align with the data better. Calculating the accuraccy for the same.

Since, this was a polynoimal graph of degree 2, we see the graph with 1 maxima or 1 minima. (1 cursve) We want to go to a grpah with higher degree (more curves)

```
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split

X = X_poly
y = ratings

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Transform the features to polynomial features (degree 3)
degree = 2
poly_features = PolynomialFeatures(degree=degree)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# Fit the polynomial regression model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Make predictions
y_pred = model.predict(X_test_poly)

# Calculate evaluation metrics
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

# Print the evaluation metrics
print(f"R-squared: {r2:.3f}")
print(f"Mean Absolute Error: {mae:.3f}")
print(f"Mean Squared Error: {mse:.3f}")
print(f"Root Mean Squared Error: {rmse:.3f}")
```

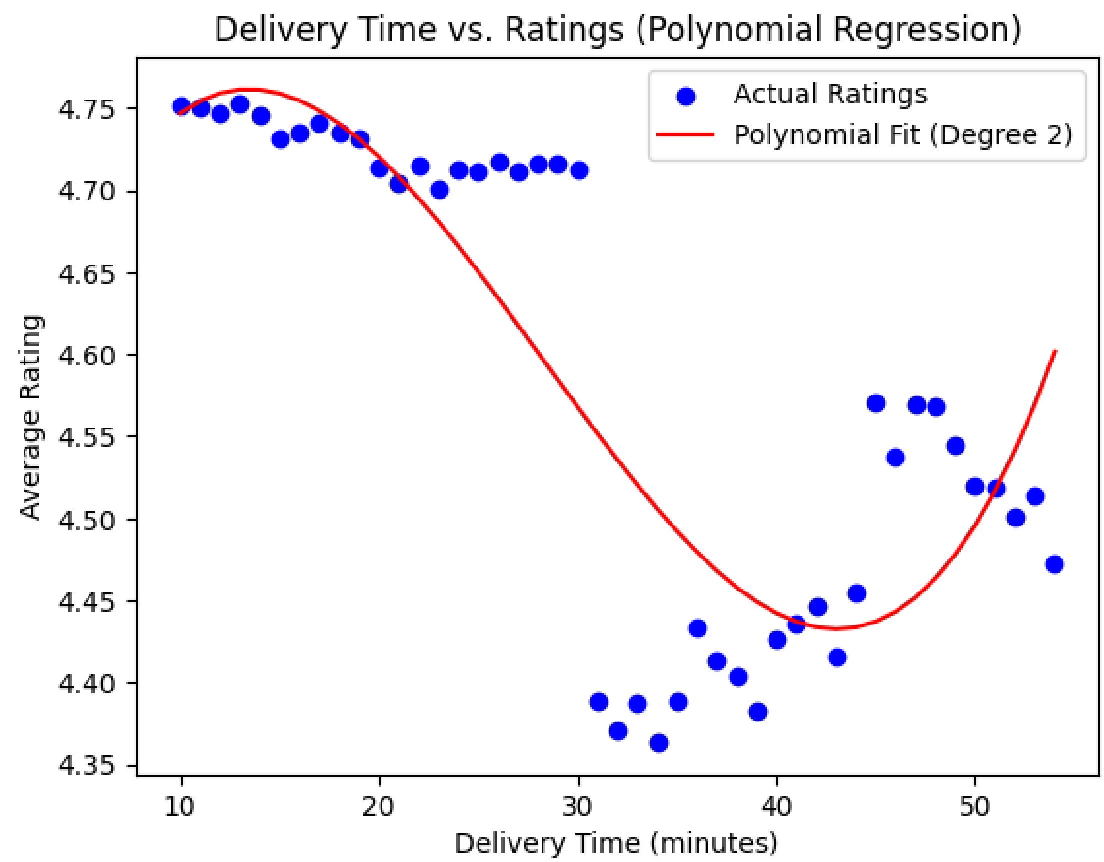
```
R-squared: 0.816
Mean Absolute Error: 0.041
```

```
Mean Squared Error: 0.003
Root Mean Squared Error: 0.053
```

We were able to get the Mean Absolute Error, Mean Squared Error and Root mean squared error. Lets try the same with a polynomial regression of higher degree.

Polynomial regression of degree 3

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression


poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(delivery_time.reshape(-1, 1))

# Fit the polynomial regression model
model = LinearRegression()
model.fit(X_poly, ratings)

# Predict values
y_pred = model.predict(X_poly)

# Plotting the data
plt.scatter(delivery_time, ratings, color='blue', label='Actual
Ratings')
plt.plot(delivery_time, y_pred, color='red', label='Polynomial Fit
(Degree 2)')
plt.title('Delivery Time vs. Ratings (Polynomial Regression)')
plt.xlabel('Delivery Time (minutes)')
plt.ylabel('Average Rating')
plt.legend()
plt.show()
```



Here we are able to get a better graph compared to a Polynomial Regression of degree 2. Let try to compute the accuracy for the same.

```
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split

X = X_poly
y = ratings

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Transform the features to polynomial features (degree 3)
degree = 3
poly_features = PolynomialFeatures(degree=degree)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# Fit the polynomial regression model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Make predictions
```

```
y_pred = model.predict(X_test_poly)

# Calculate evaluation metrics
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

# Print the evaluation metrics
print(f"R-squared: {r2:.3f}")
print(f"Mean Absolute Error: {mae:.3f}")
print(f"Mean Squared Error: {mse:.3f}")
print(f"Root Mean Squared Error: {rmse:.3f}")
```

```
R-squared: 0.931
Mean Absolute Error: 0.024
Mean Squared Error: 0.001
Root Mean Squared Error: 0.032
```

As we can see, the accuracy has increased compared to Polynimal regression of degree 2.

Here, since the grpah has a degree 3, we were able to get a graph with 2 extremes. As a result, the graph fits the data better.

Decisioin Tree Regression

```
from sklearn.tree import DecisionTreeRegressor

# Fit the decision tree regression model
tree_model = DecisionTreeRegressor(max_depth=3)  # Control depth to
prevent overfitting
tree_model.fit(delivery_time, ratings)

# Predict and plot
y_pred_tree = tree_model.predict(delivery_time)

plt.scatter(delivery_time, ratings, color='blue', label='Actual
Ratings')
plt.plot(delivery_time, y_pred_tree, color='green', label='Decision
Tree Fit')
plt.title('Decision Tree Regression')
plt.xlabel('Delivery Time (minutes)')
plt.ylabel('Average Rating')
plt.legend()
plt.show()
```

Here the model is following the graph too closely. We might run into over-fitting.

```
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor


X = delivery_time
y = ratings

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
model = DecisionTreeRegressor(random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calculate R-squared
r2 = r2_score(y_test, y_pred)
print(f"R-squared: {r2:.3f}")

# Calculate Mean Absolute Error
mae = mean_absolute_error(y_test, y_pred)
```

```
print(f"Mean Absolute Error: {mae:.3f}")

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.3f}")

