Dominos - Predictive Purchase Order System

BY

SAMUELSON G

Introduction to the Domain

The food service industry is a fast-paced sector that demands efficient management of inventory and sales forecasting to satisfy customer needs.

Companies like Dominos leverage data analytics to streamline operations, reduce waste, and enhance customer satisfaction.

Utilizing historical sales data allows for informed decision-making and improved operational efficiency.

Problem Statement

Dominos seeks to optimize its ingredient ordering process by accurately predicting future sales, thereby minimizing waste and preventing stockouts.

Data Cleaning and Preprocessing

Null Value Imputation: Missing values were filled using mean or median to maintain dataset integrity.

Data Formatting: Ensured consistent date formats and categorical variables for effective analysis.

Data Cleaning

- Handling missing Data: Delete all null values

```
[ ]  pizza_df.dropna(inplace=True)
```

```
[ ]  pizza_df.isna().sum()
```

	0
pizza_id	0
order_id	0
pizza_name_id	16
quantity	0
order_date	0
order_time	0
unit_price	0
total_price	7
pizza_size	0
pizza_category	23
pizza_ingredients	13
pizza_name	7

```
[ ]  pizza_df.isna().sum()
```

	0
pizza_id	0
order_id	0
pizza_name_id	0
quantity	0
order_date	0
order_time	0
unit_price	0
total_price	0
pizza_size	0
pizza_category	0
pizza_ingredients	0
pizza_name	0

Feature Engineering

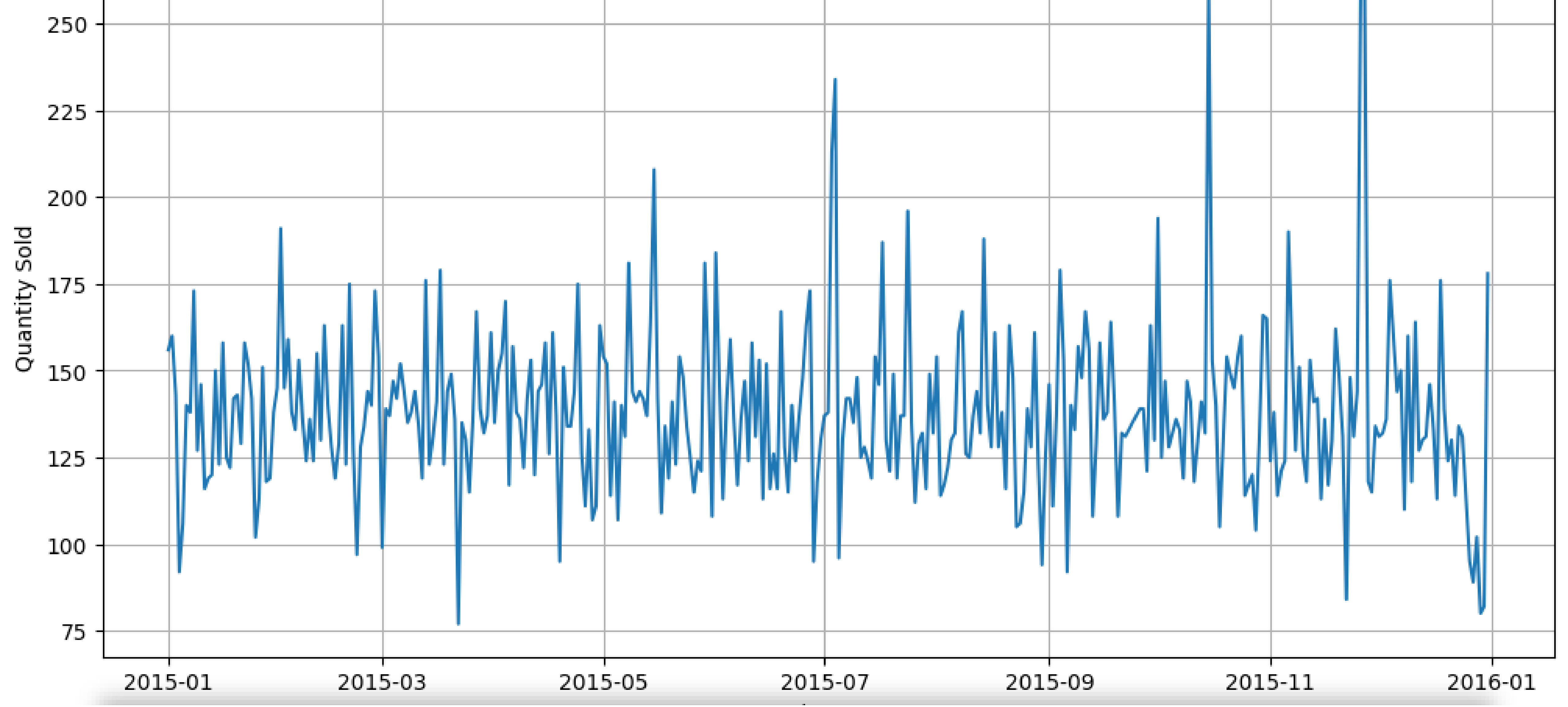
```
[ ] def parse_dates(date):
      for fmt in ('%d-%m-%Y', '%d/%m/%Y'):
        try:
          return pd.to_datetime(date, format=fmt)
        except ValueError:
          pass
      raise ValueError(f'no valid date format found for {date}')
```

```
[ ] pizza_df['order_date'] = pizza_df['order_date'].apply(parse_dates)
```

```
[ ] pizza_df['order_date'].head()
```

