
Politecnico di Torino

# Polytechnic University of Turin

## ICT for smart mobility

## Laboratory reports- Group 15

---

## Report 2

---

Authors:

| | |
|---|---|
| Mahdi Rajaee | s308497 |
| Alireza Esmaeilifar | s307765 |
| Kamaleshwaran Ravichandiran | s315468 |

Professors:
Marco MELLIA
Luca VASSIO

Academic year 2023/24

you are supposed to NOT exceed 5 pages +1 !

# 1. Introduction

In this lab, three cities were considered for the discovery of their information. These cities are Stuttgart, Columbus, and Torino. The goal of this lab is to find the best ARIMA model to predict the future number of rentals.

- 1- For each city, consider the selected period of 31 days. Extract the number of rentals recorded at each hour after properly filtering the outliers and those bookings that are not rentals.

  In the first question, we checked the booking for 31 days and considered each hour for the results. the reason for this fluctuation is because of the following:

  which period of time ?

  **Daily Patterns:** There is a daily pattern where bookings rise and fall, possibly due to typical daily routines, such as people booking rides to and from work or going out for meals and events.
  **Weekly Trends:** There may be weekly trends, with certain days of the week (like weekends) showing different patterns compared to weekdays.

  is this a trend ?

  **Special Events:** If there are days with particularly high peaks, it could be due to special events or holidays that increase the demand for bookings.
  **Time of Day:** Certain times of day (like rush hours in the morning and evening) are likely to have higher booking counts.
  **Weather Conditions:** Poor weather might increase bookings as people prefer not to walk or use personal transportation.
  **Market Changes:** The introduction of promotions, discounts, or competitors can affect booking patterns.

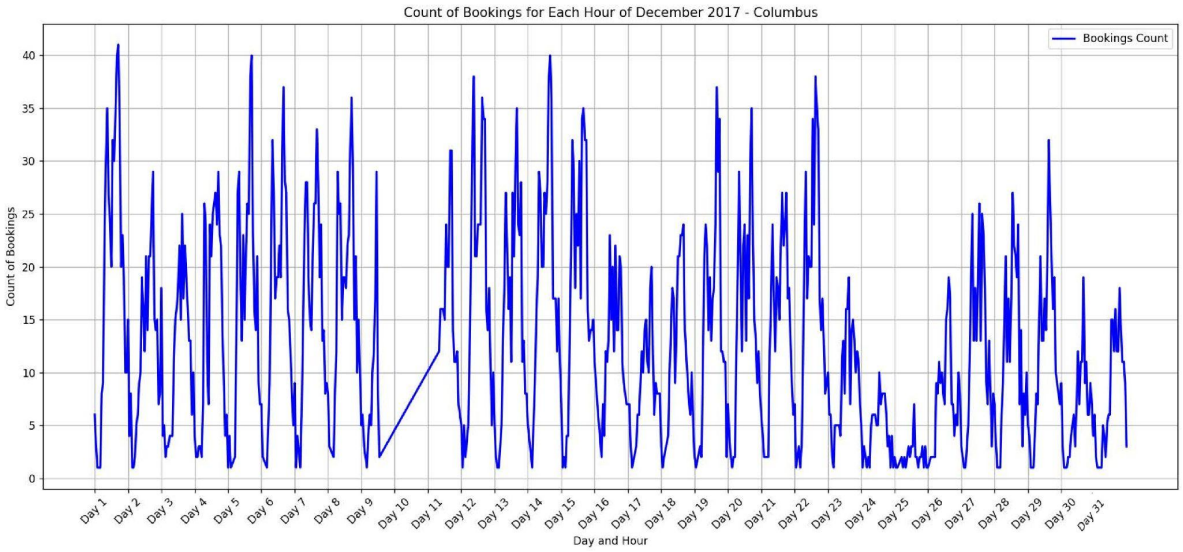


**Fig 1.1: Columbus daily and hourly booking analysis**

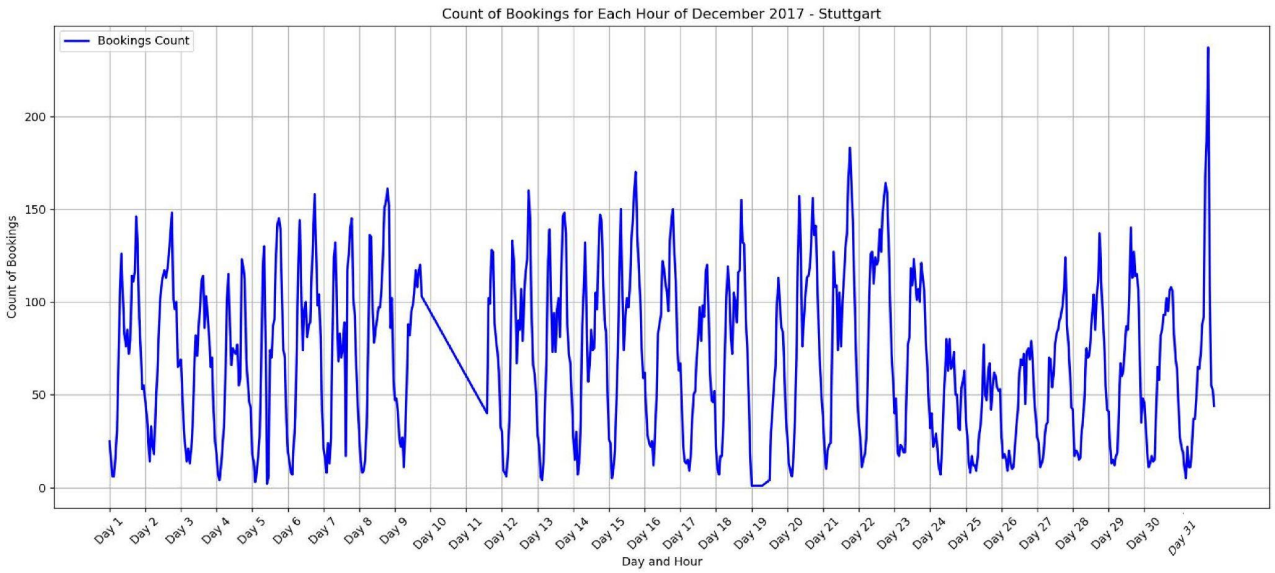



**Fig 1.1: Stuttgart daily and hourly booking analysis**

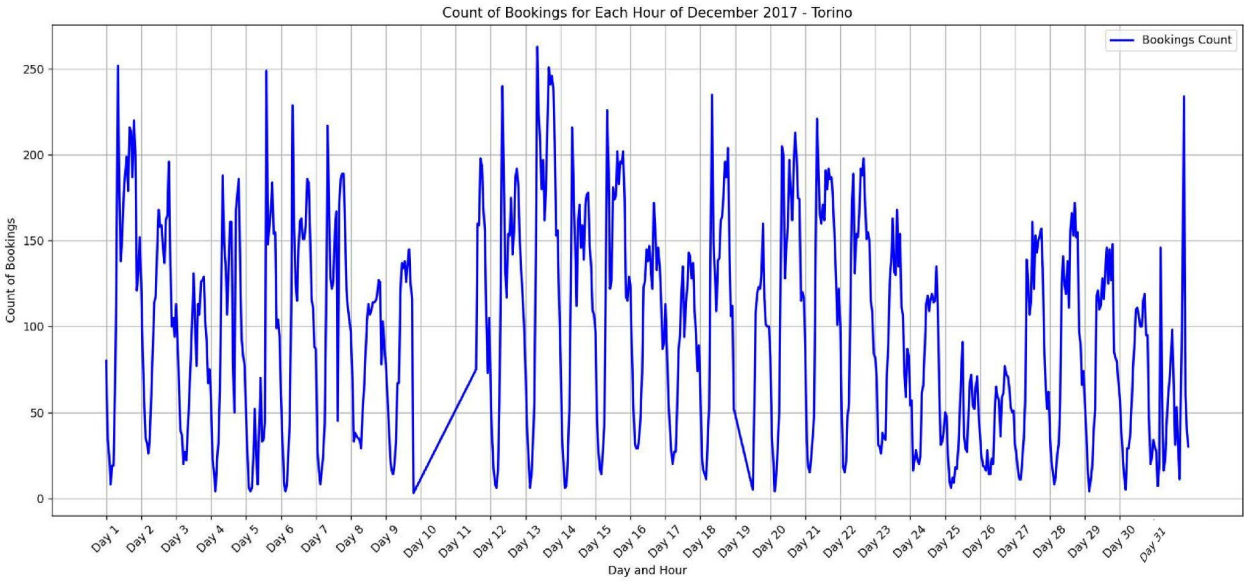



**Fig 1.3: Torino daily and hourly booking analysis**

In our methodology, we delineate the analysis by specifying both a start date and an end date. The query is designed to extract records wherein the initialization date falls within this predefined range. Additionally, we apply a criterion for selection based on the duration between the initiation and final timestamps, filtering for records where this duration does not exceed 8000 units. The data thus retrieved is then subjected to a plotting process, wherein the results are visually represented in two temporal resolutions: an hourly breakdown and a daily

what is a "unit" here (?)

aggregation. This approach enables a comprehensive examination of the data over different time scales, facilitating a nuanced understanding of temporal patterns and trends.

- 2 - Check if there are missing samples (recall, ARIMA, with no missing data) in the case of missing samples, use a policy for missing data for instance, use the last value, average value, replace with zeros, replace with the average value for the given time, etc.

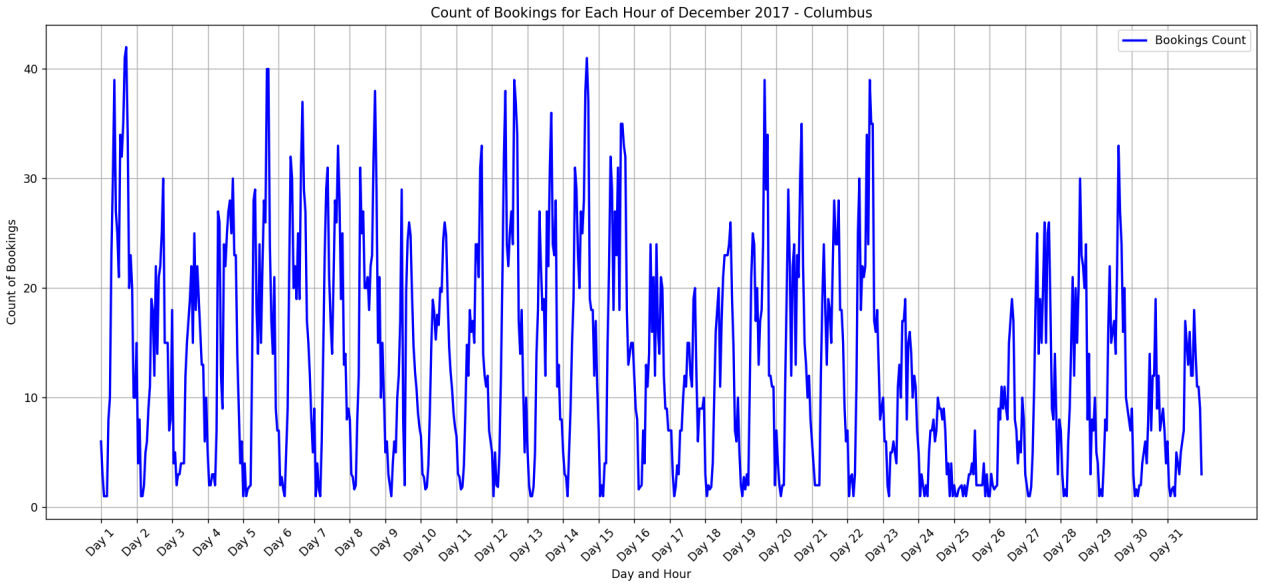



**Fig 2.1: rectified missing samples for Columbus**

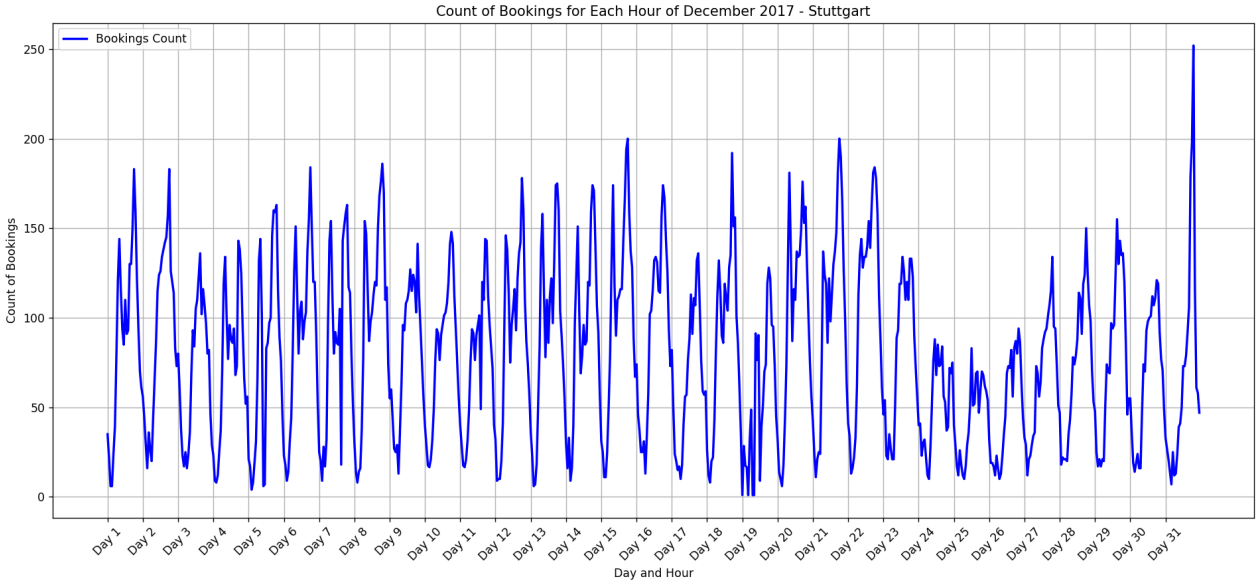



**Fig 2.2: rectified missing samples for Stuttgart**

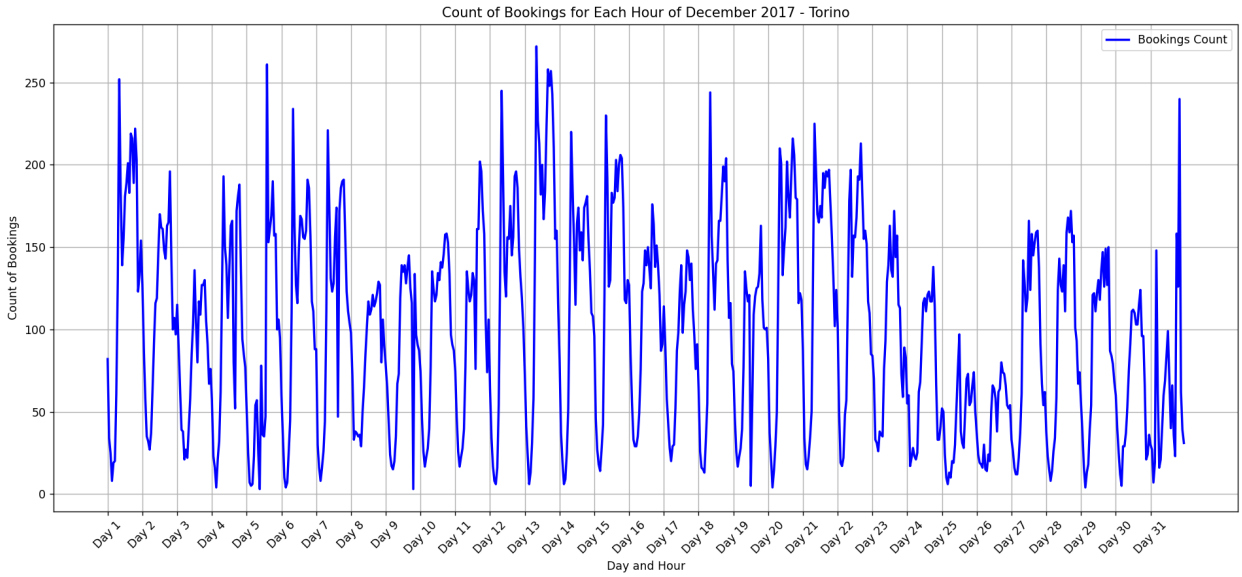



**Fig 2.3: rectified missing samples for Torino**

We have missing samples on the 10th date. To address the issue of missing data for certain days, we implemented a systematic approach encompassing several steps to ensure data completeness and accuracy:

**Creation of a Day-Hour Count Dictionary:** We initiated the process by constructing a nested dictionary, counts_dict, structured to encapsulate the count of bookings. Here, each key corresponds to a specific day of the month, and its associated value is another dictionary. This inner dictionary maps each hour of the day to the booking counts observed at that hour.