# Calculate Root Mean Squared Error
rmse = np.sqrt(mse)
print(f"Root Mean Squared Error: {rmse:.3f}")
```

```
R-squared: 0.887
Mean Absolute Error: 0.025
Mean Squared Error: 0.002
Root Mean Squared Error: 0.041
```

As we see the accuracy went down, do Polynomial regression is the best fit for the data.

We can see here, we have a case of overfitting. The accuracy for Testing data has gone down. So, to conclude, Polynomial regression of degree 3 is the best fit for the data.

Using Clustering Models For Second Hypothesis

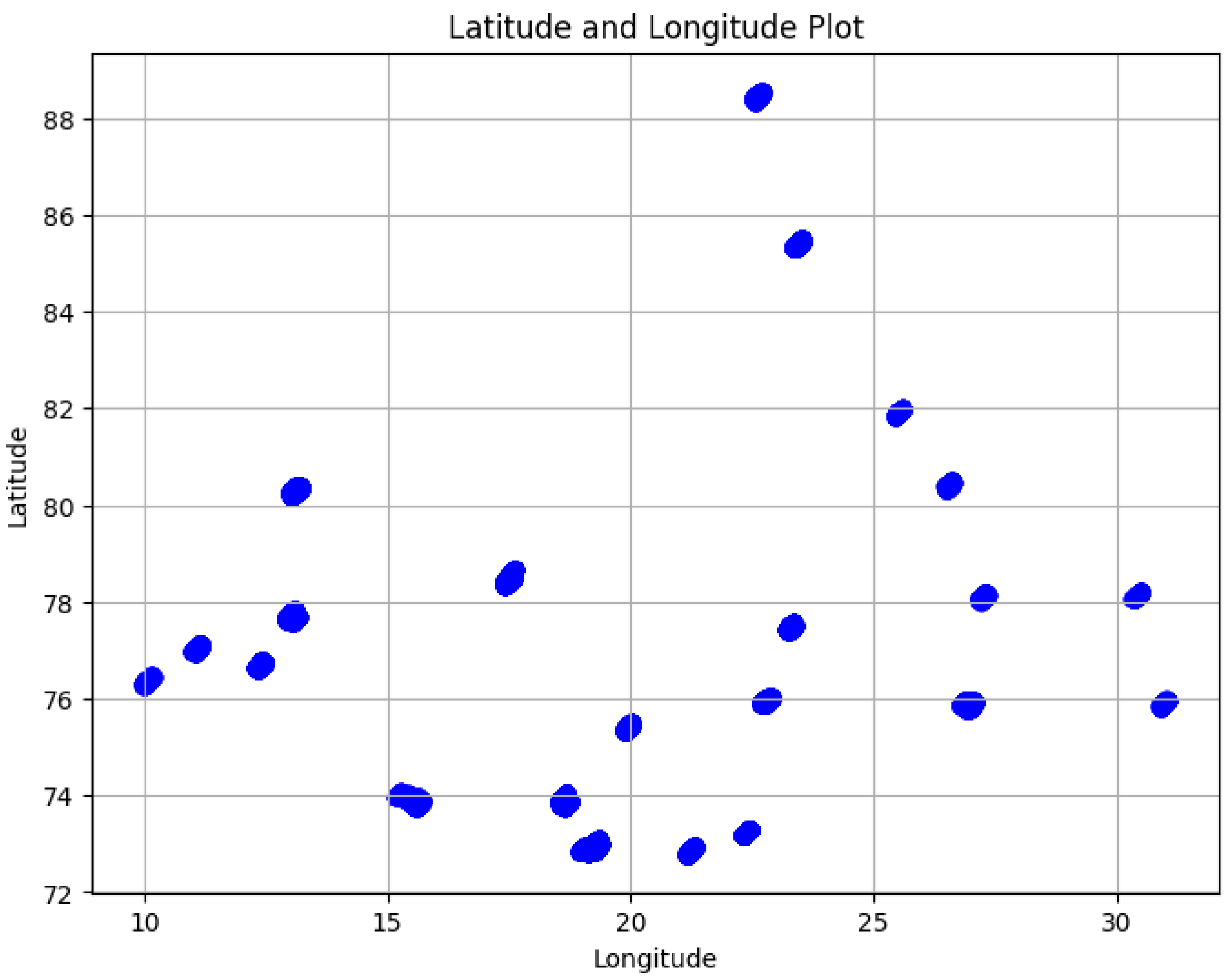
In our second hypothesis, we assumed that the as the distance increases, the delivery time increases. Here, we try to divide the observed data into clusters.

Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('../food_delivery_dataset/train.csv')

df = df.replace('NaN', np.nan, regex=True)

df['Restaurant_latitude'] =
pd.to_numeric(df['Restaurant_latitude']).abs()
df['Restaurant_longitude'] =
pd.to_numeric(df['Restaurant_longitude']).abs()
df['Delivery_location_latitude'] =
pd.to_numeric(df['Delivery_location_latitude']).abs()
df['Delivery_location_longitude'] =
pd.to_numeric(df['Delivery_location_longitude']).abs()

df = df[~((df['Restaurant_latitude'] <= 1) &
(df['Restaurant_longitude'] <= 1) &(df['Delivery_location_latitude']
<= 1) &(df['Delivery_location_longitude'] <= 1))]
```

Plotting the graph for the observed data.

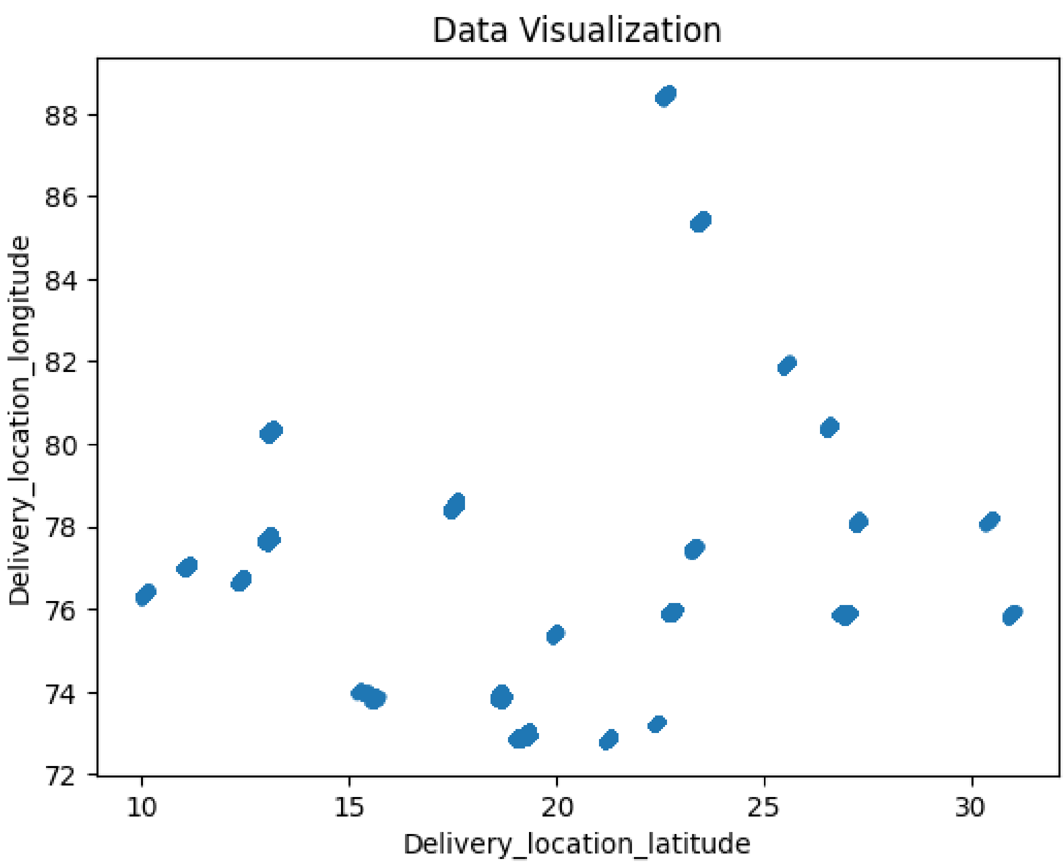
```
plt.figure(figsize=(8, 6))
# plt.scatter(df['Restaurant_latitude'], df['Restaurant_longitude'],
color='red', marker='o')
plt.scatter(df['Delivery_location_latitude'],
df['Delivery_location_longitude'], color='blue')

# Adding labels
plt.title('Latitude and Longitude Plot')
plt.xlabel('Longitude')
plt.ylabel('Latitude')

# Display the plot
plt.grid(True)
plt.show()
```

