ICT for smart mobility

Report of Lab 2

Group 10

Setareh Pourgholamali S309064

Hossein Zahedi Nezhad S309247

SeyedMohammadhossein Sajjadikaboudi S313242

January 2024

Introduction

This lab investigates ARIMA model predictions, emphasizing the impact of tuning parameters (p, d, q), selecting the training window size N, and comparing expanding versus sliding window training policies. The goal is to evaluate how these choices affect prediction accuracy.

1. Time series and Data Extraction

In the initial step, we focused on extracting the rental time series for each city, namely Roma Enjoy, Milano Enjoy, and Vancouver (Car2go). This analysis covered a 30-day period, specifically from January 1st to January 30th, 2018. This duration was chosen to encompass the entire month of January. To refine the time series, we systematically filtered out outliers, excluding invalid rentals (where the initial and final positions are the same), rentals lasting less than or equal to 3 minutes, and rentals lasting more than or equal to 150 minutes.

2. Checking Missing Values

Since, the ARIMA model is only suitable for data that does not have missing values. we checked the data to make sure that there is no missing value, and fortunately, this problem did not exist during this period (the code is in the appendix). The resulting time series are depicted in Figure 1.

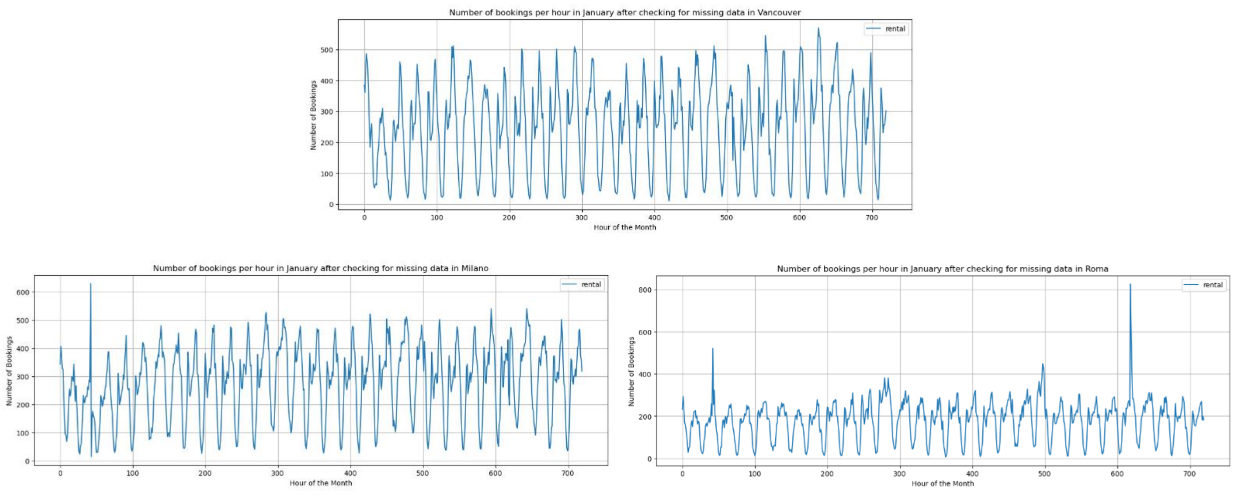


Figure 1. Number of bookings in January in each City after checking Missing data

3. stationary Checking

To ensure the stationarity of the time series, which is crucial for accurate predictions, statistical properties like Mean, Variance, and Autocorrelation are assessed. The analysis reveals that these properties remain nearly constant over time. Consequently, the time series is deemed stationary, obviating the need for differencing (d=0) in the ARIMA mode. For optimal ARIMA model performance, a stationary time series, free from trends, is mandatory. Employing the rolling statistics method with a subset of 168 hours (1 week) for each city, we confirm the constancy of moving averages over time, indicating stationarity. With this stability, differentiation (d=0) is unnecessary, leading to an updated ARIMA model as ARMA model (p, 0, q).

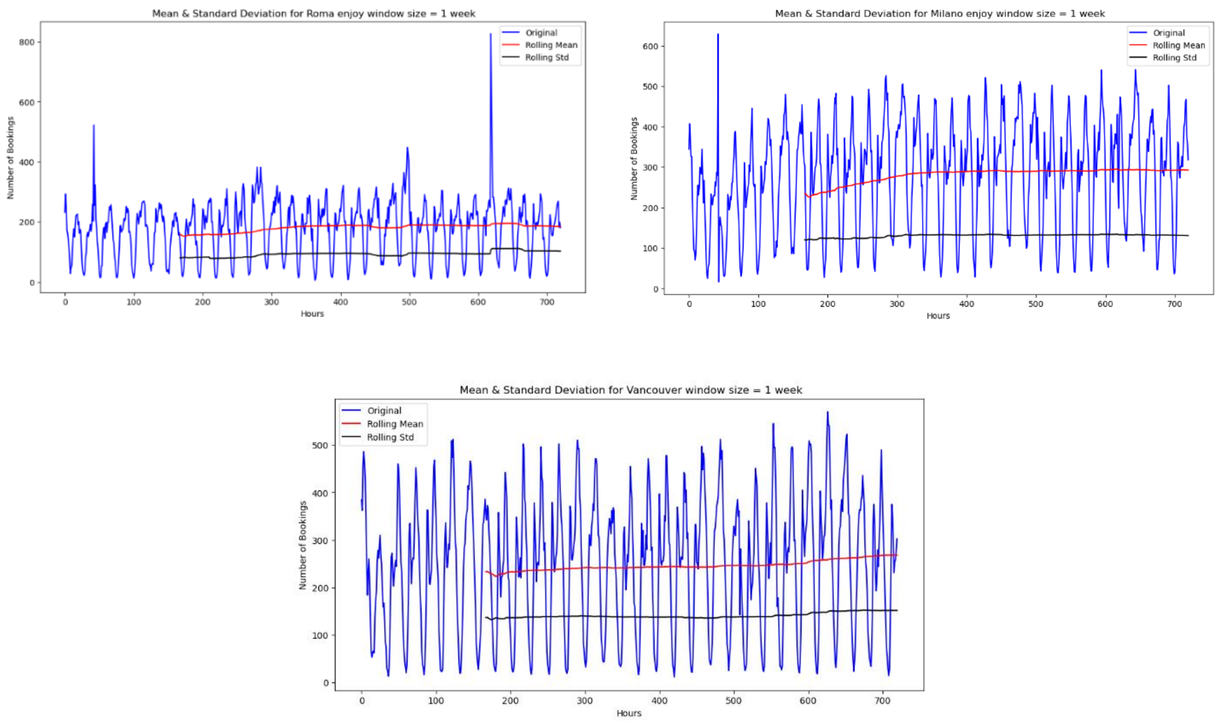


Figure 2. Stationary Checking by Rolling Mean & Standard Deviation

4. Evaluating ACF and PACF

The Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) are key for exploring time series data. They help spot patterns and see if data behaves randomly. ACF looks at how past and current data relate to each other. PACF focuses on the correlation between the gaps left after removing these relationships and the next data point. First, we plot the ACF for all the sample in each city, which is shown in the figure 3. We can see correlation in daily times (24h, 48h, 72h, …). Even correlation is observable for weekly times (week1, week2…). Then, we compute ACF and PACF in zooming mode for just 48 lags in each city. As is observable, mostly in the ACF, correlation in 24 hours and 48 hours is not negligible. On the other hand, the PACF decreases quickly and becomes negligible so it might be an AR model. That's why we assume that q=0. As ACF diagrams in 4 shown we can approximate that the P for the Vancouver, Milano and Roma is 2, 2, and 3 Respectively.

