Anomaly Detection & Time Series | **Assignment**

Question 1: What is Anomaly Detection? Explain its types (point, contextual, and collective anomalies) with examples.

Answer:

Anomaly Detection is the process of identifying data points, patterns, or observations that deviate significantly from the normal behavior of a dataset.

- **Point Anomaly:** A single data point that is far from the rest of the data.
  Example: A sudden credit card transaction of ₹5,00,000 when the user usually spends ₹1,000–₹5,000.
- **Contextual Anomaly:** A data point that is anomalous in a specific context (time, location, condition).
  Example: High electricity usage during midnight hours when usage is normally low.
- **Collective Anomaly:** A group of related data points that together form an anomaly, even if individual points appear normal.
  Example: A sequence of small but continuous network requests indicating a DDoS attack.

Question 2: Compare Isolation Forest, DBSCAN, and Local Outlier Factor in terms of their approach and suitable use cases.

Answer:

- Isolation Forest:
  Works by randomly partitioning data and isolating anomalies faster since anomalies are easier to separate.
  Use case: Large, high-dimensional datasets such as fraud detection.
- DBSCAN:
  A density-based clustering algorithm that marks low-density points as anomalies.
  Use case: Spatial data and datasets with clusters of arbitrary shape.
- Local Outlier Factor (LOF):
  Measures the local density deviation of a data point compared to its neighbors.
  Use case: Detecting local anomalies in datasets with varying densities.

Question 3: What are the key components of a Time Series? Explain each with one example.

Answer:

The key components of a Time Series are:

- **Trend:** Long-term movement in the data.
  Example: Gradual increase in online sales over several years.

- **Seasonality:** Repeating patterns at fixed intervals.
  Example: Increased ice cream sales every summer.
- **Cyclic:** Fluctuations without a fixed period, often influenced by economic factors.
  Example: Business cycles during economic expansions and recessions.
- **Irregular (Noise):** Random variations not explained by other components.
  Example: Sudden drop in sales due to a one-time event like a strike.

Question 4: Define Stationary in time series. How can you test and transform a non-stationary series into a stationary one?

Answer:

A time series is **stationary** if its statistical properties such as mean, variance, and autocovariance remain constant over time.

- **Testing stationarity:**
  - Augmented Dickey-Fuller (ADF) Test
  - KPSS Test
  - Visual inspection (rolling mean and variance)
- **Transforming a non-stationary series:**
  - Differencing the data
  - Log or square-root transformation
  - Removing trend and seasonality

Question 5: Differentiate between AR, MA, ARIMA, SARIMA, and SARIMAX models in terms of structure and application.

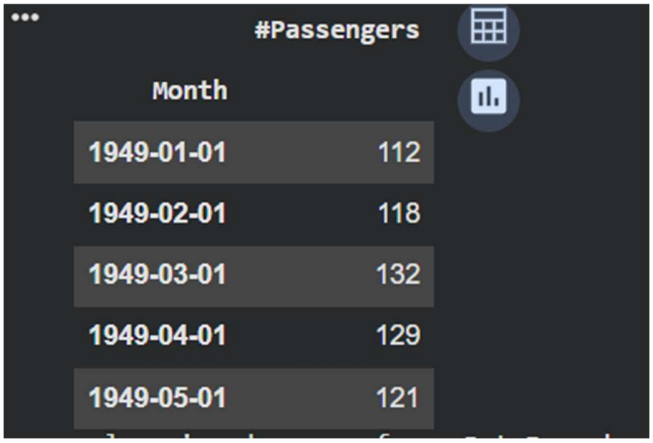
Answer:

- **AR (AutoRegressive):**
  Uses past values of the series to predict future values.
  Application: Forecasting when data depends on its own past values.
- **MA (Moving Average):**
  Uses past forecast errors to model the series.
  Application: Smoothing short-term fluctuations.
- **ARIMA:**
  Combines AR, MA, and differencing to handle non-stationary data.
  Application: General-purpose time series forecasting.
- **SARIMA:**
  Extends ARIMA by adding seasonal components.
  Application: Seasonal time series like monthly sales data.
- **SARIMAX:**
  SARIMA with exogenous variables.
  Application: Forecasting influenced by external factors such as promotions or weather.

Question 6: a Load time series dataset (e.g., AirPassengers), plot the original series, and decompose it into trend, seasonality, and residual components

Answer:

```
import pandas as pd

import matplotlib.pyplot as plt

from statsmodels.tsa.seasonal import seasonal_decompose

# Load the dataset

df = pd.read_csv('/content/AirPassengers.csv')

# Convert 'Month' to datetime and set as index

df['Month'] = pd.to_datetime(df['Month'])

df = df.set_index('Month')

# Display the first few rows and info
```