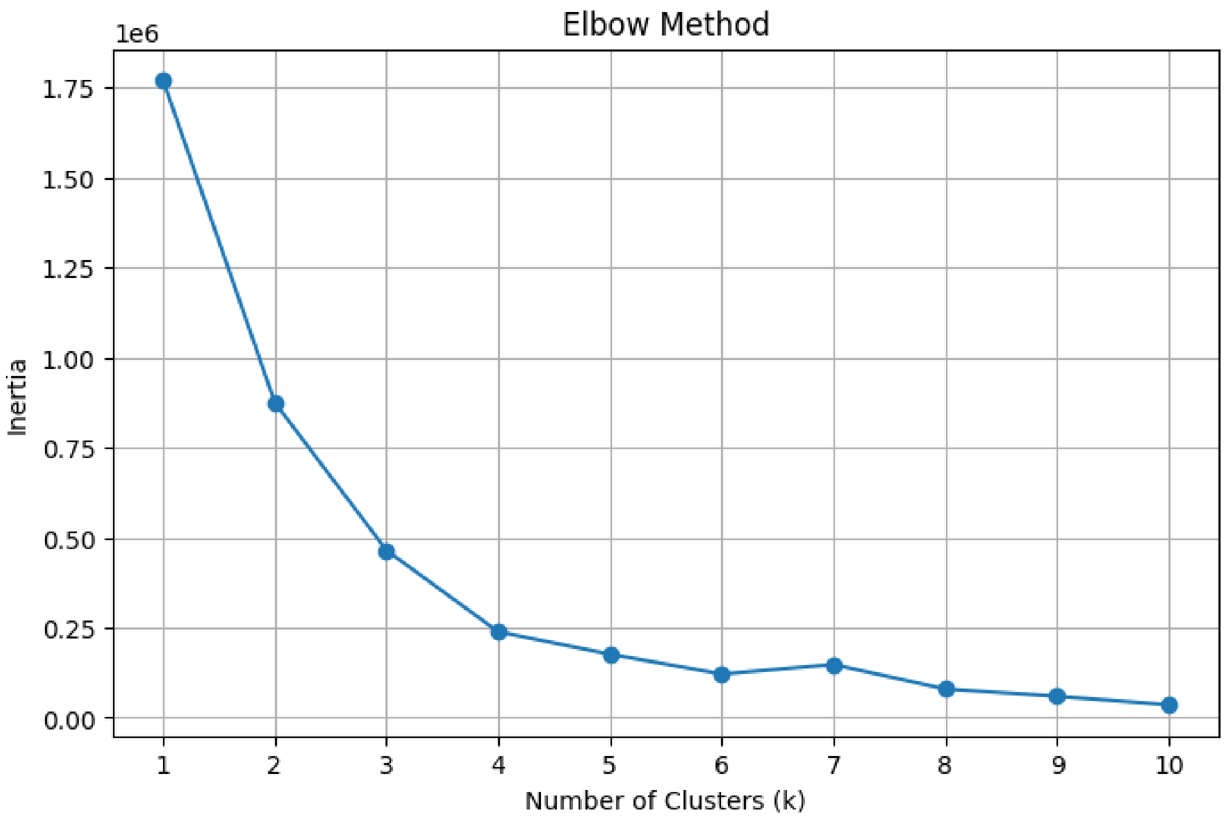


Here, we want to divide the data into different clusters. We are going to use K-Means clustering in order to classify data.

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Create a synthetic dataset
# X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)
X = df
# y = df['Restaurant_longitude']

data = pd.DataFrame(X, columns=['Delivery_location_latitude',
'Delivery_location_longitude'])

# Visualize the data
plt.scatter(data['Delivery_location_latitude'],
data['Delivery_location_longitude'], s=10)
plt.title('Data Visualization')
plt.xlabel('Delivery_location_latitude')
```

```
plt.ylabel('Delivery_location_longitude')
plt.show()

# Determine the optimal number of clusters using the Elbow Method
inertia = []
k_range = range(1, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(data)
    inertia.append(kmeans.inertia_)

# Plotting the elbow curve
plt.figure(figsize=(8, 5))
plt.plot(k_range, inertia, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia')
plt.xticks(k_range)
plt.grid()
plt.show()
```



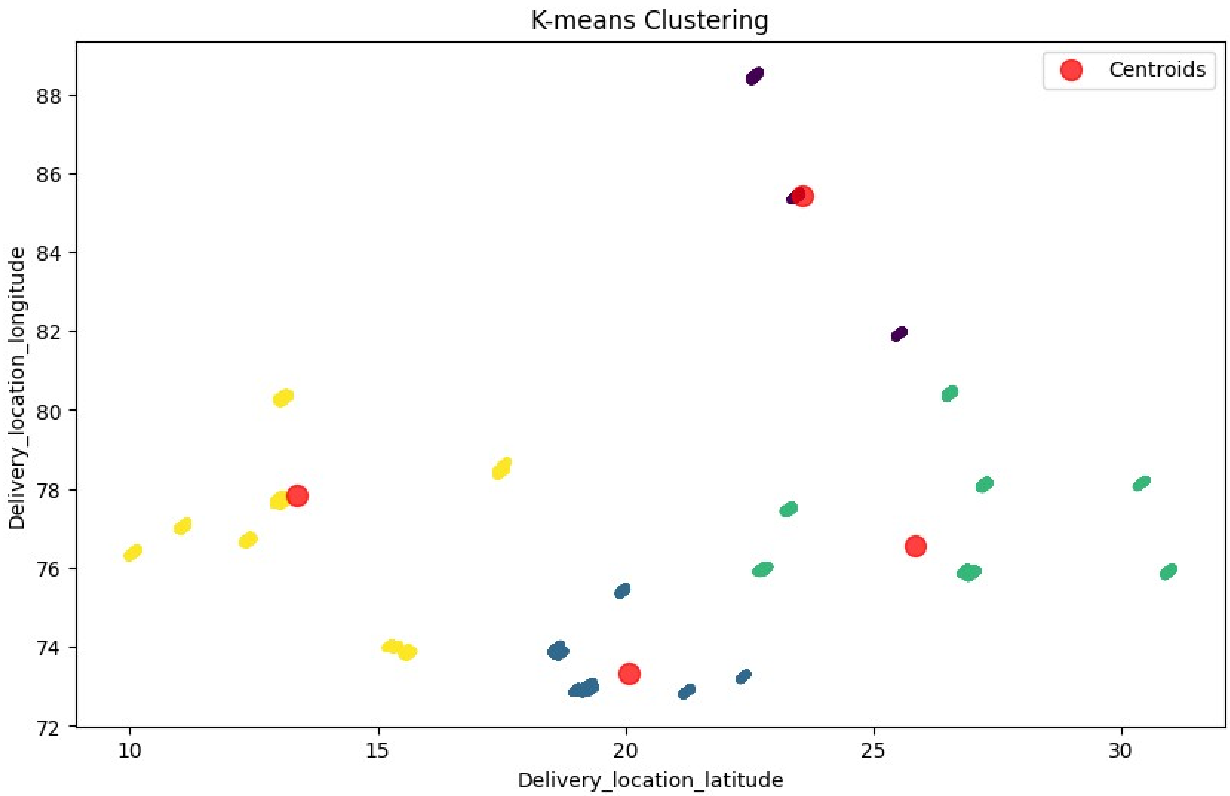


From the Elbow curce, we can see, the graph starts to flatten at k value 4. So, we can assume the optimal number of clusters to be 4.

