



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 D

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.

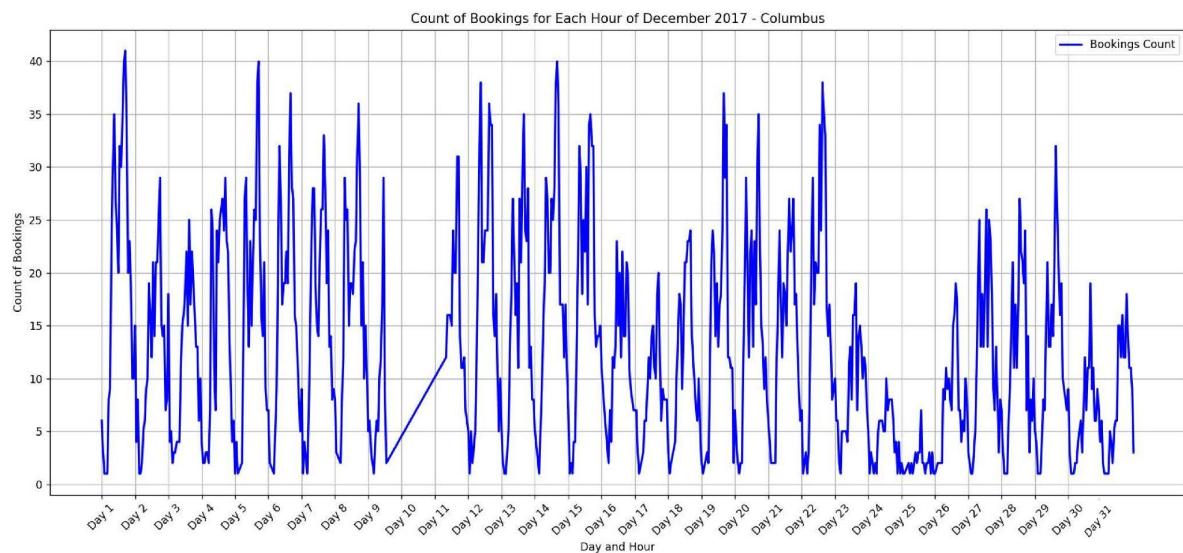


Fig 1.1: Columbus daily and hourly booking analysis

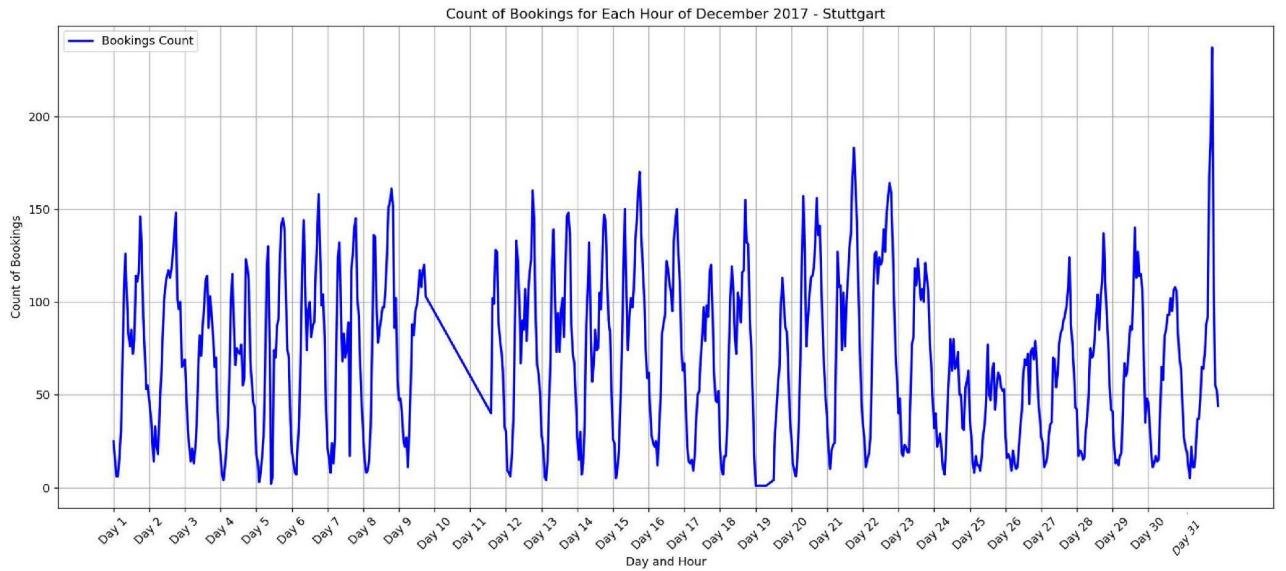


Fig 1.1: Stuttgart daily and hourly booking analysis

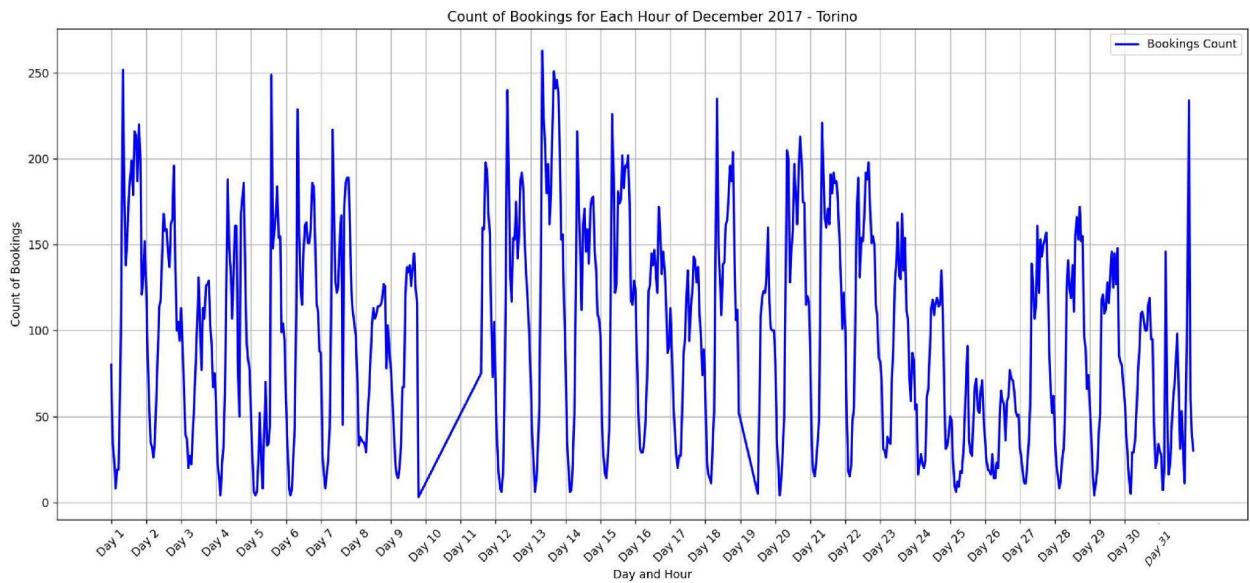


Fig 1.3: Torino daily and hourly booking analysis

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.

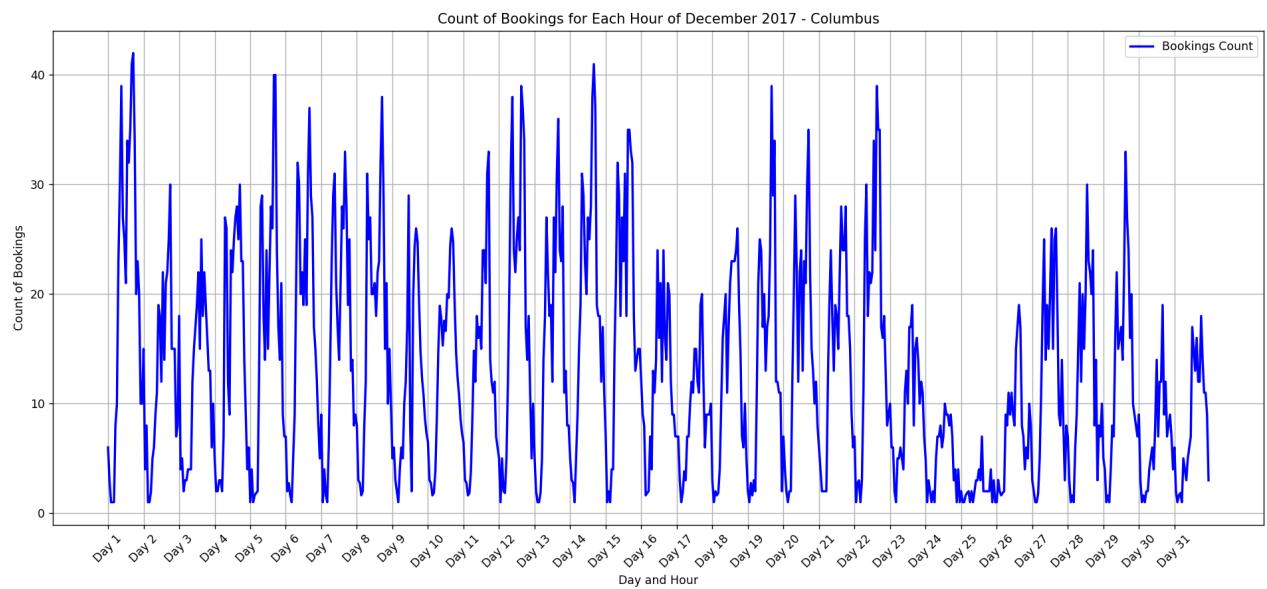


Fig 2.1: rectified missing samples for Columbus

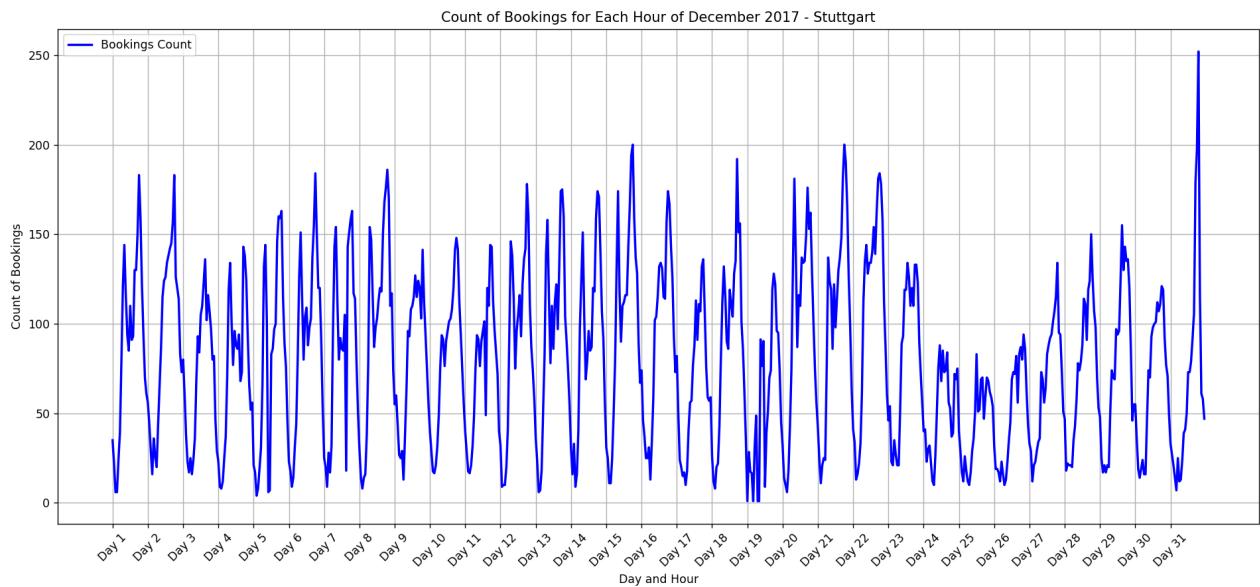


Fig 2.2: rectified missing samples for Stuttgart

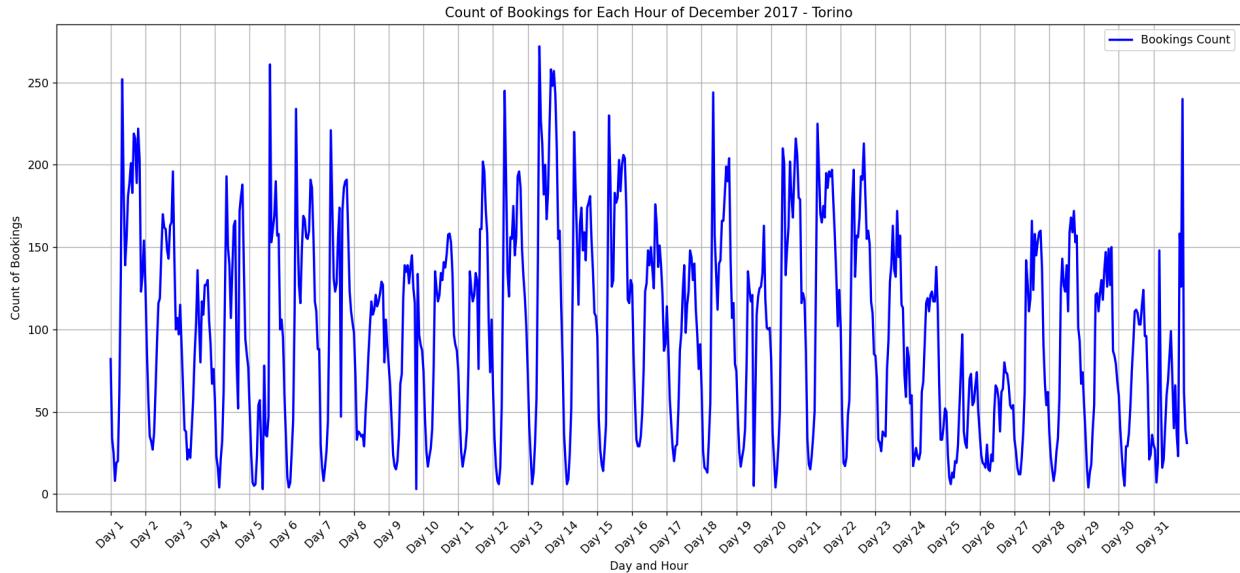


Fig 2.3: rectified missing samples for Torino

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`.