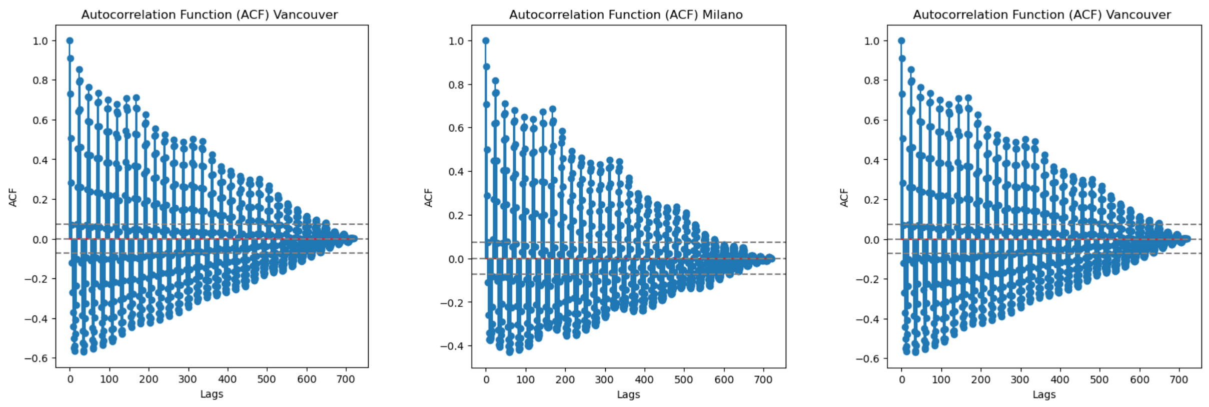


Figure 3. Autocorrelation Function for all the Samples in each city

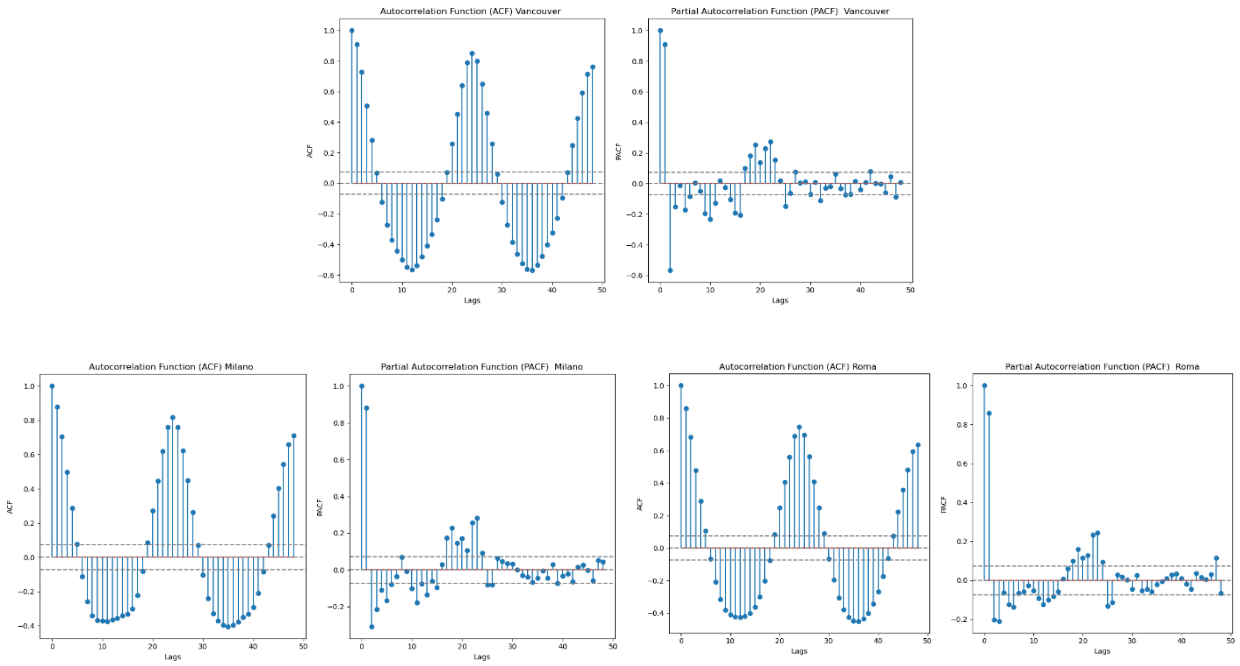


Figure 4. ACF and PACF for 48 lags for each city

So, we fitted our model (2, 0, 0) for Milano, Vancouver and (3,0,0) for Roma and then we plotted them versus actual Rentals. In following Steps, we will calculate the errors.

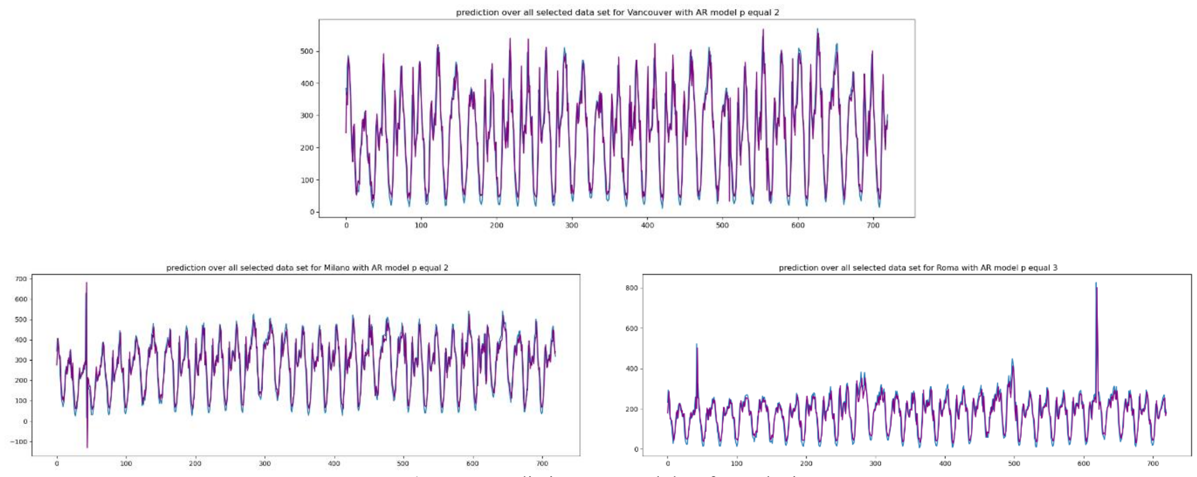


Figure 5. Prediction vs Actual data for each city

5. Data Splitting

After the choice of the initial parameter p, d and q Since our samples are aggregated per hour, we split the dataset to 504 hours for training and 48hours for testing. In fact, we use the first 3 weeks to train the model and the 2 following days to test our model.