```
k = 4  # Optimal k from the elbow method
kmeans = KMeans(n_clusters=k)
data['Cluster'] = kmeans.fit_predict(data)

# Print the cluster centers
print("Cluster Centers:\n", kmeans.cluster_centers_)

# Visualize the clusters
plt.figure(figsize=(10, 6))
plt.scatter(data['Delivery_location_latitude'],
data['Delivery_location_longitude'], c=data['Cluster'], s=10,
cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,
1], c='red', s=100, alpha=0.75, label='Centroids')
plt.title('K-means Clustering')
plt.xlabel('Delivery_location_latitude')
plt.ylabel('Delivery_location_longitude')
plt.legend()
plt.show()
```

```
Cluster Centers:
 [[ 2.35872693e+01  8.54323607e+01  3.00000000e+00]
```

```
[ 2.00590351e+01  7.33267483e+01  2.00000000e+00]
[ 2.58470781e+01  7.65467196e+01  1.33305219e+00]
[ 1.33644109e+01  7.78310401e+01 -2.68673972e-14]]
```



The above is the graph for the 4 different clusters that we received. The red dots show the centroids for each cluster.

We can vary the number of centroids to get different clusters. In our case, 4 is the optimal number of clusters that we will get form the given data.

Time Series Forecasting

Here, we will try to forecast the number of orders using Time Series Forecasting. We will cummulate the number of orders received on a daily basis.

We will then predict the number of orders for the future. We will find the prefect Time Series forecasting models to get the prediction.

Imports

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
```

Loading the CSV files

```
df1 = pd.read_csv("../food_delivery_dataset/train.csv")

df1['Order_Date'] = pd.to_datetime(df1['Order_Date'])

df1 = df1.groupby('Order_Date').size().reset_index(name='count')
```

```
/var/folders/2t/gftqwtk579jcc7_0379k_f900000gn/T/
ipykernel_64133/1130109973.py:3: UserWarning: Parsing dates in %d-%m-
%Y format when dayfirst=False (the default) was specified. Pass
`dayfirst=True` or specify a format to silence this warning.
  df1['Order_Date'] = pd.to_datetime(df1['Order_Date'])
```

```
df1.columns = ['timestamp','Count']
df1['timestamp'] = pd.to_datetime(df1['timestamp'])

df1 = df1.set_index('timestamp')
```

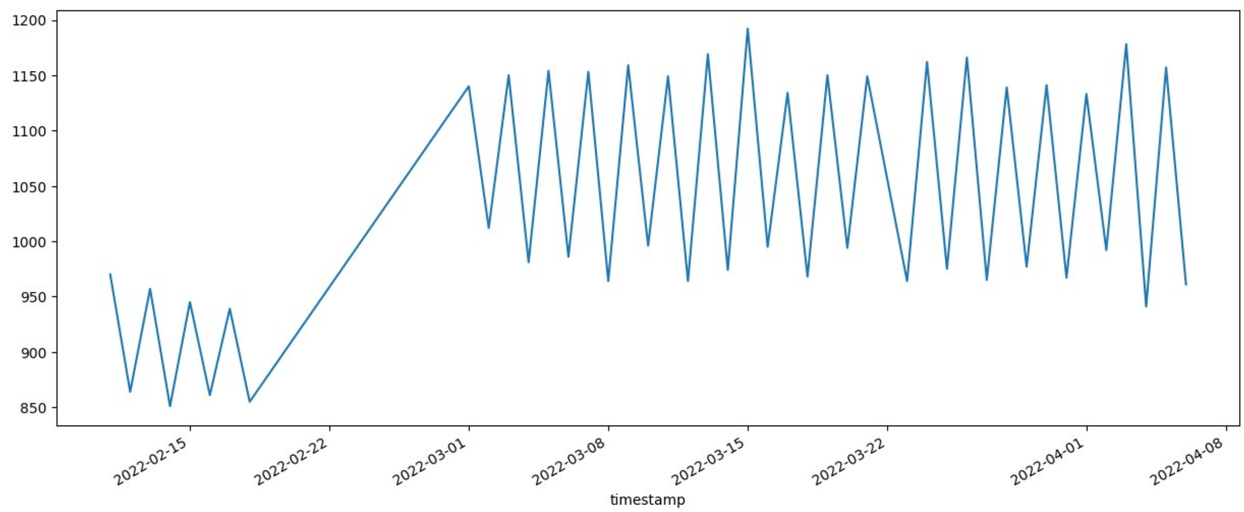
Ploting the graph with the Daily Order Count

```
y = df1['Count']
y['2022':]
```

```
timestamp
2022-02-11     970
2022-02-12     864
2022-02-13     957
```

```
2022-02-14     851
2022-02-15     945
2022-02-16     861
2022-02-17     939
2022-02-18     855
2022-03-01    1140
2022-03-02    1012
2022-03-03    1150
2022-03-04     981
2022-03-05    1154
2022-03-06     986
2022-03-07    1153
2022-03-08     964
2022-03-09    1159
2022-03-10     996
2022-03-11    1149
2022-03-12     964
2022-03-13    1169
2022-03-14     974
2022-03-15    1192
2022-03-16     995
2022-03-17    1134
2022-03-18     968
2022-03-19    1150
2022-03-20     994
2022-03-21    1149
2022-03-23     964
2022-03-24    1162
2022-03-25     975
2022-03-26    1166
2022-03-27     965
2022-03-28    1139
2022-03-29     977
2022-03-30    1141
2022-03-31     967
2022-04-01    1133
2022-04-02     992
2022-04-03    1178
2022-04-04     941
2022-04-05    1157
2022-04-06     961
Name: Count, dtype: int64
```

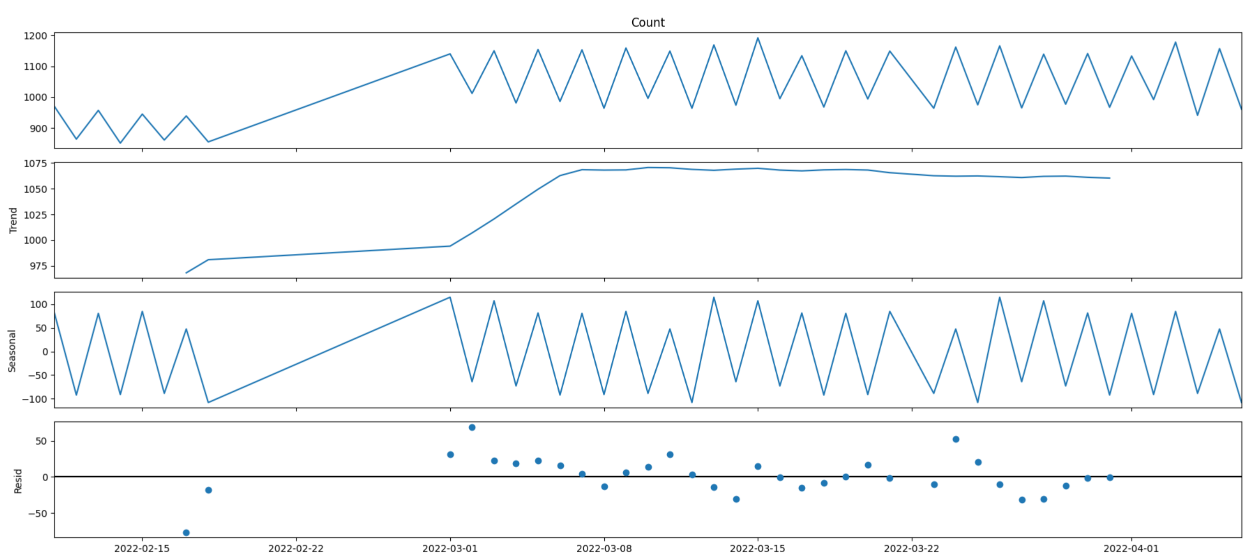
```
y.plot(figsize=(15,6))
plt.show()
```



We compute the Trend, Seaonality and Residual (Noise) of the current data.