**Population of the Dictionary with Query Data:** Subsequently, we iterated over the dataset retrieved from our query. During this iteration, counts_dict was populated with data specifying the day, hour, and corresponding count of bookings.

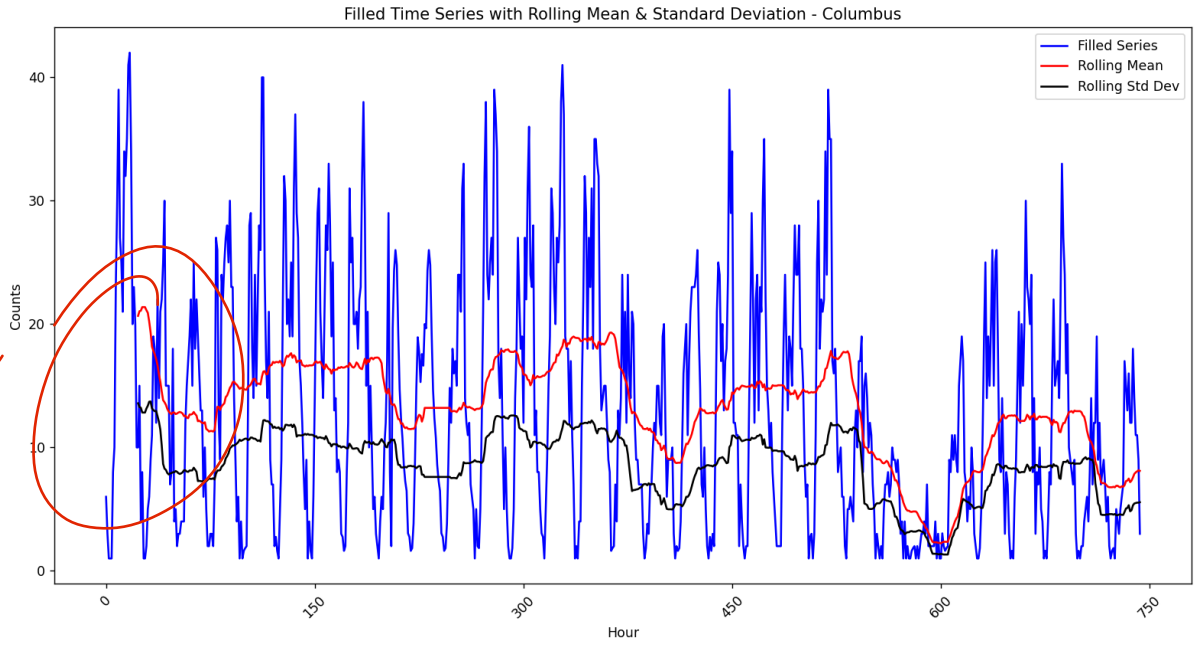
**Computation of Hourly Averages Across Days:** In the subsequent phase, we formulated another dictionary, hourly_averages, dedicated to calculating the average booking counts for each hour across all days. This dictionary maps each hour to a list of counts recorded for that hour on different days. The average count per hour is then computed and stored within hourly_averages.

**Filling Missing Hourly Data with Averages:** To address data gaps, we constructed a list, completed_counts, and filled it with the day-hour counts for each hour of every day in the specified month (e.g., December). In instances where an hour was absent in counts_dict for a given day, we substituted the missing value with the corresponding average count from hourly_averages.

**Preparation of Data for Plotting:** Finally, we separated completed_counts into two distinct lists: one for hours (hours) and another for counts (counts). This separation facilitated the plotting process, allowing for a clear and accurate visual representation of the booking data.

Through these steps, we ensured that our data was both complete and representative, thereby enhancing the reliability of our subsequent analyses and visualizations.

- 3 - Check whether the time series is stationary or not, and decide accordingly whether to use differencing or not (d=0 or not)




how did you choose the rolling window time ?

**Fig 3.1: Time series for Colombus**

```
ADF Statistic: -3.18053220207815
p-value: 0.021139570244623068
   1%, -3.4394269973845657
   5%, -2.8655458544300387
   10%, -2.568903174551249
The filled time series is stationary.
```

**Fig 3.2: Colombus data info**

what is this ?

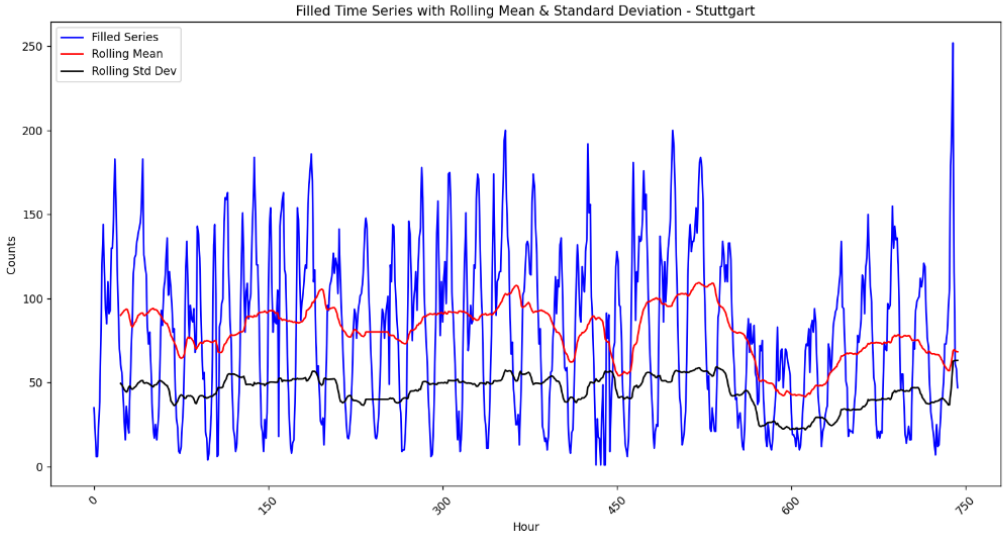



**Fig 3.3: Time series for Stuttgart**

```
ADF Statistic: -3.5855520229633684
p-value: 0.006043325862772057
   1%, -3.4394269973845657
   5%, -2.8655458544300387
   10%, -2.568903174551249
The filled time series is stationary.
```

**Fig 3.4: Stuttgart data info**

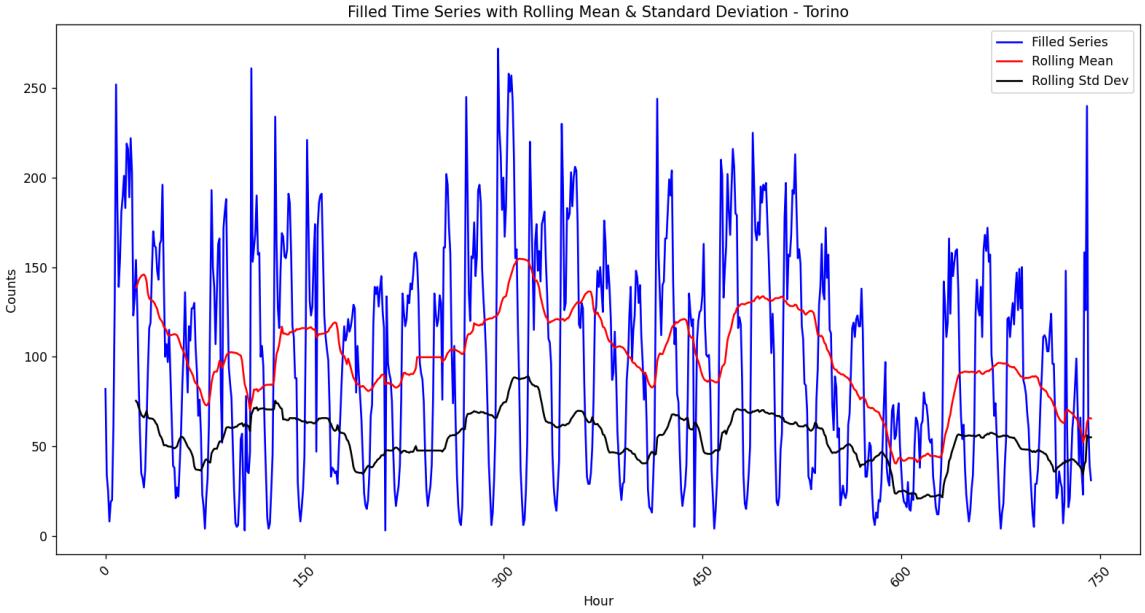



**Fig 3.5: Time series for Torino**

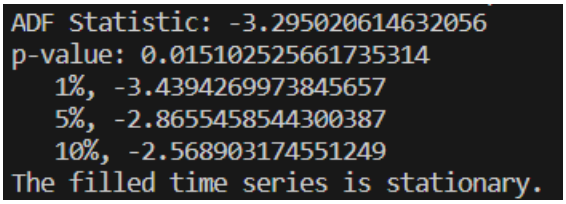



**Fig 3.6: Torino data info**

In our analysis, we conducted a thorough examination to ascertain whether there is any variation in the average (mean) and standard deviation (S.T.) over time. This assessment is crucial for determining the stationarity of the dataset.

how can you say so!?

Stationarity implies that the statistical properties of a time series, such as mean and variance, remain constant over time. In the context of our study, the absence of any discernible trend or systematic change in the average and standard deviation over the analyzed period **strongly suggests that the data is stationary**. Consequently, this led to the decision that the **differencing order (d) should be set to 0.**

The implication of d=0 is that the time series does not require differencing to achieve stationarity, which is a fundamental assumption for many time series models. This conclusion is pivotal in guiding the selection of appropriate modeling techniques and in ensuring the validity and reliability of any subsequent analyses or forecasts derived from the data.

- 4 - Compute the ACF and PACF to observe how they vanish. This is instrumental in guessing possible good values of the p and q parameters, assuming the model is pure AR or pure MA. What if your model is not pure AR or pure MA, i.e., it is an ARMA process?