	order_date
0	2015-01-01
1	2015-01-01

Modify the 'order_date' column data type from string to the appropriate date time format.

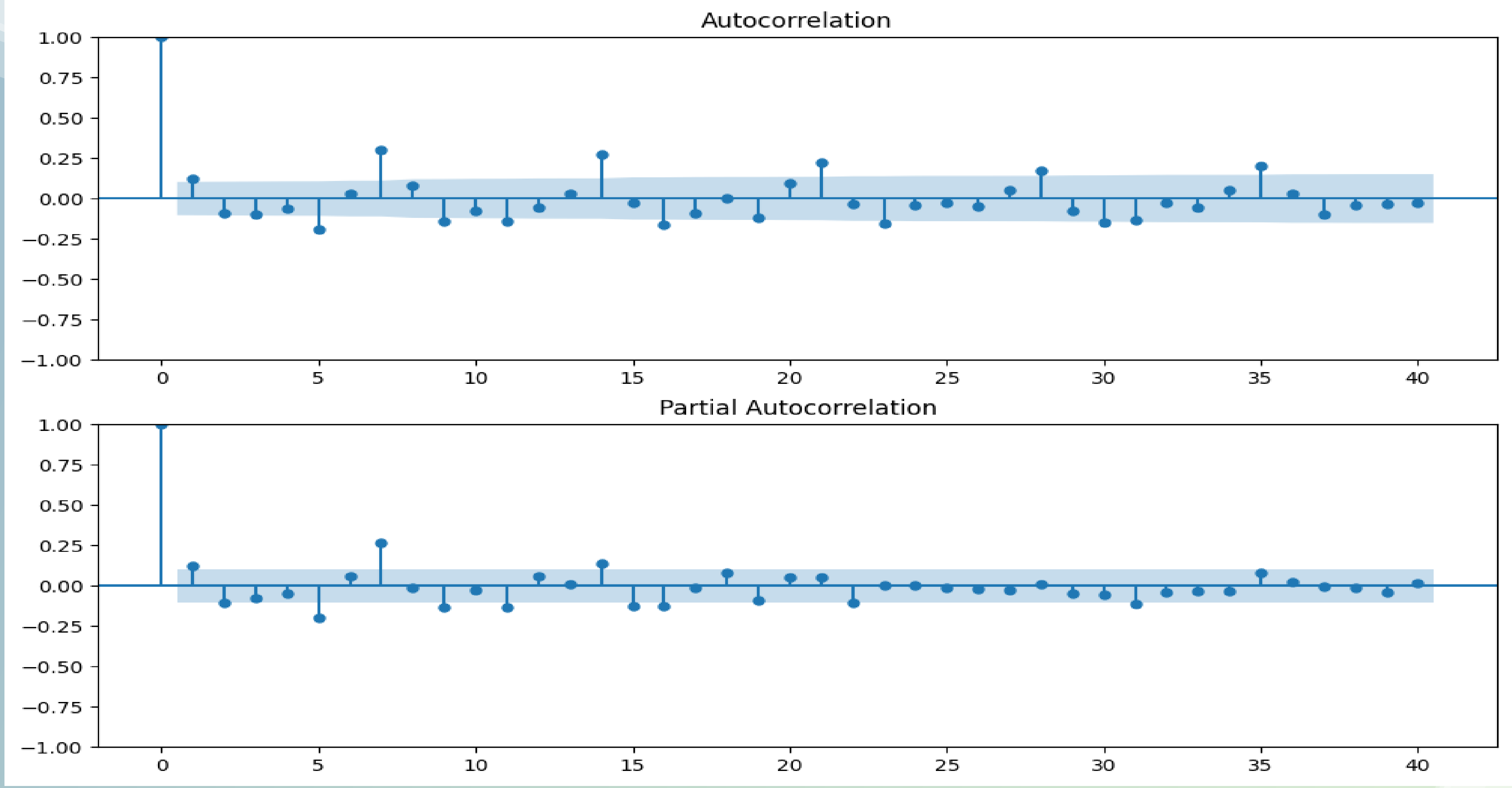


EDA(quantity if pizzas sold over time)

Autocorrelation and Partial Autocorrelation

- Autocorrelation measures the linear relationship between a time series and its lagged values. In simpler terms, it assesses how much the current value of a series depends on its past values. Autocorrelation is fundamental in time series analysis, helping identify patterns and dependencies within the data.(q)
- Partial autocorrelation removes the influence of intermediate lags, providing a clearer picture of the direct relationship between a variable and its past values. Unlike autocorrelation, partial autocorrelation focuses on the direct correlation at each lag. This is particularly useful for identifying the order of autoregressive (AR) models.(p)

Autocorrelation Function(acf) and partial Autocorrelation Function(pacf)



Q = 1

P = 1

Determining - p, q and d values

In an ARIMA (p,d,q) model, p, d, and q are parameters that specify the model's components:

p: The order of the autoregressive model, also known as the lag order,

(p): Number of lagged observations included (AR part)

d: The degree of differencing, or the number of times the raw observations are differenced,

(d): Number of times the series is differenced to achieve stationarity.

q: The order of the moving-average model, also known as the size of the moving average window,

(q): Number of lagged forecast errors included (MA part)

These parameters together help in building a model that can effectively capture the underlying patterns in the time series data and make accurate forecasts

Statistical Significance

Used the **Augmented Dickey-Fuller (ADF) test** to check for stationarity in the time series data(degree of differencing), which is crucial for time series forecasting models. The ADF test was chosen for its effectiveness in determining whether a unit root is present in the series.

```python
[ ] from statsmodels.tsa.stattools import adfuller

[ ] # H0: it is not stationary
    # H1: it is stationary

    def adf_test(sales):
      result = adfuller(sales)
      print('ADF Statistic: %f' % result[0])
      print('p-value: %f' % result[1])
      print(f'# Lags used: {result[2]}')
      print(f'No of observations used: {result[3]}')
      if result[1] <= 0.05:
        print('we reject the null hypothesis, The series is stationary')
      else:
        print('Not enough statistical evidence to reject null hypothesis, The series is not stationary')

[ ] adf_test(quantity_over_time)
```

```
ADF Statistic: -5.208670
p-value: 0.000008
# Lags used: 15
No of observations used: 342
we reject the null hypothesis, The series is stationary
```