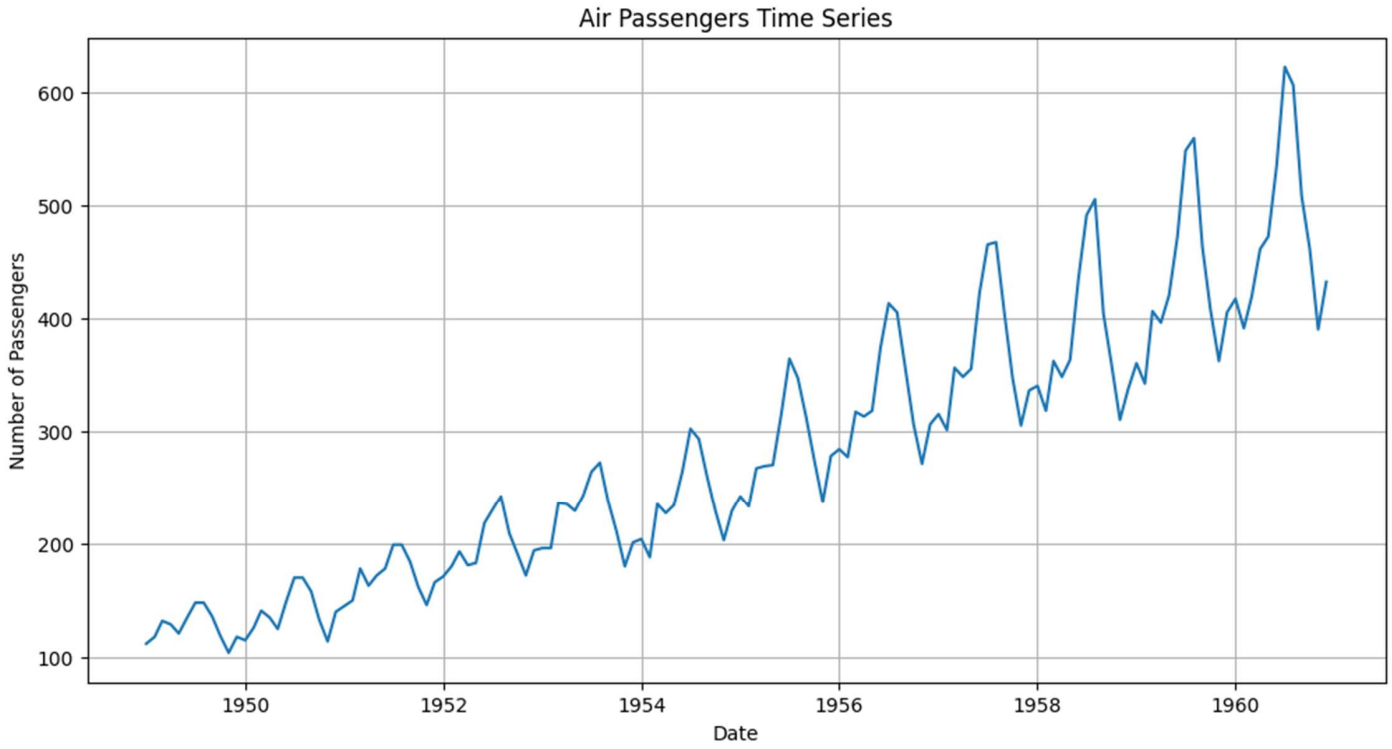


Month	#Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
display(df.head())
```

```
# Plot the original series

plt.figure(figsize=(12, 6))

plt.plot(df['#Passengers'])

plt.title('Air Passengers Time Series')

plt.xlabel('Date')

plt.ylabel('Number of Passengers')