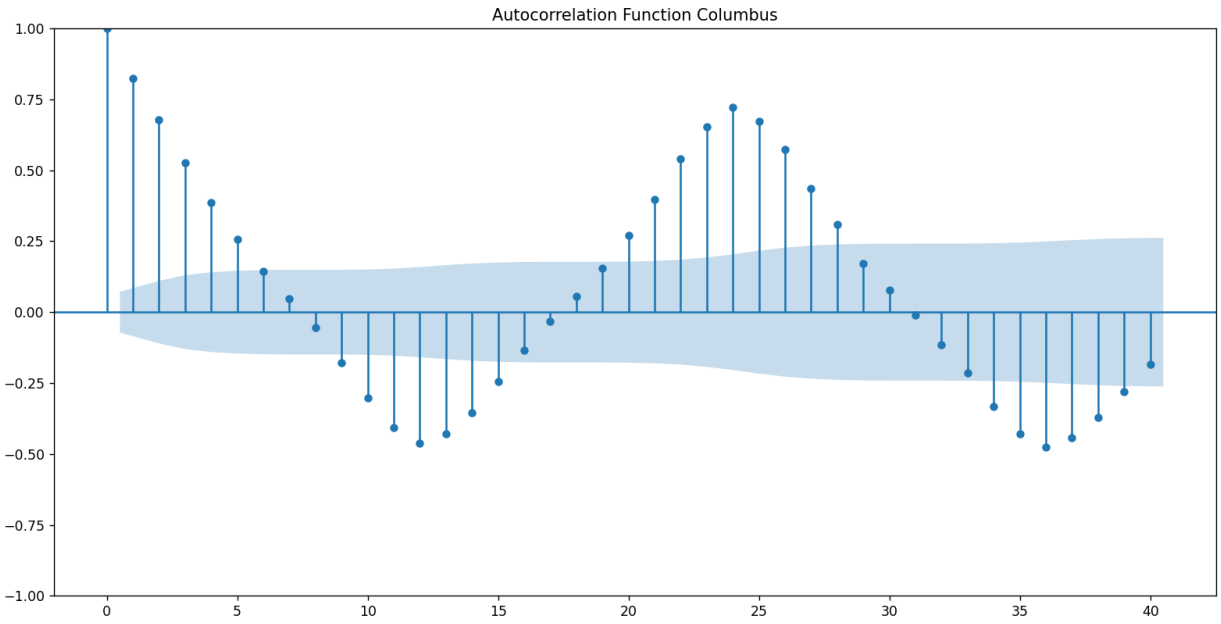



**Fig 4.1: ACF Colombus**

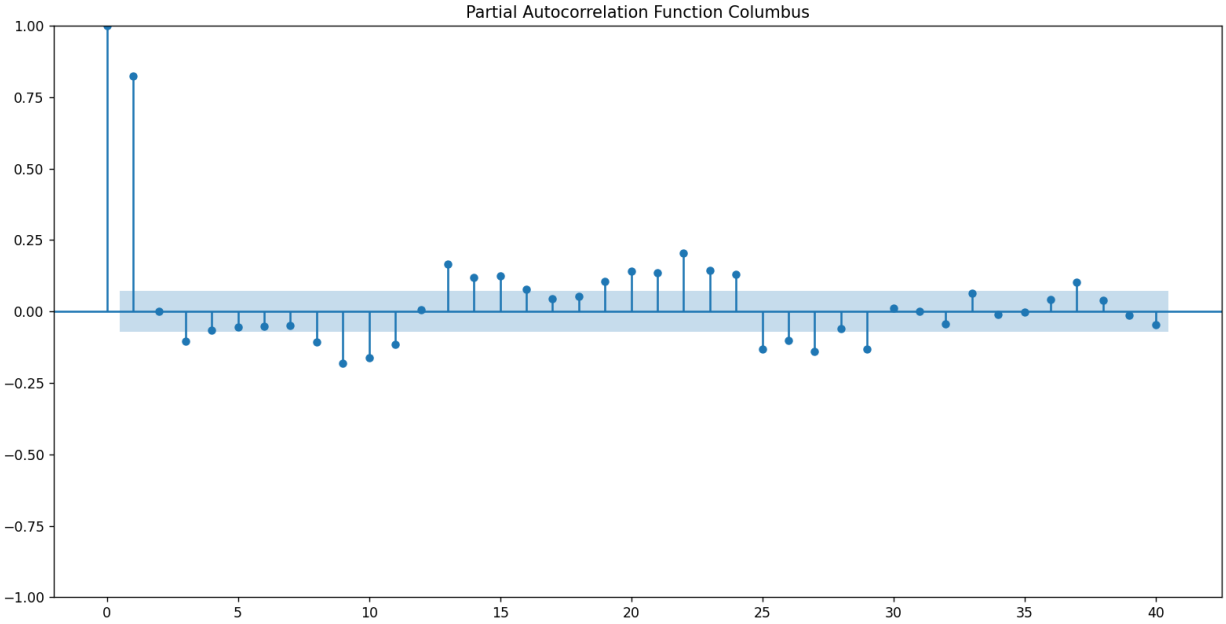



**Fig 4.2: PACF Colombus**

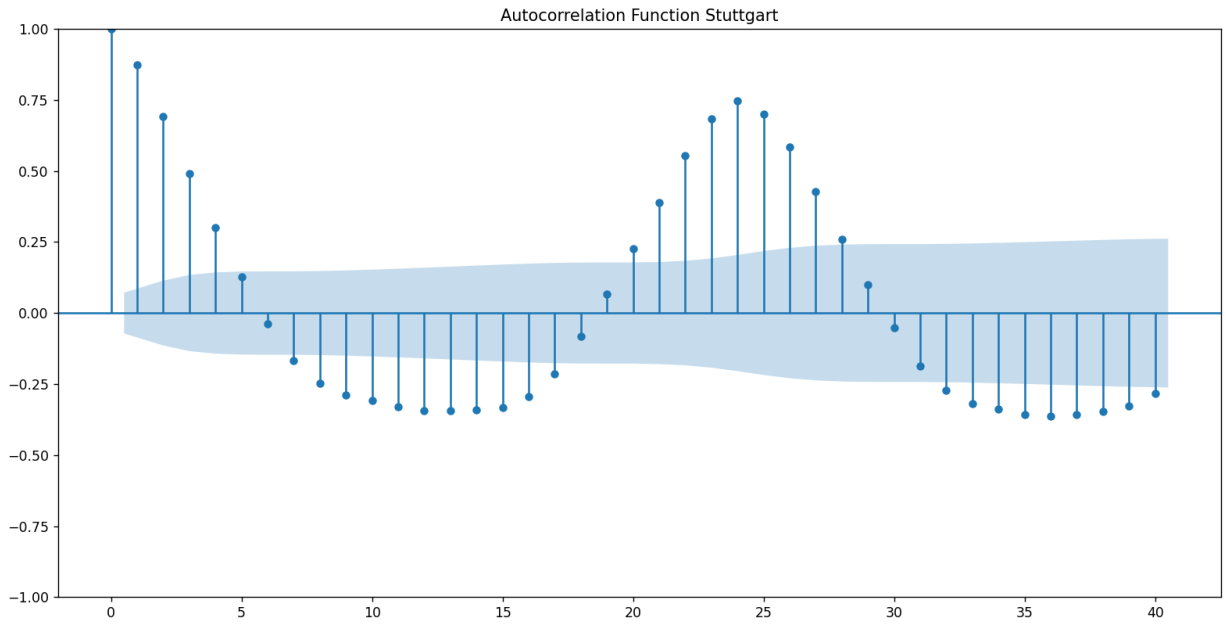



**Fig 4.3: ACF Stuttgart**

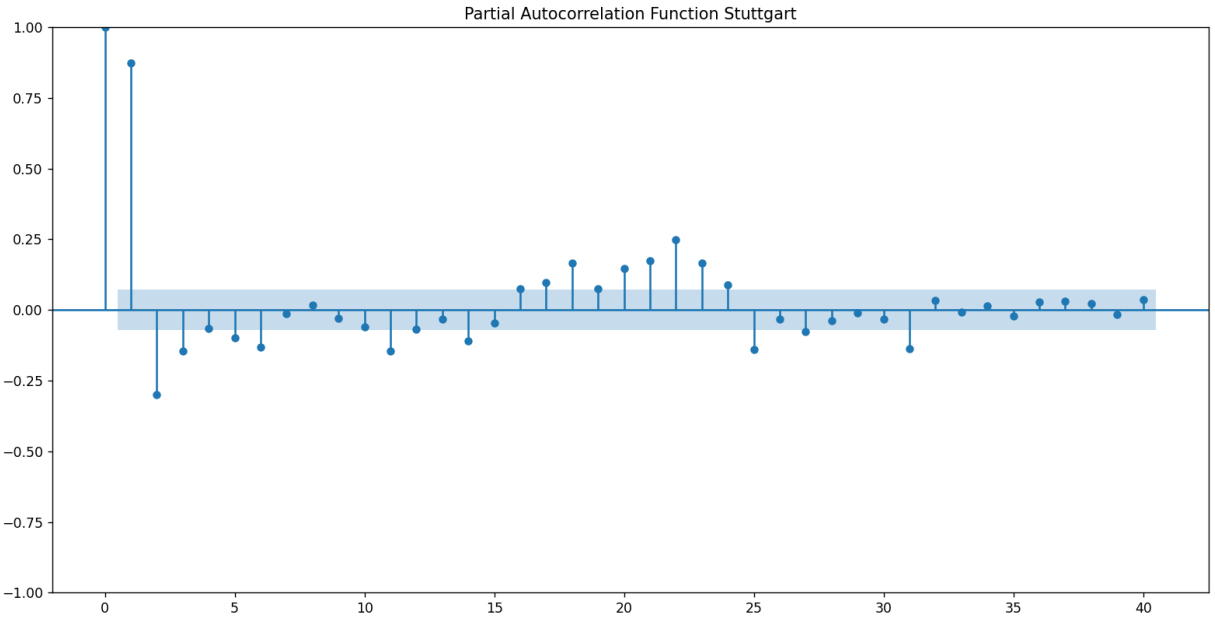



**Fig 4.4: PACF Stuttgart**

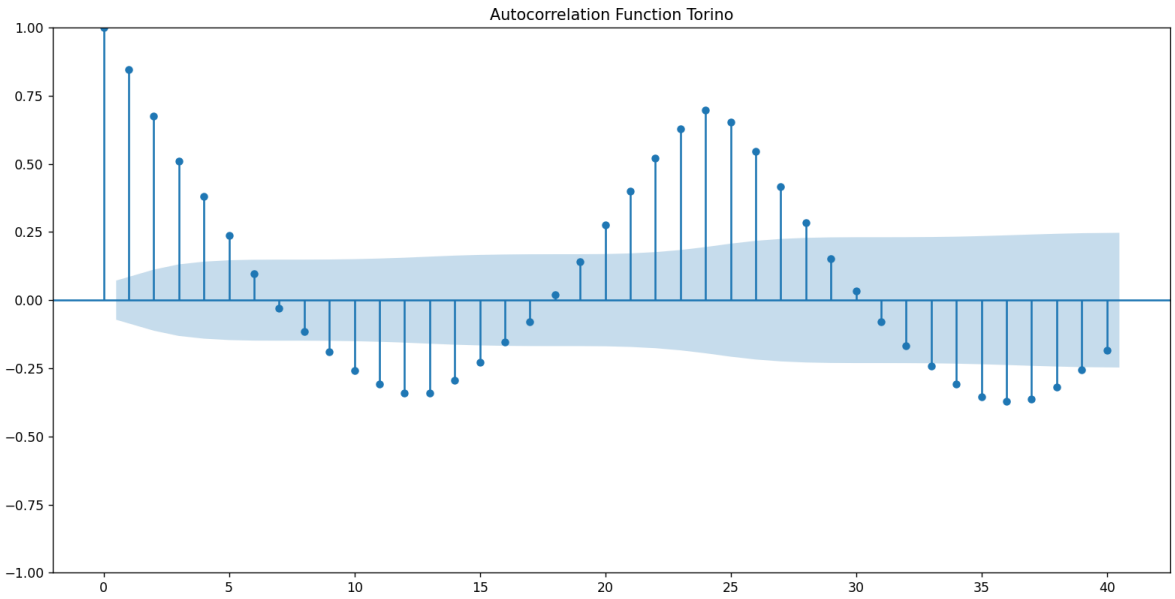



**Fig 4.5: ACF Torino**

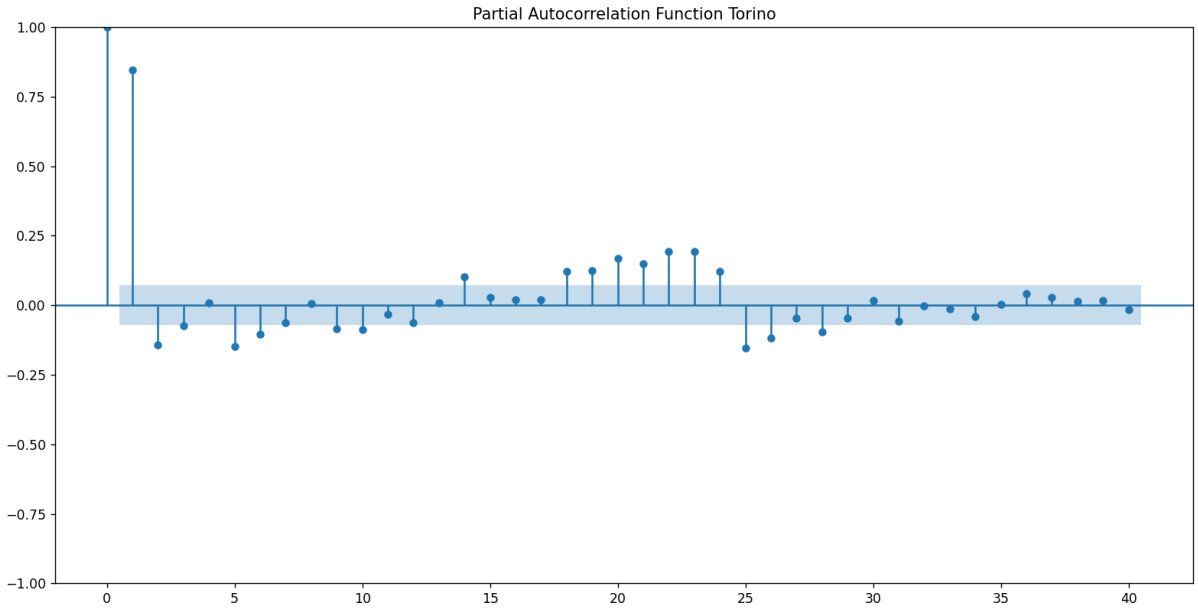



**Fig 4.6: PACF Torino**

In the analysis of the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots, we infer a differencing order (d) of 0. This finding suggests that the dataset is **inherently stationary** and negates the necessity for further differencing to achieve stationarity. The PACF plot indicates a substantial correlation at the initial two lags, followed by a pronounced truncation. This pattern leads us to select a value of 2 for the autoregressive (AR) parameter (p), implying that the current value in the series is significantly influenced by its

preceding two values. In contrast, the ACF plot reveals a notable correlation at the first lag, succeeded by a rapid decline. This observation informs our decision to adopt a value of 1 for the moving average (MA) parameter (q), despite initially considering a **value of 5**. The rationale behind this choice is the indication that the current error term is predominantly affected by the error term of the immediate past. This discernible trend in the plots suggests a model predominantly **characterized by autoregressive (AR)** elements rather than moving average (MA) elements. Furthermore, the detection of frequency within the data that corroborates its **stationarity** implies an **absence of trends** or seasonality influencing the series level over time.

- 5 - Decide the number of past samples N to use for training and how many to use for testing. Given you have 30 days of data (each with 24 hours), you can consider, for instance, training during the first week of data and testing the prediction in the second week of data
- 6 - Given *N (p,d,q),* train a model to compute the error. Consider the MPE and/or MAPE and/or other metrics – so that you can compare results for different cities (use percentage errors and not absolute errors for this comparison. Absolute errors would obviously not be directly comparable).

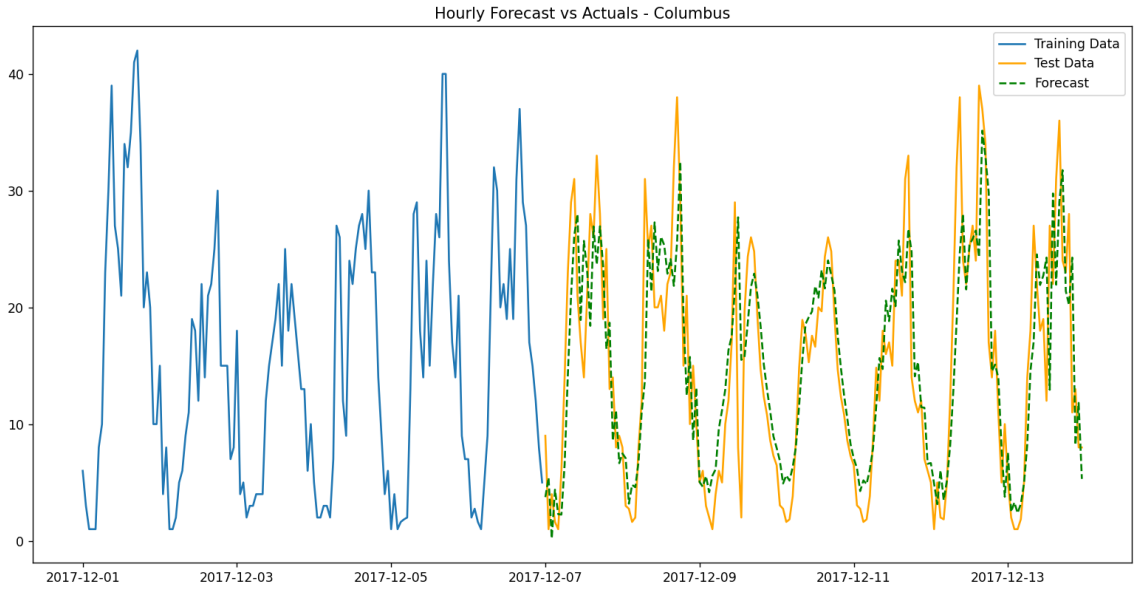


**Fig 6.1: hourly forecast for Colombus**

```
Train Time window: 2017-12-07 00:00:00
Test Time window: 2017-12-14 00:00:00
p, d, q = 2, 0, 3
{'MAPE': 54.27611390663893, 'MSE': 31.59729248409418, 'R2': 0.6755294453425318}
```