6, 7.a. Model Training and Error Computation

Given N, (p,d,q) and a trained model, the error can be computed and evaluated through different metrics mentioned below:

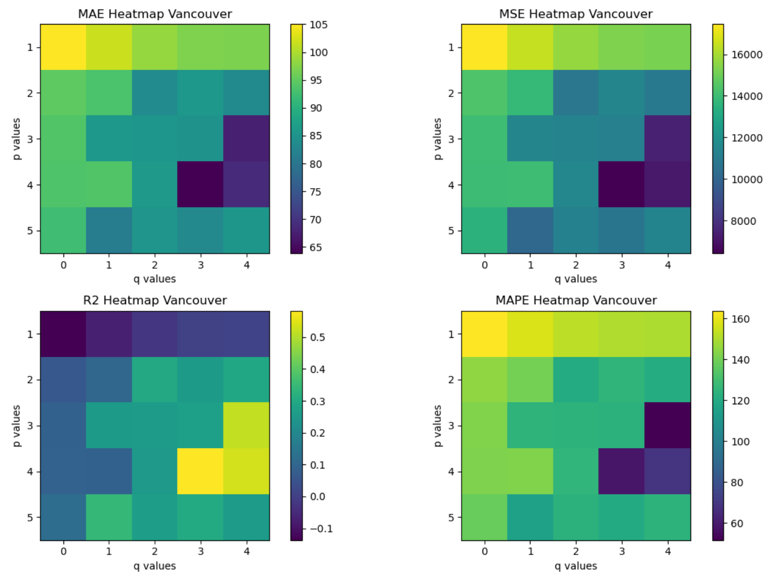
- **Mean Absolute Error (MAE)**: This is the average of the absolute errors between the predicted values and actual values.

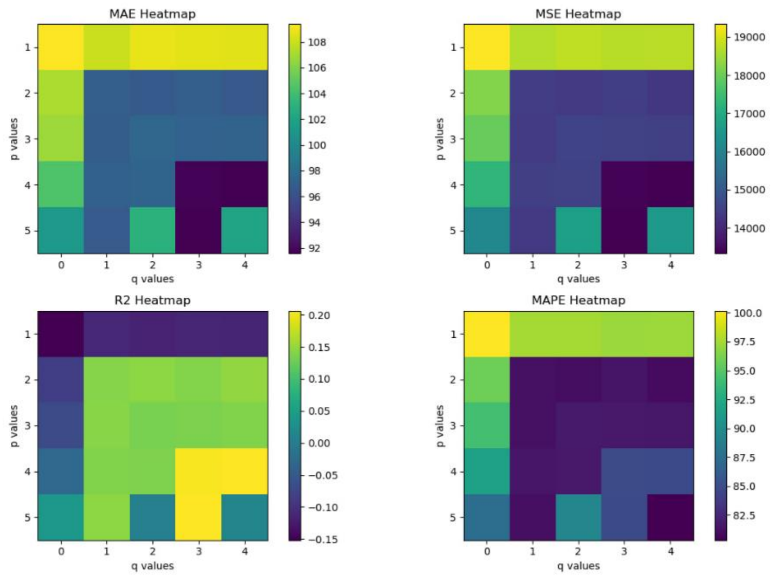
- **Mean Squared Error (MSE)**: This is the average of the squares of the errors between the predicted values and the actual values. MSE gives more weight to larger errors, due to the squaring of each term.

- **Coefficient of Determination (R^2)**: It is a statistical measure used in regression analysis. it measures how well the predictions match the actual data.

- **Mean Absolute Percentage Error (MAPE):** It is a metric for evaluating the accuracy of forecasting models, including ARIMA models. These metrics are used to assess how well a forecasting model predicts data by comparing forecasted values with the actual observed data.

In this part we assigned different values to p and q to calculate the error metrics for them, as we see our first assumption which was a pure AR model it was not a good decision because the error is too high for all three cities.





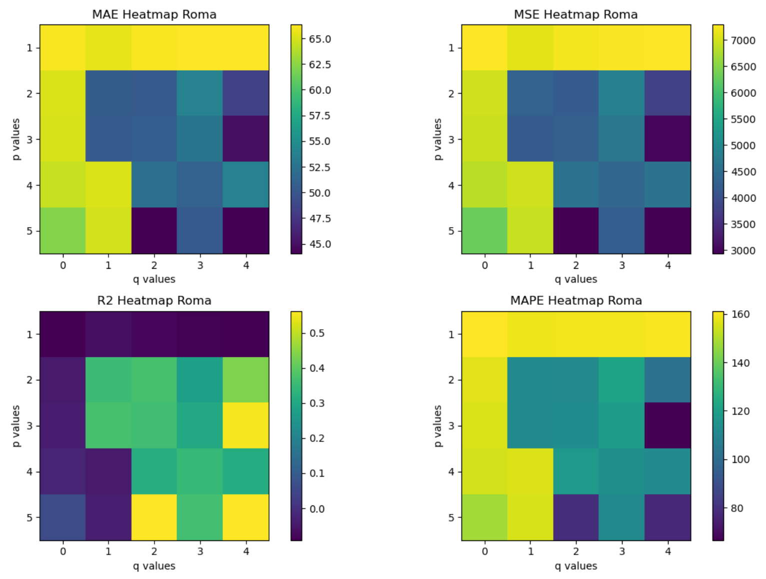


Figure 6. Heatmap of R^2 errors for different cities

When selecting the optimal values of p and q, we prioritize R^2 as our chief metric because it is scale-independent, making it a superior option for our purposes. A value nearer to 1 is preferable as it indicates a better model fit. In the R^2 heatmap for Milan, we opt for the ARMA (4,0,3) model even though its R^2 is marginally lower than that of the ARMA (5,0,2) model. We give precedence to model simplicity, and therefore, a less complex model is desirable. Similarly, for Rome, the ARMA (3,0,4) is our model of choice, and for Vancouver, we also select ARMA (4,0,3).