- The trend is the long-term movement or direction in the time series data
- Seasonality refers to the repeating and predictable patterns or cycles in the data that occur at regular intervals, typically within a year, month, or week.
- The residuals (or noise) represent the random or unexplained variation in the data after removing the trend and seasonality components

```
from pylab import rcParams
rcParams['figure.figsize'] = 18, 8
decomposition = sm.tsa.seasonal_decompose(y, model='additive',
period=12)
fig = decomposition.plot()
plt.show()
```



As we can see a pattern in the Seasonality of the data. So, we use SARIMA to predict the outputs.

```
import itertools
p = d = q = range(0, 2)
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in
list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

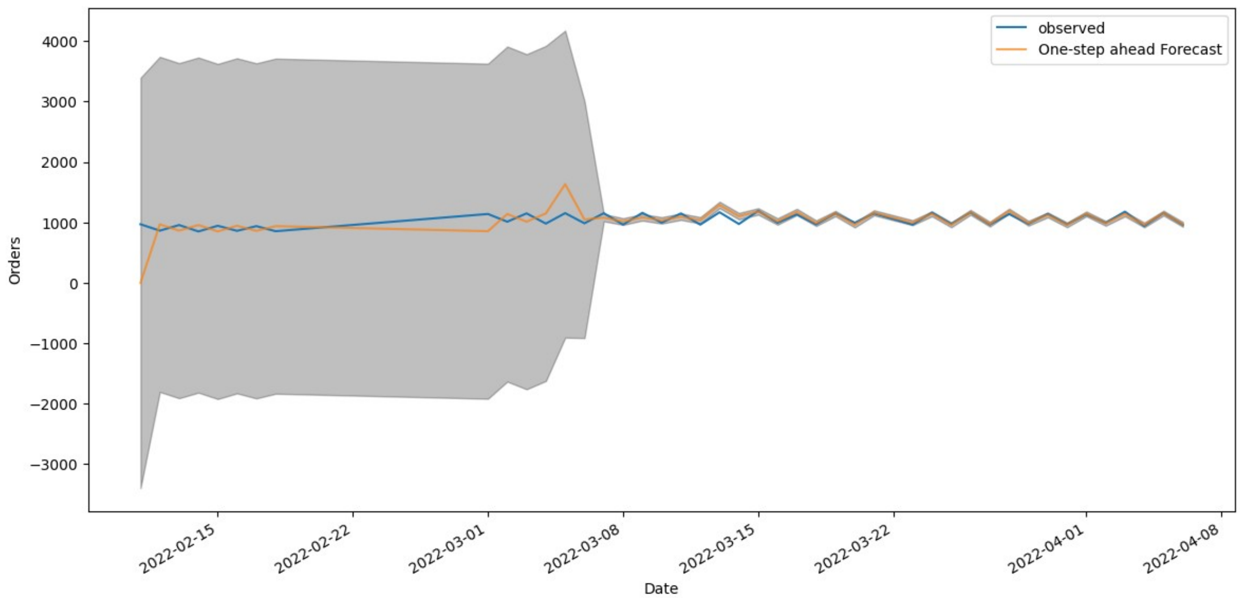
```
Examples of parameter combinations for Seasonal ARIMA...
SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
SARIMAX: (0, 0, 1) x (0, 1, 0, 12)
SARIMAX: (0, 1, 0) x (0, 1, 1, 12)
SARIMAX: (0, 1, 0) x (1, 0, 0, 12)
```

```
import warnings
warnings.filterwarnings('ignore')
for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(y,
                                            order=param,

seasonal_order=param_seasonal,

enforce_stationarity=False,

enforce_invertibility=False)
            results = mod.fit()

            print('SARIMA{}x{}12 - AIC:{}'.format(param,
param_seasonal, results.aic))
        except Exception as e:
            print(e)
            break
```

We try to plot the graph using the values obtained using SARIMA Model

```
pred = results.get_prediction(start=pd.to_datetime('2022-02-11'),
dynamic=False)
pred_ci = pred.conf_int()
ax = y['2022':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast',
```

```
alpha=.7, figsize=(14, 7))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Orders')
plt.legend()
plt.show()
```



We compute the Mean Square Error For the SARIMA model.

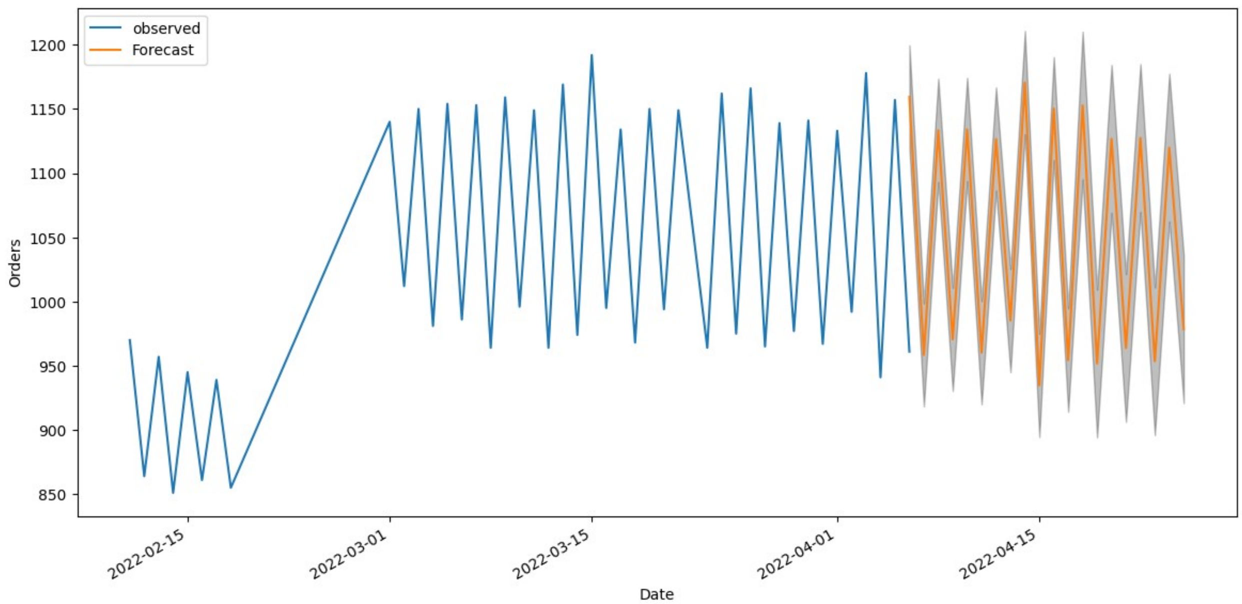
```
y_forecasted = pred.predicted_mean
y_truth = y['2022-02-11':]
mse = ((y_forecasted - y_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is
{}'.format(round(mse, 2)))

The Mean Squared Error of our forecasts is 32824.41
```

We compute the Root Mean Squared Error for the SARIMA Model

```
print('The Root Mean Squared Error of our forecasts is
{}'.format(round(np.sqrt(mse), 2)))

The Root Mean Squared Error of our forecasts is 181.18
```

We now want to plot the predicts for the given data.