Augmented dickey-fuller test

D = 0

Model Building

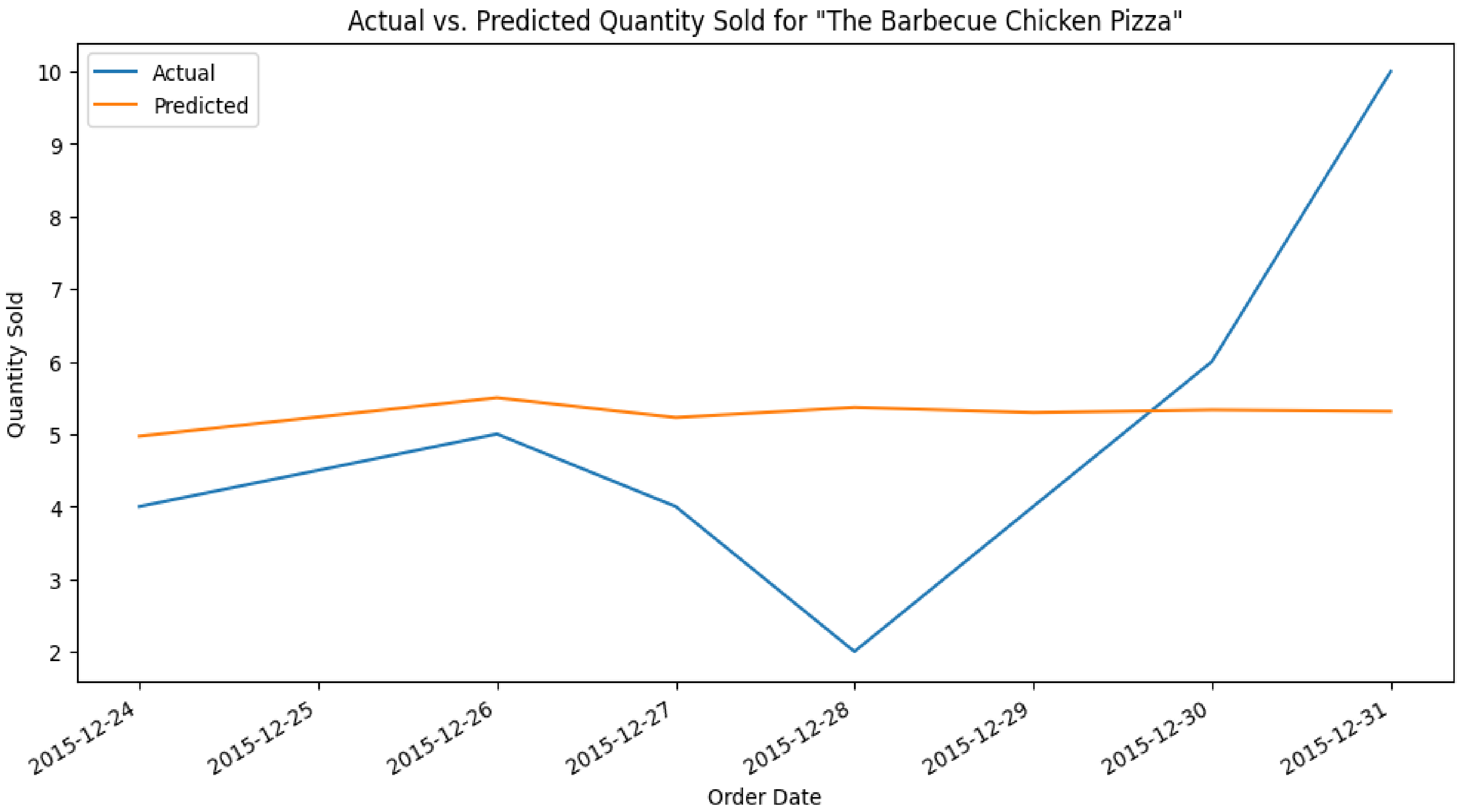
The **base model** selected was **ARIMA**, chosen for its strong capability in capturing temporal dependencies in time series data.