?

is this fundamental?

?

 **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)

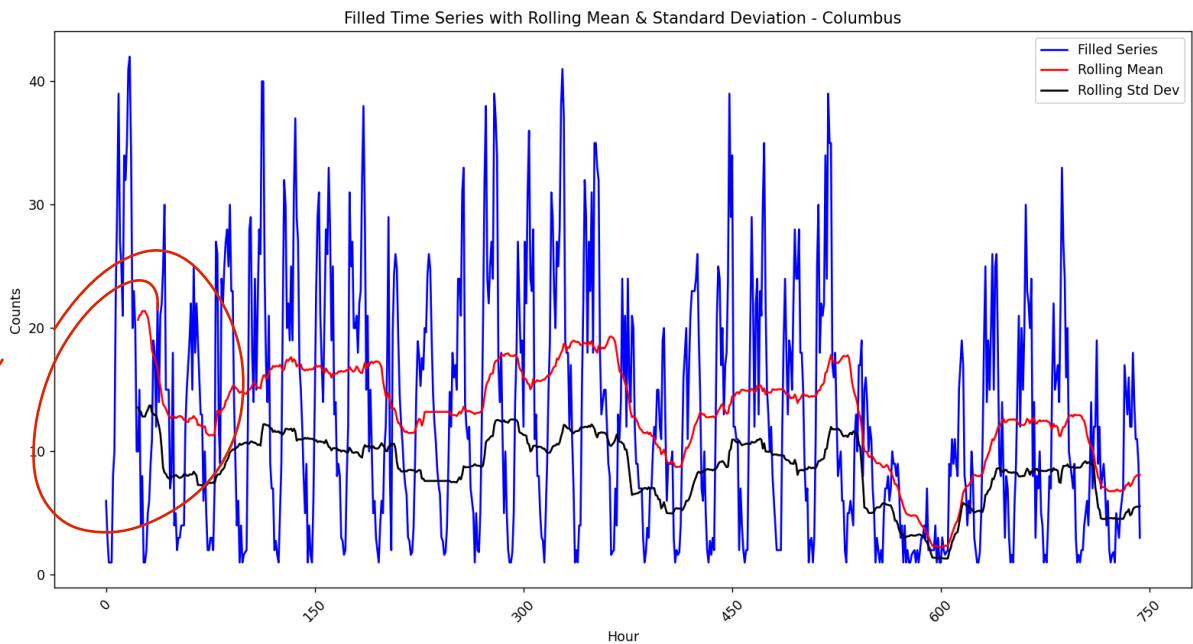


Fig 3.1: Time series for Columbus

```
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: Columbus data info

 What is this? 

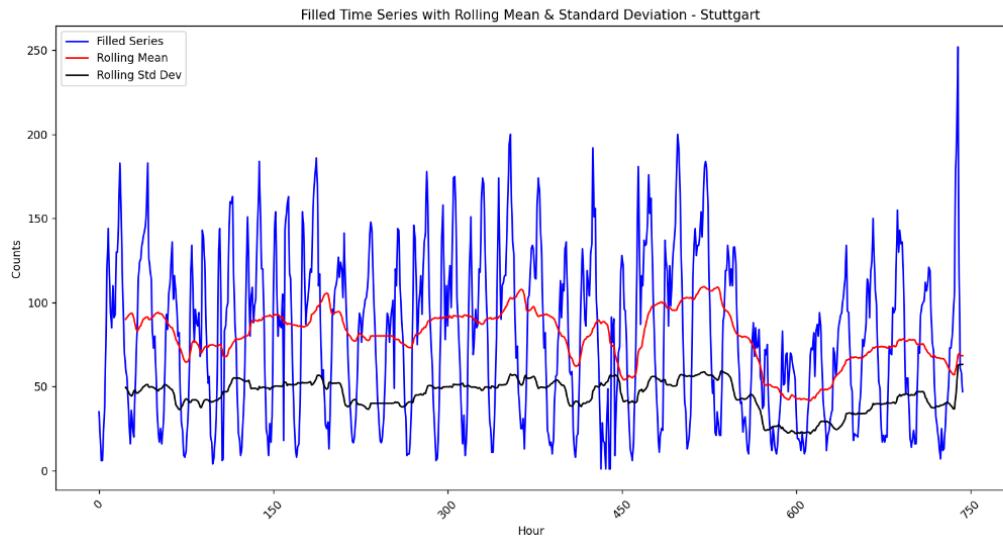


Fig 3.3: Time series for Stuttgart

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

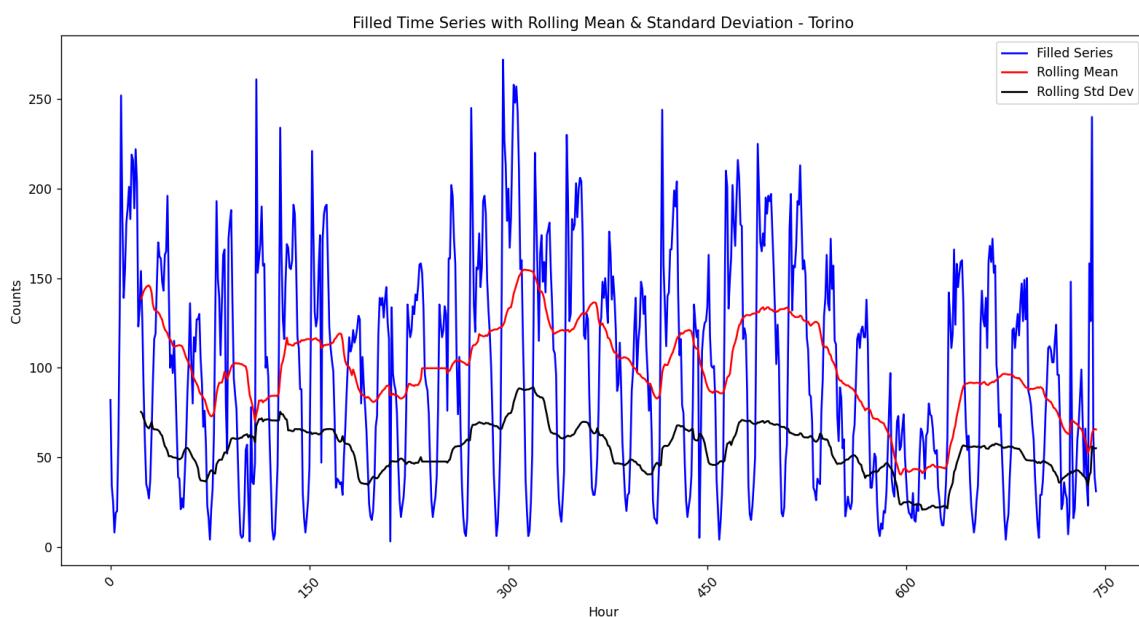


Fig 3.5: Time series for Torino