```
pred_uc = results.get_forecast(steps=20)
pred_ci = pred_uc.conf_int()
forecast_index = pd.date_range(start='2022-04-06',
periods=len(pred_uc.predicted_mean), freq='D')
pred_ci.index = pd.date_range(start='2022-04-06',
periods=len(pred_ci.index), freq='D')
ax = y.plot(label='observed', figsize=(14, 7))
forecast = pred_uc.predicted_mean
forecast.index = forecast_index
forecast.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Orders')
plt.legend()
plt.show()
```



Here, we see the orange line is the predicted value, based on the SARIMA Time Series Forecasting. The gray part is the expected noise in the data.

We can use the predicted values for various advancements and optimissatins.

One problem with the given data is the Dataset size. We have around about 2 months of data. In order to unlock the true potential of Time Series Forecasting, we need atleast 2 years of data.

We will perform the above predictions but on a bigger dataset in-order to test the performance of Time Series Forecasting.

Time Series Forecasting For A Bigger Dataset

Imports

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
```

Loading Dataset

```python
df_triain = pd.read_csv("../food_delivery_dataset/train.csv")

df_triain['Order_Date'] = pd.to_datetime(df_triain['Order_Date'])

date_counts = 
df_triain.groupby('Order_Date').size().reset_index(name='count')

print(date_counts)
```

```
   Order_Date  count
0  2022-02-11    970
1  2022-02-12    864
2  2022-02-13    957
3  2022-02-14    851
4  2022-02-15    945
5  2022-02-16    861
6  2022-02-17    939
7  2022-02-18    855
8  2022-03-01   1140
9  2022-03-02   1012
10 2022-03-03   1150
11 2022-03-04    981
12 2022-03-05   1154
13 2022-03-06    986
14 2022-03-07   1153
15 2022-03-08    964
16 2022-03-09   1159
17 2022-03-10    996
18 2022-03-11   1149
19 2022-03-12    964
20 2022-03-13   1169
21 2022-03-14    974
22 2022-03-15   1192
23 2022-03-16    995
24 2022-03-17   1134
25 2022-03-18    968
```

```
26 2022-03-19   1150
27 2022-03-20    994
28 2022-03-21   1149
29 2022-03-23    964
30 2022-03-24   1162
31 2022-03-25    975
32 2022-03-26   1166
33 2022-03-27    965
34 2022-03-28   1139
35 2022-03-29    977
36 2022-03-30   1141
37 2022-03-31    967
38 2022-04-01   1133
39 2022-04-02    992
40 2022-04-03   1178
41 2022-04-04    941
42 2022-04-05   1157
43 2022-04-06    961
```

```
/var/folders/2t/gftqwtk579jcc7_0379k_f900000gn/T/
ipykernel_39335/997718512.py:3: UserWarning: Parsing dates in %d-%m-%Y
format when dayfirst=False (the default) was specified. Pass
`dayfirst=True` or specify a format to silence this warning.
  df_triain['Order_Date'] = pd.to_datetime(df_triain['Order_Date'])
```

```
df = pd.read_csv("../food_delivery_dataset/olist_orders_dataset.csv")
cols =
['order_id','customer_id','order_status','order_approved_at','order_de
livered_carrier_date','order_delivered_customer_date','order_estimated
_delivery_date']
df.drop(cols,axis=1,inplace=True)
df['order_purchase_timestamp'] =
pd.to_datetime(df['order_purchase_timestamp'])
df['date'] = df['order_purchase_timestamp']
df.head()
```

```
  order_purchase_timestamp                date
0      2017-10-02 10:56:33 2017-10-02 10:56:33
1      2018-07-24 20:41:37 2018-07-24 20:41:37
2      2018-08-08 08:38:49 2018-08-08 08:38:49
3      2017-11-18 19:28:06 2017-11-18 19:28:06
4      2018-02-13 21:18:39 2018-02-13 21:18:39
```

```
df.isnull().sum()
```

```
order_purchase_timestamp    0
date                        0
dtype: int64
```

```
df1 = pd.read_csv('../food_delivery_dataset/data3.csv')
df1.columns = ['orderId','timestamp','Count']
```

```
df1['timestamp'] = pd.to_datetime(df1['timestamp'])
df1.tail(20)
```

```
     orderId  timestamp  Count
710      971 2018-12-12     35
711      606 2018-12-13     72
712      134 2018-12-14    214
713        9 2018-12-15    340
714      315 2018-12-16    135
715      745 2018-12-17     55
716      917 2018-12-18     40
717      398 2018-12-19    112
718      470 2018-12-20     99
719      163 2018-12-21    200
720       10 2018-12-22    338
721      376 2018-12-23    118
722      111 2018-12-24    229
723     9992 2018-12-25    178
724      731 2018-12-26     56
725      227 2018-12-27    165
726      387 2018-12-28    116
727      210 2018-12-29    173
728      292 2018-12-30    142
729       20 2018-12-31    314
```

```
df1 = df1.set_index('timestamp')
df1.tail()
```

```
            orderId  Count
timestamp
2018-12-27      227    165
2018-12-28      387    116
2018-12-29      210    173
2018-12-30      292    142
2018-12-31       20    314
```

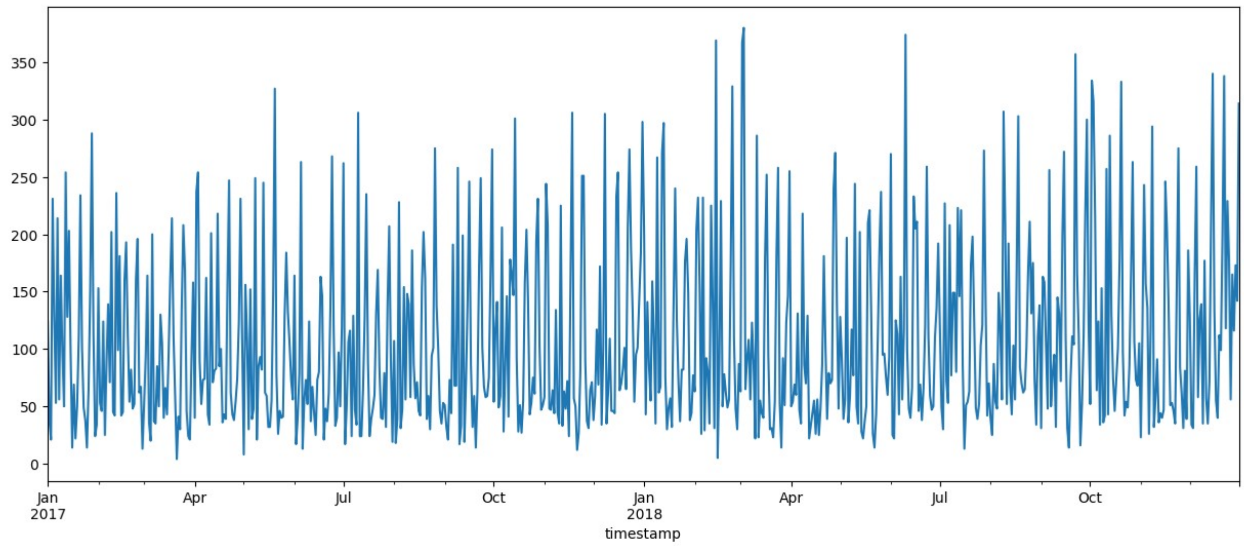
```
y = df1['Count']
y['2017':]
```

```
timestamp
2017-01-01    124
2017-01-02     38
2017-01-03     21
2017-01-04    231
2017-01-05    129
             ...
2018-12-27    165
2018-12-28    116
2018-12-29    173
2018-12-30    142
```

```
2018-12-31    314
Name: Count, Length: 730, dtype: int64
```