**Fig 6.2: Colombus Information**

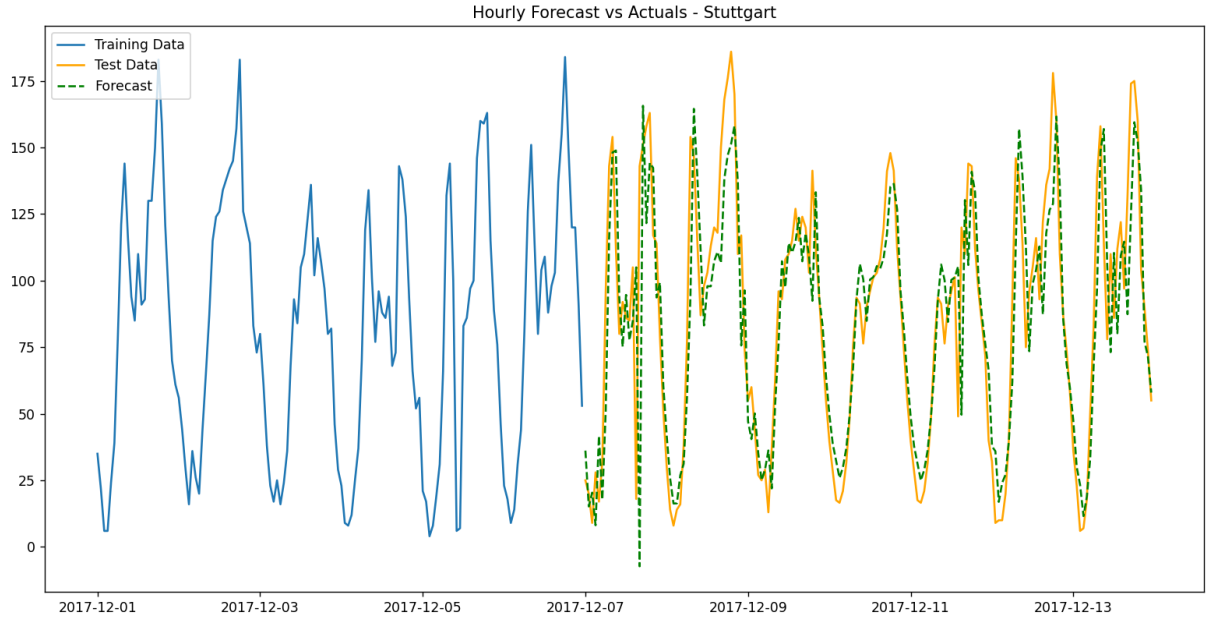



**Fig 6.3: hourly forecast for Stuttgart**

```
Train Time window: 2017-12-07 00:00:00
Test Time window: 2017-12-14 00:00:00
p, d, q = 2, 0, 3
{'MAPE': 31.547617007835697, 'MSE': 609.3090377131448, 'R2': 0.7201609985016826}
```

**Fig 6.4: Stuttgart information**

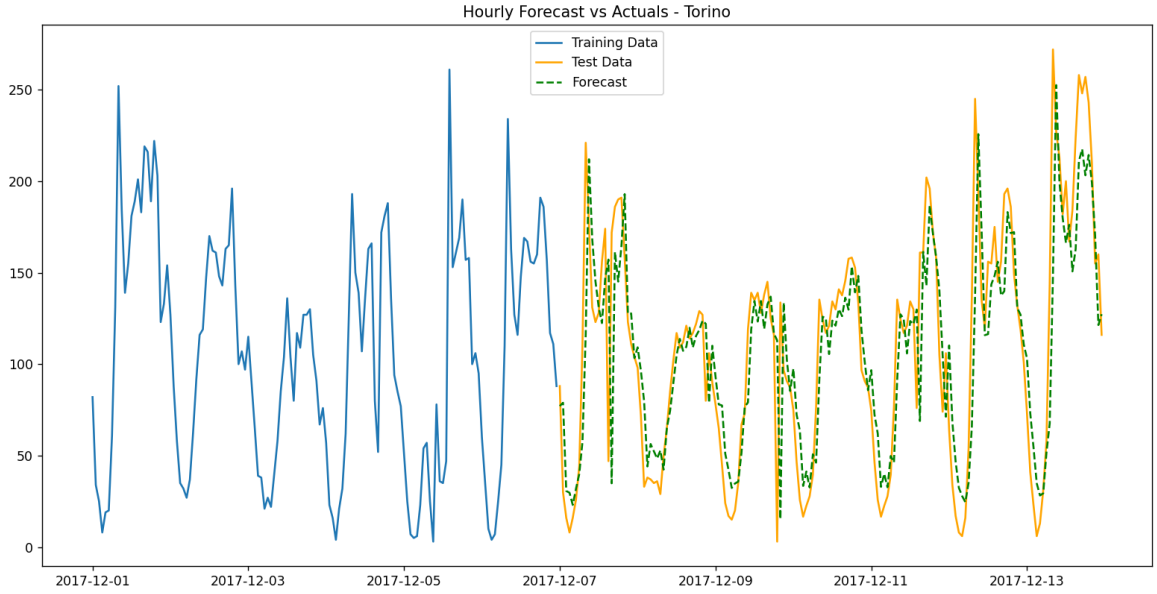



**Fig 6.5: hourly forecast for Torino**

```
Train Time window: 2017-12-07 00:00:00
Test Time window: 2017-12-14 00:00:00
p, d, q = 2, 0, 3
{'MAPE': 61.87594354852002, 'MSE': 1228.71402686918, 'R2': 0.6934091293943981}
```

**Fig 6.6: Torino information**

In this section of our analysis, we focus on the data pertaining to December 2017, following a systematic process for data preparation, model fitting, and evaluation:

**Data Filtering:** Initially, we isolated the dataset to include only the data from December 2017. This filtering step is crucial for focusing our analysis on a specific time frame.

**Data Processing for Average Count Calculation:** We then processed the filtered data to compute the average booking count for each hour. This step involved dealing with missing data, where any absent hourly counts for a given day were substituted with the corresponding hourly averages.

**Conversion to Pandas DataFrame:** The processed data was then converted into a Pandas DataFrame. This transformation is essential for leveraging Pandas' powerful data manipulation and analysis capabilities.

**Data Splitting into Training and Testing Sets:** We partitioned the DataFrame into two subsets: a training set and a testing set. This split was based on the dates, ensuring that the model is trained on one segment of the data and tested on another, distinct segment.

**Fitting an ARIMA Model:** On the training set, we applied an AutoRegressive Integrated Moving Average (ARIMA) model. ARIMA is a popular statistical method for time series forecasting, capable of capturing various temporal dynamics in the data.

**Prediction on Test Set:** Utilizing the ARIMA model fitted to the training data, we conducted predictions on the test set. This step is critical for assessing the model's forecasting capabilities on unseen data.
**Calculation of Mean Squared Error (MSE):** To evaluate the accuracy of our forecasts, we calculated the mean squared error between the predicted values and the actual values in the test set. MSE provides a quantitative measure of the model's prediction accuracy.
**Plotting the Results:** Finally, we visualized the results, plotting both the actual and predicted values. This graphical representation allows for an intuitive assessment of the model's performance and the identification of any discrepancies between predictions and actual observations.
Through these steps, we comprehensively analyze the dataset, develop a predictive model, and evaluate its performance, thereby gaining valuable insights into the temporal dynamics of the data.

what are the result then (?)

- 7 - Now check the impact of parameters:
  a- Keep *N* fixed, and do a grid search varying *(p,d,q)* and observe how the error changes. Choose the best *(p,d,q)* parameter tuple for each city. Justify your choice.
  b- Given the best parameter configuration, change *N* and the learning strategy (expanding versus sliding window). Always keep the testing set on the same portion of the data. For instance, if you use 1 week for testing, you can use from 1 day to 3 weeks for training.
  c- Compare results for the different cities. How does the relative error change w.r.t. the absolute number of rentals?

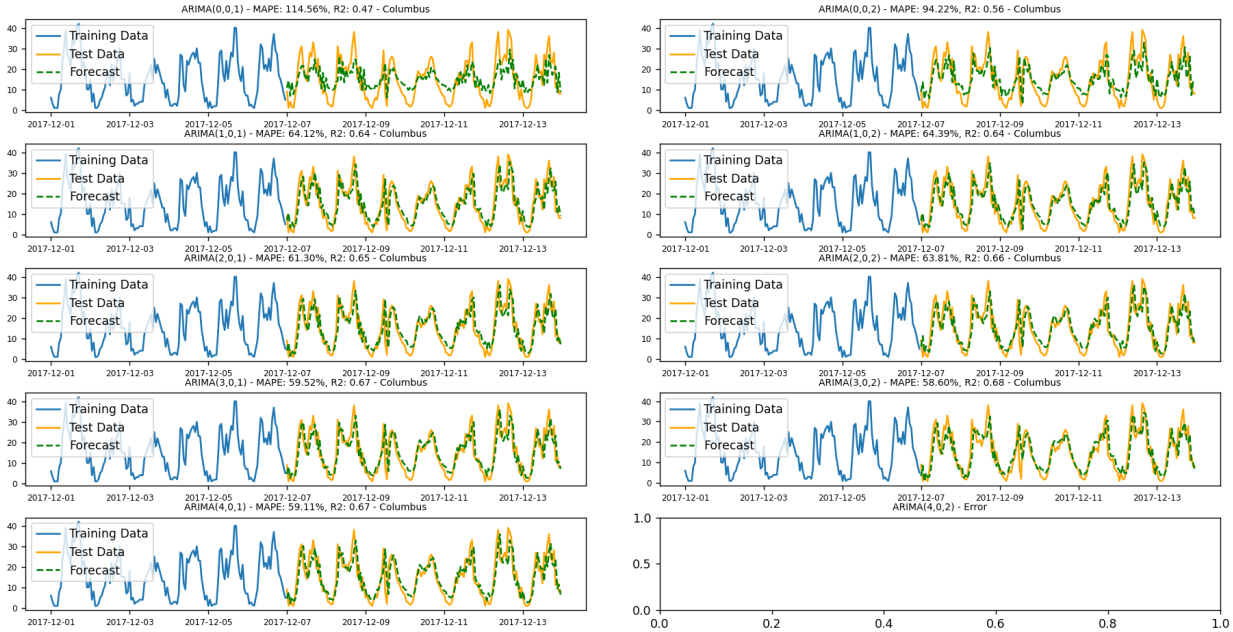



**Fig 7.1: Colombus error changes**

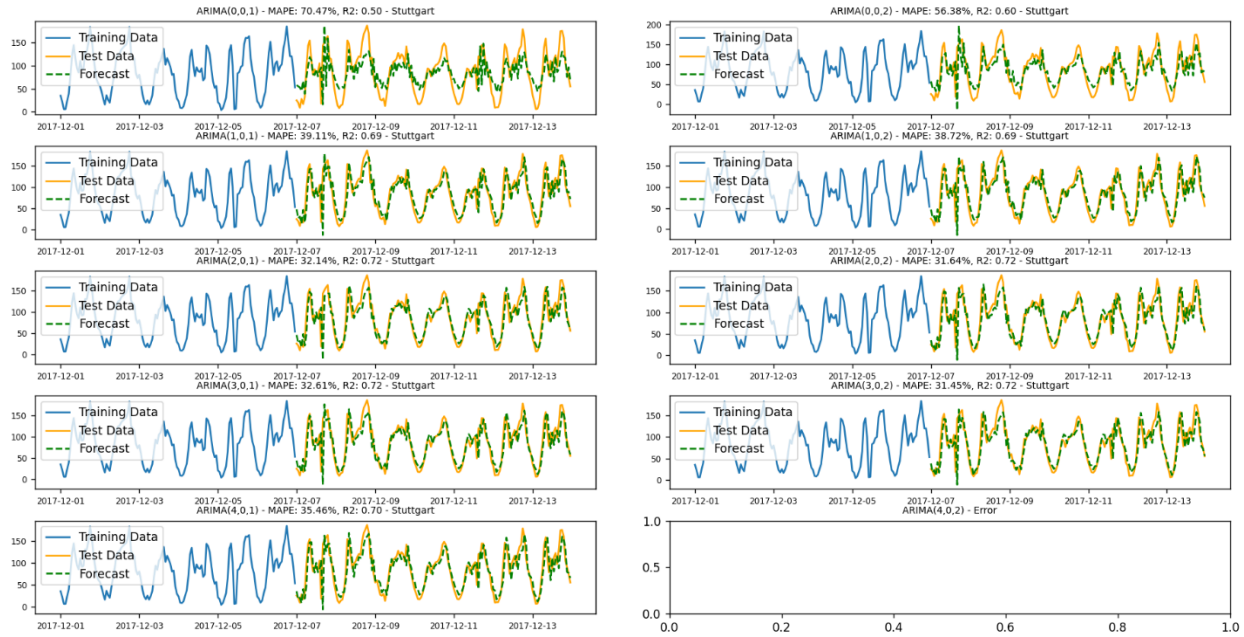



**Fig 7.2: Stuttgart error changes**

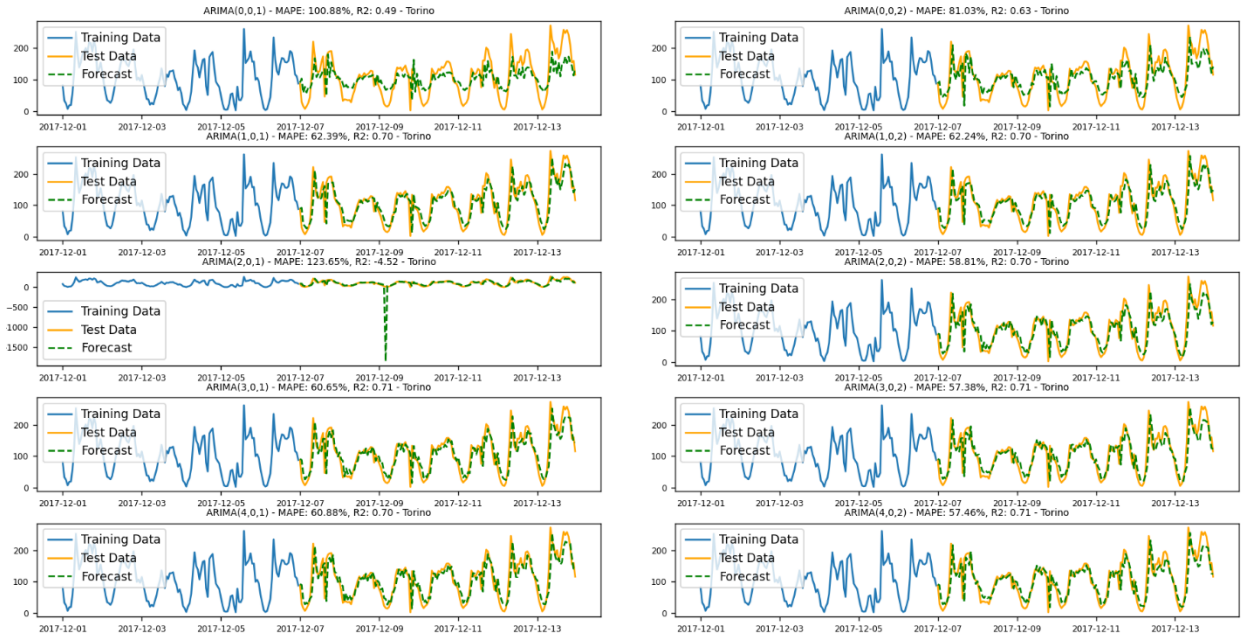



**Fig 7.3: Torino error changes**

In this segment of our analysis, we explored various configurations of the ARIMA model parameters, specifically focusing on the range of values for the autoregressive (AR) term p (ranging from 0 to 4) and the moving average (MA) term q (considering the values 1 and 2). This exploration resulted in a total of 10 distinct model scenarios. Our evaluation criteria included the Mean Absolute Percentage Error (MAPE) and the coefficient of determination R^2.