```

ADF Statistic: -3.295020614632056
p-value: 0.015102525661735314
1%, -3.4394269973845657
5%, -2.8655458544300387
10%, -2.568903174551249
The filled time series is stationary.

```

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.

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.

*how can you say
so?*

- 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?

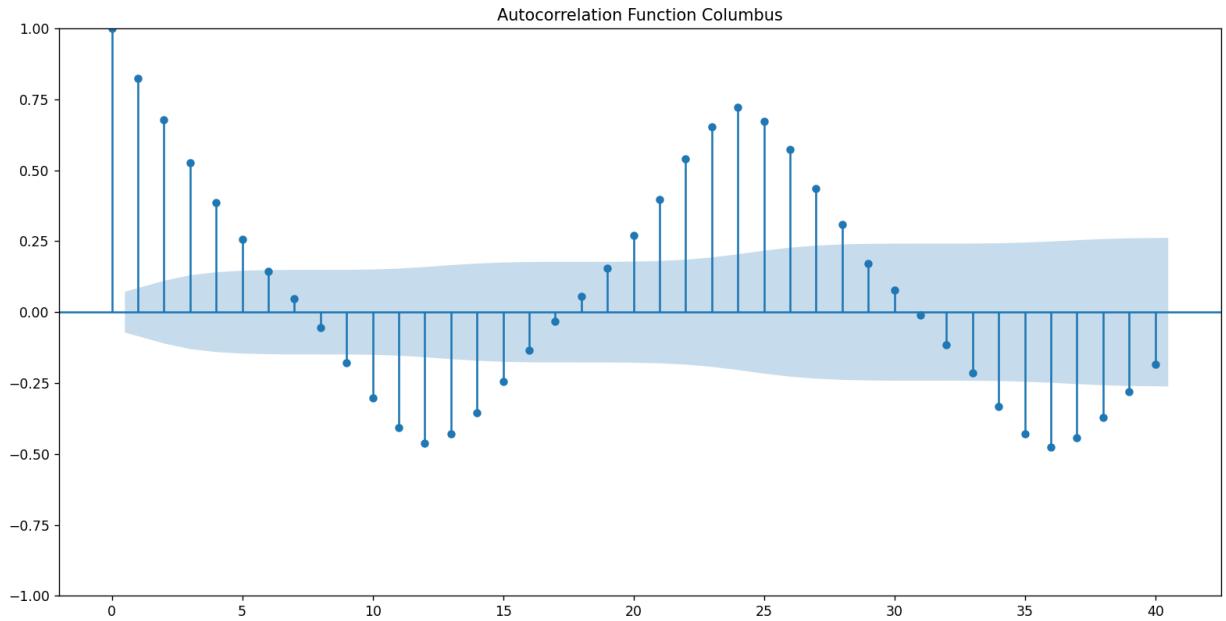


Fig 4.1: ACF Columbus

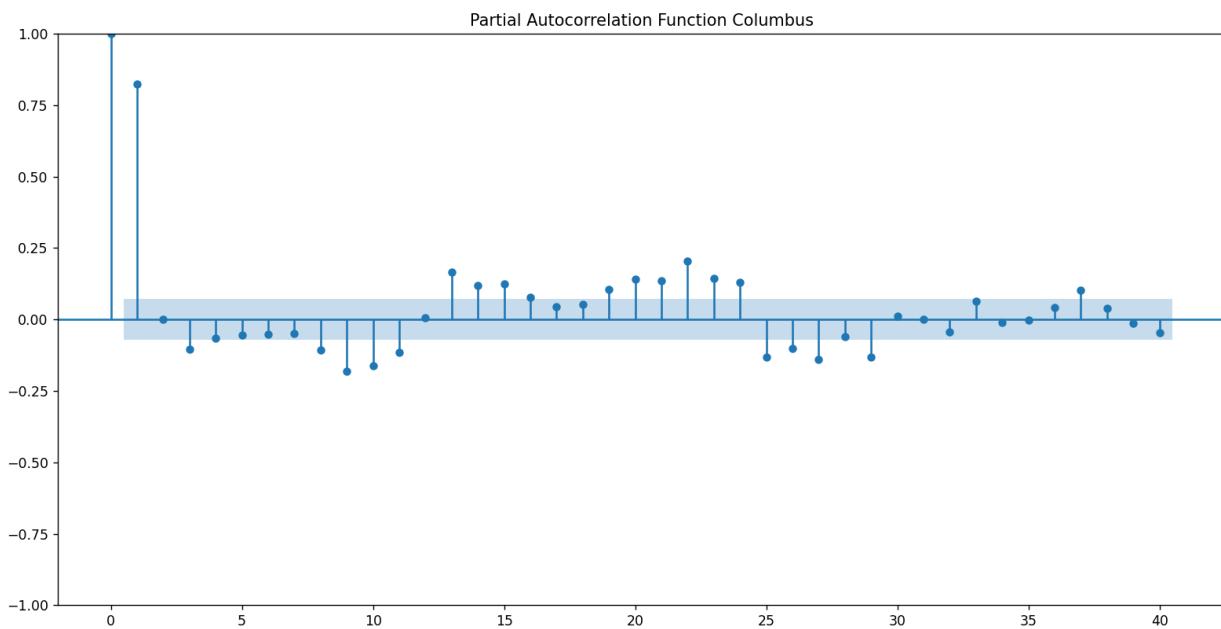


Fig 4.2: PACF Columbus

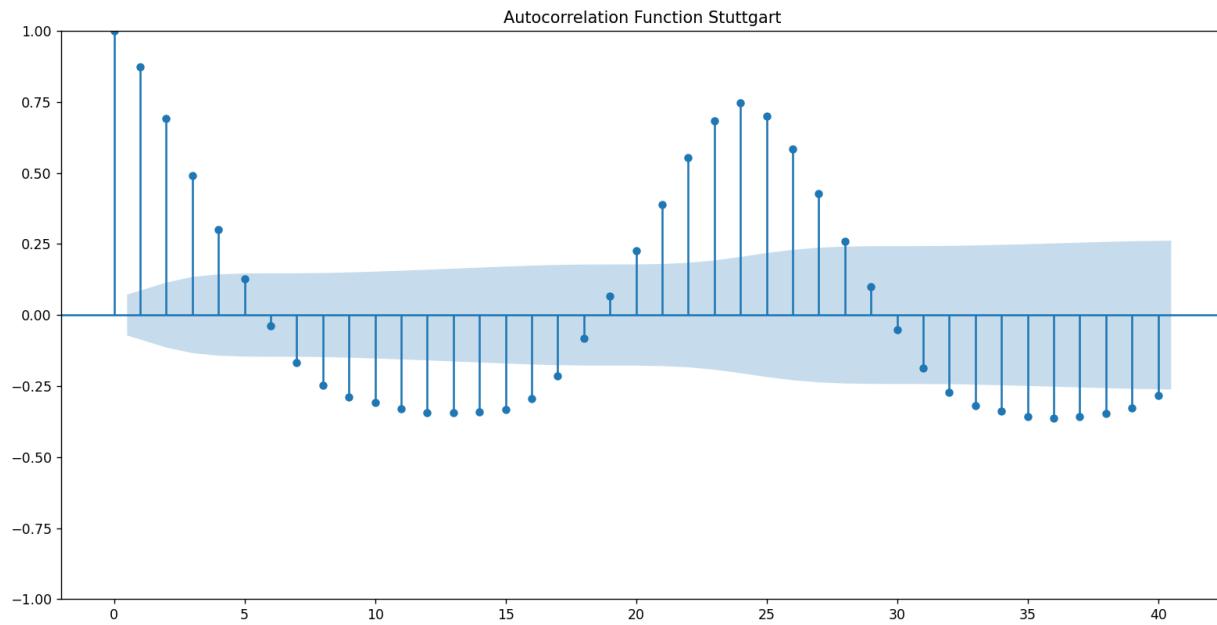


Fig 4.3: ACF Stuttgart

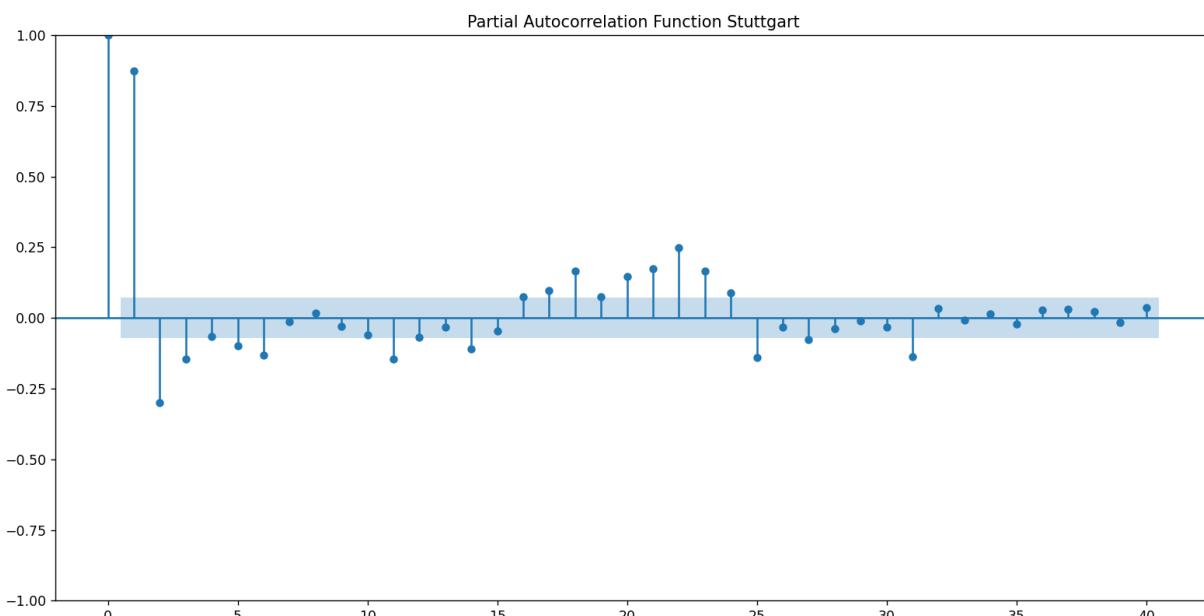


Fig 4.4: PACF Stuttgart

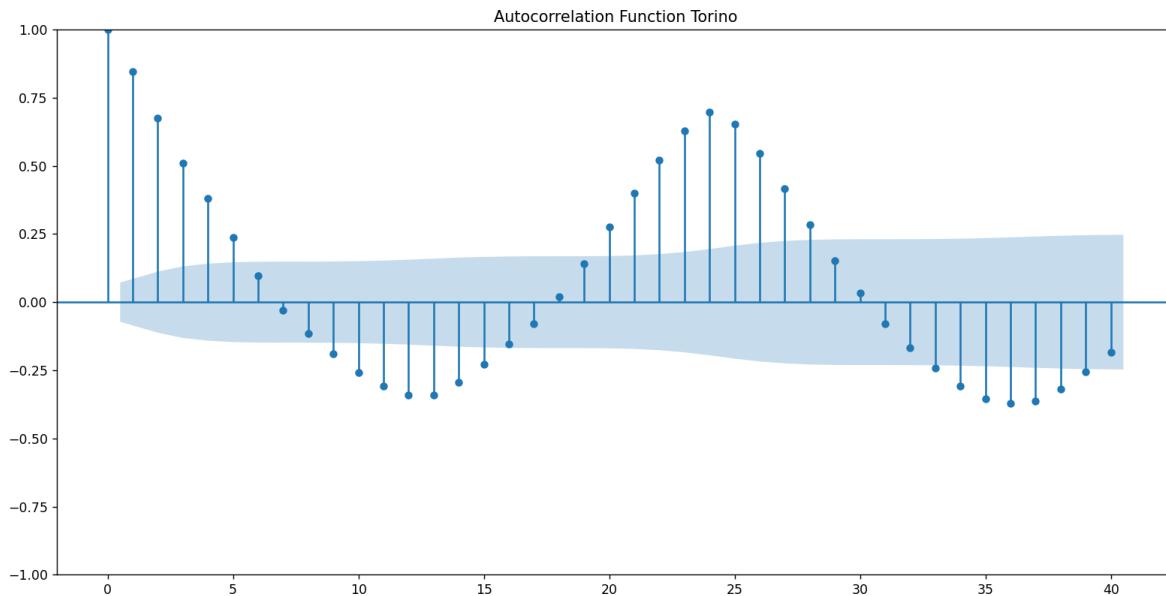


Fig 4.5: ACF Torino

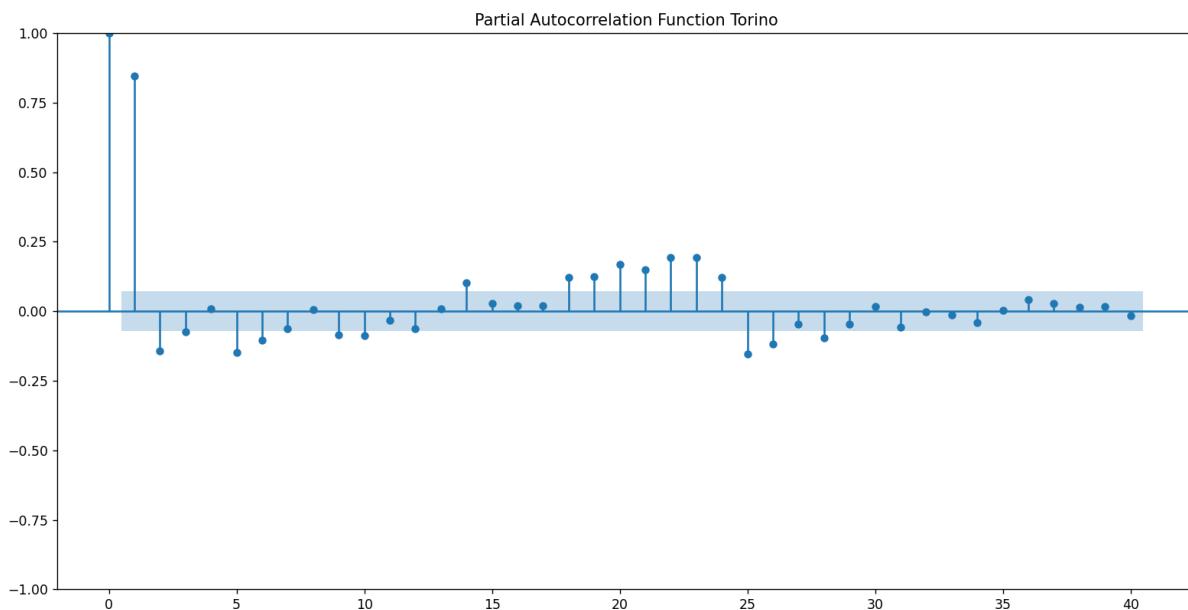


Fig 4.6: PACF Torino

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

which assumption
allow you to claim this!

the
opposite

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).

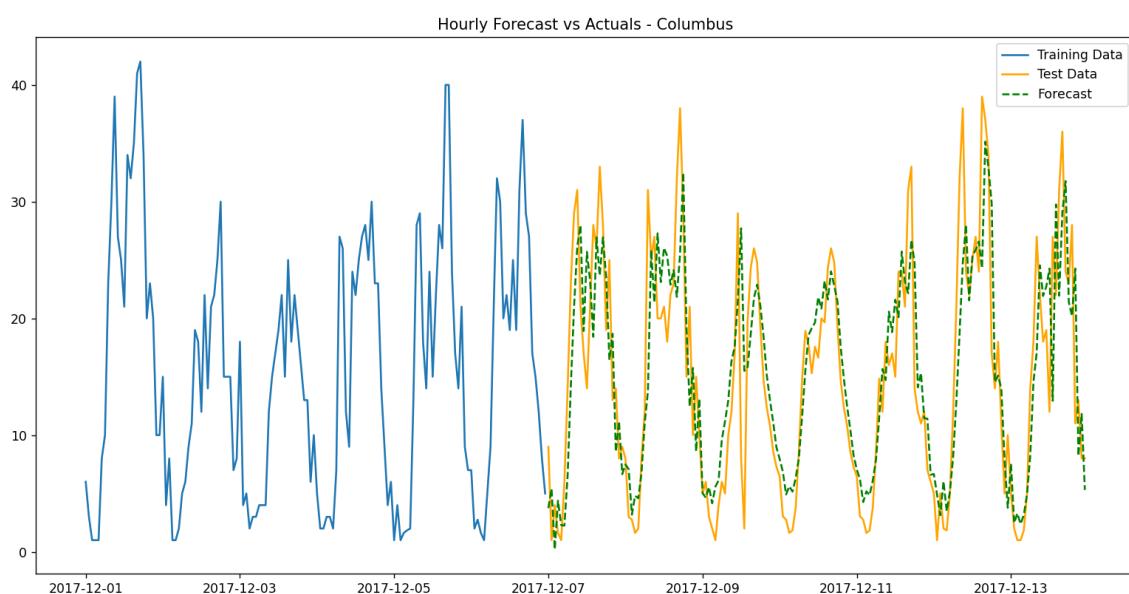


Fig 6.1: hourly forecast for Columbus



```
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: Columbus Information

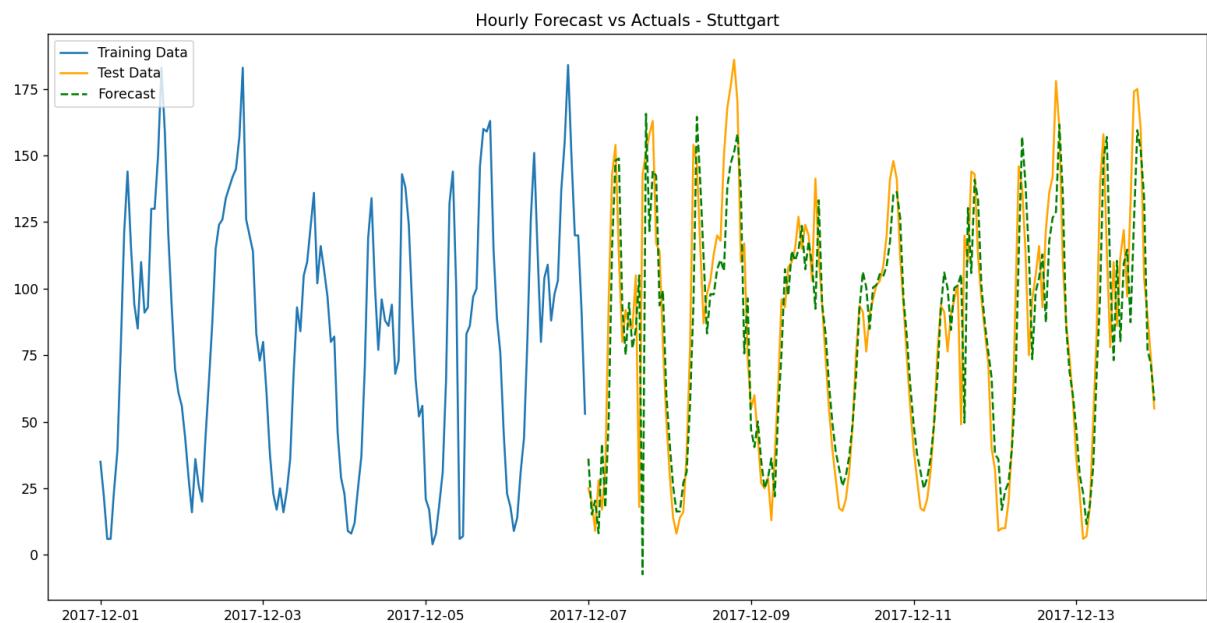


Fig 6.3: hourly forecast for Stuttgart

```
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

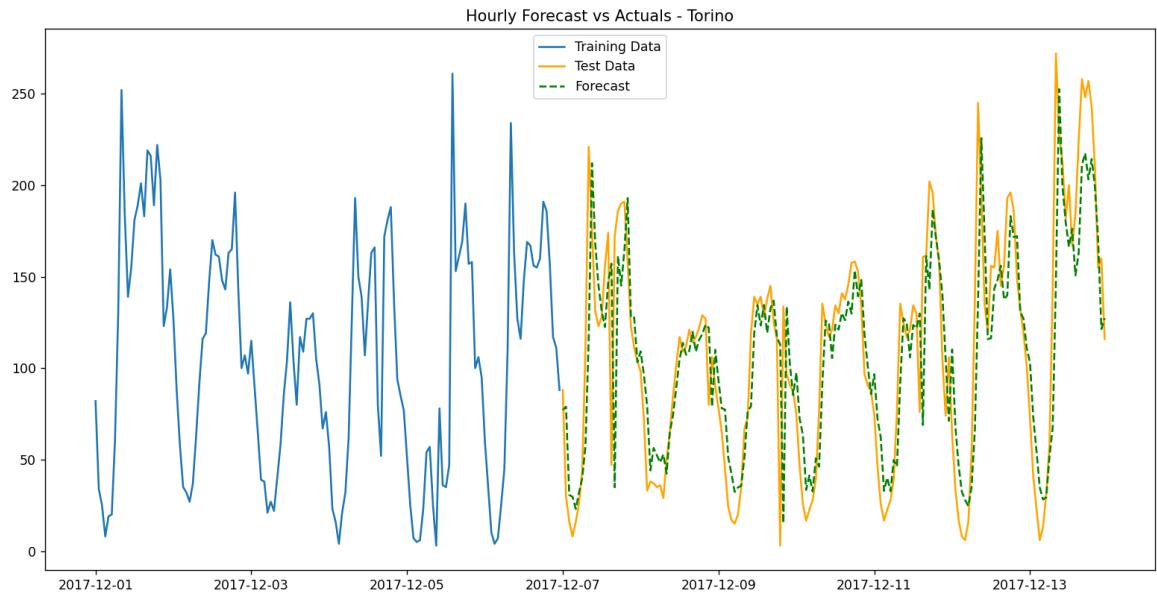


Fig 6.5: hourly forecast for Torino

```

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.

how (?)