plt.grid(True)
```

plt.show()



 # Decompose the series

decomposition = seasonal_decompose(df['#Passengers'], model='multiplicative', period=12)

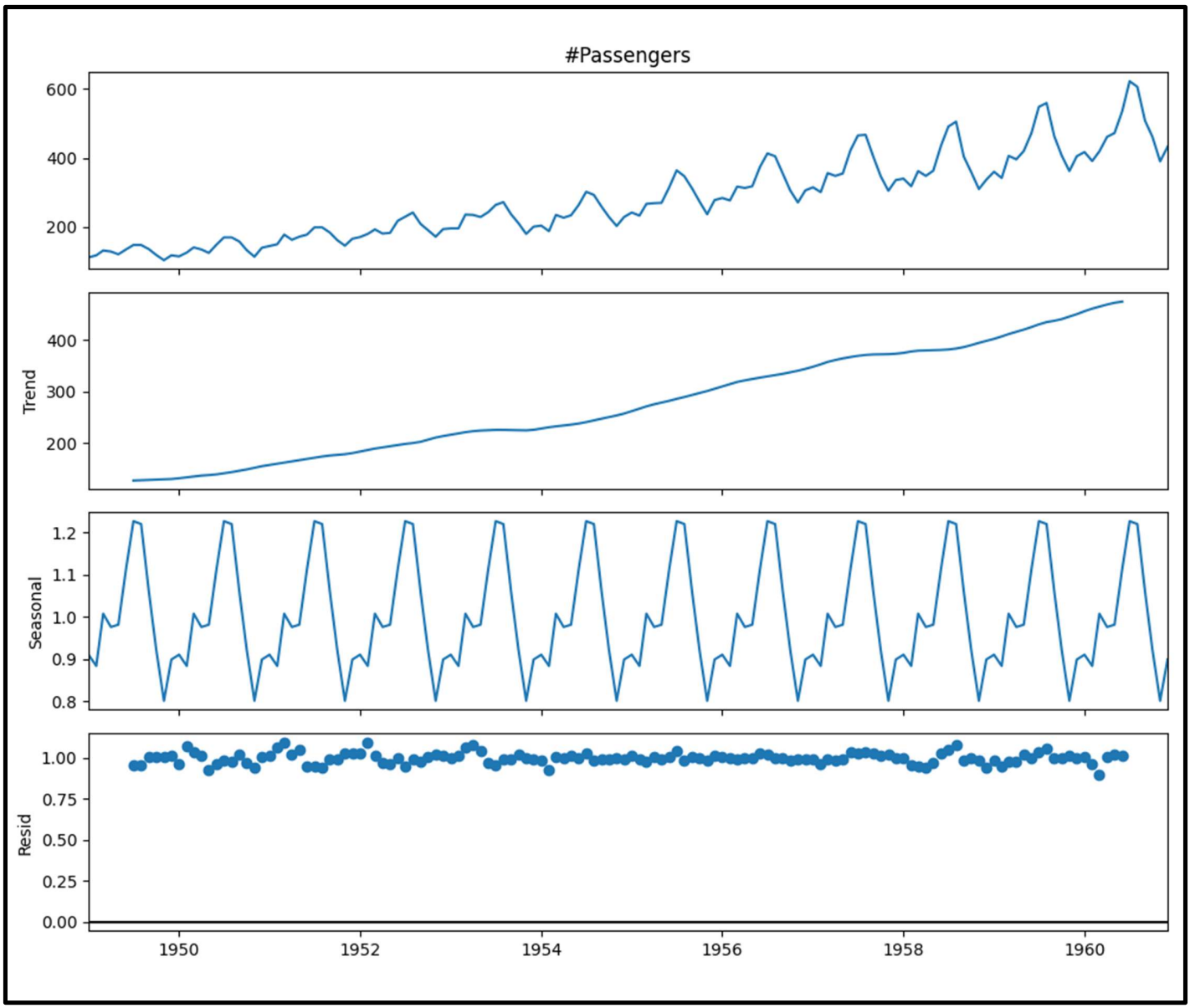
 # Plot the decomposed components

fig = decomposition.plot()

fig.set_size_inches(10, 8)

plt.tight_layout()

plt.show()



Question 7: Apply Isolation Forest on a numerical dataset (e.g., NYC Taxi Fare) to detect anomalies. Visualize the anomalies on a 2D scatter plot.

Answer:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.ensemble import IsolationForest

# Load the NYC Taxi Fare dataset

df_taxi = pd.read_csv('/content/NYC_taxi_fare_data.csv')
```

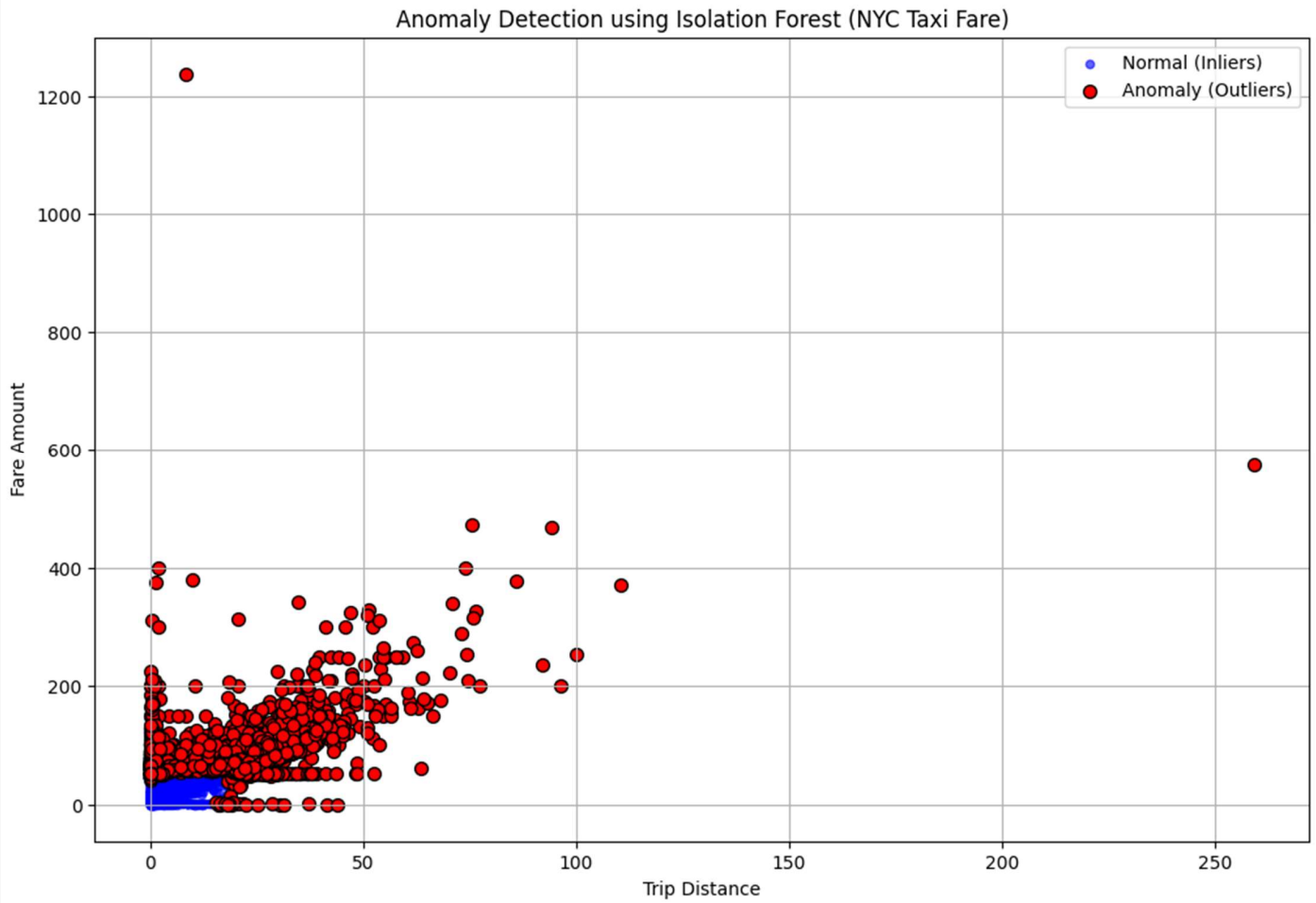
```
# Display initial information about the dataset

display(df_taxi.head())

display(df_taxi.info())
```