City	***ARIMA(p,d,q)***	***MAE***	***MSE***	***R^2***	***MAPE***
Vancouver	*(4,0,3)*	*44.93*	*2982.58*	***0.55***	*66.52%*
Milano	*(4,0,3)*	*91.78*	*13396.24*	***0.20***	*84.83%*
Roma	*(3,0,4)*	*63.84*	*6413.31*	***0.58***	*58.40%*

Table 1. Calculated errors with different metrics for each city

7.b, 7.c. Changing Learning Strategy and Comparison

After choosing the best set of hyper parameters for the ARIMA models, the size of the training data window is varied to see how changing the number of days, N, affects the predictions. Assuming total number of samples are 3 weeks and to evaluate performance, two distinct training approaches are used. The first approach uses a sliding window technique, and the second uses an expanding window technique. for first learning strategy which is sliding we consider 2 weeks for training and one day for test and sliding window equal to 24 hours. For expanding window, we assume the initial size of training is 2 weeks and test is one day but the training set expands each time by 24 hours (1 day). With the expanding window approach, the model is trained on an incrementally larger set of data at each step. Ultimately, the training set for the expanding window approach will contain more data than in the sliding window approach so for all three cities the expanding window method show acceptable result compare to sliding window as bellow table 2 and then plotting the prediction values compare to actual rentals in this period shows in figure 7.

Expanding window	***MAE***	***MAPE***	***R^2***
Vancouver	*82.25*	*58%*	*0.675*
Milano	*69.44*	*38%*	*0.562*
Roma	*49.99*	*36%*	*0.877*

Table 2. Metrics over expanding windows in each city

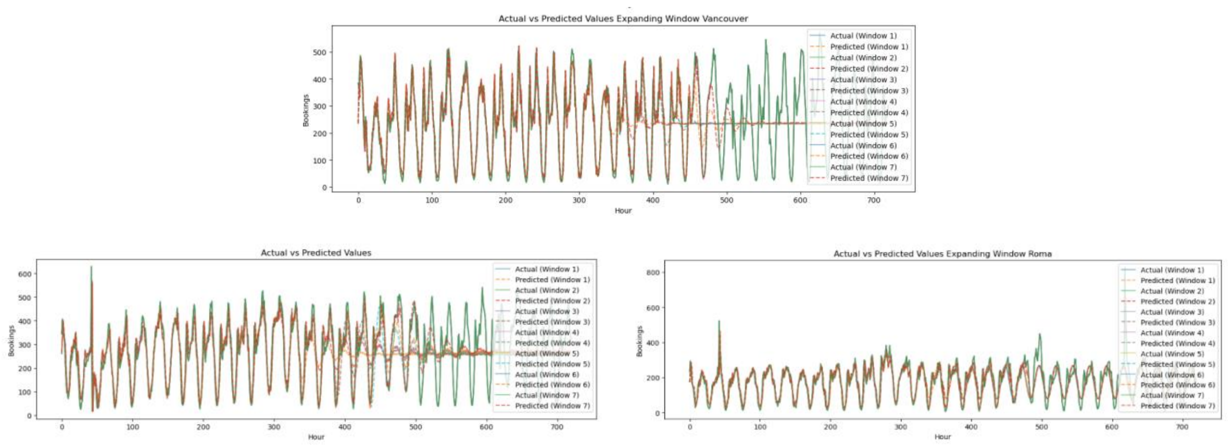


Figure 7. Prediction vs Actual data for Expanding Windows

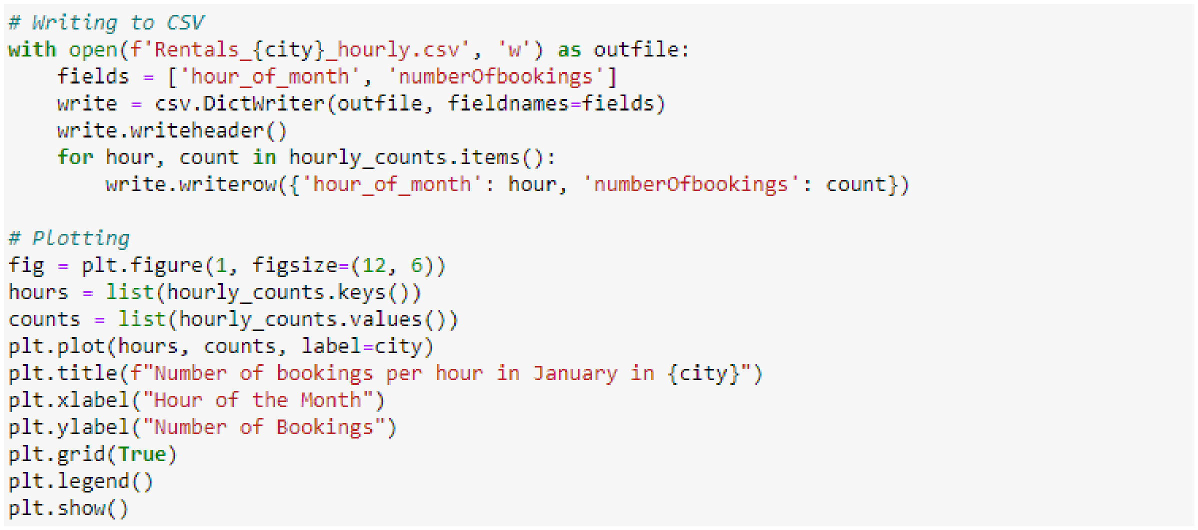
Appendix

TASK 1

```
import matplotlib.pyplot as plt
import pprint
import datetime
import time
import warnings
import pymongo as pm #import MongoClient only
import os
import pandas as pd
from sklearn.metrics import (mean_squared_error, mean_absolute_error, r2_score)
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
from statsmodels.tsa.stattools import acf, pacf
import seaborn as sn
import csv
import numpy as np
client = pm.MongoClient('bigdatadb.polito.it',
 ssl=True,
 authSource = 'carsharing',
 username = 'ictts',
 password ='Ict4SM22!',
 tlsAllowInvalidCertificates=True)
db = client['carsharing']
PB_C2 = db['PermanentBookings']
AB_C2 = db['ActiveBookings']
PP_C2 = db['PermanentParkings']
AP_C2 = db['ActiveParkings']
PB_EJ = db['enjoy_PermanentBookings']
AB_EJ = db['enjoy_ActiveBookings']
PP_EJ = db['enjoy_PermanentParkings']
AP_EJ = db['enjoyActiveParkings']
cities = ["Vancouver", "Milano", "Roma"]
```