Models Used

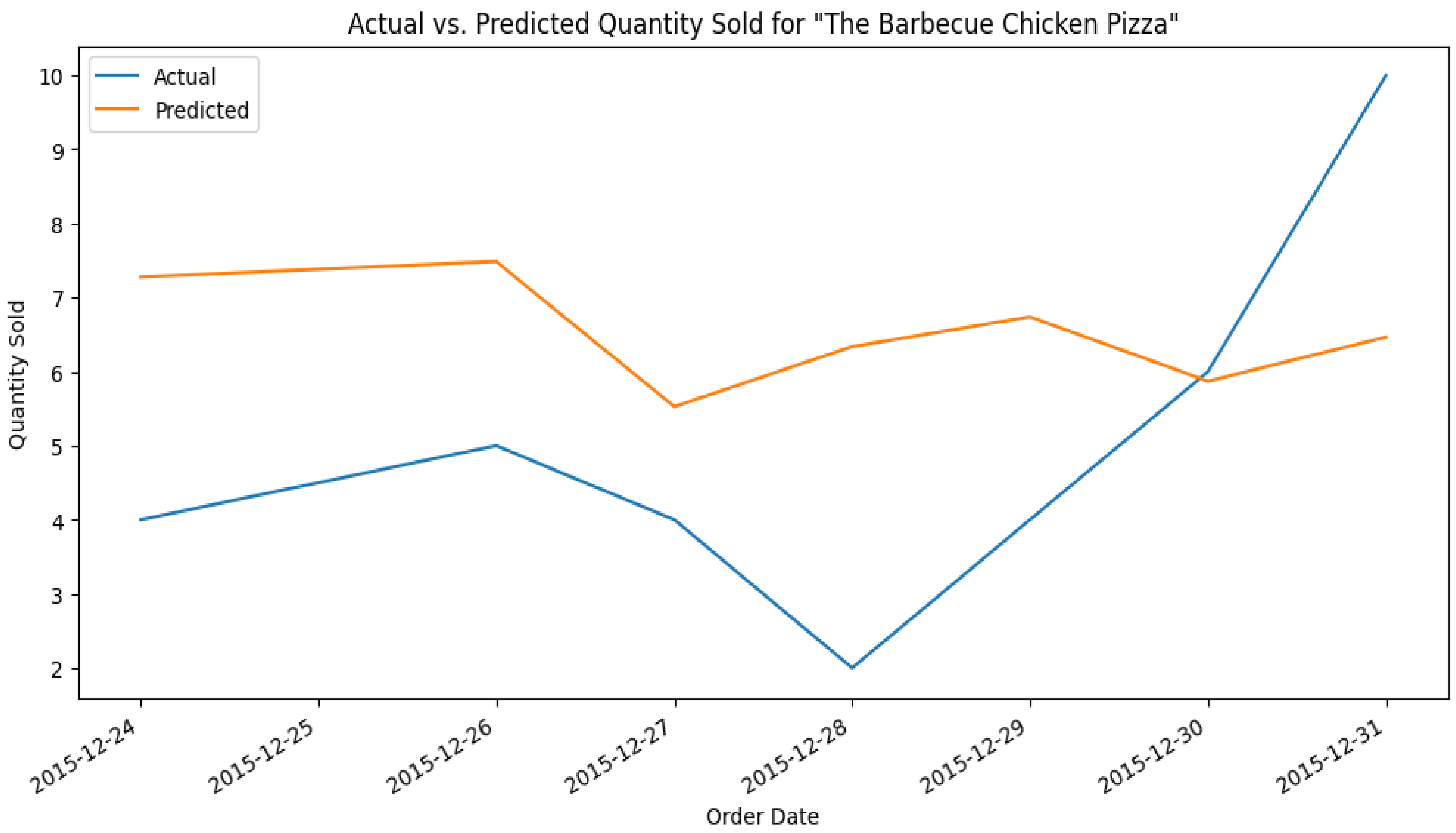
Evaluated multiple models including:

- **ARIMA(AutoRegressive Integrated Moving Average)**: For its simplicity and effectiveness.
- **SARIMA(Seasonal ARIMA)**: To account for seasonality.
- **Prophet**: Selected for its ability to handle missing data and incorporate seasonality and holiday effects easily.