**MAPE Analysis:** The MAPE metric, which measures the accuracy of the model in terms of percentage errors, was not within acceptable limits, as it exceeded the threshold of 30. A MAPE value higher than 30 indicates a relatively high level of prediction error, which is not ideal for forecasting accuracy.

**R^2 Analysis:** In contrast, the R^2 values were comparatively higher, suggesting a decent level of agreement between the predicted values and the **actual data**. A higher R^2 indicates that the model can explain a substantial proportion of the variance in the observed data.

**Interpretation of Results:** It is important to note that if the MAPE were lower (i.e., better), it would indicate superior predictive accuracy, making the model highly desirable. The optimal scenario for model selection would be one where both MAPE is low (indicating high accuracy) and R^2 is high (indicating a good fit to the data).

**Effect of Increasing p**: We observed that models with higher values of p generally yielded better predictions. This improvement is evidenced by lower MAPE values and

cut & paste from chatGPT (?)

↳ where are the results (?)

higher R^2 scores. An increase in p implies that the model incorporates more past values, potentially capturing more of the time series dynamics.
while the high R^2 values suggest that the models fit well with the data, the elevated MAPE values indicate a lack of accuracy in the predictions. The ideal model would strike a balance with both a low MAPE and a high R^2, and increasing p seems to improve the model's performance in this regard.