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID	payment_type	fare_amount	extra	mta_tax	tip_amount	tolls_amount	improvement_surcharge	total_amount	congestion_surcharge
0	1	2020-01-01 00:28:15	2020-01-01 00:33:03	1.0	1.2	1.0	N	238.0	239.0	1.0	6.0	3.0	0.5	1.47	0.0	0.3	11.27	2.5
1	1	2020-01-01 00:35:39	2020-01-01 00:43:04	1.0	1.2	1.0	N	239.0	238.0	1.0	7.0	3.0	0.5	1.50	0.0	0.3	12.30	2.5
2	1	2020-01-01 00:47:41	2020-01-01 00:53:52	1.0	0.6	1.0	N	238.0	238.0	1.0	6.0	3.0	0.5	1.00	0.0	0.3	10.80	2.5
3	1	2020-01-01 00:55:23	2020-01-01 01:00:14	1.0	0.8	1.0	N	238.0	151.0	1.0	5.5	0.5	0.5	1.36	0.0	0.3	8.16	0.0
4	2	2020-01-01 00:01:58	2020-01-01 00:04:16	1.0	0.0	1.0	N	193.0	193.0	2.0	3.5	0.5	0.5	0.00	0.0	0.3	4.80	0.0

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 759898 entries, 0 to 759897
Data columns (total 18 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   VendorID               759898 non-null  int64
 1   tpep_pickup_datetime   759898 non-null  object
 2   tpep_dropoff_datetime  759898 non-null  object
 3   passenger_count        759897 non-null  float64
 4   trip_distance          759897 non-null  float64
 5   RatecodeID             759897 non-null  float64
 6   store_and_fwd_flag     759897 non-null  object
 7   PULocationID           759897 non-null  float64
 8   DOLocationID           759897 non-null  float64
 9   payment_type           759897 non-null  float64
```

```
 10  fare_amount          759897 non-null  float64
 11  extra                759897 non-null  float64
 12  mta_tax              759897 non-null  float64
 13  tip_amount           759897 non-null  float64
 14  tolls_amount         759897 non-null  float64
 15  improvement_surcharge  759897 non-null  float64
 16  total_amount         759897 non-null  float64
 17  congestion_surcharge   759897 non-null  float64
dtypes: float64(14), int64(1), object(3)
memory usage: 104.4+ MB
None

data = df_taxi[['fare_amount', 'trip_distance']].copy()
# Handle potential missing values by dropping rows (for simplicity in this example)
data.dropna(inplace=True)
data = data[data['fare_amount'] > 0]
data = data[data['trip_distance'] > 0]
model = IsolationForest(contamination=0.01, random_state=42)
model.fit(data)
# Predict anomalies (-1 for outliers, 1 for inliers)
data['anomaly'] = model.predict(data)
# Separate inliers and outliers for plotting
inliers = data[data['anomaly'] == 1]
outliers = data[data['anomaly'] == -1]

# Display the number of anomalies found
print(f"Number of inliers: {len(inliers)}")
print(f"Number of outliers: {len(outliers)}")

Number of inliers: 742103
Number of outliers: 7438

# Visualize the anomalies on a 2D scatter plot
plt.figure(figsize=(12, 8))

# Plot inliers in blue
plt.scatter(inliers['trip_distance'], inliers['fare_amount'],
        c='blue', label='Normal (Inliers)', s=20, alpha=0.6)

# Plot outliers in red
plt.scatter(outliers['trip_distance'], outliers['fare_amount'],
        c='red', label='Anomaly (Outliers)', s=50, edgecolors='k')

plt.title('Anomaly Detection using Isolation Forest (NYC Taxi Fare)')
plt.xlabel('Trip Distance')
plt.ylabel('Fare Amount')
plt.legend()
plt.grid(True)
```

plt.show()



Question 8: Train a SARIMA model on the monthly airline passengers dataset. Forecast the next 12 months and visualize the results.

(*Include your Python code and output in the code box below.*)

Answer:

```
import pmdarima as pm
from statsmodels.tsa.statespace.sarimax import SARIMAX
import matplotlib.pyplot as plt
import pandas as pd
print("Finding optimal SARIMA orders...")
stepwise_fit = pm.auto_arima(df['#Passengers'], start_p=1, start_q=1,
                             max_p=3, max_q=3, m=12,
                             start_P=0, seasonal=True,
                             d=1, D=1, trace=True,
                             error_action='ignore',  # don't want to know if a run failed
                             suppress_warnings=True, # don't want convergence warnings
                             stepwise=True)

print("Optimal SARIMA orders found:")
print(stepwise_fit.summary())
```

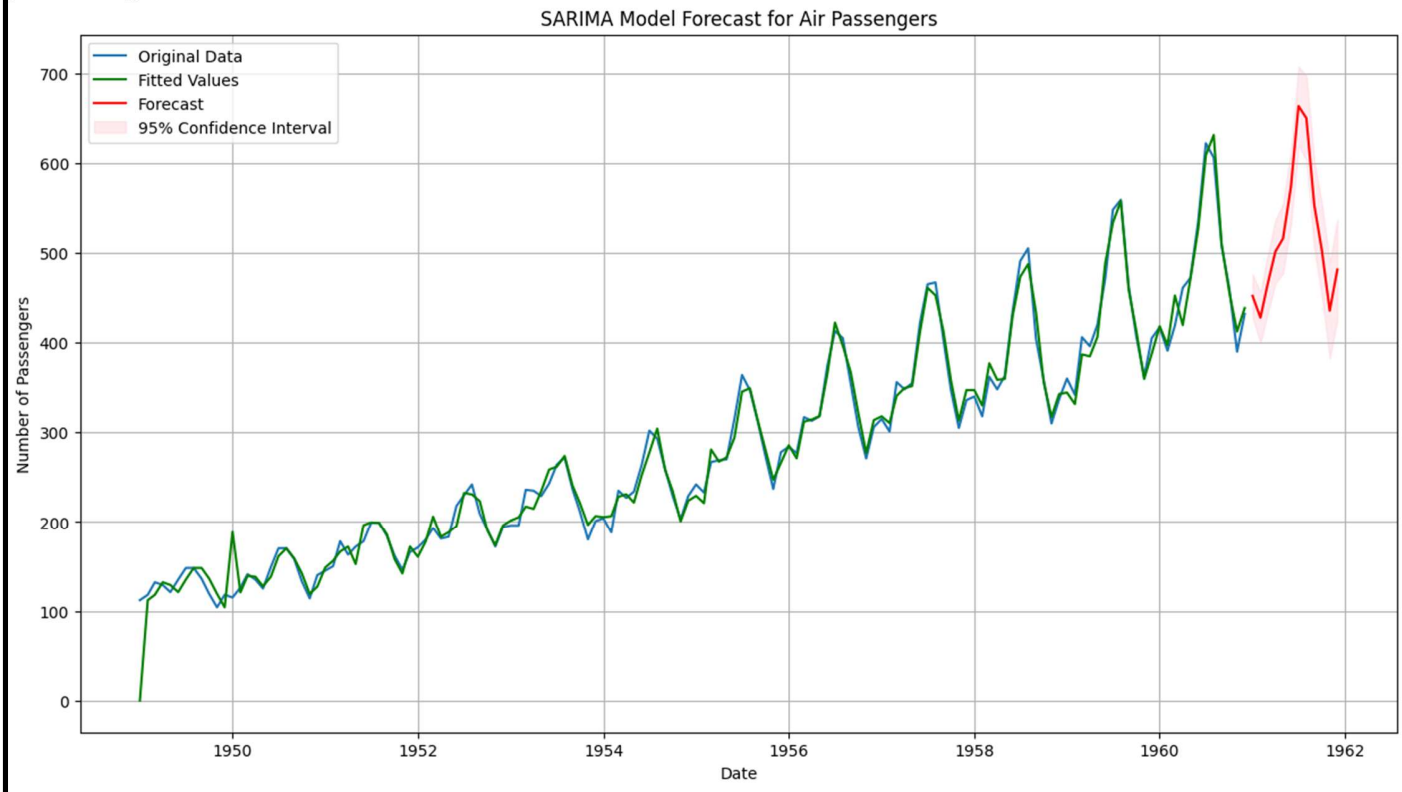
```
# Extract the best orders
order = stepwise_fit.order
seasonal_order = stepwise_fit.seasonal_order

print(f"Using SARIMA order: {order} and seasonal order: {seasonal_order}")

# Train the SARIMA model with the determined orders
model = SARIMAX(df['#Passengers'], order=order, seasonal_order=seasonal_order,
enforce_stationarity=False, enforce_invertibility=False)
model_fit = model.fit(disp=False)

# Forecast the next 12 months
forecast_steps = 12
forecast = model_fit.get_forecast(steps=forecast_steps)
forecast_ci = forecast.conf_int()

# Create a DataFrame for the forecast
forecast_index = pd.date_range(start=df.index[-1], periods=forecast_steps + 1, freq='MS')[1:]
forecast_df = pd.DataFrame({'#Passengers': forecast.predicted_mean.values}, index=forecast_index)
forecast_df['lower #Passengers'] = forecast_ci.iloc[:, 0].values
forecast_df['upper #Passengers'] = forecast_ci.iloc[:, 1].values

print("Forecast for the next 12 months:")
display(forecast_df)