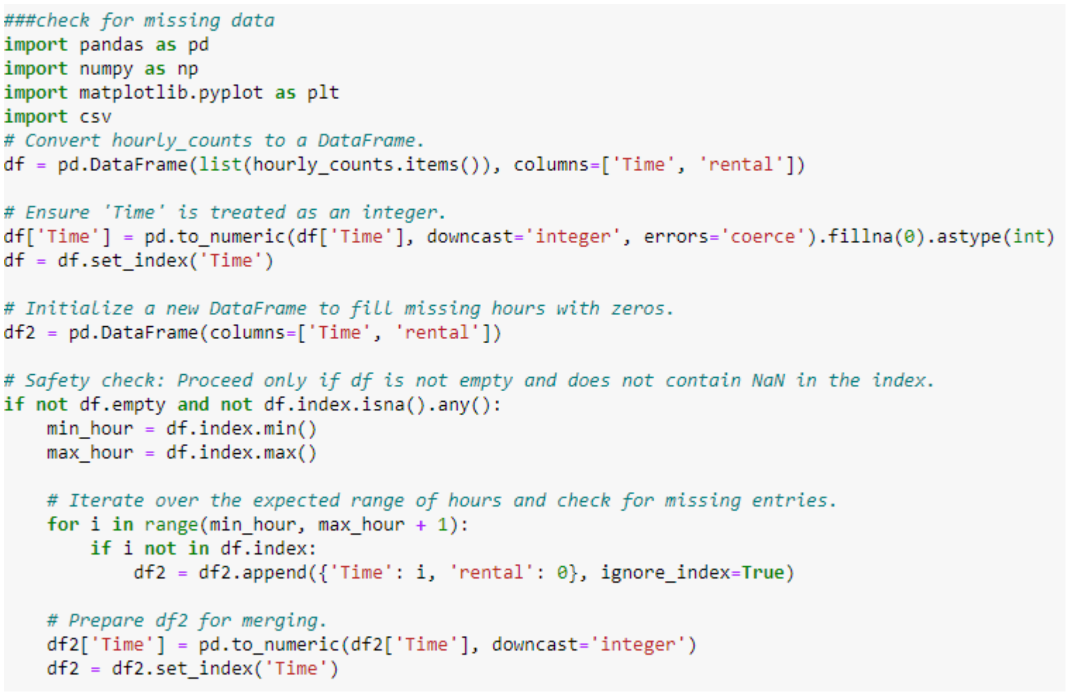
```python
city = "Milano"
start_date = datetime.datetime(2018, 1, 1, 0, 0, 0)
end_date = datetime.datetime(2018, 1, 30, 23, 59, 59)

# Convert dates to Unix timestamps
init_jan = int(start_date.timestamp())
final_jan = int(end_date.timestamp())

# Aggregation pipeline
bookings = PB_EJ.aggregate([
    {
        "$match": {
            "$and": [
                {"city": city},
                {"init_time": {"$gte": init_jan}},
                {"init_time": {"$lte": final_jan}}
            ]
        }
    },
    {
        "$project": {
            "_id": 0,
            "moved": {
                "$ne": [
                    {"$arrayElemAt": ["$origin_destination.coordinates", 0]},
                    {"$arrayElemAt": ["$origin_destination.coordinates", 1]}
                ]
            },
            "bookingDuration": {"$subtract": ["$final_time", "$init_time"]},
            "init_time": 1
        }
```

```python
    },
    {
        "$match": {
            "$and": [
                {"moved": True},
                {"bookingDuration": {"$gte": 180}},
                {"bookingDuration": {"$lte": 9000}}
            ]
        }
    }
])


# Convert the query result to a list
temp = list(bookings)

# Sort bookings by initial time
sortedBookings = sorted(temp, key=lambda i: i['init_time'])

# Helper function to convert timestamp to hour of the month
def timestamp_to_hour(timestamp, start_timestamp):
    return int((timestamp - start_timestamp) / 3600)

# Count bookings per hour
hourly_counts = {}
for booking in sortedBookings:
    hour_of_month = timestamp_to_hour(booking['init_time'], init_jan)
    hourly_counts[hour_of_month] = hourly_counts.get(hour_of_month, 0) + 1
```

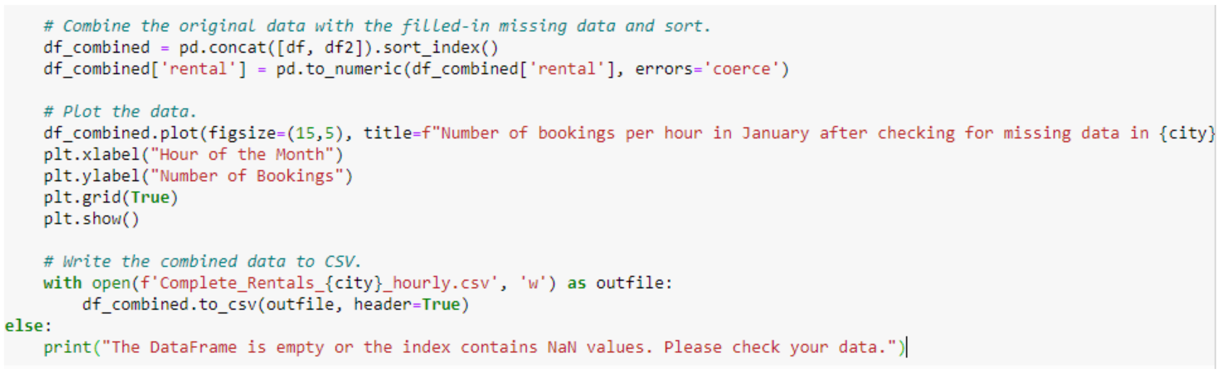


```python
# Writing to CSV
with open(f'Rentals_{city}_hourly.csv', 'w') as outfile:
    fields = ['hour_of_month', 'numberOfbookings']
    write = csv.DictWriter(outfile, fieldnames=fields)
    write.writeheader()
    for hour, count in hourly_counts.items():
        write.writerow({'hour_of_month': hour, 'numberOfbookings': count})

# Plotting
fig = plt.figure(1, figsize=(12, 6))
hours = list(hourly_counts.keys())
counts = list(hourly_counts.values())
plt.plot(hours, counts, label=city)
plt.title(f"Number of bookings per hour in January in {city}")
plt.xlabel("Hour of the Month")
plt.ylabel("Number of Bookings")
plt.grid(True)
plt.legend()
plt.show()
```