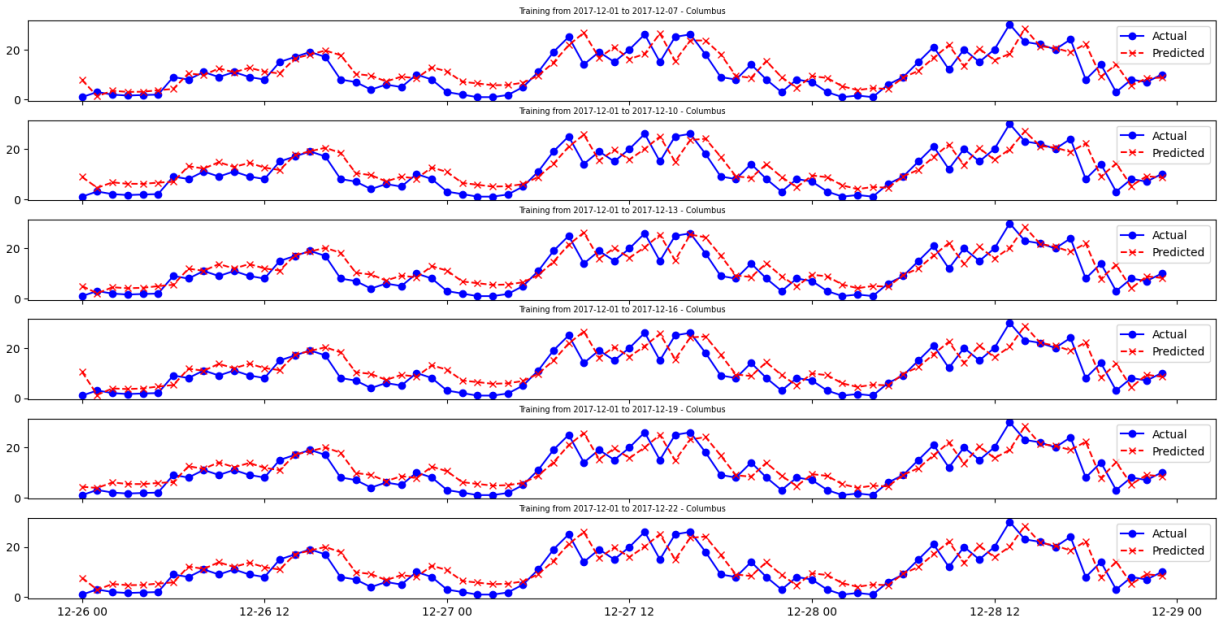



**Fig 7.4: Colombus Expanding**

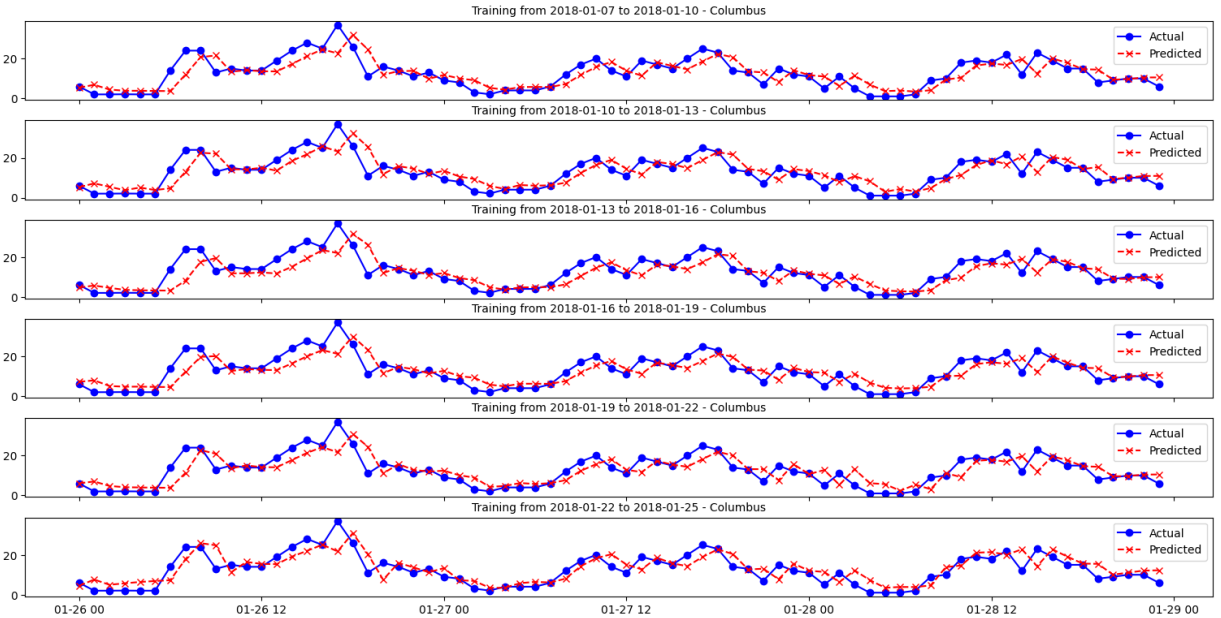



**Fig 7.5: Colombus sliding**

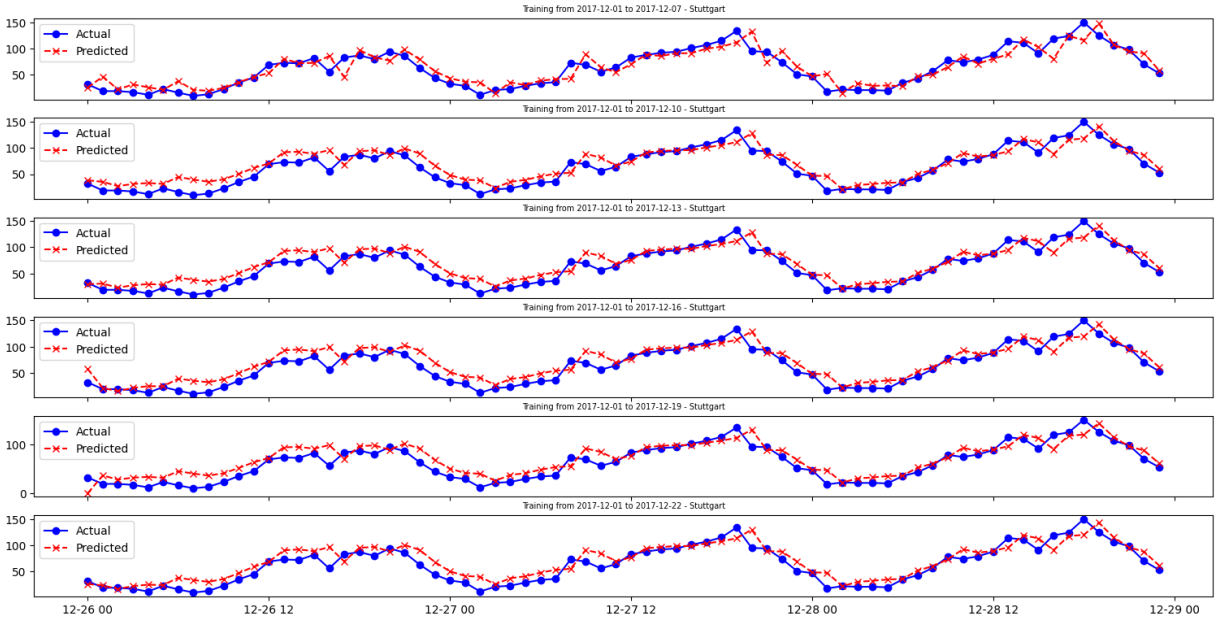



**Fig 7.6: Stuttgart Expanding**

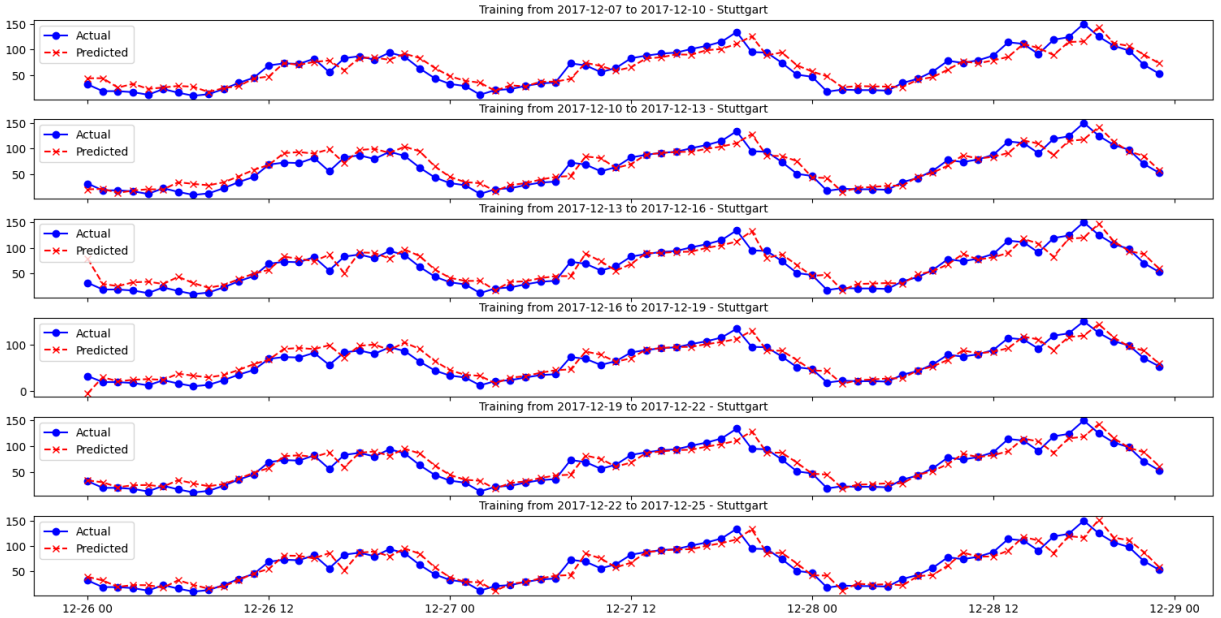



**Fig 7.7: Stuttgart sliding**

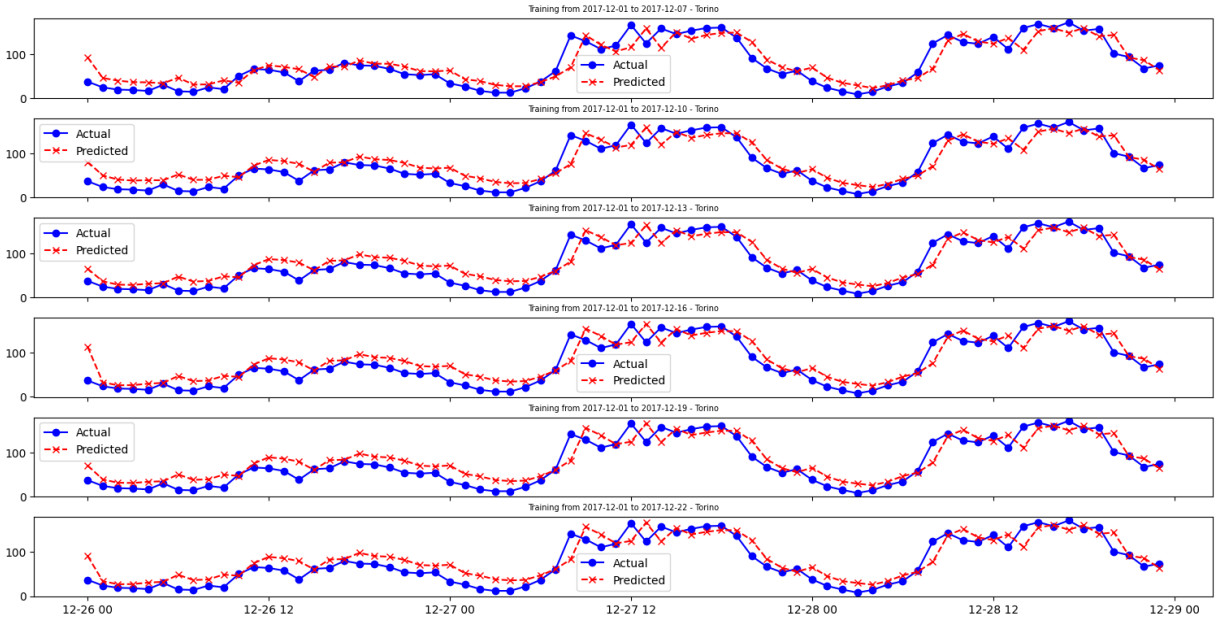



**Fig 7.8: Torino expanding**

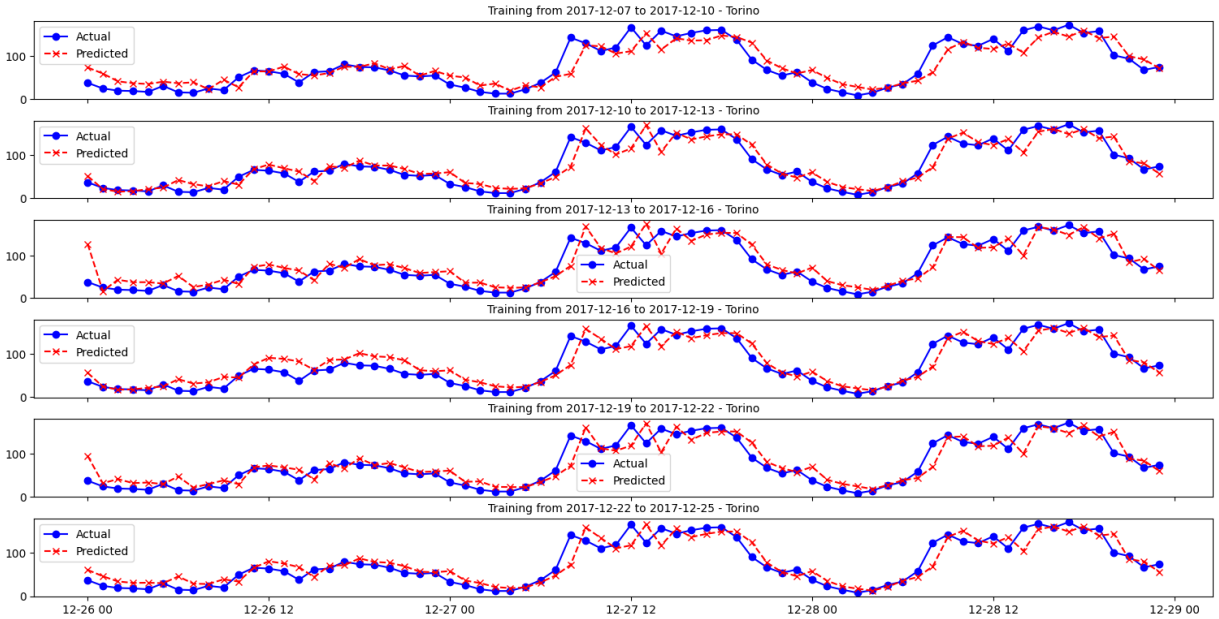



**Fig 7.9: Torino sliding**

In our time series analysis, we employed two distinct windowing techniques to partition the data for model training and evaluation: sliding windows and expanding windows. Both methods are designed to systematically vary the amount of training data, but they do so in different ways.

**Sliding Windows Approach:**

**Window Creation:** We created multiple sliding windows for the training data, specifically on the 10th, 13th, 16th, 19th, and 22nd days. This approach implies that each window encompasses a span of 3 days.

**Purpose and Evaluation:** The primary objective of using sliding windows is to understand how the model's performance varies with different consecutive subsets of the data. By shifting the training window by fixed intervals (every 3 days in this case), we can evaluate the model's stability and performance consistency across different time frames within the month.

**Expanding Windows Approach:**

**Window Creation:** In this method, we started with a window beginning on the 7th of December and expanded this window incrementally until the 22nd of December. Each subsequent window includes all the data from the previous windows plus additional data up to the new end date.

**Purpose and Evaluation:** The expanding window technique is useful for assessing the impact of increasing the volume of training data on the model's predictive performance. As the window expands, the model is trained on an increasingly larger dataset,

culminating in the window with the most comprehensive dataset (ending on the 22nd). This approach is particularly **beneficial for understanding how the accumulation of data** over time influences the model's ability to train and **forecast effectively**.
In both techniques, the key difference lies in how the data is segmented and how the training set grows over time. The sliding window approach provides insights into the **model's performance over short, fixed intervals**, while the expanding window offers an understanding of the model's behavior as it is exposed to a progressively larger dataset. Each method has its unique benefits and can be chosen based on the specific requirements and characteristics of the time series data under study.

- 8 - Try to see how the time horizon *h* of the prediction impacts the performance. Instead of predicting the number of rentals at *t+1*, use the model to predict the future rentals at *t+h, h* in [1:24] or more. What happens if you change the parameter *p*? Which value could allow a better long-term prediction?

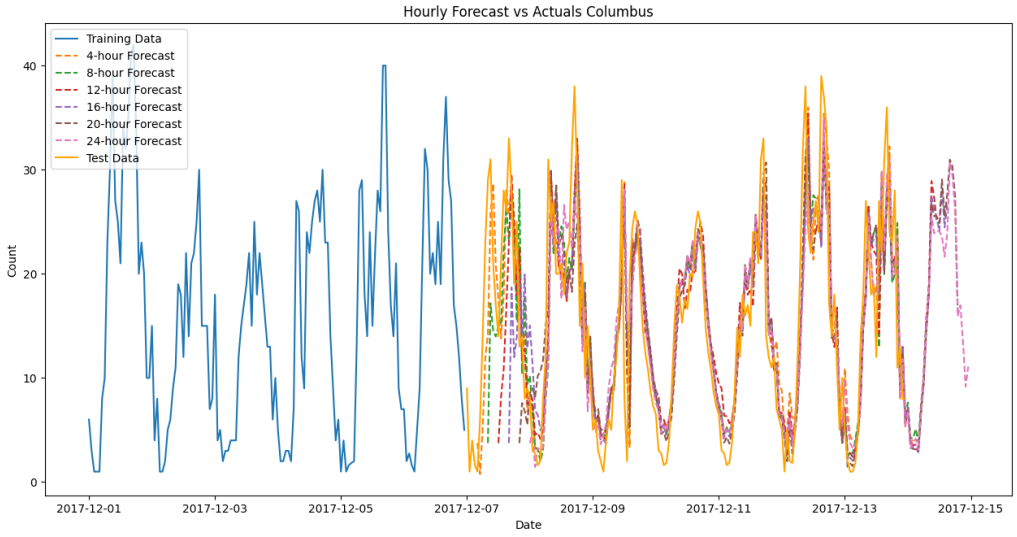



**Fig 8.1: Hourly forecast vs actual for Colombus**

```
metrics for 4 hour forecast: {'MAPE': 56.076853176535245, 'MSE': 33.674418052638615, 'R2': 0.655407187607846} in Stuttgart
metrics for 8 hour forecast: {'MAPE': 51.11923150837574, 'MSE': 37.058769647125644, 'R2': 0.6163283125015886} in Stuttgart
metrics for 12 hour forecast: {'MAPE': 54.61966190587497, 'MSE': 34.84443040405798, 'R2': 0.6403262089990818} in Stuttgart
metrics for 16 hour forecast: {'MAPE': 52.204635021957266, 'MSE': 37.627583935006086, 'R2': 0.6255321322737291} in Stuttgart
metrics for 20 hour forecast: {'MAPE': 50.503092576201865, 'MSE': 31.567820088262444, 'R2': 0.6961077507606495} in Stuttgart
metrics for 24 hour forecast: {'MAPE': 49.931569061822096, 'MSE': 30.117227020578206, 'R2': 0.7090421935139573} in Stuttgart
```

**Fig 8.2: Metrics for Colombus**

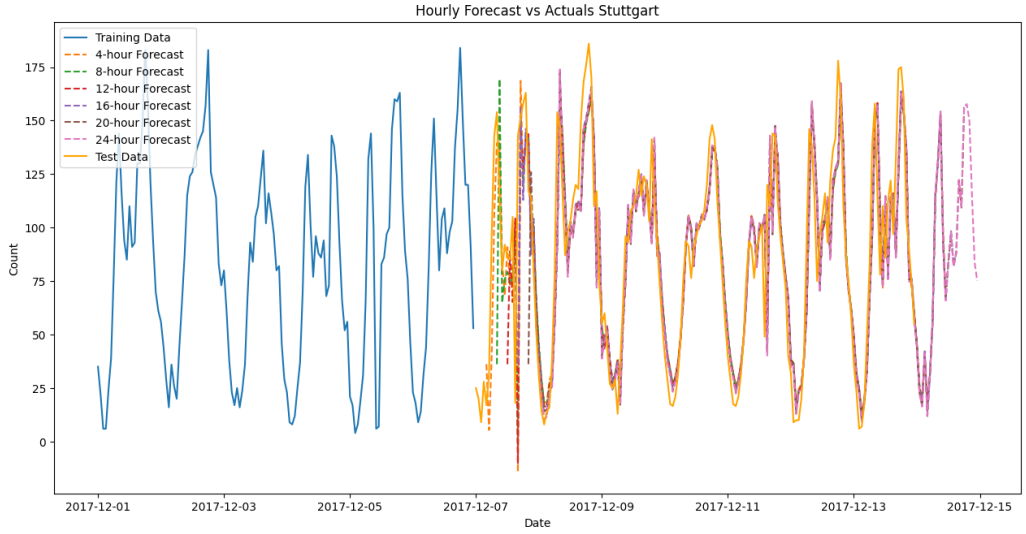



**Fig 8.3: Hourly forecast vs actual for Stuttgart**

```
metrics for 4 hour forecast: {'MAPE': 34.81234252676552, 'MSE': 638.7754672219424, 'R2': 0.7061450796151566} in Stuttgart
metrics for 8 hour forecast: {'MAPE': 33.8483601337088, 'MSE': 690.129400300127, 'R2': 0.6803887083949016} in Stuttgart
metrics for 12 hour forecast: {'MAPE': 33.223880066186155, 'MSE': 636.9632487503134, 'R2': 0.7046310471623265} in Stuttgart
metrics for 16 hour forecast: {'MAPE': 29.028043285622502, 'MSE': 523.9273857571113, 'R2': 0.7544648228570027} in Stuttgart
metrics for 20 hour forecast: {'MAPE': 28.266365529051747, 'MSE': 494.10921796471087, 'R2': 0.7701856184965322} in Stuttgart
metrics for 24 hour forecast: {'MAPE': 26.99591238619436, 'MSE': 456.9347145213912, 'R2': 0.7882154435937465} in Stuttgart
```

**Fig 8.4: metrics for Stuttgart**

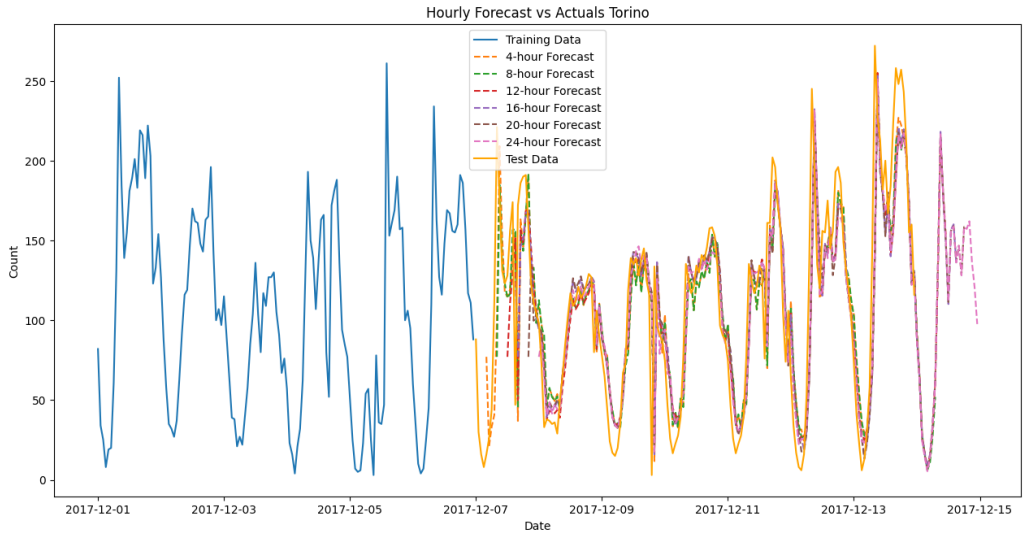



**Fig 8.5: Hourly forecast vs actual for Turin**

```
metrics for 4 hour forecast: {'MAPE': 63.321367167014365, 'MSE': 1254.3109307549041, 'R2': 0.6865155905925555} in Stuttgart
metrics for 8 hour forecast: {'MAPE': 61.17310964372186, 'MSE': 1266.5357102592704, 'R2': 0.6838509596667441} in Stuttgart
metrics for 12 hour forecast: {'MAPE': 57.40114277221002, 'MSE': 1141.1108283674978, 'R2': 0.716152984170098} in Stuttgart
metrics for 16 hour forecast: {'MAPE': 54.94943042570542, 'MSE': 1052.3595549733923, 'R2': 0.7384231396237146} in Stuttgart
metrics for 20 hour forecast: {'MAPE': 55.0340353459419, 'MSE': 1049.4362618302077, 'R2': 0.7360368762823039} in Stuttgart
metrics for 24 hour forecast: {'MAPE': 53.1750858404464, 'MSE': 964.0699015926643, 'R2': 0.757532633946032} in Stuttgart
```

**Fig 8.6: metrics for Turin**

In our comparative analysis of time series predictions across various cities, we observed a consistent pattern: the 24-hour forecast period consistently yielded the most accurate results. This observation is substantiated by two key performance metrics:

**Coefficient of Determination R^2:** The R^2 values for the 24-hour prediction period were the highest among all forecast intervals. A high R^2 value indicates that the model's predictions closely match the actual data, suggesting a strong fit of the model to the observed data points.

**Mean Squared Error (MSE):** The MSE, a measure of the average squared difference between the predicted and actual values, was found to be the lowest for the 24-hour prediction interval. Lower MSE values imply greater prediction accuracy, as the predicted values deviate less from the actual values.

The superior performance of the 24-hour prediction can be attributed to a couple of key factors:

**Data Density:** The 24-hour period likely benefits from a higher density of data points. More data within this interval provides a richer set of information for the model to learn from, leading to more accurate predictions.

**Temporal Proximity:** Predictions over a 24-hour period are based on more recent and, therefore, more relevant data. As the prediction interval extends beyond 24 hours, the model relies on data that is increasingly distant in time. This temporal gap can introduce greater uncertainty and variability, leading to lower R^2 values and higher MSE.