Finding optimal SARIMA orders...
Performing stepwise search to minimize aic
 ARIMA(1,1,1)(0,1,1)[12]             : AIC=1022.896, Time=0.54 sec
 ARIMA(0,1,0)(0,1,0)[12]             : AIC=1031.508, Time=0.04 sec
 ARIMA(1,1,0)(1,1,0)[12]             : AIC=1020.393, Time=0.14 sec
 ARIMA(0,1,1)(0,1,1)[12]             : AIC=1021.003, Time=0.47 sec
 ARIMA(1,1,0)(0,1,0)[12]             : AIC=1020.393, Time=0.35 sec
 ARIMA(1,1,0)(2,1,0)[12]             : AIC=1019.239, Time=1.12 sec
 ARIMA(1,1,0)(2,1,1)[12]             : AIC=inf, Time=8.89 sec
 ARIMA(1,1,0)(1,1,1)[12]             : AIC=1020.493, Time=0.94 sec
 ARIMA(0,1,0)(2,1,0)[12]             : AIC=1032.120, Time=0.66 sec
 ARIMA(2,1,0)(2,1,0)[12]             : AIC=1021.120, Time=1.25 sec
 ARIMA(1,1,1)(2,1,0)[12]             : AIC=1021.032, Time=1.80 sec
 ARIMA(0,1,1)(2,1,0)[12]             : AIC=1019.178, Time=1.05 sec
 ARIMA(0,1,1)(1,1,0)[12]             : AIC=1020.425, Time=0.66 sec
 ARIMA(0,1,1)(2,1,1)[12]             : AIC=inf, Time=7.24 sec
 ARIMA(0,1,1)(1,1,1)[12]             : AIC=1020.327, Time=0.52 sec
 ARIMA(0,1,2)(2,1,0)[12]             : AIC=1021.148, Time=0.43 sec
 ARIMA(1,1,2)(2,1,0)[12]             : AIC=1022.805, Time=0.72 sec
 ARIMA(0,1,1)(2,1,0)[12] intercept   : AIC=1021.017, Time=0.68 sec

Best model:  ARIMA(0,1,1)(2,1,0)[12]
Total fit time: 27.544 seconds
Optimal SARIMA orders found:
                               SARIMAX Results
==========================================================================================
=======
Dep. Variable:                           y   No. Observations:                144
Model:           SARIMAX(0, 1, 1)x(2, 1, [], 12)   Log Likelihood             -505.589
Date:                       Sat, 03 Jan 2026   AIC                         1019.178
Time:                               16:06:48   BIC                         1030.679
```

```
Sample:                         01-01-1949   HQIC                        1023.851
                              - 12-01-1960
Covariance Type:                        opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1         -0.3634      0.074     -4.945      0.000      -0.508      -0.219
ar.S.L12      -0.1239      0.090     -1.372      0.170      -0.301       0.053
ar.S.L24       0.1911      0.107      1.783      0.075      -0.019       0.401
sigma2       130.4480     15.527      8.402      0.000     100.016     160.880
===================================================================================
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):                 4.59
Prob(Q):                              0.92   Prob(JB):                         0.10
Heteroskedasticity (H):               2.70   Skew:                             0.15
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.87
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
Using SARIMA order: (0, 1, 1) and seasonal order: (2, 1, 0, 12)
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No
frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No
frequency information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
Forecast for the next 12 months:
```

	#Passengers	lower #Passengers	upper #Passengers
1961-01-01	451.989521	428.074555	475.904488
1961-02-01	427.824552	399.537902	456.111201
1961-03-01	464.538194	432.470399	496.605988
1961-04-01	501.450587	466.002705	536.898469
1961-05-01	516.002519	477.469918	554.535119
1961-06-01	573.173309	531.785264	614.561353
1961-07-01	663.464218	619.405404	707.523033
1961-08-01	649.895396	603.318706	696.472086
1961-09-01	553.266165	504.300903	602.231428
1961-10-01	502.719737	451.477119	553.962355
1961-11-01	435.631558	382.208578	489.054538
1961-12-01	481.394983	425.877205	536.912762

```
import matplotlib.pyplot as plt

# Plot the original series, fitted values, and forecast
plt.figure(figsize=(15, 8))
plt.plot(df['#Passengers'], label='Original Data')
plt.plot(model_fit.fittedvalues, color='green', label='Fitted Values')

plt.plot(forecast_df.index, forecast_df['#Passengers'], color='red', label='Forecast')
plt.fill_between(forecast_df.index,
```