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?

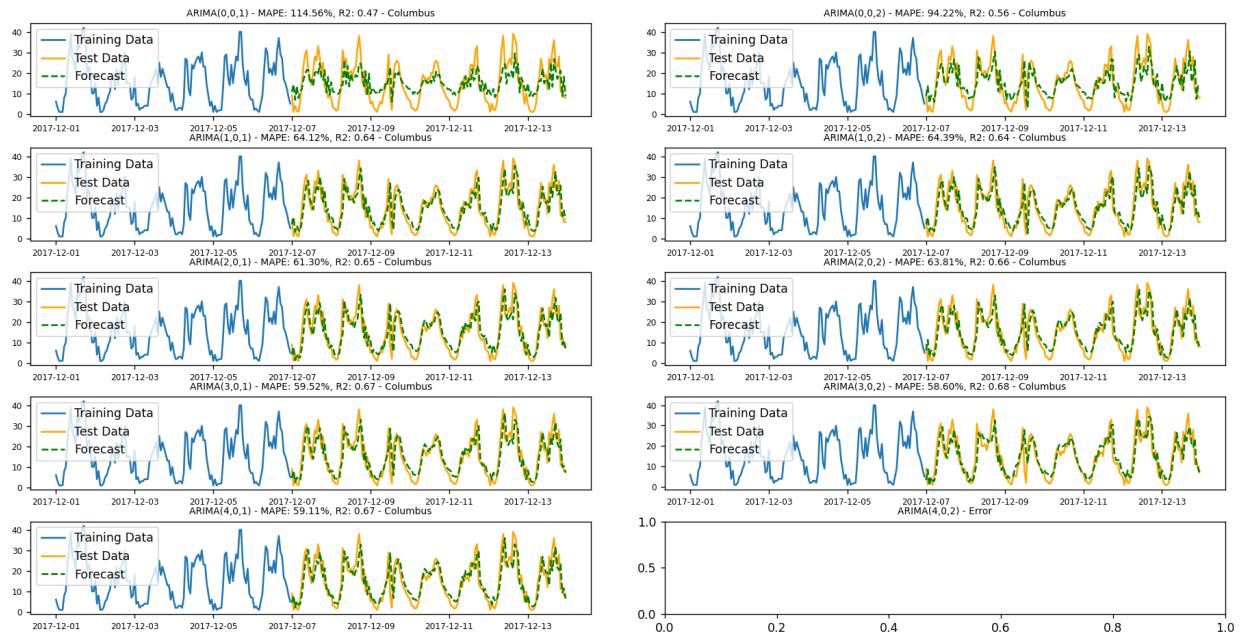


Fig 7.1: Columbus error changes

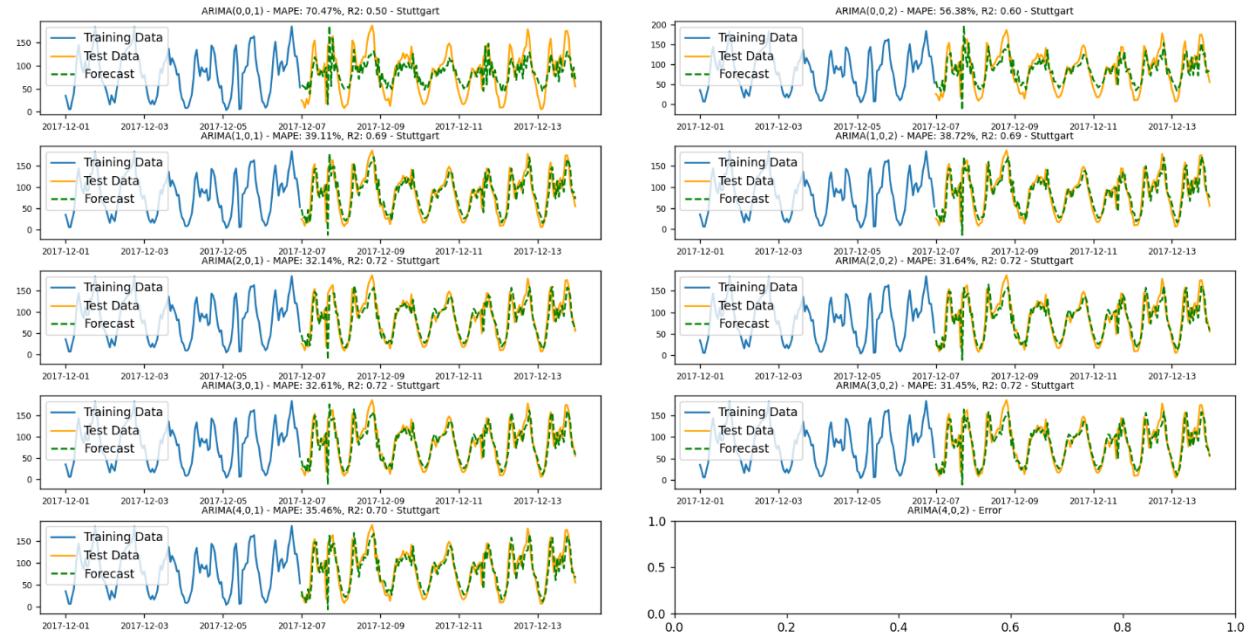


Fig 7.2: Stuttgart error changes

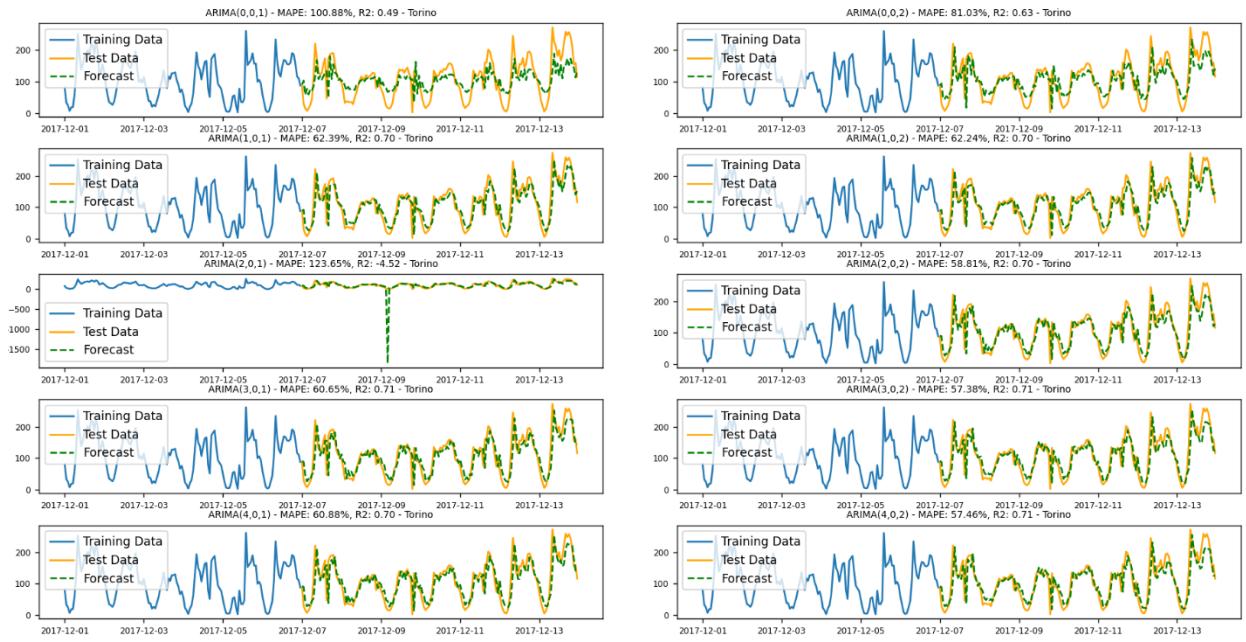


Fig 7.3: Torino error changes

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

↳ where on the result (?)

cut &
paste
from
chatGPT
(?)

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.