- max page limit not respected by far
- very confused and with unclear result
- choice of plots is not clear and does not convey a clear message
- lot of repetitions and useless descriptions

REDO

Appendixes:

Q1:

```python
from datetime import datetime
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt


client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    '$and': [
        {'init_date': {'$gte': start_date}},
        {'init_date': {'$lt': end_date}},
        {'$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]}},
        {'$expr': {'$ne': ['$init_address', '$final_address']}}
    ]
}
```

```
# Aggregation pipeline
pipeline = [
    [
        '$match': query_conditions,
        "city": "Torino"
    },
    [
        '$group': {
            '_id': {
                'day': {'$dayOfMonth': '$init_date'},
                'hour': {'$hour': '$init_date'}
            },
            'count': {'$sum': 1}
        }
    },
    {
        '$sort': {
            '_id.day': 1,
            '_id.hour': 1
        }
    }
]

# Execute the query to retrieve the records
records = db["PermanentBookings"].aggregate(pipeline)

# Prepare the data for plotting
hours = []
counts = []
for record in records:
    # Convert day and hour to a single hour number for the entire month
    day_hour = (record['_id']['day'] - 1) * 24 + record['_id']['hour']
    hours.append(day_hour)
    counts.append(record['count'])
```

```
# Prepare the data for plotting
hours = []
counts = []
for record in records:
    # Convert day and hour to a single hour number for the entire month
    day_hour = (record['_id']['day'] - 1) * 24 + record['_id']['hour']
    hours.append(day_hour)
    counts.append(record['count'])

# Plotting
plt.figure(figsize=(15, 6))

# Connect the points with lines
plt.plot(hours, counts, color='blue', linestyle='-', linewidth="2", marker='', label='Bookings Count')

# Label formatting to show day and hour
tick_positions = range(0, 24 * 30, 24)  # One tick per day
tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
plt.xticks(tick_positions, tick_labels, rotation=45)  # Set x-ticks to be every day and rotate for readability

plt.xlabel('Day and Hour')
plt.ylabel('Count of Bookings')
plt.title('Count of Bookings for Each Hour of November 2017')
plt.grid(True)  # Add grid for better readability
plt.legend()

# Show the plot
plt.tight_layout()  # Adjust layout so everything fits without overlapping
plt.show()
```

Q2:

D: > class files > 3.semester > mobility > mellia > lab 2 > lab2 - docs > 2 > 2.py

```python
from datetime import datetime, timedelta
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt

client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Torino",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
    '$expr': {'$ne': ['$init_address', '$final_address']}
}
```

```python
# Aggregation pipeline
pipeline = [
    {
        '$match': query_conditions
    },
    {
        '$group': {
            '_id': {
                'day': {'$dayOfMonth': '$init_date'},
                'hour': {'$hour': '$init_date'}
            },
            'count': {'$sum': 1}
        }
    },
    {
        '$sort': {
            '_id.day': 1,
            '_id.hour': 1
        }
    }
]

# Execute the query to retrieve the records
records = list(db["PermanentBookings"].aggregate(pipeline))

# Create a dictionary to hold the counts for each day and hour
counts_dict = defaultdict(lambda: defaultdict(int))
```

```
  # Populate the dictionary with the query results
for record in records:
    day = record['_id']['day']
    hour = record['_id']['hour']
    count = record['count']
    counts_dict[day][hour] = count

# Calculate the average count for each hour across all days
hourly_averages = defaultdict(list)
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        hourly_averages[hour].append(count)

# Compute averages
hourly_averages = {hour: np.mean(counts) for hour, counts in hourly_averages.items()}

# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days
    for hour in range(24):
        if hour in counts_dict[day]:
            completed_counts.append(((day - 1) * 24 + hour, counts_dict[day][hour]))
        else:
            completed_counts.append(((day - 1) * 24 + hour, hourly_averages[hour]))

# Unzip the completed_counts for plotting
hours, counts = zip(*completed_counts)
```

```
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days
    for hour in range(24):
        if hour in counts_dict[day]:
            completed_counts.append(((day - 1) * 24 + hour,  (variable) hourly_averages: dict
        else:
            completed_counts.append(((day - 1) * 24 + hour, hourly_averages[hour]))

# Unzip the completed_counts for plotting
hours, counts = zip(*completed_counts)

# Plotting
plt.figure(figsize=(15, 6))
plt.plot(hours, counts, color='blue', linestyle='-', linewidth="2", marker='', label='Bookings Count')

# Label formatting to show day and hour
tick_positions = range(0, 24 * 31, 24)  # One tick per day
tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
plt.xticks(tick_positions, tick_labels, rotation=45)  # Set x-ticks to be every day and rotate for readability

plt.xlabel('Day and Hour')
plt.ylabel('Count of Bookings')
plt.title('Count of Bookings for Each Hour of December 2017')
plt.grid(True)  # Add grid for better readability
plt.legend()

# Show the plot
plt.tight_layout()  # Adjust layout so everything fits without overlapping
plt.show()
```

Q3:

```python
from datetime import datetime
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
import matplotlib.dates as mdates

client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Torino",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
    '$expr': {'$ne': ['$init_address', '$final_address']}
}

# Aggregation pipeline
pipeline = [
    {
        '$match': query_conditions
    },
```

```
    if day not in counts_dict:
        counts_dict[day] = {}  # Create a new dictionary for the day if it doesn't exist

    # Assuming that there will be only one count per day and hour,
    # otherwise, you need logic to sum or handle multiple counts
    counts_dict[day][hour] = count


# Initialize an empty dictionary to hold lists of counts for each hour
hourly_averages = {}

# Populate the dictionary with counts for each hour
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        if hour not in hourly_averages:
            hourly_averages[hour] = []  # Initialize a new list if the hour key doesn't exist
        hourly_averages[hour].append(count)

# Compute averages using numpy for mean calculation
hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):



        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))
```

```
        {
            '$match': query_conditions
        },
        {
            '$group': {
                '_id': {
                    'day': {'$dayOfMonth': '$init_date'},
                    'hour': {'$hour': '$init_date'}
                },
                'count': {'$sum': 1}
            }
        },
        {
            '$sort': {
                '_id.day': 1,
                '_id.hour': 1
            }
        }
    ]

    # Execute the query to retrieve the records
    records = list(db["PermanentBookings"].aggregate(pipeline))

    # Create a standard dictionary to hold the counts for each day and hour
    counts_dict = {}

    # Populate the dictionary with the query results
    for record in records:
        day = record['_id']['day']
        hour = record['_id']['hour']
        count = record['count']
```

```
        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))

        completed_counts.append(((day - 1) * 24 + hour, count))

# Convert completed_counts into a DataFrame
df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])

# Convert 'hours_since_start' to a datetime index starting from 'start_date'
df_completed['datetime'] = pd.to_datetime(df_completed['hours_since_start'], unit='h', origin=start_date)

# Perform the ADF test
adf_test_filled = adfuller(df_completed['count'])

# Output the ADF statistic and p-value
print(f"ADF Statistic: {adf_test_filled[0]}")
print(f"p-value: {adf_test_filled[1]}")
for key, value in adf_test_filled[4].items():
    print(f"   {key}, {value}")

# Check the stationarity of the filled series
if adf_test_filled[1] < 0.05:
    print("The filled time series is stationary.")
else:
    print("The filled time series is not stationary.")

# Convert 'count' to a series for rolling operations
counts_series_filled = df_completed['count']
```

```
# Check the stationarity of the filled series
if adf_test_filled[1] < 0.05:
    print("The filled time series is stationary.")
else:
    print("The filled time series is not stationary.")

# Convert 'count' to a series for rolling operations
counts_series_filled = df_completed['count']

# Calculate the rolling mean and the rolling standard deviation
rolling_mean_filled = counts_series_filled.rolling(window=24).mean()
rolling_std_filled = counts_series_filled.rolling(window=24).std()

# Plot the filled time series with rolling statistics
fig, ax = plt.subplots(figsize=(14, 7))
ax.plot(counts_series_filled, color='blue', label='Filled Series')
ax.plot(rolling_mean_filled, color='red', label='Rolling Mean')
ax.plot(rolling_std_filled, color='black', label='Rolling Std Dev')

ax.set_title('Filled Time Series with Rolling Mean & Standard Deviation')
ax.set_xlabel('Hour')
ax.set_ylabel('Counts')

# To improve readability of the x-axis
plt.xticks(rotation=45)
ax.xaxis.set_major_locator(plt.MaxNLocator(8))

ax.legend()
plt.show()
```

Q4:

```python
from datetime import datetime
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
import matplotlib.dates as mdates
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Torino",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
    '$expr': {'$ne': ['$init_address', '$final_address']}
}

# Aggregation pipeline
pipeline = [
    {
        '$match': query_conditions
```

```python
        '$group': {
            '_id': {
                'day': {'$dayOfMonth': '$init_date'},
                'hour': {'$hour': '$init_date'}
            },
            'count': {'$sum': 1}
        }
    },
    {
        '$sort': {
            '_id.day': 1,
            '_id.hour': 1
        }
    }
]

# Execute the query to retrieve the records
records = list(db["PermanentBookings"].aggregate(pipeline))

# Create a standard dictionary to hold the counts for each day and hour
counts_dict = {}

# Populate the dictionary with the query results
for record in records:
    day = record['_id']['day']
    hour = record['_id']['hour']
    count = record['count']

    if day not in counts_dict:
        counts_dict[day] = {}  # Create a new dictionary for the day if it doesn't exist
```

```
    if day not in counts_dict:
        counts_dict[day] = {}  # Create a new dictionary for the day if it doesn't exist

    # Assuming that there will be only one count per day and hour,
    # otherwise, you need logic to sum or handle multiple counts
    counts_dict[day][hour] = count


# Initialize an empty dictionary to hold lists of counts for each hour
hourly_averages = {}

# Populate the dictionary with counts for each hour
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        if hour not in hourly_averages:
            hourly_averages[hour] = []  # Initialize a new list if the hour key doesn't exist
        hourly_averages[hour].append(count)

# Compute averages using numpy for mean calculation
hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):

        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))
```

```python
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):


        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))

        completed_counts.append(((day - 1) * 24 + hour, count))

# Convert completed_counts into a DataFrame
df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])


# Assuming 'y' is the column in 'df' DataFrame containing your time series data
time_series = df_completed['count']

# Plot ACF
plt.figure()
plot_acf(time_series, lags=40, alpha=0.05)  # Adjust lags as needed
plt.title('Autocorrelation Function')
plt.show()

# Plot PACF
plt.figure()
plot_pacf(time_series, lags=40, alpha=0.05, method='ywm')  # Adjust lags and method as needed
plt.title('Partial Autocorrelation Function')
plt.show()
```

Q5:

```python
from datetime import datetime, timedelta
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt

1 usage
def split_data(records, train_start, train_end, test_start, test_end):
    train_data = [record for record in records if train_start <= record['_id']['day'] < train_end]
    test_data = [record for record in records if test_start <= record['_id']['day'] < test_end]
    return train_data, test_data

client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime( year: 2017,  month: 12,  day: 1)
end_date = datetime( year: 2018,  month: 1,  day: 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Stuttgart",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
```

```
    '$expr': {'$ne': ['$init_address', '$final_address']}
}

# Aggregation pipeline
pipeline = [
    {
        '$match': query_conditions
    },
    {
        '$group': {
            '_id': {
                'day': {'$dayOfMonth': '$init_date'},
                'hour': {'$hour': '$init_date'}
            },
            'count': {'$sum': 1}
        }
    },
    {
        '$sort': {
            '_id.day': 1,
            '_id.hour': 1
        }
    }
]
# Execute the query to retrieve the records
records = list(db["PermanentBookings"].aggregate(pipeline))
```

```
# Split data into training and testing sets
train_start, train_end = 1, 8  # Training set: first week
test_start, test_end = 8, 15  # Testing set: second week
train_data, test_data = split_data(records, train_start, train_end, test_start, test_end)

train_counts_dict = defaultdict(lambda: defaultdict(int))

# Populate the dictionary with the training data
for record in train_data:
    day = record['_id']['day']
    hour = record['_id']['hour']
    count = record['count']
    train_counts_dict[day][hour] = count

# Calculate the average count for each hour across all days for training data
train_hourly_averages = defaultdict(list)
for day, hours in train_counts_dict.items():
    for hour, count in hours.items():
        train_hourly_averages[hour].append(count)

# Compute averages for training data
train_hourly_averages = {hour: np.mean(counts) for hour, counts in train_hourly_averages.items()}

# Fill in missing hours for each day with the average counts for training data
completed_train_counts = []
for day in range(train_start, train_end):
```

```
    for hour in range(24):
        if hour in train_counts_dict[day]:
            completed_train_counts.append(((day - 1) * 24 + hour, train_counts_dict[day][hour]))
        else:
            completed_train_counts.append(((day - 1) * 24 + hour, train_hourly_averages[hour]))

# Unzip the completed_train_counts for plotting
train_hours, train_counts = zip(*completed_train_counts)

# Plotting for training data
plt.figure(figsize=(15, 6))
plt.plot( *args: train_hours, train_counts, color='blue', linestyle='-', linewidth="2", marker='', label='Training Data')

# Create a dictionary to hold the counts for each day and hour for testing data
test_counts_dict = defaultdict(lambda: defaultdict(int))

# Populate the dictionary with the testing data
for record in test_data:
    day = record['_id']['day']
    hour = record['_id']['hour']
    count = record['count']
    test_counts_dict[day][hour] = count

# Fill in missing hours for each day with the average counts for testing data
completed_test_counts = []
for day in range(test_start, test_end):
```

```
    for hour in range(24):
        if hour in test_counts_dict[day]:
            completed_test_counts.append(((day - 1) * 24 + hour, test_counts_dict[day][hour]))
        else:
            completed_test_counts.append(((day - 1) * 24 + hour, train_hourly_averages[hour]))  # Filling missing values with training data averages

# Unzip the completed_test_counts for plotting
test_hours, test_counts = zip(*completed_test_counts)

# Plotting for testing data
plt.plot( *args: test_hours, test_counts, color='red', linestyle='-', linewidth="2", marker='', label='Testing Data')

# Label formatting to show day and hour
tick_positions = range(0, 24 * (train_end - train_start), 24)  # One tick per day for training data
tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
plt.xticks(tick_positions, tick_labels, rotation=45)  # Set x-ticks to be every day and rotate for readability

plt.xlabel('Day and Hour')
plt.ylabel('Count of Bookings')
plt.title('Count of Bookings for Each Hour of December 2017')
plt.grid(True)  # Add grid for better readability
plt.legend()

# Show the plot
plt.tight_layout()  # Adjust layout so everything fits without overlapping
plt.show()
```

Q6:

```python
from datetime import datetime, timedelta
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# Connect to the MongoDB client
client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Stuttgart",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
    '$expr': {'$ne': ['$init_address', '$final_address']}
}

# Aggregation pipeline
pipeline = [
    {
        '$group': {
            '_id': {
                'day': {'$dayOfMonth': '$init_date'},
                'hour': {'$hour': '$init_date'}
            },
            'count': {'$sum': 1}
        }
    },
    {
        '$sort': {
            '_id.day': 1,
            '_id.hour': 1
        }
    }
]

# Execute the query to retrieve the records
records = list(db["PermanentBookings"].aggregate(pipeline))

# Create a standard dictionary to hold the counts for each day and hour
counts_dict = {}

# Populate the dictionary with the query results
for record in records:
    day = record['_id']['day']
    hour = record['_id']['hour']
    count = record['count']

    if day not in counts_dict:
        counts_dict[day] = {}  # Create a new dictionary for the day if it doesn't exist

    # Assuming that there will be only one count per day and hour,
    # otherwise, you need logic to sum or handle multiple counts
```

```
    counts_dict[day][hour] = count


# Initialize an empty dictionary to hold lists of counts for each hour
hourly_averages = {}

# Populate the dictionary with counts for each hour
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        if hour not in hourly_averages:
            hourly_averages[hour] = []  # Initialize a new list if the hour key doesn't exist
        hourly_averages[hour].append(count)

# Compute averages using numpy for mean calculation
hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):



        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))

        completed_counts.append(((day - 1) * 24 + hour, count))

# Convert completed_counts into a DataFrame
df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])