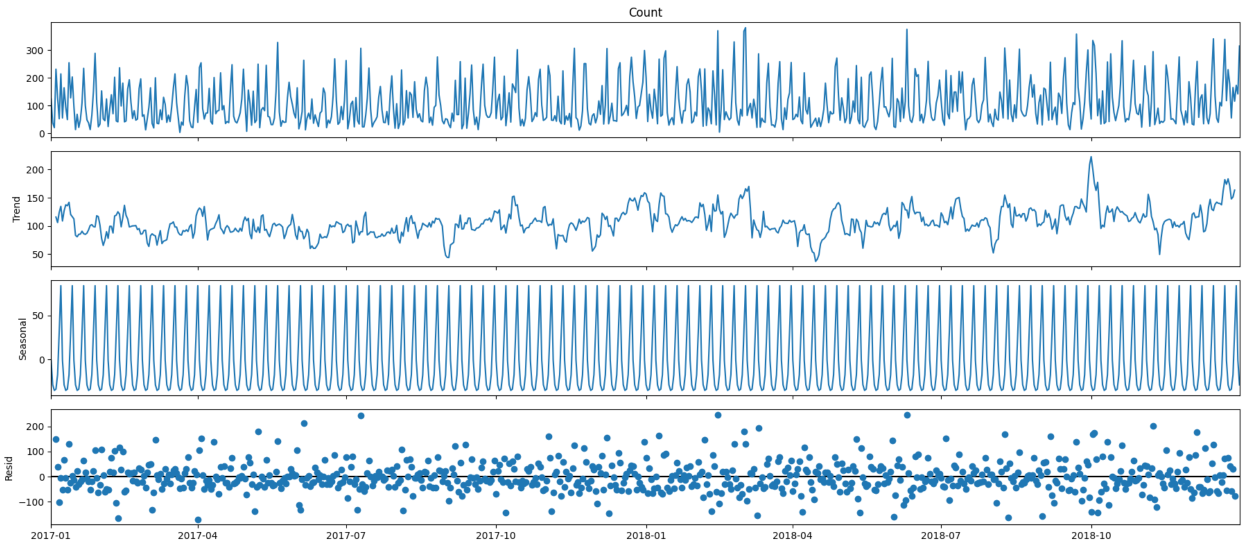
Plotting the Daily Order Count

```
y.plot(figsize=(15,6))
plt.show()
```



We compute the Trend, Seaonality and Residual (Noise) of the current data.

```
from pylab import rcParams
rcParams['figure.figsize'] = 18, 8
decomposition = sm.tsa.seasonal_decompose(y, model='additive')
fig = decomposition.plot()
plt.show()
```




```
import itertools
p = d = q = range(0, 2)
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in
list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

```
Examples of parameter combinations for Seasonal ARIMA...
SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
SARIMAX: (0, 0, 1) x (0, 1, 0, 12)
SARIMAX: (0, 1, 0) x (0, 1, 1, 12)
SARIMAX: (0, 1, 0) x (1, 0, 0, 12)
```

```
import warnings
warnings.filterwarnings('ignore')
for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(y,
                                            order=param,

seasonal_order=param_seasonal,

enforce_stationarity=False,

enforce_invertibility=False)
            results = mod.fit()

            print('ARIMA{}x{}12 - AIC:{}'.format(param,
```

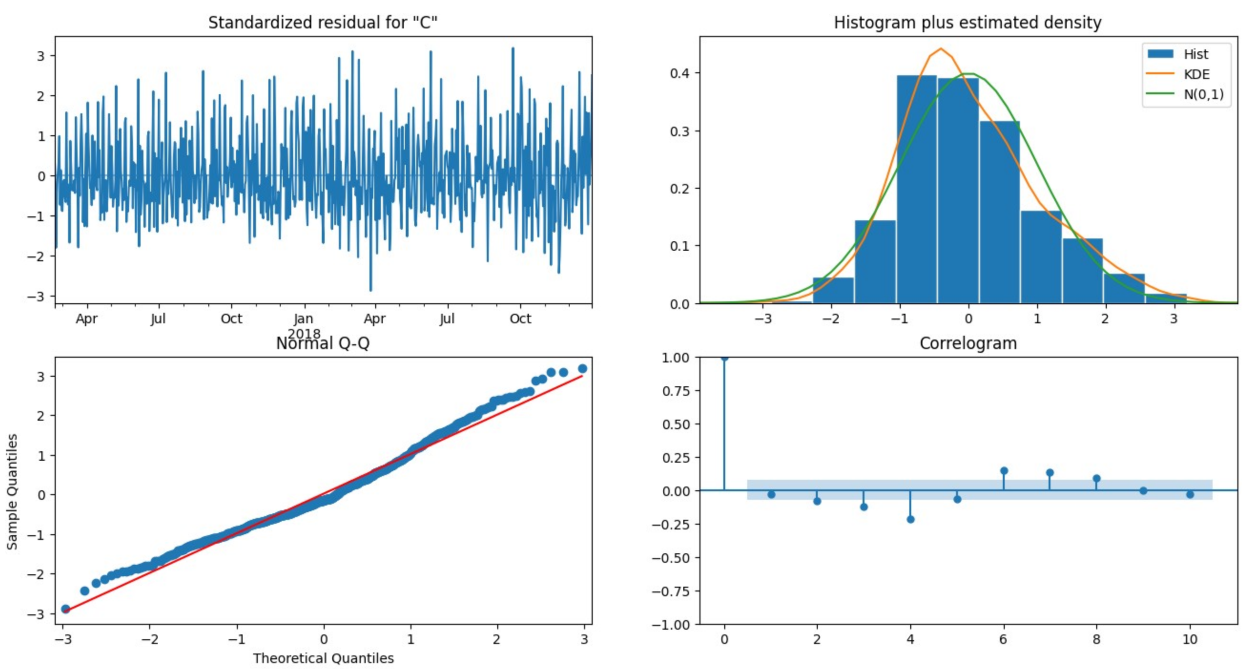
```
param_seasonal, results.aic))
        except:
            continue

mod = sm.tsa.statespace.SARIMAX(y,
                                order=(1, 0, 1),
                                seasonal_order=(1, 1, 0, 24),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results = mod.fit()
print(results.summary().tables[1])
```

```
 This problem is unconstrained.

RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            4     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  5.61851D+00    |proj g|=  7.71799D-02

At iterate    5    f=  5.60800D+00    |proj g|=  9.00686D-02

At iterate   10    f=  5.60095D+00    |proj g|=  2.84891D-03

At iterate   15    f=  5.60014D+00    |proj g|=  2.80912D-02

At iterate   20    f=  5.56566D+00    |proj g|=  1.22589D-01

At iterate   25    f=  5.56124D+00    |proj g|=  1.30559D-03

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    4     27     37      1     0     0   3.223D-06   5.561D+00
  F =   5.5612345985354974
```