Task 2



```python
###check for missing data
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv
# Convert hourly_counts to a DataFrame.
df = pd.DataFrame(list(hourly_counts.items()), columns=['Time', 'rental'])

# Ensure 'Time' is treated as an integer.
df['Time'] = pd.to_numeric(df['Time'], downcast='integer', errors='coerce').fillna(0).astype(int)
df = df.set_index('Time')

# Initialize a new DataFrame to fill missing hours with zeros.
df2 = pd.DataFrame(columns=['Time', 'rental'])

# Safety check: Proceed only if df is not empty and does not contain NaN in the index.
if not df.empty and not df.index.isna().any():
    min_hour = df.index.min()
    max_hour = df.index.max()

    # Iterate over the expected range of hours and check for missing entries.
    for i in range(min_hour, max_hour + 1):
        if i not in df.index:
            df2 = df2.append({'Time': i, 'rental': 0}, ignore_index=True)

    # Prepare df2 for merging.
    df2['Time'] = pd.to_numeric(df2['Time'], downcast='integer')
    df2 = df2.set_index('Time')
```



```python
    # Combine the original data with the filled-in missing data and sort.
    df_combined = pd.concat([df, df2]).sort_index()
    df_combined['rental'] = pd.to_numeric(df_combined['rental'], errors='coerce')

    # Plot the data.
    df_combined.plot(figsize=(15,5), title=f"Number of bookings per hour in January after checking for missing data in {city}
    plt.xlabel("Hour of the Month")
    plt.ylabel("Number of Bookings")
    plt.grid(True)
    plt.show()

    # Write the combined data to CSV.
    with open(f'Complete_Rentals_{city}_hourly.csv', 'w') as outfile:
        df_combined.to_csv(outfile, header=True)
else:
    print("The DataFrame is empty or the index contains NaN values. Please check your data.")
```

Task3

```
###task3 it is stationary so assume that d=0
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])

# Set the 'Hour' column as the index
df.set_index('Hour', inplace=True)

# Calculate the rolling mean and standard deviation with a 1 week window
rolling_mean = df['Bookings'].rolling(window=168).mean()
rolling_std = df['Bookings'].rolling(window=168).std()

# Plotting the original time series, rolling mean, and rolling standard deviation
plt.figure(figsize=(12, 6))
plt.plot(df['Bookings'], color='blue', label='Original')
plt.plot(rolling_mean, color='red', label='Rolling Mean')
plt.plot(rolling_std, color='black', label='Rolling Std ')
plt.legend(loc='best')
plt.title(' Mean & Standard Deviation for Milano enjoy window size = 1 week')
plt.xlabel('Hours')
plt.ylabel('Number of Bookings')
plt.show()
```

Task 4

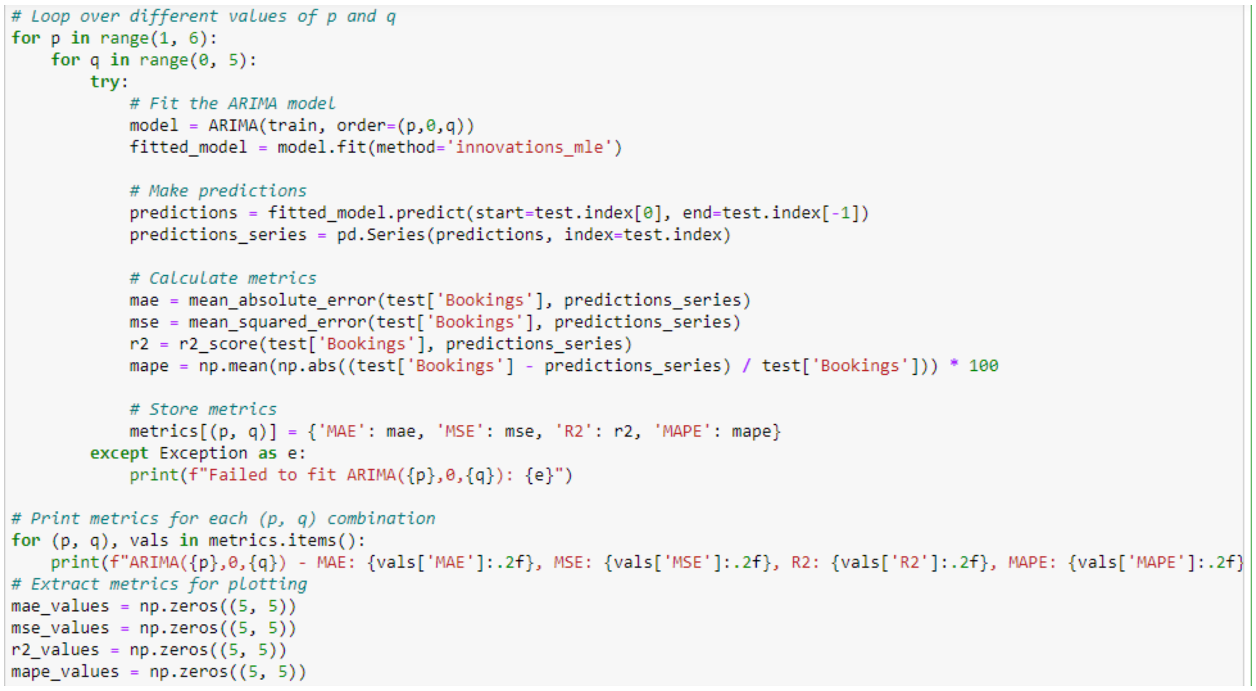
```
###task4 acf for all sample
###correlation in daily times,also in same say and also weakly baisis so there is priodicity
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])

# Set the 'Hour' column as the index
df.set_index('Hour', inplace=True)

# Compute ACF and PACF
lag_acf = acf(df['Bookings'], nlags=744)
#lag_pacf = pacf(df['Bookings'], nlags=48, method='ols')

# Plotting ACF using stem plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.stem(lag_acf, use_line_collection=True)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.title('Autocorrelation Function (ACF) Milano ')
plt.xlabel('Lags')
plt.ylabel('ACF')

###as the ACF better corrlation in 24h and 48h not negligible
##the PCAF is quickly drops to almost negligible so it maybe is AR model
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])

# Set the 'Hour' column as the index
df.set_index('Hour', inplace=True)
```

```
# Compute ACF and PACF
lag_acf = acf(df['Bookings'], nlags=48)
lag_pacf = pacf(df['Bookings'], nlags=48, method='ols')

# Plotting ACF using stem plot
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.stem(lag_acf, use_line_collection=True)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.title('Autocorrelation Function (ACF) Milano ')
plt.xlabel('Lags')
plt.ylabel('ACF')

# Plotting PACF using stem plot
plt.subplot(122)
plt.stem(lag_pacf, use_line_collection=True)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(df['Bookings'])), linestyle='--', color='gray')
plt.title('Partial Autocorrelation Function (PACF)  Milano')
plt.xlabel('Lags')
plt.ylabel('PACF')
plt.tight_layout()
plt.show()
```