```
            forecast_df['lower #Passengers'],
            forecast_df['upper #Passengers'],
            color='pink', alpha=0.3, label='95% Confidence Interval')

plt.title('SARIMA Model Forecast for Air Passengers')
plt.xlabel('Date')
plt.ylabel('Number of Passengers')
plt.legend()
plt.grid(True)
plt.show()
```



Question 9: Apply Local Outlier Factor (LOF) on any numerical dataset to detect anomalies and visualize them using matplotlib.

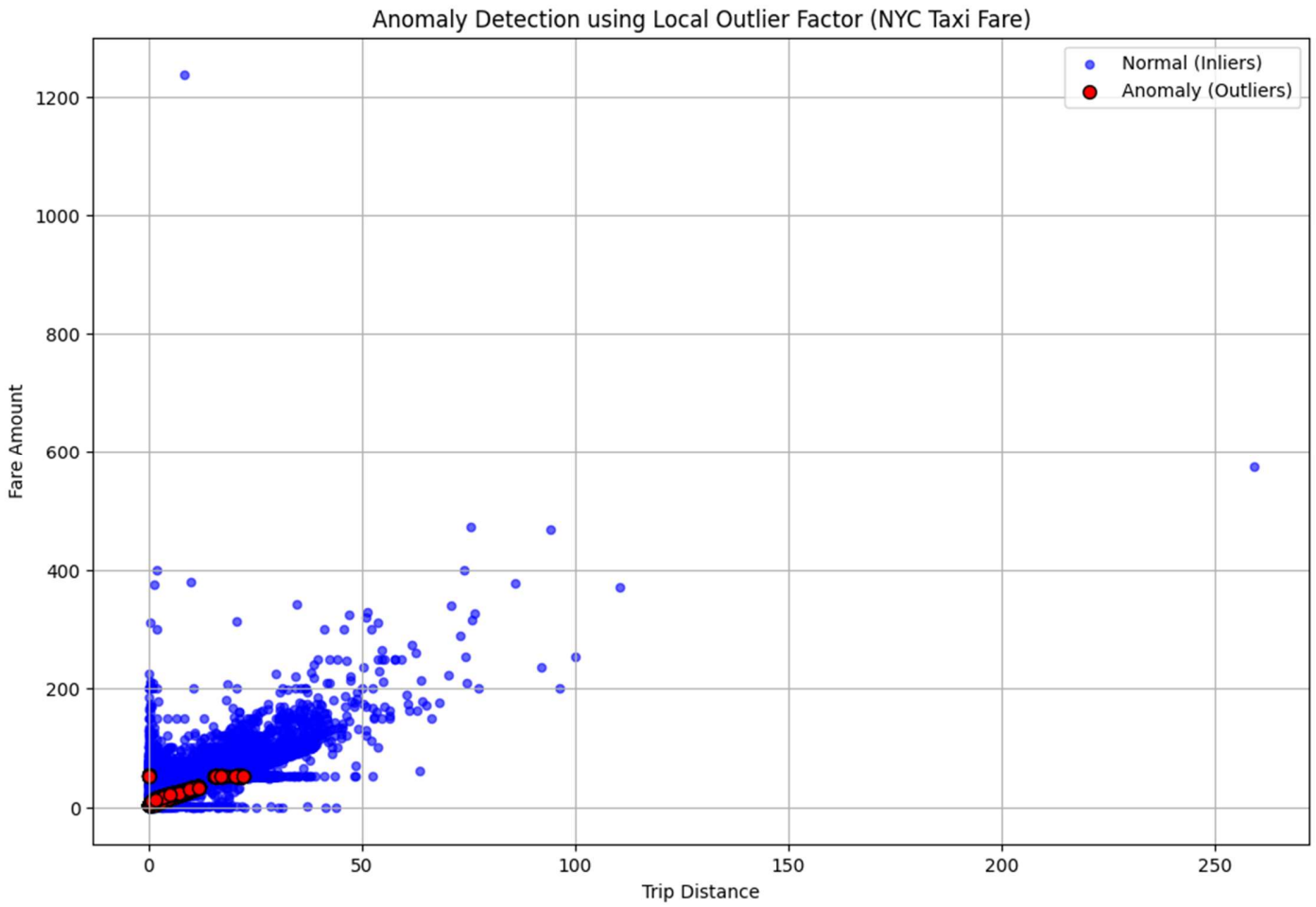
(*Include your Python code and output in the code box below.*)

Answer:

```
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor
import pandas as pd

lof = LocalOutlierFactor(n_neighbors=100, contamination=0.01)

# Fit the model and predict the anomaly scores (negative_outlier_factor_)
# The fit_predict method returns -1 for outliers and 1 for inliers.
data['anomaly_lof'] = lof.fit_predict(data[['fare_amount', 'trip_distance']])

# Separate inliers and outliers for plotting
inliers_lof = data[data['anomaly_lof'] == 1]
outliers_lof = data[data['anomaly_lof'] == -1]

# Display the number of anomalies found
print(f"Number of inliers (LOF): {len(inliers_lof)}")
print(f"Number of outliers (LOF): {len(outliers_lof)}")
```

```
Number of inliers (LOF): 742050
Number of outliers (LOF): 7491
/usr/local/lib/python3.12/dist-packages/sklearn/neighbors/_lof.py:322: UserWarning: Duplicate values are leading to
incorrect results. Increase the number of neighbors for more accurate results.
  warnings.warn(
```

```
import matplotlib.pyplot as plt

# Visualize the anomalies detected by LOF on a 2D scatter plot
plt.figure(figsize=(12, 8))

# Plot inliers in blue
plt.scatter(inliers_lof['trip_distance'], inliers_lof['fare_amount'],
            c='blue', label='Normal (Inliers)', s=20, alpha=0.6)