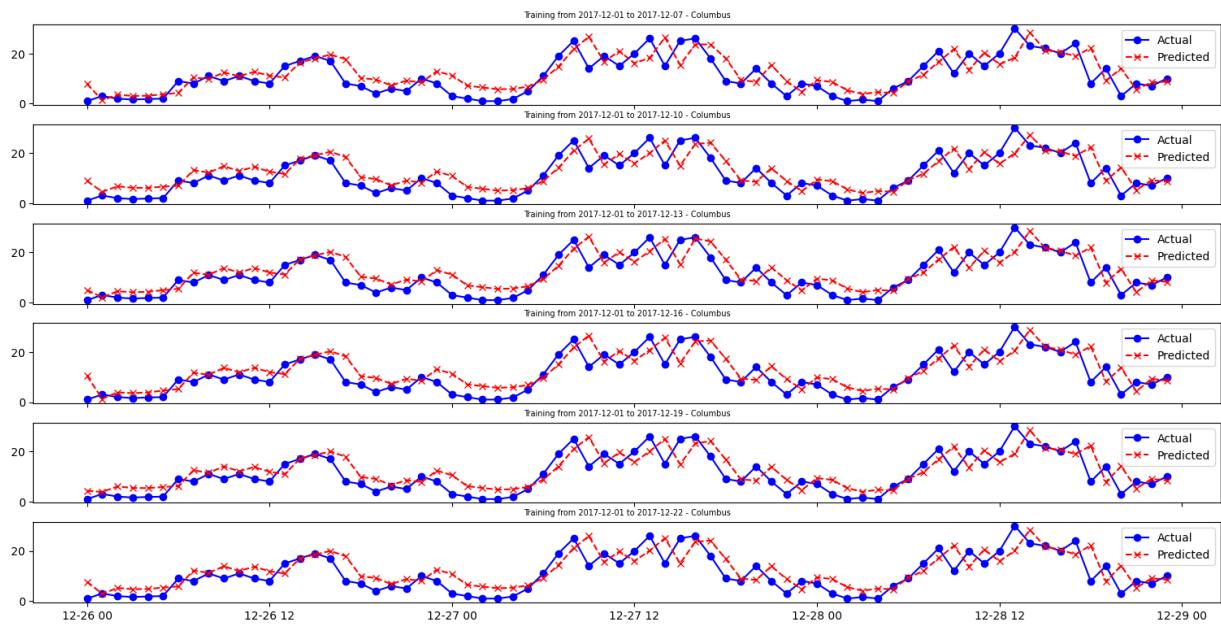


Fig 7.4: Columbus Expanding

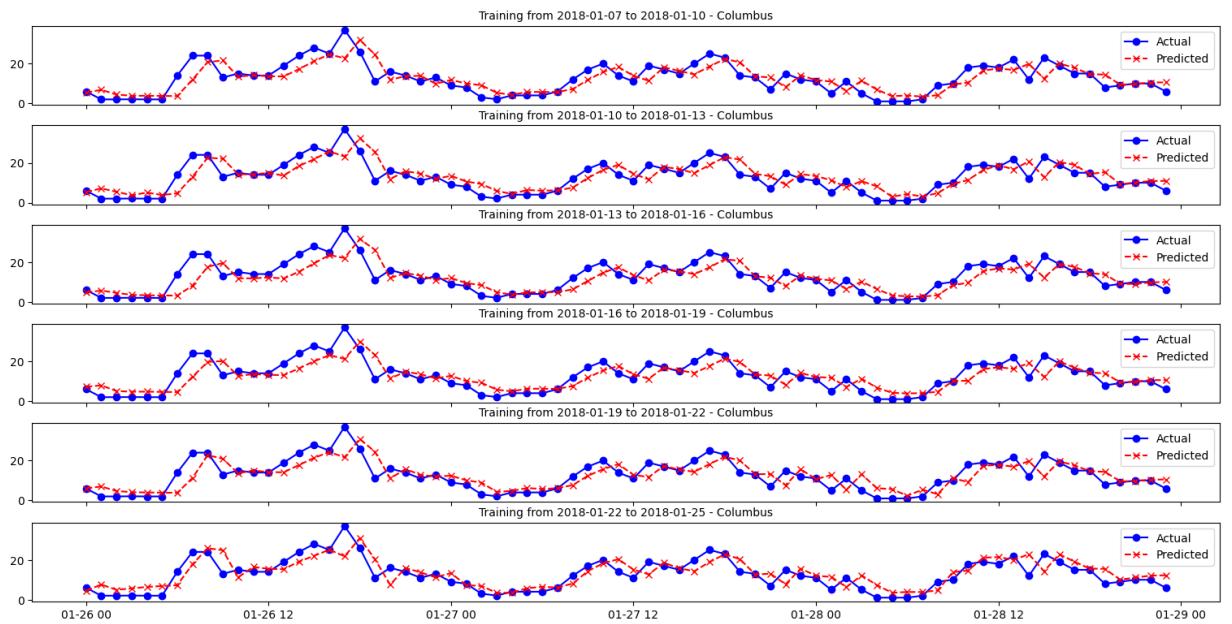


Fig 7.5: Columbus sliding

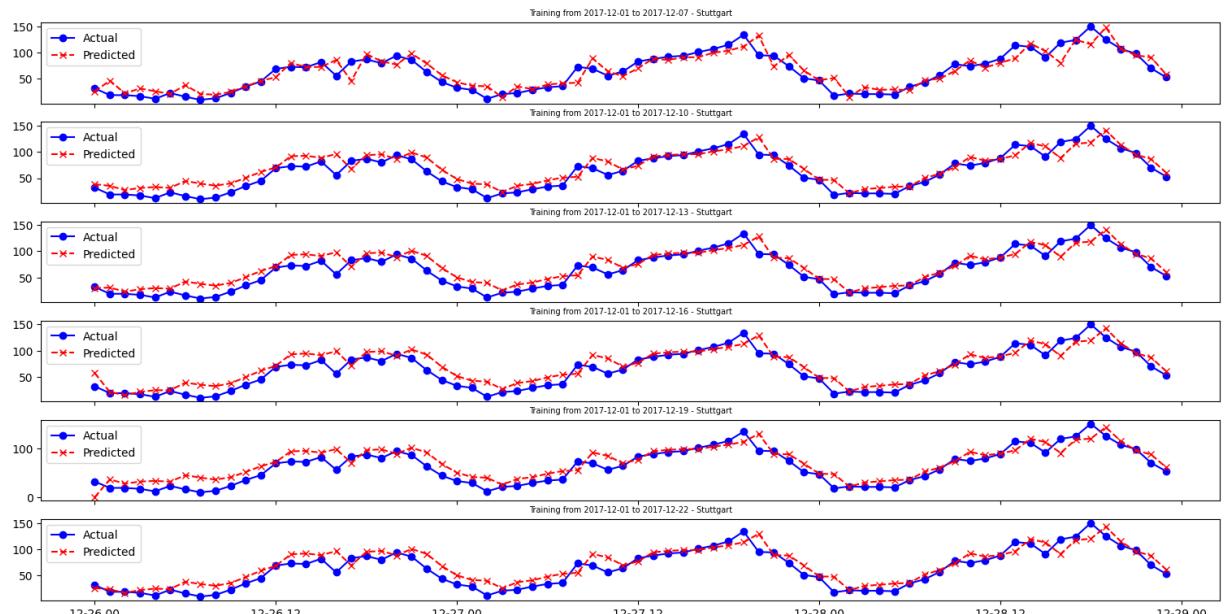


Fig 7.6: Stuttgart Expanding

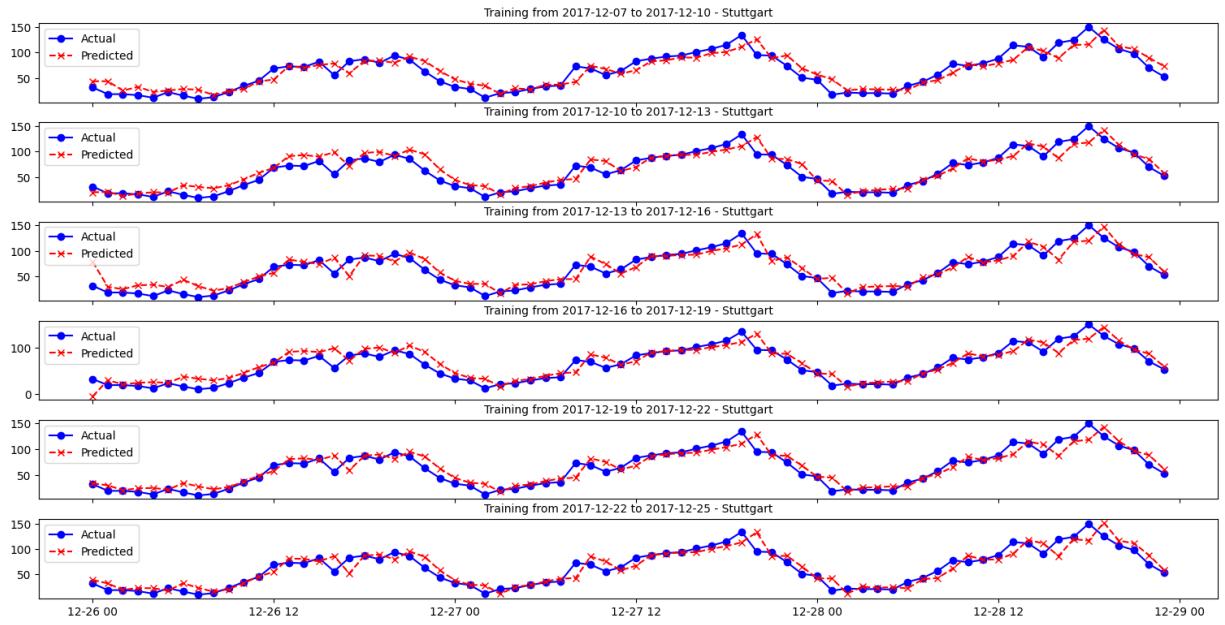


Fig 7.7: Stuttgart sliding

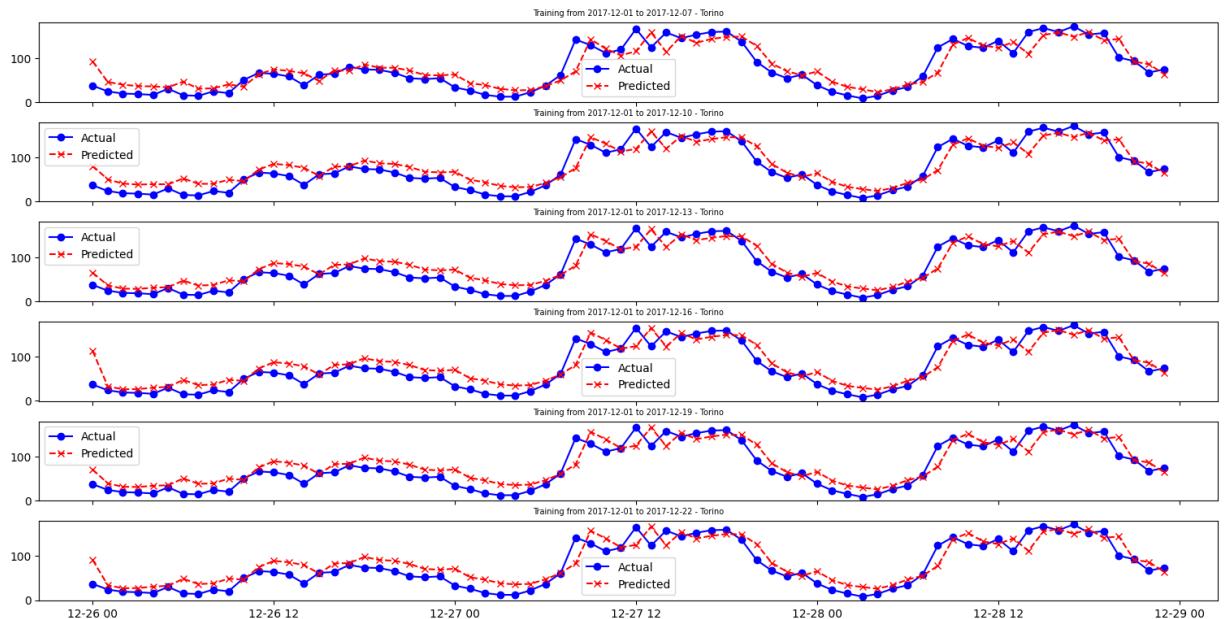


Fig 7.8: Torino expanding

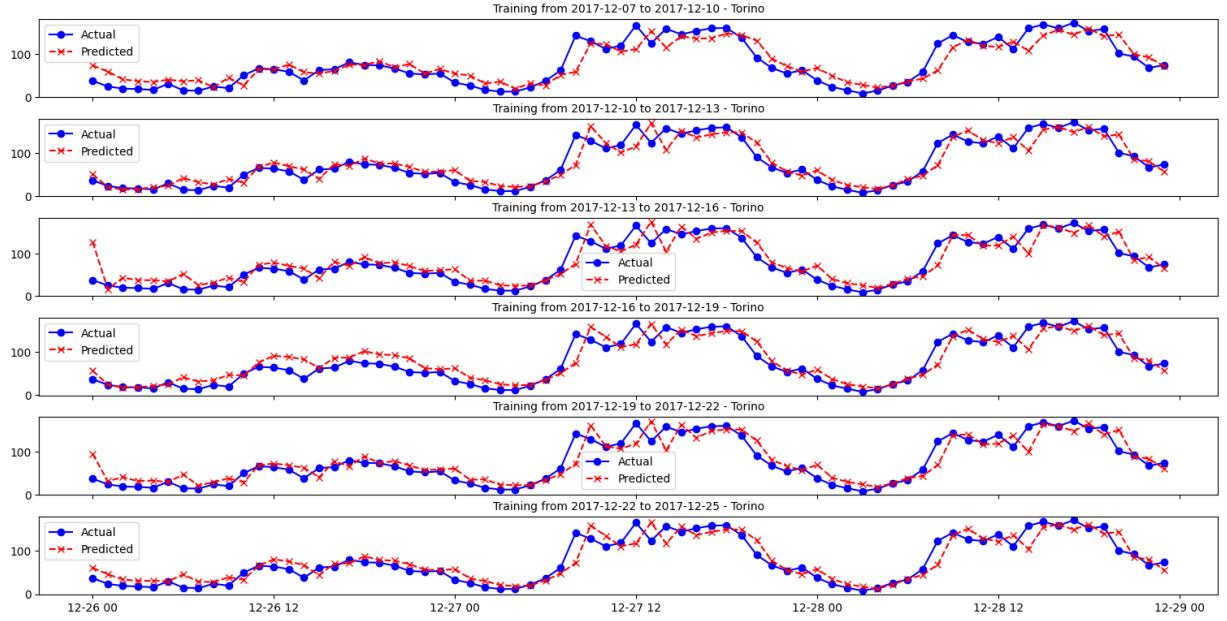


Fig 7.9: Torino sliding

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?

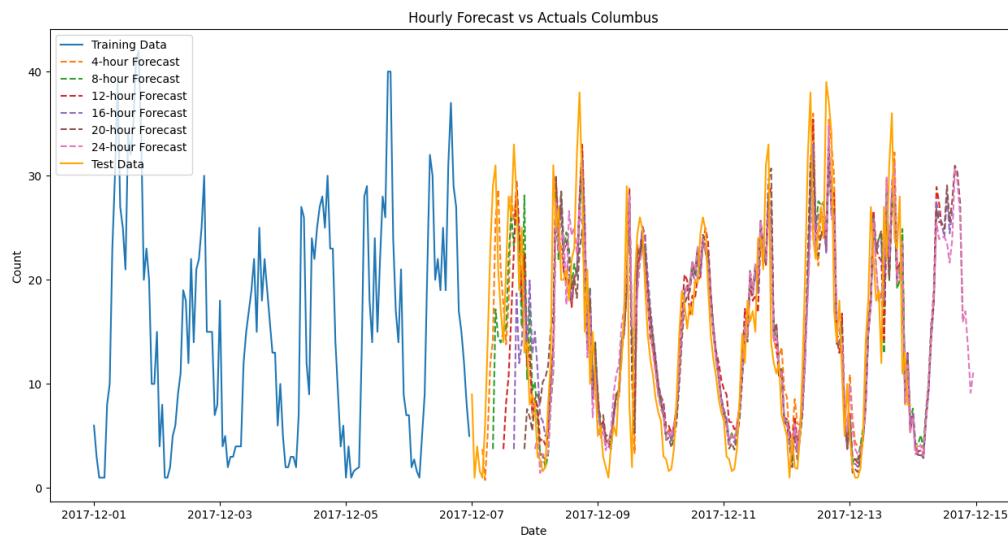


Fig 8.1: Hourly forecast vs actual for Columbus

```
metrics for 4 hour forecast: {'MAPE': 56.07685317653245, '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 Columbus

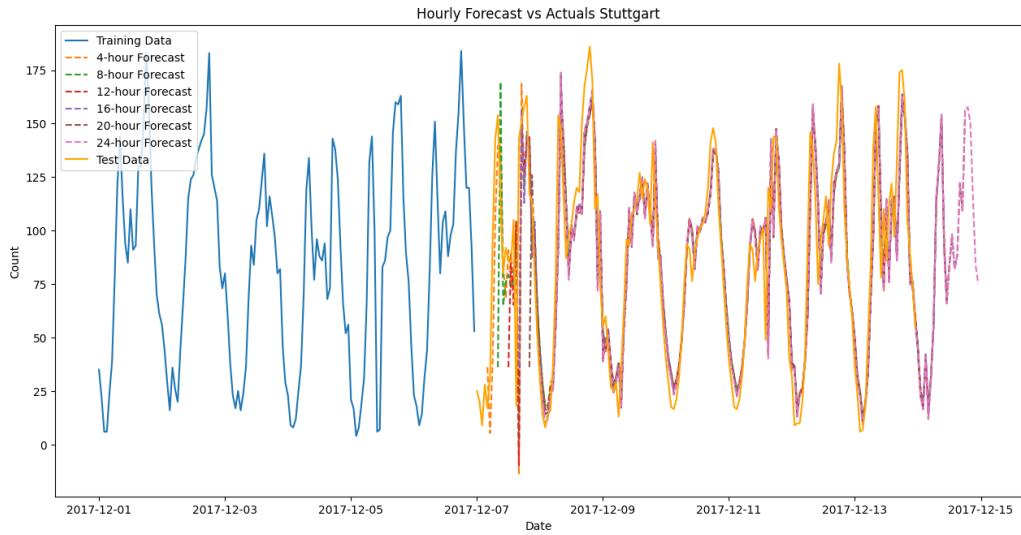


Fig 8.3: Hourly forecast vs actual for Stuttgart