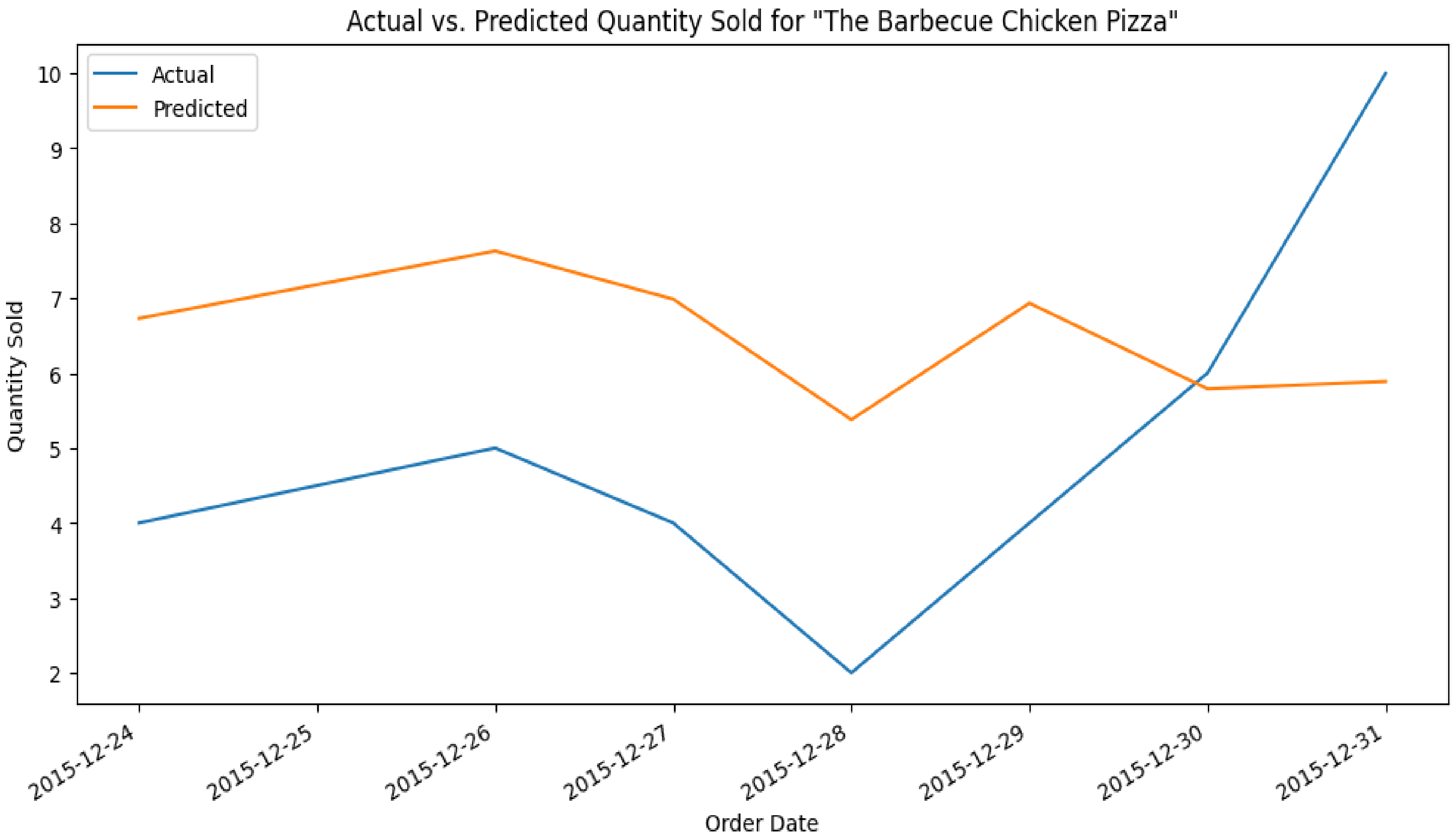
Arima model evaluation



Sarimax model



Prophet model



Model Evaluation Metric

Mean Absolute Error (MAE), **Mean Squared Error (MSE)**, and **Root Mean Squared Error (RMSE)** were used as evaluation metrics to assess the model's performance and accuracy in predictions.

```
[ ] # Evaluate the model
    mae = mean_absolute_error(test, ari_fore_values)
    mse = mean_squared_error(test, ari_fore_values)
    rmse = np.sqrt(mse)

    print(f'ARIMA MAE: {mae}')
    print(f'ARIMA MSE: {mse}')
    print(f'ARIMA RMSE: {rmse}')
```

```
ARIMA MAE: 1.8163972880345898
ARIMA MSE: 5.446642862241798
ARIMA RMSE: 2.3338043753155056
```

```
[ ] # Evaluate Sarima
    mae = mean_absolute_error(test, sari_fore_values)
    mse = mean_squared_error(test, sari_fore_values)
    rmse = np.sqrt(mse)

    print(f'SARIMAX MAE: {mae}')
    print(f'SARIMAX MSE: {mse}')
    print(f'SARIMAX RMSE: {rmse}')
```

```
SARIMAX MAE: 2.5756288583705804
SARIMAX MSE: 8.295832525545224
SARIMAX RMSE: 2.880248691614185
```

```
[ ] # Evaluate the model
    mae = mean_absolute_error(pr_test['y'], pr_forecast_values)
    mse = mean_squared_error(pr_test['y'], pr_forecast_values)
    rmse = np.sqrt(mse)

    print(f'prophet MAE: {mae}')
    print(f'prophet MSE: {mse}')
    print(f'prophet RMSE: {rmse}')
```

```
prophet MAE: 2.7110449928541485
prophet MSE: 8.6065420624295
prophet RMSE: 2.933690860065099
```

Comparison of Errors

Final Model

The final model chosen was **ARIMA**, as it provided the best performance in terms of accuracy and effectively captured both trends and seasonal components in the sales data.

Model Training

```
[ ]  # reshape data for time series modeling
     sales_pivot = sales_summary.pivot(index='order_date', columns='pizza_name', values='quantity').fillna(0)
```

```
[ ]  arima_models = {}

     for pizza_name in sales_pivot.columns:
       try:
         model = ARIMA(sales_pivot[pizza_name], order=(1, 1, 0))
         model_fit = model.fit()
         arima_models[pizza_name] = model_fit
       except:
         print(f'ARIMA model for {pizza_name} failed to fit')
```

```
[ ]  # Generate predictions for one week
     prediction_days = 7
     predictions_arima = {}

     for pizza_name, model in arima_models.items():
       predictions_arima[pizza_name] = model.predict(start=len(sales_pivot), end=len(sales_pivot) + prediction_days - 1)
```

Ingredient Calculation

```
[ ]  # Create a dictionary to store the ingredient quantities