# Plot outliers in red
plt.scatter(outliers_lof['trip_distance'], outliers_lof['fare_amount'],
            c='red', label='Anomaly (Outliers)', s=50, edgecolors='k')

plt.title('Anomaly Detection using Local Outlier Factor (NYC Taxi Fare)')
plt.xlabel('Trip Distance')
plt.ylabel('Fare Amount')
plt.legend()
plt.grid(True)
plt.show()
```



Question 10: You are working as a data scientist for a power grid monitoring company. Your goal is to forecast energy demand and also detect abnormal spikes or drops in real-time consumption data collected every 15 minutes. The dataset includes features like timestamp, region, weather conditions, and energy usage.

Explain your real-time data science workflow:

- How would you detect anomalies in this streaming data (Isolation Forest / LOF / DBSCAN)?
- Which time series model would you use for short-term forecasting (ARIMA / SARIMA / SARIMAX)?
- How would you validate and monitor the performance over time?
- How would this solution help business decisions or operations?

(*Include your Python code and output in the code box below.*)

Answer:

Real-Time Data Science Workflow (High Level)
Pipeline

1. **Data ingestion** – Streaming data every 15 minutes (Kafka / Spark Streaming / AWS Kinesis)
2. **Preprocessing** – Handle missing values, scale features, encode region
3. **Anomaly detection** – Detect abnormal spikes/drops in real time
4. **Short-term forecasting** – Predict next 1–24 hours of demand
5. **Monitoring & retraining** – Track drift, accuracy, false alarms
6. **Business action layer** – Alerts, load balancing, pricing decisions

---

2. Anomaly Detection in Streaming Data
Why Isolation Forest (Best Choice for Real-Time)

Model	Use Case	Verdict
Isolation Forest	Fast, scalable, works well on high-volume streams	✅ **Best**
LOF	Needs dense neighborhoods, slow in streaming	⚠️ Limited
DBSCAN	Not suitable for real-time & high-dimensional data	❌ Avoid

Strategy

- Train Isolation Forest on **recent rolling window** (e.g., last 7 days)
- Use **energy usage + weather + time features**
- Flag anomalies instantly

---

3. Short-Term Forecasting Model
Model Comparison

Model	When to Use
ARIMA	No seasonality
SARIMA	Strong daily/weekly seasonality
SARIMAX	Seasonality + weather & external features

Final Choice → SARIMAX
Because:

- Energy demand has **daily & weekly patterns**
- Weather (temperature, humidity) strongly affects usage

---

4. Validation & Monitoring Strategy
Forecasting

- Rolling window validation
- Metrics: **RMSE, MAE, MAPE**
- Retrain weekly or when drift detected

Anomaly Detection

- Track anomaly rate over time
- Human-verified alerts to reduce false positives
- Concept drift detection (e.g., KS test)

---

5. Business & Operational Impact

✅ **Grid Stability** – Prevent blackouts
✅ **Cost Optimization** – Efficient power generation planning
✅ **Preventive Maintenance** – Detect faulty meters or transformers
✅ **Dynamic Pricing** – Demand-based tariff decisions
✅ **Regulatory Compliance** – Reliable reporting

Code:

```
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_absolute_error
```

```
# ----------------------------
# Simulated Streaming Data
# ----------------------------
np.random.seed(42)

date_range = pd.date_range("2025-01-01", periods=500, freq="15T")
energy = 100 + 10*np.sin(np.arange(500)/24) + np.random.normal(0, 3, 500)

# Inject anomalies
energy[120] += 40
energy[300] -= 35

temperature = 25 + np.random.normal(0, 2, 500)

df = pd.DataFrame({
    "timestamp": date_range,
    "energy_usage": energy,
    "temperature": temperature
})

# ----------------------------
# Anomaly Detection
# ----------------------------
iso_forest = IsolationForest(contamination=0.02, random_state=42)
df["anomaly"] = iso_forest.fit_predict(df[["energy_usage", "temperature"]])

df["anomaly"] = df["anomaly"].map({1: "Normal", -1: "Anomaly"})

print("Anomaly Count:")
print(df["anomaly"].value_counts())

# ----------------------------
# SARIMAX Forecasting
# ----------------------------
train = df.iloc[:-48]
test = df.iloc[-48:]

model = SARIMAX(
    train["energy_usage"],
    exog=train[["temperature"]],
    order=(1,1,1),
    seasonal_order=(1,1,1,96)
)

model_fit = model.fit(disp=False)

forecast = model_fit.forecast(
    steps=48,
    exog=test[["temperature"]]
)

mae = mean_absolute_error(test["energy_usage"], forecast)

print("\nForecast MAE:", round(mae, 2))
```

Output

```
/tmp/ipython-input-3250117876.py:12: FutureWarning: 'T' is deprecated and will be removed in a future version, please use 'min' instead.
  date_range = pd.date_range("2025-01-01", periods=500, freq="15T")
Anomaly Count:
```

```
anomaly
Normal     490
Anomaly     10
Name: count, dtype: int64
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/sarimax.py:1009: UserWarning: Non-invertible starting seasonal moving average Using zeros as starting parameters.
  warn('Non-invertible starting seasonal moving average'
Forecast MAE: 11.19
```