```

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

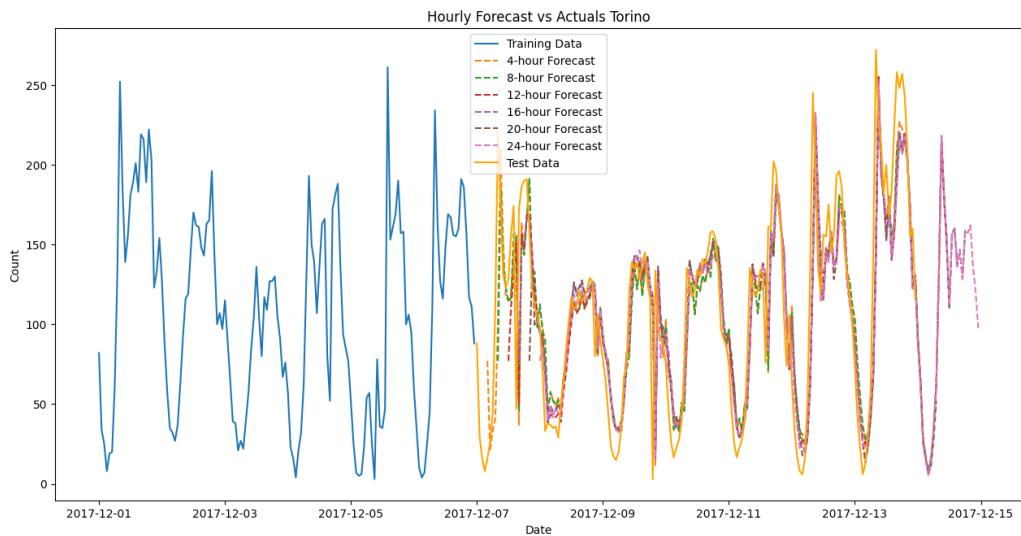


Fig 8.5: Hourly forecast vs actual for Turin

```

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 R²: The R² values for the 24-hour prediction period were the highest among all forecast intervals. A high R² 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 R² values and higher MSE.

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

REDO

Appendices:

Q1:

```
1  from datetime import datetime
2  import numpy as np
3  import pandas as pd
4  from pymongo import MongoClient
5  import matplotlib.pyplot as plt
6
7
8  client = MongoClient("mongodb://ictts:Ict4SM22!@bigdataadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
9
10 # Get the database
11 db = client['carsharing']
12
13 # Define the start and end dates for the query
14 start_date = datetime(2017, 12, 1)
15 end_date = datetime(2018, 1, 1)
16
17 # Define the query conditions
18 query_conditions = {
19     '$and': [
20         {'init_date': {'$gte': start_date}},
21         {'init_date': {'$lt': end_date}},
22         {'$expr': {'$lt': ['$subtract': ['$final_time', '$init_time'], 8000]}},
23         {'$expr': {'$ne': ['$init_address', '$final_address']}}
24     ]
25 }
```

```

27  # Aggregation pipeline
28  pipeline = [
29      [
30          {
31              '$match': query_conditions,
32              "city": "Torino"
33          },
34          [
35              {
36                  '$group': {
37                      '_id': {
38                          'day': {'$dayOfMonth': '$init_date'},
39                          'hour': {'$hour': '$init_date'}
40                      },
41                      'count': {'$sum': 1}
42                  }
43              },
44              {
45                  '$sort': {
46                      '_id.day': 1,
47                      '_id.hour': 1
48                  }
49              }
50          ]
51      ],
52      [
53          {
54              '$sort': {
55                  '_id.day': 1,
56                  '_id.hour': 1
57              }
58          }
59      ]
60  ]
61
62  # Execute the query to retrieve the records
63  records = db["PermanentBookings"].aggregate(pipeline)
64
65  # Prepare the data for plotting
66  hours = []
67  counts = []
68  for record in records:
69      # Convert day and hour to a single hour number for the entire month
70      day_hour = (record['_id'][['day']] - 1) * 24 + record['_id'][['hour']]
71      hours.append(day_hour)
72      counts.append(record[['count']])
73
74  # Prepare the data for plotting
75  hours = []
76  counts = []
77  for record in records:
78      # Convert day and hour to a single hour number for the entire month
79      day_hour = (record['_id'][['day']] - 1) * 24 + record['_id'][['hour']]
80      hours.append(day_hour)
81      counts.append(record[['count']])
82
83  # Plotting
84  plt.figure(figsize=(15, 6))
85
86  # Connect the points with lines
87  plt.plot(hours, counts, color='blue', linestyle='-', linewidth=2, marker='', label='Bookings Count')
88
89  # Label formatting to show day and hour
90  tick_positions = range(0, 24 * 30, 24) # one tick per day
91  tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
92  plt.xticks(tick_positions, tick_labels, rotation=45) # Set x-ticks to be every day and rotate for readability
93
94  plt.xlabel('Day and Hour')
95  plt.ylabel('Count of Bookings')
96  plt.title('Count of Bookings for Each Hour of November 2017')
97  plt.grid(True) # Add grid for better readability
98  plt.legend()
99
100 # Show the plot
101 plt.tight_layout() # Adjust layout so everything fits without overlapping
102 plt.show()

```

Q2:

```
D:\> class files > 3.semester > mobility > mella > lab 2 > lab2 - docs > 2 > 2.py
 1  from datetime import datetime, timedelta
 2  from collections import defaultdict
 3  import numpy as np
 4  import pandas as pd
 5  from pymongo import MongoClient
 6  import matplotlib.pyplot as plt
 7
 8  client = MongoClient("mongodb://ictts:Ict4SM22!@bigdataadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
 9
10 # Get the database
11 db = client['carsharing']
12
13 # Define the start and end dates for the query
14 start_date = datetime(2017, 12, 1)
15 end_date = datetime(2018, 1, 1)
16
17 # Define the query conditions
18 query_conditions = {
19     '$init_date': {'$gte': start_date, '$lt': end_date},
20     'city': "Torino",
21     '$expr': {'$lt': [{"$subtract": ['$final_time', '$init_time']}, 8000]},
22     '$expr': {'$ne': ['$init_address', '$final_address']}
23 }
24
25 # Aggregation pipeline
26 pipeline = [
27     {
28         '$match': query_conditions
29     },
30     {
31         '$group': {
32             '_id': {
33                 'day': {'$dayOfMonth': '$init_date'},
34                 'hour': {'$hour': '$init_date'}
35             },
36             'count': {'$sum': 1}
37         }
38     },
39     {
40         '$sort': {
41             '_id.day': 1,
42             '_id.hour': 1
43         }
44     }
45 ]
46
47 # Execute the query to retrieve the records
48 records = list(db["PermanentBookings"].aggregate(pipeline))
49
50 # Create a dictionary to hold the counts for each day and hour
51 counts_dict = defaultdict(lambda: defaultdict(int))
```

```

53     # Populate the dictionary with the query results
54     for record in records:
55         day = record['_id']['day']
56         hour = record['_id']['hour']
57         count = record['count']
58         counts_dict[day][hour] = count
59
60     # Calculate the average count for each hour across all days
61     hourly_averages = defaultdict(list)
62     for day, hours in counts_dict.items():
63         for hour, count in hours.items():
64             hourly_averages[hour].append(count)
65
66     # Compute averages
67     hourly_averages = {hour: np.mean(counts) for hour, counts in hourly_averages.items()}
68
69     # Fill in missing hours for each day with the average counts
70     completed_counts = []
71     for day in range(1, 32): # Assuming December has 31 days
72         for hour in range(24):
73             if hour in counts_dict[day]:
74                 completed_counts.append(((day - 1) * 24 + hour, counts_dict[day][hour]))
75             else:
76                 completed_counts.append(((day - 1) * 24 + hour, hourly_averages[hour]))
77
78     # Unzip the completed_counts for plotting
79     hours, counts = zip(*completed_counts)

70     completed_counts = []
71     for day in range(1, 32): # Assuming December has 31 days
72         for hour in range(24):
73             if hour in counts_dict[day]:
74                 completed_counts.append(((day - 1) * 24 + hour, hourly_averages[dict])) # (variable) hourly_averages: dict
75             else:
76                 completed_counts.append(((day - 1) * 24 + hour, hourly_averages[hour]))
77
78     # Unzip the completed_counts for plotting
79     hours, counts = zip(*completed_counts)
80
81     # Plotting
82     plt.figure(figsize=(15, 6))
83     plt.plot(hours, counts, color='blue', linestyle='-', linewidth=2, marker='', label='Bookings Count')
84
85     # Label formatting to show day and hour
86     tick_positions = range(0, 24 * 31, 24) # One tick per day
87     tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
88     plt.xticks(tick_positions, tick_labels, rotation=45) # Set x-ticks to be every day and rotate for readability
89
90     plt.xlabel('Day and Hour')
91     plt.ylabel('Count of Bookings')
92     plt.title('Count of Bookings for Each Hour of December 2017')
93     plt.grid(True) # Add grid for better readability
94     plt.legend()
95
96     # Show the plot
97     plt.tight_layout() # Adjust layout so everything fits without overlapping
98     plt.show()

```

Q3:

```
 1  from datetime import datetime
 2  from collections import defaultdict
 3  import numpy as np
 4  import pandas as pd
 5  from pymongo import MongoClient
 6  import matplotlib.pyplot as plt
 7  from statsmodels.tsa.stattools import adfuller
 8  import matplotlib.dates as mdates
 9
10 client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
11
12 # Get the database
13 db = client['carsharing']
14
15 # Define the start and end dates for the query
16 start_date = datetime(2017, 12, 1)
17 end_date = datetime(2018, 1, 1)
18
19 # Define the query conditions
20 query_conditions = {
21     'init_date': {'$gte': start_date, '$lt': end_date},
22     'city': "Torino",
23     '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
24     '$expr': {'$ne': ['$init_address', '$final_address']}
25 }
26
27 # Aggregation pipeline
28 pipeline = [
29     {
30         '$match': query_conditions
31     }
```

```

61     if day not in counts_dict:
62         counts_dict[day] = {} # Create a new dictionary for the day if it doesn't exist
63
64     # Assuming that there will be only one count per day and hour,
65     # otherwise, you need logic to sum or handle multiple counts
66     counts_dict[day][hour] = count
67
68
69     # Initialize an empty dictionary to hold lists of counts for each hour
70     hourly_averages = {}
71
72     # Populate the dictionary with counts for each hour
73     for day, hours in counts_dict.items():
74         for hour, count in hours.items():
75             if hour not in hourly_averages:
76                 hourly_averages[hour] = [] # Initialize a new list if the hour key doesn't exist
77                 hourly_averages[hour].append(count)
78
79     # Compute averages using numpy for mean calculation
80     hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
81     # Fill in missing hours for each day with the average counts
82     completed_counts = []
83     for day in range(1, 32): # Assuming December has 31 days
84
85         for hour in range(24):
86
87
88         hours_dict = counts_dict.get(day, {})
89
90         # Use get to provide the hourly average as a default if the hour is not in hours_dict
91         count = hours_dict.get(hour, hourly_averages.get(hour, 0))
92
93     [
94         {
95             '$match': query_conditions
96         },
97         {
98             '$group': {
99                 '_id': {
100                     'day': {'$dayOfMonth': '$init_date'},
101                     'hour': {'$hour': '$init_date'}
102                 },
103                 'count': {'$sum': 1}
104             }
105         },
106         {
107             '$sort': {
108                 '_id.day': 1,
109                 '_id.hour': 1
110             }
111         }
112     ]
113
114     # Execute the query to retrieve the records
115     records = list(db["PermanentBookings"].aggregate(pipeline))
116
117     # Create a standard dictionary to hold the counts for each day and hour
118     counts_dict = {}
119
120     # Populate the dictionary with the query results
121     for record in records:
122         day = record['_id']['day']
123         hour = record['_id']['hour']
124         count = record['count']
125

```

```

89     hours_dict = counts_dict.get(day, {})
90
91     # Use get to provide the hourly average as a default if the hour is not in hours_dict
92     count = hours_dict.get(hour, hourly_averages.get(hour, 0))
93
94     completed_counts.append(((day - 1) * 24 + hour, count))
95
96     # Convert completed_counts into a DataFrame
97     df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])
98
99     # Convert 'hours_since_start' to a datetime index starting from 'start_date'
100    df_completed['datetime'] = pd.to_datetime(df_completed['hours_since_start'], unit='h', origin=start_date)
101
102    # Perform the ADF test
103    adf_test_filled = adfuller(df_completed['count'])
104
105    # Output the ADF statistic and p-value
106    print(f"ADF Statistic: {adf_test_filled[0]}")
107    print(f"p-value: {adf_test_filled[1]}")
108    for key, value in adf_test_filled[4].items():
109        print(f"  {key}, {value}")
110
111    # Check the stationarity of the filled series
112    if adf_test_filled[1] < 0.05:
113        print("The filled time series is stationary.")
114    else:
115        print("The filled time series is not stationary.")
116
117    # Convert 'count' to a series for rolling operations
118    counts_series_filled = df_completed['count']
119
120    # Check the stationarity of the filled series
121    if adf_test_filled[1] < 0.05:
122        print("The filled time series is stationary.")
123    else:
124        print("The filled time series is not stationary.")
125
126    # Convert 'count' to a series for rolling operations
127    counts_series_filled = df_completed['count']
128
129    # Calculate the rolling mean and the rolling standard deviation
130    rolling_mean_filled = counts_series_filled.rolling(window=24).mean()
131    rolling_std_filled = counts_series_filled.rolling(window=24).std()
132
133    # Plot the filled time series with rolling statistics
134    fig, ax = plt.subplots(figsize=(14, 7))
135    ax.plot(counts_series_filled, color='blue', label='Filled Series')
136    ax.plot(rolling_mean_filled, color='red', label='Rolling Mean')
137    ax.plot(rolling_std_filled, color='black', label='Rolling Std Dev')
138
139    ax.set_title('Filled Time Series with Rolling Mean & Standard Deviation')
140    ax.set_xlabel('Hour')
141    ax.set_ylabel('Counts')
142
143    # To improve readability of the x-axis
144    plt.xticks(rotation=45)
145    ax.xaxis.set_major_locator(plt.MaxNLocator(8))
146
147    ax.legend()
148    plt.show()

```

Q4:

```

1  from datetime import datetime
2  from collections import defaultdict
3  import numpy as np
4  import pandas as pd
5  from pymongo import MongoClient
6  import matplotlib.pyplot as plt
7  from statsmodels.tsa.stattools import adfuller
8  import matplotlib.dates as mdates
9  from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
10
11 client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
12
13 # Get the database
14 db = client['carsharing']
15
16 # Define the start and end dates for the query
17 start_date = datetime(2017, 12, 1)
18 end_date = datetime(2018, 1, 1)
19
20 # Define the query conditions
21 query_conditions = {
22     '$init_date': {'$gte': start_date, '$lt': end_date},
23     'city': "Torino",
24     '$expr': {'$lt': [{'$subtract': ['$final_time', '$init_time']}, 8000]},
25     '$expr': {'$ne': ['$init_address', '$final_address']}
26 }
27
28 # Aggregation pipeline
29 pipeline = [
30     {
31         '$match': query_conditions

```

```

34         '$group': {
35             '_id': {
36                 'day': {'$dayOfMonth': '$init_date'},
37                 'hour': {'$hour': '$init_date'}
38             },
39             'count': {'$sum': 1}
40         }
41     },
42     {
43         '$sort': {
44             '_id.day': 1,
45             '_id.hour': 1
46         }
47     }
48 ]
49
50 # Execute the query to retrieve the records
51 records = list(db["PermanentBookings"].aggregate(pipeline))
52
53 # Create a standard dictionary to hold the counts for each day and hour
54 counts_dict = {}
55
56 # Populate the dictionary with the query results
57 for record in records:
58     day = record['_id']['day']
59     hour = record['_id']['hour']
60     count = record['count']
61
62     if day not in counts_dict:
63         counts_dict[day] = {} # Create a new dictionary for the day if it doesn't exist
64

```

```
62     if day not in counts_dict:
63         counts_dict[day] = {} # Create a new dictionary for the day if it doesn't exist
64
65     # Assuming that there will be only one count per day and hour,
66     # otherwise, you need logic to sum or handle multiple counts
67     counts_dict[day][hour] = count
68
69
70 # Initialize an empty dictionary to hold lists of counts for each hour
71 hourly_averages = {}
72
73 # Populate the dictionary with counts for each hour
74 for day, hours in counts_dict.items():
75     for hour, count in hours.items():
76         if hour not in hourly_averages:
77             hourly_averages[hour] = [] # Initialize a new list if the hour key doesn't exist
78             hourly_averages[hour].append(count)
79
80 # Compute averages using numpy for mean calculation
81 hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
82 # Fill in missing hours for each day with the average counts
83 completed_counts = []
84 for day in range(1, 32): # Assuming December has 31 days
85
86     for hour in range(24):
87
88         |
89
90         hours_dict = counts_dict.get(day, {})
91
92         # Use get to provide the hourly average as a default if the hour is not in hours_dict
93         count = hours_dict.get(hour, hourly_averages.get(hour, 0))
```

```
82 # Fill in missing hours for each day with the average counts
83 completed_counts = []
84 for day in range(1, 32): # Assuming December has 31 days
85
86     for hour in range(24):
87
88         |
89
90         hours_dict = counts_dict.get(day, {})
91
92         # Use get to provide the hourly average as a default if the hour is not in hours_dict
93         count = hours_dict.get(hour, hourly_averages.get(hour, 0))
94
95         completed_counts.append((day - 1) * 24 + hour, count)
96
97 # Convert completed_counts into a DataFrame
98 df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])
99
100
101 # Assuming 'y' is the column in 'df' DataFrame containing your time series data
102 time_series = df_completed['count']
103
104
105 # Plot ACF
106 plt.figure()
107 plot_acf(time_series, lags=40, alpha=0.05) # Adjust lags as needed
108 plt.title('Autocorrelation Function')
109 plt.show()
110
111 # Plot PACF
112 plt.figure()
113 plot_pacf(time_series, lags=40, alpha=0.05, method='ywmm') # Adjust lags and method as needed
114 plt.title('Partial Autocorrelation Function')
115 plt.show()
```

Q5:

```
1  from datetime import datetime, timedelta
2  from collections import defaultdict
3  import numpy as np
4  import pandas as pd
5  from pymongo import MongoClient
6  import matplotlib.pyplot as plt
7
8  usage
9  def split_data(records, train_start, train_end, test_start, test_end):
10     train_data = [record for record in records if train_start <= record['_id']['day'] < train_end]
11     test_data = [record for record in records if test_start <= record['_id']['day'] < test_end]
12     return train_data, test_data
13
14 client = MongoClient("mongodb://ictts:Ict4SM22!@bigdatadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
15
16 # Get the database
17 db = client['carsharing']
18
19 # Define the start and end dates for the query
20 start_date = datetime(year=2017, month=12, day=1)
21 end_date = datetime(year=2018, month=1, day=1)
22
23 # Define the query conditions
24 query_conditions = {
25     'init_date': {'$gte': start_date, '$lt': end_date},
26     'city': "Stuttgart",
27     '$expr': {'$lt': ['$subtract': ['$final_time', '$init_time']], 8000]},
```

```

27     '$expr': {'$ne': ['$init_address', '$final_address']}
28 }
29
30 # Aggregation pipeline
31 pipeline = [
32     {
33         '$match': query_conditions
34     },
35     {
36         '$group': {
37             '_id': {
38                 'day': {'$dayOfMonth': '$init_date'},
39                 'hour': {'$hour': '$init_date'}
40             },
41             'count': {'$sum': 1}
42         }
43     },
44     {
45         '$sort': {
46             '_id.day': 1,
47             '_id.hour': 1
48         }
49     }
50 ]
51 # Execute the query to retrieve the records
52 records = list(db['PermanentBookings'].aggregate(pipeline))
53

```

```

54 # Split data into training and testing sets
55 train_start, train_end = 1, 8 # Training set: first week
56 test_start, test_end = 8, 15 # Testing set: second week
57 train_data, test_data = split_data(records, train_start, train_end, test_start, test_end)
58
59 train_counts_dict = defaultdict(lambda: defaultdict(int))
60
61 # Populate the dictionary with the training data
62 for record in train_data:
63     day = record['_id']['day']
64     hour = record['_id']['hour']
65     count = record['count']
66     train_counts_dict[day][hour] = count
67
68 # Calculate the average count for each hour across all days for training data
69 train_hourly_averages = defaultdict(list)
70 for day, hours in train_counts_dict.items():
71     for hour, count in hours.items():
72         train_hourly_averages[hour].append(count)
73
74 # Compute averages for training data
75 train_hourly_averages = {hour: np.mean(counts) for hour, counts in train_hourly_averages.items()}
76
77 # Fill in missing hours for each day with the average counts for training data
78 completed_train_counts = []
79 for day in range(train_start, train_end):

```

```

80     for hour in range(24):
81         if hour in train_counts_dict[day]:
82             completed_train_counts.append(((day - 1) * 24 + hour, train_counts_dict[day][hour]))
83         else:
84             completed_train_counts.append(((day - 1) * 24 + hour, train_hourly_averages[hour]))
85
86     # Unzip the completed_train_counts for plotting
87     train_hours, train_counts = zip(*completed_train_counts)
88
89     # Plotting for training data
90     plt.figure(figsize=(15, 6))
91     plt.plot(*args: train_hours, train_counts, color='blue', linestyle='-', linewidth=2, marker='', label='Training Data')
92
93     # Create a dictionary to hold the counts for each day and hour for testing data
94     test_counts_dict = defaultdict(lambda: defaultdict(int))
95
96     # Populate the dictionary with the testing data
97     for record in test_data:
98         day = record['_id']['day']
99         hour = record['_id']['hour']
100        count = record['count']
101        test_counts_dict[day][hour] = count
102
103    # Fill in missing hours for each day with the average counts for testing data
104    completed_test_counts = []
105    for day in range(test_start, test_end):
106        for hour in range(24):
107            if hour in test_counts_dict[day]:
108                completed_test_counts.append(((day - 1) * 24 + hour, test_counts_dict[day][hour]))
109            else:
110                completed_test_counts.append(((day - 1) * 24 + hour, train_hourly_averages[hour])) # Filling missing values with training data averages
111
112    # Unzip the completed_test_counts for plotting
113    test_hours, test_counts = zip(*completed_test_counts)
114
115    # Plotting for testing data
116    plt.plot(*args: test_hours, test_counts, color='red', linestyle='-', linewidth=2, marker='', label='Testing Data')
117
118    # Label formatting to show day and hour
119    tick_positions = range(0, 24 * (train_end - train_start), 24) # One tick per day for training data
120    tick_labels = [f'Day {i // 24 + 1}' for i in tick_positions]
121    plt.xticks(tick_positions, tick_labels, rotation=45) # Set x-ticks to be every day and rotate for readability
122
123    plt.xlabel('Day and Hour')
124    plt.ylabel('Count of Bookings')
125    plt.title('Count of Bookings for Each Hour of December 2017')
126    plt.grid(True) # Add grid for better readability
127    plt.legend()
128
129    # Show the plot
130    plt.tight_layout() # Adjust layout so everything fits without overlapping
131    plt.show()

```

Q6:

```

1  from datetime import datetime, timedelta
2  from collections import defaultdict
3  import numpy as np
4  import pandas as pd
5  from pymongo import MongoClient
6  import matplotlib.pyplot as plt
7  from sklearn.metrics import r2_score
8  from statsmodels.tsa.arima.model import ARIMA
9  from sklearn.metrics import mean_squared_error
10
11 # Connect to the MongoDB client
12 client = MongoClient("mongodb://ict4sm22@bigdataadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
13
14 # Get the database
15 db = client['carsharing']
16
17 # Define the start and end dates for the query
18 start_date = datetime(2017, 12, 1)
19 end_date = datetime(2018, 1, 1)
20
21 # Define the query conditions
22 query_conditions = {
23     '$init_date': {'$gte': start_date, '$lt': end_date},
24     'city': "Stuttgart",
25     '$expr': ['$lt': [{"$subtract": ['$final_time', '$init_time']}, 8000]],
26     '$expr': ['$ne': ['$init_address', '$final_address']]}
27 }
28
29 # Aggregation pipeline
30 pipeline = [
31     {

```

```

35     '$group': {
36         '_id': {
37             'day': {'$dayOfMonth': '$init_date'},
38             'hour': {'$hour': '$init_date'}
39         },
40         'count': {'$sum': 1}
41     },
42 },
43 {
44     '$sort': {
45         '_id.day': 1,
46         '_id.hour': 1
47     }
48 }
49 ]
50
51 # Execute the query to retrieve the records
52 records = list(db["PermanentBookings"].aggregate(pipeline))
53
54 # Create a standard dictionary to hold the counts for each day and hour
55 counts_dict = {}
56
57 # Populate the dictionary with the query results
58 for record in records:
59     day = record['_id']['day']
60     hour = record['_id']['hour']
61     count = record['count']
62
63     if day not in counts_dict:
64         counts_dict[day] = {} # Create a new dictionary for the day if it doesn't exist
65
66     # Assuming that there will be only one count per day and hour,
67     # otherwise, you need logic to sum or handle multiple counts

```

```

68     counts_dict[day][hour] = count
69
70
71 # Initialize an empty dictionary to hold lists of counts for each hour
72 hourly_averages = {}
73
74 # Populate the dictionary with counts for each hour
75 for day, hours in counts_dict.items():
76     for hour, count in hours.items():
77         if hour not in hourly_averages:
78             hourly_averages[hour] = [] # Initialize a new list if the hour key doesn't exist
79             hourly_averages[hour].append(count)
80
81 # Compute averages using numpy for mean calculation
82 hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
83 # Fill in missing hours for each day with the average counts
84 completed_counts = []
85 for day in range(1, 32): # Assuming December has 31 days
86
87     for hour in range(24):
88
89
90         hours_dict = counts_dict.get(day, {})
91
92         # Use get to provide the hourly average as a default if the hour is not in hours_dict
93         count = hours_dict.get(hour, hourly_averages.get(hour, 0))
94
95         completed_counts.append(((day - 1) * 24 + hour, count))
96
97 # Convert completed_counts into a DataFrame
98 df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])
99
100

```

```

102 if not isinstance(df_completed.index, pd.DatetimeIndex):
103     df_completed.index = pd.to_datetime([datetime(2017, 12, 1) + timedelta(hours=x) for x in df_completed.index])
104
105 def predict_arima(train_data, test_data, order):
106     # Ensure the test_data is not empty and does not contain zero values
107     if test_data.empty or (test_data == 0).any():
108         raise ValueError("Test data must not be empty and must not contain zeros when calculating MAPE.")
109
110
111     predictions = []
112     history = list(train_data)
113
114     for (variable, model: Any):
115         for a):
116             model = ARIMA(history, order=order)
117             model_fit = model.fit()
118             yhat = model_fit.forecast()[0]
119             predictions.append(yhat)
120             history.append(test_data.iloc[t])
121
122     # Convert predictions to a numpy array for error calculation
123     predictions = np.array(predictions)
124
125     # Calculate error metrics
126     metrics = {
127         "MAPE": np.mean(np.abs(test_data - predictions) / test_data) * 100,
128         "MSE": mean_squared_error(test_data, predictions),
129         "R2": r2_score(test_data, predictions)
130     }
131
132     return predictions, metrics
133
134
135     # Now, use the function in your given context
136     train_end = pd.Timestamp('2017-12-07 00:00:00')

```

```

135 test_end = pd.Timestamp('2017-12-14 00:00:00')
136
137 train_data = df_completed.loc[df_completed.index < train_end, 'count']
138 test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']
139
140 # Call the predict_arima function
141 predictions, metrics = predict_arima(train_data, test_data, (2, 0, 3))
142
143 # Create a pandas Series for the forecasted values
144 forecast_series = pd.Series(predictions, index=test_data.index)
145
146 # Plot the training data, test data, and forecast
147 plt.figure(figsize=(14, 7))
148 plt.plot(train_data, label='Training Data')
149 plt.plot(test_data, label='Test Data', color='orange')
150 plt.plot(forecast_series, label='Forecast', linestyle='--', color='green')
151 plt.legend()
152 plt.title('Hourly Forecast vs Actuals - Stuttgart')
153 plt.show()
154
155 # Optionally, print out the error metrics
156 print("Train Time window: 2017-12-07 00:00:00 ")
157 print("Test Time window: 2017-12-14 00:00:00")
158 print("p, d, q = 2, 0, 3")
159 print(metrics)

```

Q7:

```

1 from datetime import datetime, timedelta
2 from collections import defaultdict
3 import numpy as np
4 import pandas as pd
5 from pymongo import MongoClient
6 import matplotlib.pyplot as plt
7 from sklearn.metrics import r2_score
8 from statsmodels.tsa.arima.model import ARIMA
9 from sklearn.metrics import mean_squared_error
10
11 # Connect to the MongoDB client
12 client = MongoClient("mongodb://ictts:ict4SM22!@bigdataadb.polito.it:27017/carsharing?ssl=true&authSource=carsharing&tlsAllowInvalidCertificates=true")
13
14 # Get the database
15 db = client['carsharing']
16
17 # Define the start and end dates for the query
18 start_date = datetime(2017, 12, 1)
19 end_date = datetime(2018, 1, 1)
20
21 # Define the query conditions
22 query_conditions = {
23     '$init_date': {'$gte': start_date, '$lt': end_date},
24     'city': "Stuttgart",
25     '$expr': ['$lt': [{$subtract: ['$final_time', '$init_time']}, 8000]],
26     '$expr': ['$ne': ['$init_address', '$final_address']]
27 }
28
29 # Aggregation pipeline
30 pipeline = [
31     {
32         '$match': query_conditions

```

The screenshot shows a Microsoft Visual Studio Code (VS Code) window with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Search.
- Status Bar:** Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More. Ln 66, Col 6. Spaces: 4. UTF-8. CRLF. ENG. 6:15 PM. 1/3/2024.
- Sidebar:** Explorer (No folder opened), Outline, Timeline, and a search bar.
- Code Editor:** The main area displays Python code for data processing. The code uses MongoDB's aggregation framework to group documents by date and hour, counts the documents, sorts them, and then processes the results to create a standard dictionary where each key is a day and hour combination, and the value is the count. It then initializes an empty dictionary for hourly averages, iterates through the counts dictionary to add counts to the hourly averages dictionary, and finally computes the average for each hour using NumPy's mean function. The code also handles missing hours for each day by filling them with the average counts from the hourly averages dictionary.

```
File Edit Selection View Go Run Terminal Help < - > Search

Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More

EXPLORER
NO FOLDER OPENED
D: > class files > 3.semester > mobility > mellia > lab 2 > lab2 - docs > 7 > 7-a.py > ...

33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96

You have not yet opened a folder.
Open Folder
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96

# 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

# 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)

Ln 66, Col 6. Spaces: 4. UTF-8. CRLF. ENG. 6:15 PM. 1/3/2024
```

```

67     # otherwise, you need logic to sum or handle multiple counts
68     counts_dict[day][hour] = count
69
70
71 # Initialize an empty dictionary to hold lists of counts for each hour
72 hourly_averages = {}
73
74 # Populate the dictionary with counts for each hour
75 for day, hours in counts_dict.items():
76     for hour, count in hours.items():
77         if hour not in hourly_averages:
78             hourly_averages[hour] = [] # Initialize a new list if the hour key doesn't exist
79             hourly_averages[hour].append(count)
80
81 # Compute averages using numpy for mean calculation
82 hourly_averages = {hour: round(np.mean(counts), 2) for hour, counts in hourly_averages.items()}
83 # Fill in missing hours for each day with the average counts
84 completed_counts = []
85 for day in range(1, 32): # Assuming December has 31 days
86
87     for hour in range(24):
88
89     |
90
91     hours_dict = counts_dict.get(day, {})
92
93     # Use get to provide the hourly average as a default if the hour is not in hours_dict
94     count = hours_dict.get(hour, hourly_averages.get(hour, 0))
95
96     completed_counts.append(((day - 1) * 24 + hour, count))
97
98 # Convert completed_counts into a DataFrame
99 df_completed = pd.DataFrame(completed_counts, columns=['hours_since_start', 'count'])
100
101 train_end = pd.Timestamp('2017-12-07 00:00:00')
102 test_end = pd.Timestamp('2017-12-14 00:00:00')
103
104 train_data = df_completed.loc[df_completed.index < train_end, 'count']
105 test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']
106
107
108 def mean_percentage_error(y_true, y_pred):
109     y_true, y_pred = np.array(y_true), np.array(y_pred)
110     return np.mean((y_true - y_pred) / y_true) * 100
111
112 def mean_absolute_percentage_error(y_true, y_pred):
113     y_true, y_pred = np.array(y_true), np.array(y_pred)
114     return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
115
116 def calculate_r_squared(y_true, y_pred):
117     return r2_score(y_true, y_pred) * 100
118
119 # Set the range of values for p, d, and q
120 p_values = range(0, 5)
121 q_values = range(1, 3)
122
123 best_r2 = float('-inf') # Initialize with a low value since we're maximizing R2
124 best_order = None
125 best_metrics = {}
126
127
128 # Create subplots
129 fig, axes = plt.subplots(len(p_values), len(q_values), figsize=(20, 20))
130 fig.tight_layout(pad=5.0)
131
132 # Perform grid search
133 for i, p in enumerate(p_values):
134     for j, q in enumerate(q_values):
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165

```

```
163 # Perform grid search
164 for i, p in enumerate(p_values):
165     for j, q in enumerate(q_values):
166         try:
167             # Fit the ARIMA model on the training set
168             predictions, metrics = predict_arima(train_data, test_data, (p, 0, q))
169
170             # Calculate metrics
171             mse = metrics["MSE"]
172             r2 = metrics["R2"]
173             mape = metrics["MAPE"]
174
175             # Plot the training data, test data, and forecast
176             ax = axes[i][j] if axes.ndim > 1 else axes[max(i, j)]
177             ax.plot(train_data, label='Training Data')
178             ax.plot(test_data, label='Test Data', color='orange')
179             ax.plot(test_data.index, predictions, label='Forecast', linestyle='--', color='green')
180             # Set the font size of the tick labels
181             ax.tick_params(axis='both', which='major', labelsize=7)
182             # Set the font size title
183             ax.set_title(f'ARIMA({{p}},{0},{{q}}) - MAPE: {{mape:.2f}}%, R2: {{r2:.2f}} - Stuttgart', fontsize=8)
184             ax.{{class}} Exception
185
186         except Exception as e:
187             print(f'An error occurred for model with (p, d, q) = ({p}, 0, {q}): {e}')
188             ax = axes[i][j] if axes.ndim > 1 else axes[max(i, j)]
189             ax.set_title(f'ARIMA({{p}},{0},{{q}}) - Error', fontsize=8)
190
191 plt.show()
```

```
147 |     return predictions, metrics
148 |
149 # Define the initial training period
150 initial_train_end = pd.Timestamp('2017-12-07 00:00:00')
151
152 start_date = '2017-12-01'
153 end_date = '2017-12-25'
154 expanding_windows = [pd.Timestamp(initial_train_end) + pd.Timedelta(days=i) for i in range(0, (pd.Timestamp(end_date) - pd.Timestamp(initial_train_end)).days + 1)]
155
156 # Ensure we only take the first 6 windows
157 expanding_windows = expanding_windows[:6]
158
159 # Set up the matplotlib figure - 6 subplots (one for each expanding window)
160 fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)
161
162 # Define the test period outside the loop as they are the same for all windows
163 test_start = pd.Timestamp('2017-12-26 00:00:00')
164 test_end = pd.Timestamp('2017-12-28 23:59:59')
165 test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']
166
167 for i, expand_day in enumerate(expanding_windows):
168     train_end = expand_day
169
170     # Split the data into train and test sets according to the current window
171     train_data = df_completed.loc[df_completed.index <= train_end, 'count']
172
173     # Use the provided predict_arima function
174     predictions, metrics = predict_arima(train_data, test_data, (2,0,1))
175
176     # Plot actual vs predicted values
177     # The test_data.index gives you the correct timestamps for the x-axis
178     axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
179     axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')
```

```

157     expanding_windows = expanding_windows[:6]
158
159     # Set up the matplotlib figure - 6 subplots (one for each expanding window)
160     fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)
161
162     # Define the test period outside the loop as they are the same for all windows
163     test_start = pd.Timestamp('2017-12-26 00:00:00')
164     test_end = pd.Timestamp('2017-12-28 23:59:59')
165     test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']
166
167     for i, expand_day in enumerate(expanding_windows):
168         train_end = expand_day
169
170         # Split the data into train and test sets according to the current window
171         train_data = df_completed.loc[df_completed.index <= train_end, 'count']
172
173         # Use the provided predict_arima function
174         predictions, metrics = predict_arima(train_data, test_data, (2,0,1))
175
176         # Plot actual vs predicted values
177         # The test_data.index gives you the correct timestamps for the x-axis
178         axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
179         axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')
180
181         # Set title for each subplot
182         axes[i].set_title(f'Training from 2017-12-01 to {expand_day.strftime("%Y-%m-%d")}' - Columbus', fontsize=7)
183
184         # Add legend
185         axes[i].legend()
186
187     # Improve plot layout
188     plt.tight_layout()
189     plt.show()

```

Sliding:

```

149     # Define the initial training start and end dates
150     initial_train_start = pd.Timestamp('2017-12-07 00:00:00')
151     initial_train_end = initial_train_start + pd.Timedelta(days=7)
152
153     # Set the sliding step in days
154     slide_step = 3
155
156     # Calculate the number of windows by dividing the total number of days by the step
157     total_days = (pd.Timestamp('2017-12-25') - initial_train_start).days
158     num_windows = total_days // slide_step
159
160     # Generate start and end dates for each sliding window
161     sliding_windows = [(initial_train_start + pd.Timedelta(days=i*slide_step),
162                         initial_train_start + pd.Timedelta(days=(i+1)*slide_step))
163                         for i in range(num_windows)]
164
165     # Ensure we only take the first 6 windows
166     sliding_windows = sliding_windows[:6]
167
168     # Set up the matplotlib figure - 6 subplots (one for each sliding window)
169     fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)
170
171     # Define the test period outside the loop as they are the same for all windows
172     test_start = pd.Timestamp('2017-12-26 00:00:00')
173     test_end = pd.Timestamp('2017-12-28 23:59:59')
174     test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']
175

```

```

165 # Ensure we only take the first 6 windows
166 sliding_windows = sliding_windows[:6]
167
168 # Set up the matplotlib figure - 6 subplots (one for each sliding window)
169 fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 12), sharex=True)
170
171 # Define the test period outside the loop as they are the same for all windows
172 test_start = pd.Timestamp('2017-12-26 00:00:00')
173 test_end = pd.Timestamp('2017-12-28 23:59:59')
174 test_data = df_completed.loc[(df_completed.index >= test_start) & (df_completed.index <= test_end), 'count']
175
176 for i, (train_start, train_end) in enumerate(sliding_windows):
177     # Split the data into train and test sets according to the current window
178     train_data = df_completed.loc[(df_completed.index >= train_start) & (df_completed.index <= train_end), 'count']
179
180     # Use the provided predict_arima function
181     predictions, metrics = predict_arima(train_data, test_data, (2,0,1))
182
183     # Plot actual vs predicted values
184     # The test_data.index gives you the correct timestamps for the x-axis
185     axes[i].plot(test_data.index, test_data, label='Actual', color='blue', marker='o')
186     axes[i].plot(test_data.index, predictions, label='Predicted', color='red', linestyle='--', marker='x')
187
188     # Set title for each subplot
189     axes[i].set_title(f'Training from {train_start.strftime("%Y-%m-%d")} to {train_end.strftime("%Y-%m-%d")}) - Stuttgart', fontsize=10)
190
191     # Add legend
192     axes[i].legend()
193
194     # Improve plot layout
195 plt.tight_layout()
196 plt.show()

```

Q8:

```

140 # Now, use the function in your given context
141 train_end = pd.Timestamp('2017-12-7 00:00:00')
142 test_end = pd.Timestamp('2017-12-14 00:00:00')
143
144 forecast_hours = [4, 8, 12, 16, 20, 24]
145
146 # Start a new plot
147 plt.figure(figsize=(14, 7))
148
149 # Plot the training data only once
150 train_data = df_completed.loc[df_completed.index < train_end, 'count']
151 plt.plot(train_data, label='Training Data')
152
153 final_metrics = []
154
155 # Loop through each forecast horizon and plot the forecasts
156 for h in forecast_hours:
157     # Define the start and end of the test_data for each forecast
158     test_data_start = train_end + pd.Timedelta(hours=h)
159     test_data_end = test_end + pd.Timedelta(hours=h)
160     test_data = df_completed.loc[(df_completed.index >= test_data_start) & (df_completed.index < test_data_end), 'count']
161
162     # Call the predict_arima function to make predictions
163     predictions, metrics = predict_arima(train_data, test_data, (2, 0, 3))
164
165     # Create a pandas Series for the forecasted values with the correct index
166     forecast_index = test_data.index
167     forecast_series = pd.Series(predictions, index=forecast_index)
168
169     # Plot the forecast
170     plt.plot(forecast_series, label=f'{h}-hour Forecast', linestyle='--')
171
172
173
174

```

```
157 |     forecast_series = parseres(p, collections, index=forecast_index)
158 |
159 |     # Plot the forecast
160 |     plt.plot(forecast_series, label=f'{h}-hour Forecast', linestyle='--')
161 |
162 |     final_metrics.append(f"metrics for {h} hour forecast: {metrics} in Stuttgart")
163 |
164 |     # Plot the actual test data
165 |     actual_test_data = df_completed.loc[(df_completed.index >= train_end) & (df_completed.index < test_end), 'count']
166 |     plt.plot(actual_test_data, label='Test Data', color='orange', linestyle='-.')
167 |
168 |     # Finalize the plot
169 |     plt.legend()
170 |     plt.title('Hourly Forecast vs Actuals Stuttgart')
171 |     plt.xlabel('Date')
172 |     plt.ylabel('Count')
173 |     plt.show()
174 |
175 |     for i in range(len(final_metrics)):
176 |         print(final_metrics[i], "\n")
```