```

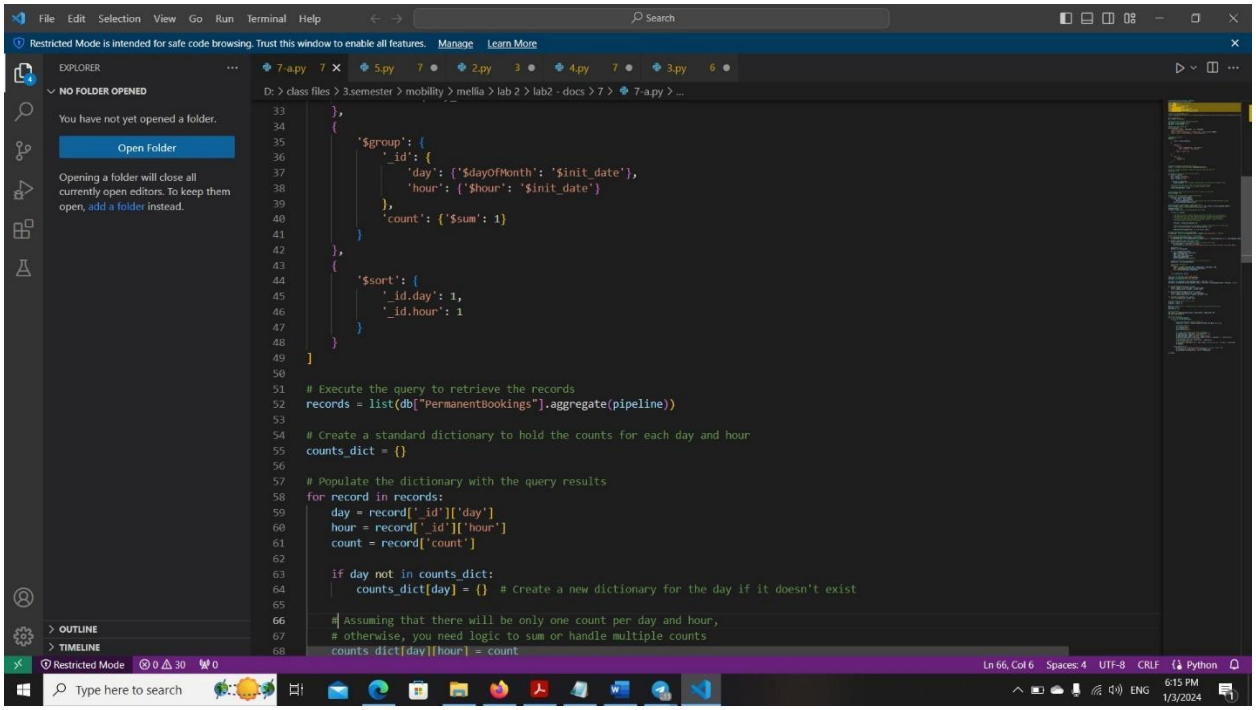
```
if not isinstance(df_completed.index, pd.DatetimeIndex):
    df_completed.index = pd.to_datetime([datetime(2017, 12, 1) + timedelta(hours=x) for x in df_completed.index])

def predict_arima(train_data, test_data, order):
    # Ensure the test_data is not empty and does not contain zero values
    if test_data.empty or (test_data == 0).any():
        raise ValueError("Test data must not be empty and must not contain zeros when calculating MAPE.")


    predictions = []
    history = list(train_data)
    for                                  a)):
        model = ARIMA(history, order=order)
        model_fit = model.fit()
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test_data.iloc[t])

    # Convert predictions to a numpy array for error calculation
    predictions = np.array(predictions)

    # Calculate error metrics
    metrics = {
        "MAPE": np.mean(np.abs(test_data - predictions) / test_data) * 100,
        "MSE": mean_squared_error(test_data, predictions),
        "R2": r2_score(test_data, predictions)
    }

    return predictions, metrics

# Now, use the function in your given context
train_end = pd.Timestamp('2017-12-07 00:00:00')
```

(variable) model: Any

```
test_end = pd.Timestamp('2017-12-14 00:00:00')

train_data = df_completed.loc[df_completed.index < train_end, 'count']
test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']

# Call the predict_arima function
predictions, metrics = predict_arima(train_data, test_data, (2, 0, 3))

# Create a pandas Series for the forecasted values
forecast_series = pd.Series(predictions, index=test_data.index)

# Plot the training data, test data, and forecast
plt.figure(figsize=(14, 7))
plt.plot(train_data, label='Training Data')
plt.plot(test_data, label='Test Data', color='orange')
plt.plot(forecast_series, label='Forecast', linestyle='--', color='green')
plt.legend()
plt.title('Hourly Forecast vs Actuals - Stuttgart')
plt.show()

# Optionally, print out the error metrics
print("Train Time window: 2017-12-07 00:00:00 ")
print("Test Time window: 2017-12-14 00:00:00")
print("p, d, q = 2, 0, 3")
print(metrics)
```

Q7:

```
from datetime import datetime, timedelta
from collections import defaultdict
import numpy as np
import pandas as pd
from pymongo import MongoClient
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# Connect to the MongoDB client
client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")

# Get the database
db = client['carsharing']

# Define the start and end dates for the query
start_date = datetime(2017, 12, 1)
end_date = datetime(2018, 1, 1)

# Define the query conditions
query_conditions = {
    'init_date': {'$gte': start_date, '$lt': end_date},
    'city': "Stuttgart",
    '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
    '$expr': {'$ne': ['$init_address', '$final_address']}
}

# Aggregation pipeline
pipeline = [
    {
        '$match': query_conditions
```




```
    # otherwise, you need logic to sum or handle multiple counts
    counts_dict[day][hour] = count


# Initialize an empty dictionary to hold lists of counts for each hour
hourly_averages = {}

# Populate the dictionary with counts for each hour
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        if hour not in hourly_averages:
            hourly_averages[hour] = []  # Initialize a new list if the hour key doesn't exist
        hourly_averages[hour].append(count)

# Compute averages using numpy for mean calculation
hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):


        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))

        completed_counts.append(((day - 1) * 24 + hour, count))
```

```
    # otherwise, you need logic to sum or handle multiple counts
    counts_dict[day][hour] = count


# Initialize an empty dictionary to hold lists of counts for each hour
hourly_averages = {}

# Populate the dictionary with counts for each hour
for day, hours in counts_dict.items():
    for hour, count in hours.items():
        if hour not in hourly_averages:
            hourly_averages[hour] = []  # Initialize a new list if the hour key doesn't exist
        hourly_averages[hour].append(count)

# Compute averages using numpy for mean calculation
hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
# Fill in missing hours for each day with the average counts
completed_counts = []
for day in range(1, 32):  # Assuming December has 31 days

    for hour in range(24):


        hours_dict = counts_dict.get(day, {})

        # Use get to provide the hourly average as a default if the hour is not in hours_dict
        count = hours_dict.get(hour, hourly_averages.get(hour, 0))

        completed_counts.append(((day - 1) * 24 + hour, count))

# Convert completed_counts into a DataFrame
df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])
```

```
train_end = pd.Timestamp('2017-12-07 00:00:00')
test_end = pd.Timestamp('2017-12-14 00:00:00')

train_data = df_completed.loc[df_completed.index < train_end, 'count']
test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']


def mean_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean((y_true - y_pred) / y_true) * 100

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def calculate_r_squared(y_true, y_pred):
    return r2_score(y_true, y_pred) * 100

# Set the range of values for p, d, and q
p_values = range(0, 5)
q_values = range(1, 3)

best_r2 = float('-inf')  # Initialize with a low value since we're maximizing R²
best_order = None
best_metrics = {}

# Create subplots
fig, axes = plt.subplots(len(p_values), len(q_values), figsize=(20, 20))
fig.tight_layout(pad=5.0)

# Perform grid search
for i, p in enumerate(p_values):
    for j, q in enumerate(q_values):
```

```
# Perform grid search
for i, p in enumerate(p_values):
    for j, q in enumerate(q_values):
        try:
            # Fit the ARIMA model on the training set
            predictions, metrics = predict_arima(train_data, test_data, (p, 0, q))

            # Calculate metrics
            mse = metrics["MSE"]
            r2 = metrics["R2"]
            mape = metrics["MAPE"]

            # Plot the training data, test data, and forecast
            ax = axes[i][j] if axes.ndim > 1 else axes[max(i, j)]
            ax.plot(train_data, label='Training Data')
            ax.plot(test_data, label='Test Data', color='orange')
            ax.plot(test_data.index, predictions, label='Forecast', linestyle='--', color='green')
            # Set the font size of the tick labels
            ax.tick_params(axis='both', which='major', labelsize=7)
            # Set the font size title
            ax.set_title(f'ARIMA({p},0,{q}) - MAPE: {mape:.2f}%, R2: {r2:.2f} - Stuttgart', fontsize=8)
            ax.
                (class) Exception

        except Exception as e:
            print(f"An error occurred for model with (p, d, q) = ({p}, 0, {q}): {e}")
            ax = axes[i][j] if axes.ndim > 1 else axes[max(i, j)]
            ax.set_title(f'ARIMA({p},0,{q}) - Error', fontsize=8)

plt.show()
```

7B:

```
    return predictions, metrics

# Define the initial training period
initial_train_end = pd.Timestamp('2017-12-07 00:00:00')

start_date = '2017-12-01'
end_date = '2017-12-25'
expanding_windows = [pd.Timestamp(initial_train_end) + pd.Timedelta(days=i) for i in range(0, (pd.Timestamp(end_date) - pd.Timesta

# Ensure we only take the first 6 windows
expanding_windows = expanding_windows[:6]

# Set up the matplotlib figure - 6 subplots (one for each expanding window)
fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)

# Define the test period outside the loop as they are the same for all windows
test_start = pd.Timestamp('2017-12-26 00:00:00')
test_end = pd.Timestamp('2017-12-28 23:59:59')
test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']

for i, expand_day in enumerate(expanding_windows):
    train_end = expand_day

    # Split the data into train and test sets according to the current window
    train_data = df_completed.loc[df_completed.index <= train_end, 'count']

    # Use the provided predict_arima function
    predictions, metrics = predict_arima(train_data, test_data, (2,0,1))

    # Plot actual vs predicted values
    # The test_data.index gives you the correct timestamps for the x-axis
    axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
    axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')
```

```
expanding_windows = expanding_windows[:6]

# Set up the matplotlib figure - 6 subplots (one for each expanding window)
fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)

# Define the test period outside the loop as they are the same for all windows
test_start = pd.Timestamp('2017-12-26 00:00:00')
test_end = pd.Timestamp('2017-12-28 23:59:59')
test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']

for i, expand_day in enumerate(expanding_windows):
    train_end = expand_day

    # Split the data into train and test sets according to the current window
    train_data = df_completed.loc[df_completed.index <= train_end, 'count']

    # Use the provided predict_arima function
    predictions, metrics = predict_arima(train_data, test_data, (2,0,1))

    # Plot actual vs predicted values
    # The test_data.index gives you the correct timestamps for the x-axis
    axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
    axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')

    # Set title for each subplot
    axes[i].set_title(f'Training from 2017-12-01 to {expand_day.strftime("%Y-%m-%d")} - Columbus', fontsize=7)

    # Add legend
    axes[i].legend()

# Improve plot layout
plt.tight_layout()
plt.show()
```

Sliding:

```
# Define the initial training start and end dates
initial_train_start = pd.Timestamp('2017-12-07 00:00:00')
initial_train_end = initial_train_start + pd.Timedelta(days=7)

# Set the sliding step in days
slide_step = 3

# Calculate the number of windows by dividing the total number of days by the step
total_days = (pd.Timestamp('2017-12-25') - initial_train_start).days
num_windows = total_days // slide_step

# Generate start and end dates for each sliding window
sliding_windows = [(initial_train_start + pd.Timedelta(days=i*slide_step),
                    initial_train_start + pd.Timedelta(days=(i+1)*slide_step))
                   for i in range(num_windows)]

# Ensure we only take the first 6 windows
sliding_windows = sliding_windows[:6]

# Set up the matplotlib figure - 6 subplots (one for each sliding window)
fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)

# Define the test period outside the loop as they are the same for all windows
test_start = pd.Timestamp('2017-12-26 00:00:00')
test_end = pd.Timestamp('2017-12-28 23:59:59')
test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']
```

```python
# Ensure we only take the first 6 windows
sliding_windows = sliding_windows[:6]

# Set up the matplotlib figure - 6 subplots (one for each sliding window)
fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)

# Define the test period outside the loop as they are the same for all windows
test_start = pd.Timestamp('2017-12-26 00:00:00')
test_end = pd.Timestamp('2017-12-28 23:59:59')
test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']

for i, (train_start, train_end) in enumerate(sliding_windows):
    # Split the data into train and test sets according to the current window
    train_data = df_completed.loc[(df_completed.index >= train_start) & (df_completed.index <= train_end), 'count']

    # Use the provided predict_arima function
    predictions, metrics = predict_arima(train_data, test_data, (2,0,1))

    # Plot actual vs predicted values
    # The test_data.index gives you the correct timestamps for the x-axis
    axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
    axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')

    # Set title for each subplot
    axes[i].set_title(f'Training from {train_start.strftime("%Y-%m-%d")} to {train_end.strftime("%Y-%m-%d")} - Stuttgart', fontsize=10)

    # Add legend
    axes[i].legend()

# Improve plot layout
plt.tight_layout()
plt.show()
```

Q8:

```python
# Now, use the function in your given context
train_end = pd.Timestamp('2017-12-7 00:00:00')
test_end = pd.Timestamp('2017-12-14 00:00:00')

forecast_hours = [4, 8, 12, 16, 20, 24]

# Start a new plot
plt.figure(figsize=(14, 7))

# Plot the training data only once
train_data = df_completed.loc[df_completed.index < train_end, 'count']
plt.plot(train_data, label='Training Data')

final_metrics = []

# Loop through each forecast horizon and plot the forecasts
for h in forecast_hours:
    # Define the start and end of the test_data for each forecast
    test_data_start = train_end + pd.Timedelta(hours=h)
    test_data_end = test_end + pd.Timedelta(hours=h)
    test_data = df_completed.loc[(df_completed.index >= test_data_start) & (df_completed.index < test_data_end), 'count']

    # Call the predict_arima function to make predictions
    predictions, metrics = predict_arima(train_data, test_data, (2, 0, 3))

    # Create a pandas Series for the forecasted values with the correct index
    forecast_index = test_data.index
    forecast_series = pd.Series(predictions, index=forecast_index)

    # Plot the forecast
    plt.plot(forecast_series, label=f'{h}-hour Forecast', linestyle='--')
```

```
    forecast_series = pd.Series(predictions, index=forecast_index)

    # Plot the forecast
    plt.plot(forecast_series, label=f'{h}-hour Forecast', linestyle='--')

    final_metrics.append(f"metrics for {h} hour forecast: {metrics} in Stuttgart")

# Plot the actual test data
actual_test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']
plt.plot(actual_test_data, label='Test Data', color='orange', linestyle='-')

# Finalize the plot
plt.legend()
plt.title('Hourly Forecast vs Actuals Stuttgart')
plt.xlabel('Date')
plt.ylabel('Count')
plt.show()

for i in range(len(final_metrics)):
    print(final_metrics[i], "\n")
```