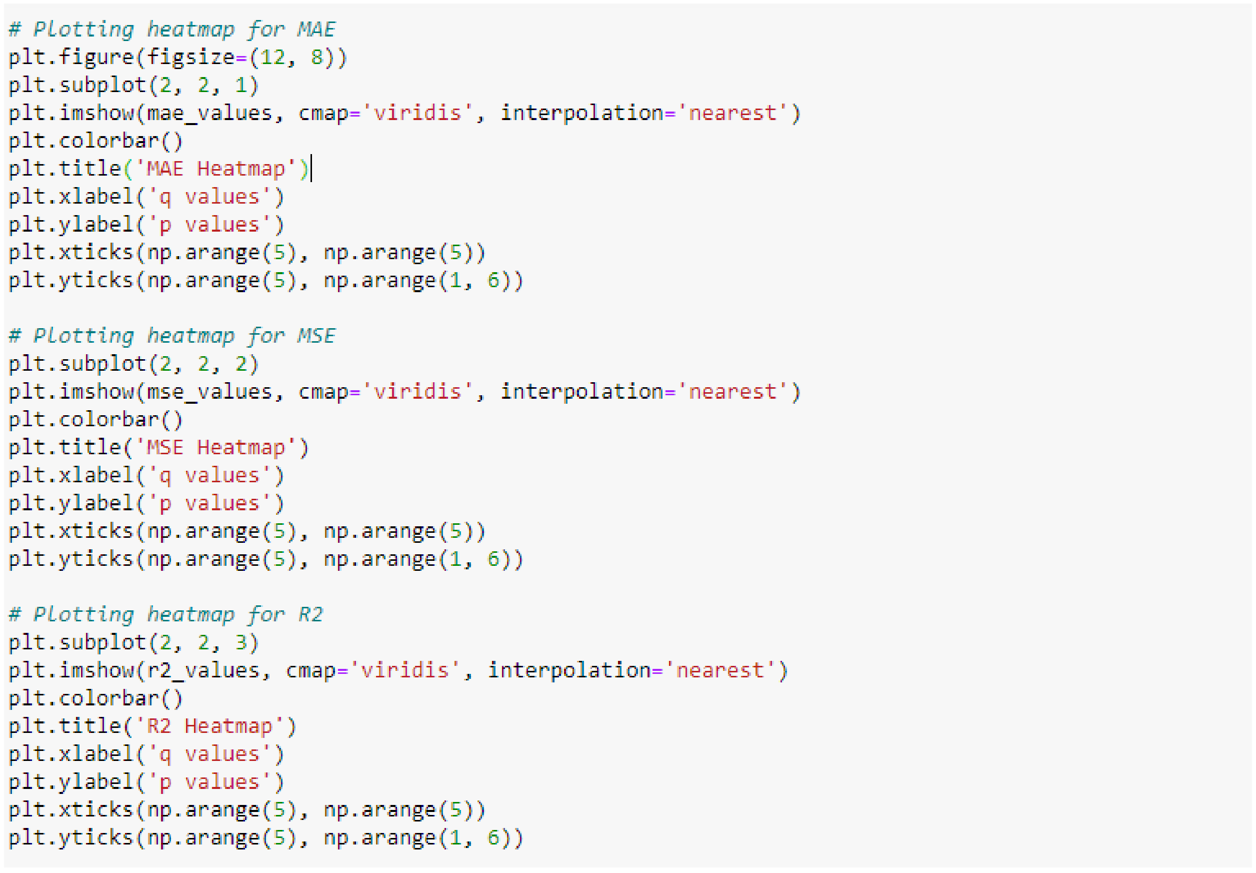
```
from statsmodels.tsa.arima.model import ARIMA
p=2; d=0;q=0
#fit model
model= ARIMA(df.astype(float), order=(p,d,q))#time series that i convert to float and creat a model
model_fit=model.fit()
fig=plt.figure(figsize=(15,5))
plt.plot(df)
plt.plot(model_fit.fittedvalues , color='purple')
plt.title(' prediction over all selected data set for Milano with AR model p equal 2')
```

Task 5, 6, 7a

```
#task5,6 7.a assuming fixed train and test set and changing p and q and calculating the error metrics
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Assuming hourly_counts is a dictionary with hours as keys and booking counts as values
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])
df.set_index('Hour', inplace=True)

# Splitting the data into training and test sets
train = df.iloc[:504]  # For example, first 504 hours for training
test = df.iloc[504:552]  # Following hours for testing

# Initialize a dictionary to store the metrics
metrics = {}
```



```python
# Loop over different values of p and q
for p in range(1, 6):
    for q in range(0, 5):
        try:
            # Fit the ARIMA model
            model = ARIMA(train, order=(p,0,q))
            fitted_model = model.fit(method='innovations_mle')

            # Make predictions
            predictions = fitted_model.predict(start=test.index[0], end=test.index[-1])
            predictions_series = pd.Series(predictions, index=test.index)

            # Calculate metrics
            mae = mean_absolute_error(test['Bookings'], predictions_series)
            mse = mean_squared_error(test['Bookings'], predictions_series)
            r2 = r2_score(test['Bookings'], predictions_series)
            mape = np.mean(np.abs((test['Bookings'] - predictions_series) / test['Bookings'])) * 100

            # Store metrics
            metrics[(p, q)] = {'MAE': mae, 'MSE': mse, 'R2': r2, 'MAPE': mape}
        except Exception as e:
            print(f"Failed to fit ARIMA({p},0,{q}): {e}")

# Print metrics for each (p, q) combination
for (p, q), vals in metrics.items():
    print(f"ARIMA({p},0,{q}) - MAE: {vals['MAE']:.2f}, MSE: {vals['MSE']:.2f}, R2: {vals['R2']:.2f}, MAPE: {vals['MAPE']:.2f}
# Extract metrics for plotting
mae_values = np.zeros((5, 5))
mse_values = np.zeros((5, 5))
r2_values = np.zeros((5, 5))
mape_values = np.zeros((5, 5))
```



```python
# Plotting heatmap for MAE
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.imshow(mae_values, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('MAE Heatmap')
plt.xlabel('q values')
plt.ylabel('p values')
plt.xticks(np.arange(5), np.arange(5))
plt.yticks(np.arange(5), np.arange(1, 6))

# Plotting heatmap for MSE
plt.subplot(2, 2, 2)
plt.imshow(mse_values, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('MSE Heatmap')
plt.xlabel('q values')
plt.ylabel('p values')
plt.xticks(np.arange(5), np.arange(5))
plt.yticks(np.arange(5), np.arange(1, 6))

# Plotting heatmap for R2
plt.subplot(2, 2, 3)
plt.imshow(r2_values, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('R2 Heatmap')
plt.xlabel('q values')
plt.ylabel('p values')
plt.xticks(np.arange(5), np.arange(5))
plt.yticks(np.arange(5), np.arange(1, 6))
```

```python
# Plotting heatmap for MAPE
plt.subplot(2, 2, 4)
plt.imshow(mape_values, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('MAPE Heatmap')
plt.xlabel('q values')
plt.ylabel('p values')
plt.xticks(np.arange(5), np.arange(5))
plt.yticks(np.arange(5), np.arange(1, 6))

plt.tight_layout()
plt.show()
```