     ingredient_quantities = {}

     # Iterate through each pizza in the predictions
     for pizza_name in predictions_df.columns:
       # Get the predicted quantity for the pizza
       predicted_quantity = predictions_df[pizza_name].sum()

       # Get the ingredients for the pizza
       pizza_ingredients = ingredients_df[ingredients_df['pizza_name'] == pizza_name]

       # Iterate through each ingredient for the pizza
       for index, row in pizza_ingredients.iterrows():
         ingredient = row['pizza_ingredients']
         ingredient_qty = row['items_qty']

         # Calculate the required quantity of the ingredient
         required_quantity = predicted_quantity * ingredient_qty

         # Add the required quantity to the dictionary
         if ingredient not in ingredient_quantities:
           ingredient_quantities[ingredient] = 0
         ingredient_quantities[ingredient] += required_quantity
```

Purchase Order generation

```
# Print the purchase order table
print(purchase_order_df.to_string())
```

```
Purchase Order:
----------------
                              quantity   unit
Barbecued Chicken          5404.165210  grams
Red Peppers               11341.551998  grams
Green Peppers              8030.393870  grams
Tomatoes                  34984.718341  grams
Red Onions                54797.556512  grams
Barbecue Sauce             1801.388403  grams
Bacon                     19992.004764  grams
Pepperoni                 24192.916429  grams
Italian Sausage             343.954622  grams
Chorizo Sausage            1719.773109  grams
Brie Carre Cheese           260.292444  grams
Prosciutto                  260.292444  grams
Caramelized Onions                 NaN  grams
Pears                        86.764148  grams
Thyme                        43.382074  grams
Garlic                    17939.075392  grams
?duja Salami               1586.898271  grams
Pancetta                   2380.347406  grams
Friggitello Peppers         396.724568  grams
```

Conclusion

Feature Importance: Analysis indicated that promotional periods and seasonal effects were significant predictors of sales, guiding inventory management decisions.

Business Suggestions/Solution

Implementing a predictive purchase order system based on the model’s forecasts will enable Dominos to optimize ingredient inventory levels, reduce waste, and enhance overall operational efficiency, ultimately leading to increased profitability and customer satisfaction.