```
CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
--------
ar.L1         -0.5877      0.096     -6.123      0.000      -0.776
-0.400
ma.L1          0.7788      0.073     10.661      0.000       0.636
0.922
ar.S.L24      -0.5877      0.031    -18.976      0.000      -0.648
-0.527
sigma2      8819.8890    463.007     19.049      0.000    7912.413
9727.365
======================================================================
========
```

Plotting Different Diagnosis For Forecasting

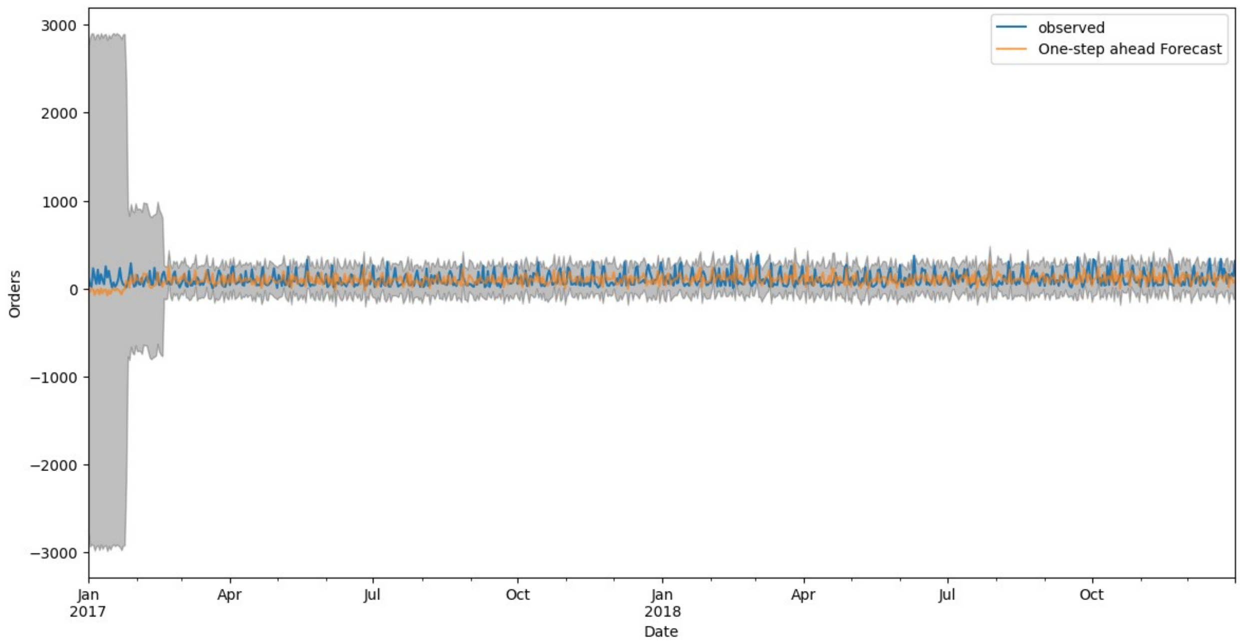
```
results.plot_diagnostics(figsize=(16, 8))
plt.show()
```



We try to plot the graph using the values obtained using SARIMA Model

```
pred = results.get_prediction(start=pd.to_datetime('2017-01-01'),
dynamic=False)
```

```
pred_ci = pred.conf_int()
ax = y['2017':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast',
alpha=.7, figsize=(14, 7))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Orders')
plt.legend()
plt.show()
```

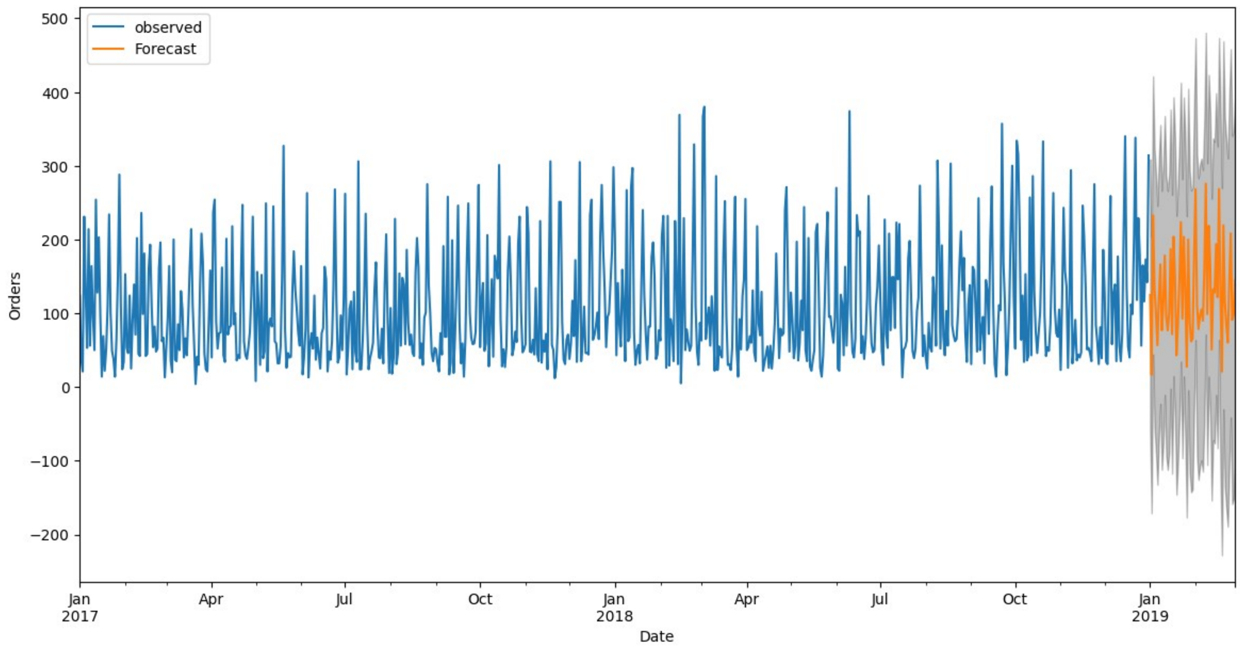


Here, since the number of data points is huge, we get a more plausible graph when predicting the values.

Computing the Squared Error

```
y_forecasted = pred.predicted_mean
y_truth = y['2017-01-01':]
mse = ((y_forecasted - y_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is
{}'.format(round(mse, 2)))

The Mean Squared Error of our forecasts is 9288.75

print('The Root Mean Squared Error of our forecasts is
{}'.format(round(np.sqrt(mse), 2)))

The Root Mean Squared Error of our forecasts is 96.38
```

```
pred_uc = results.get_forecast(steps=59)
pred_ci = pred_uc.conf_int()
ax = y.plot(label='observed', figsize=(14, 7))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Orders')
plt.legend()
plt.show()
```



Here, we see the orange line is the predicted value, based on the SARIMA Time Series Forecasting. The gray part is the expected noise in the data.

Here the predictions is uniform compared to a smaller dataset.

```
pred_uc.predicted_mean.to_csv('../food_delivery_dataset/abc.csv',index
= False, header=True)
```

We can extract the values form the above prediction value. This value can be fed to an application, so that the concerned organisation can take necessary action.

```
pred_uc.predicted_mean
```

2019-01-01	124.717859
2019-01-02	16.251489
2019-01-03	232.714261
2019-01-04	134.477453
2019-01-05	100.182027
2019-01-06	56.580006
2019-01-07	121.191987
2019-01-08	166.156955
2019-01-09	76.856444
2019-01-10	112.225858
2019-01-11	178.322048
2019-01-12	95.417908
2019-01-13	76.742520
2019-01-14	100.637279
2019-01-15	186.990528
2019-01-16	71.557686
2019-01-17	203.738412
2019-01-18	148.610483
2019-01-19	43.073988
2019-01-20	86.248305
2019-01-21	145.385349
2019-01-22	223.540349
2019-01-23	92.634765
2019-01-24	203.514866
2019-01-25	133.111396
2019-01-26	27.269647
2019-01-27	199.971828
2019-01-28	88.945033
2019-01-29	61.875495
2019-01-30	65.642108
2019-01-31	175.733938
2019-02-01	268.322030
2019-02-02	111.026584
2019-02-03	78.595037
2019-02-04	97.032154
2019-02-05	105.162967
2019-02-06	89.822924
2019-02-07	159.031332
2019-02-08	275.736641
2019-02-09	98.851171
2019-02-10	218.584287
2019-02-11	165.882283
2019-02-12	50.670421
2019-02-13	132.529553
2019-02-14	128.115998
2019-02-15	193.838507
2019-02-16	121.646009
2019-02-17	268.445403
2019-02-18	128.178634
2019-02-19	20.794432

```
2019-02-20    219.214089
2019-02-21    115.703787
2019-02-22     84.387697
2019-02-23     60.316440
2019-02-24    143.680412
2019-02-25    208.281073
2019-02-26     90.945281
2019-02-27     98.359391
2019-02-28    144.805065
Freq: D, Name: predicted_mean, dtype: float64
```