Task 7.b, 7.c

```python
#task7.b sliding window
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score

# Assuming hourly_counts is a dictionary with hours as keys and booking counts as values
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])
df.set_index('Hour', inplace=True)

# Define the start of the test set
test_start = 504

# Initialize a list to store results
results = []

# Window size for sliding window
window_size = 24  # Assuming each window represents one day (24 hours)

# Fix values for ARIMA model
p_fixed = 4
d_fixed = 0
q_fixed = 3

# Initialize lists to store plotting data
end_train_values = []
mae_values = []
mape_values = []
r2_values = []
```

```python
# Sliding the window one day at a time
for end_train in range(initial_train_size, test_start, 24):  # Assuming each step slides one day (24 hours)
    train = df.iloc[end_train - window_size:end_train]
    test = df.iloc[end_train:end_train+24]  # Next day for testing

    try:
        # Fit the ARIMA model with fixed values
        model = ARIMA(train, order=(p_fixed, d_fixed, q_fixed))
        fitted_model = model.fit()

        # Make predictions for the next day
        predictions = fitted_model.predict(start=train.index[-1] + 1, end=train.index[-1] + len(test))
        predictions_series = pd.Series(predictions, index=test.index)

        # Calculate metrics
        mae = mean_absolute_error(test['Bookings'], predictions_series)
        mape = np.mean(np.abs((test['Bookings'] - predictions_series) / test['Bookings'])) * 100
        r2 = r2_score(test['Bookings'], predictions_series)

        # Store metrics
        results.append({
            'end_train': end_train,
            'p': p_fixed,
            'd': d_fixed,
            'q': q_fixed,
            'MAE': mae,
            'MAPE': mape,
            'R2': r2
        })
```

```python
        # Store data for plotting
        end_train_values.append(end_train)
        mae_values.append(mae)
        mape_values.append(mape)
        r2_values.append(r2)
    except Exception as e:
        print(f"Failed to fit ARIMA({p_fixed},{d_fixed},{q_fixed}): {e}")

# Convert results to DataFrame for easier analysis
results_df = pd.DataFrame(results)
# Print the results
print(results_df)

# Plotting section
plt.figure(figsize=(12, 8))

# Plot MAE
plt.subplot(3, 1, 1)
plt.plot(end_train_values, mae_values, marker='o', linestyle='-', color='blue')
plt.title('Mean Absolute Error (MAE) Over Sliding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('MAE')

# Plot MAPE
plt.subplot(3, 1, 2)
plt.plot(end_train_values, mape_values, marker='o', linestyle='-', color='orange')
plt.title('Mean Absolute Percentage Error (MAPE) Over Sliding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('MAPE')
```

```python
# Plot R2
plt.subplot(3, 1, 3)
plt.plot(end_train_values, r2_values, marker='o', linestyle='-', color='green')
plt.title('R-squared (R2) Over Sliding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('R2')

plt.tight_layout()
plt.show()
```

```python
#expanding window with arima choice
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score

# Assuming hourly_counts is a dictionary with hours as keys and booking counts as values
df = pd.DataFrame(list(hourly_counts.items()), columns=['Hour', 'Bookings'])
df.set_index('Hour', inplace=True)

# Define the start of the test set
test_start = 504

# Initialize a list to store results
results = []

# Initial training set size
initial_train_size = 336  # Start with the first week for training

# Fix values for ARIMA model
p_fixed = 4
d_fixed = 0
q_fixed = 3

# Initialize lists to store plotting data
end_train_values = []
mae_values = []
mape_values = []
r2_values = []
```

```python
# Expanding the window one day at a time
for end_train in range(initial_train_size, test_start, 24):  # Assuming each step adds one day (24 hours)
    train = df.iloc[:end_train]
    test = df.iloc[end_train:end_train+24]  # Next day for testing

    try:
        # Fit the ARIMA model with fixed values
        model = ARIMA(train, order=(p_fixed, d_fixed, q_fixed))
        fitted_model = model.fit()

        # Make predictions for the next day
        predictions = fitted_model.predict(start=train.index[-1] + 1, end=train.index[-1] + len(test))
        predictions_series = pd.Series(predictions, index=test.index)

        # Calculate metrics
        mae = mean_absolute_error(test['Bookings'], predictions_series)
        mape = np.mean(np.abs((test['Bookings'] - predictions_series) / test['Bookings'])) * 100
        r2 = r2_score(test['Bookings'], predictions_series)

        # Store metrics
        results.append({
            'end_train': end_train,
            'p': p_fixed,
            'd': d_fixed,
            'q': q_fixed,
            'MAE': mae,
            'MAPE': mape,
            'R2': r2
        })
```

```python
        # Store data for plotting
        end_train_values.append(end_train)
        mae_values.append(mae)
        mape_values.append(mape)
        r2_values.append(r2)
    except Exception as e:
        print(f"Failed to fit ARIMA({p_fixed},{d_fixed},{q_fixed}): {e}")

# Convert results to DataFrame for easier analysis
results_df = pd.DataFrame(results)
# Print the results
print(results_df)

# Plotting section
plt.figure(figsize=(12, 16))

# Plot MAE, MAPE, and R2
plt.subplot(4, 1, 1)
plt.plot(end_train_values, mae_values, marker='o', linestyle='-', color='blue', label='MAE')
plt.title('Mean Absolute Error (MAE) Over Expanding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('MAE')
plt.legend()

plt.subplot(4, 1, 2)
plt.plot(end_train_values, mape_values, marker='o', linestyle='-', color='orange', label='MAPE')
plt.title('Mean Absolute Percentage Error (MAPE) Over Expanding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('MAPE')
plt.legend()
plt.subplot(4, 1, 3)
plt.plot(end_train_values, r2_values, marker='o', linestyle='-', color='green', label='R2')
plt.title('R-squared (R2) Over Expanding Window')
plt.xlabel('End of Training Set Index')
plt.ylabel('R2')
plt.legend()

# Plot Actual and Predicted Values
plt.subplot(4, 1, 4)
for i in range(len(all_actual_values)):
    plt.plot(df.index, all_actual_values[i], label=f"Actual (Window {i + 1})", alpha=0.5)
    plt.plot(df.index, all_predicted_values[i], label=f"Predicted (Window {i + 1})", linestyle='dashed', alpha=0.7)

plt.title('Actual vs Predicted Values')
plt.xlabel('Hour')
plt.ylabel('Bookings')
plt.legend()

plt.tight_layout()
plt.show()
```