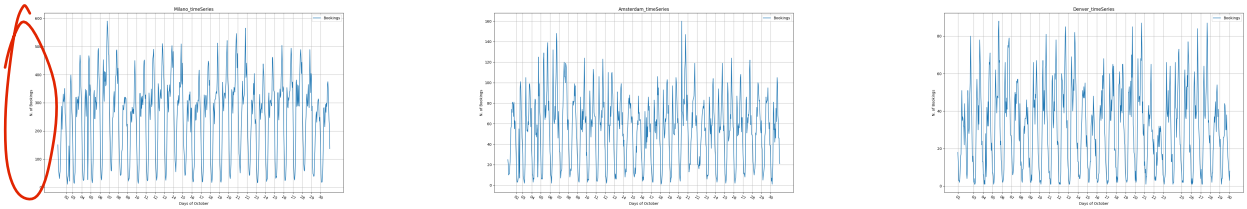
ICT for smart mobility

Kevin Gumina s320054, S.Ten. Federico Urso s316888, Alex Umberto Benedetti s319318

January 6, 2024

1 TASK 1: Prediction using ARIMA models

The first task of this step, was to extract the time series of rentals for each city (i.e., Amsterdam, Milano and Denver), considering a period of 30 days. We decided to modify our observation period with respect to the previous analysis because it was not suitable for our purpose.
We decided to consider the time period of 30 days from 1st of October to 30th of October. We choose this period because the period of December was not suitable since the last days of December may be characterized by strange results due to the Christmas holiday and in the period of November there are missing data in a lot of days.



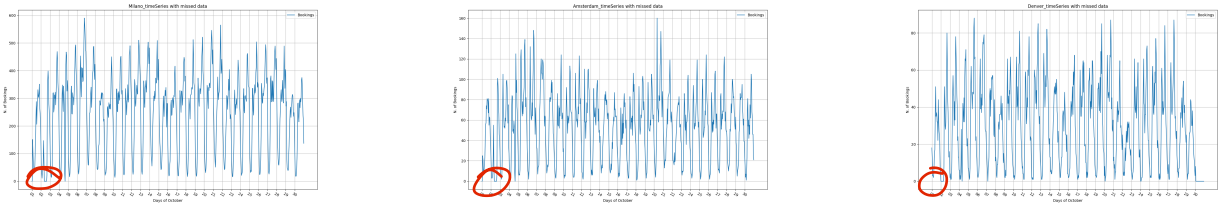
(a) Milano TimeSeries (b) Amsterdam TimeSeries (c) Denver TimeSeries

Figure 1: Original TimeSeries

From figures 1a and 1b it is possible to notice that there are some data of some hours missed; i.e. on the 2nd of October and the day before due to a system error. Instead in Denver it is possible to see from the figure 1c are missed some hours between the 1st and the 3rd and between the 23rd and the 25th of October.

2 TASK 2: Clean Missing Values

ARIMA models works only with regular time series without missed data , for this reason it will be important to fit the data. To fit the data firstly we had identified the missing data and after we replaced those missing data with zero. As it is possible to see from figure 2a and figure 2b now the missing data before the 2nd of October have been replaced with zero values. In fact, some lines now are exactly in zero. This cleaning works also for Denver as it is possible to see from the figure 2c.



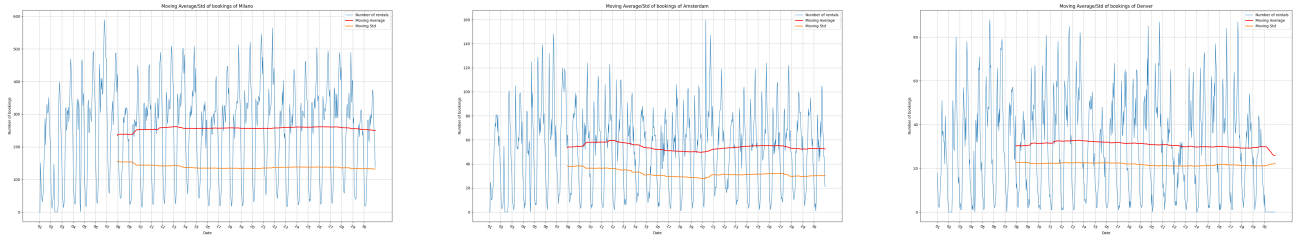
(a) Milano TimeSeries (b) Amsterdam TimeSeries (c) Denver TimeSeries

Figure 2: TimeSeries with missed data

From figures 2 it is possible to see that the cleaning process has been effective since it doesn't affect the periodicity.

3 TASK 3: Stationarity Checking

In order to check the stationarity or not of a time series it is possible to plot the rolling statistics: average and standard deviation over the time series that we have plotted in figures 1. A process can be defined stationary if the mean and standard deviation are costant over time.



(a) Milano Stationarity (b) Amsterdam Stationarity (c) Denver Stationarity

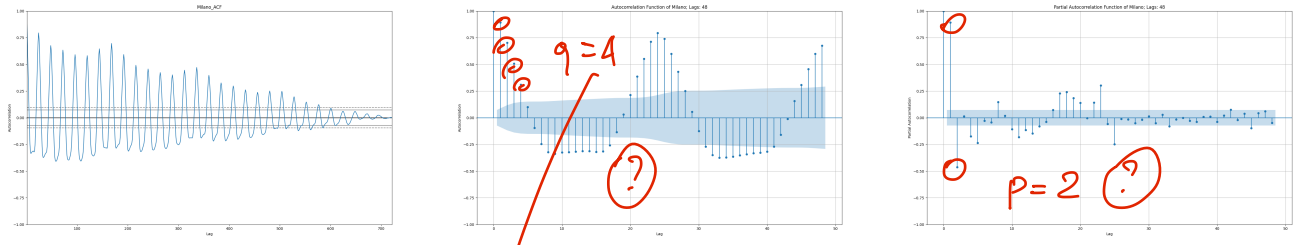
Figure 3: Stationarity

From the figure 3 is possible to understand that both average and standard deviation for all the cities are almost constant over the time. Therefore we deduced that the process is stationary. Due to this stationarity it is possible to set the parameter d=0. Hence we will have a model (p,0,q) which represent an ARMA model.

4 TASK 4: ACF and PACF

In this section we calculated the Autocorrelation function **(ACF)** and the Partial Autocorrelation Function **(PACF)** which is useful to choose an initial value for the hyperparameters of the model. We have three parameters :

- **p**: The number of lag observations included in the model,also called the lag order.
- **d**: The number of times that the raw observations are differenced,also called the degree of differencing. In our case it is zero as we see before.
- **q**: The size of the moving average window, also called the order of moving average.



(a) Autocorrelation function (b) ACF - zoom (c) PACF - zoom

Figure 4: Milano autocorrelation and partial autocorrelation

From the ACF 4a it is possible to notice the 24 hours periodicity and also the weekly periodicity. Therefore as seen before d is zero, now we have to set the parameter p and q by looking at the figures. In fact looking at figure 4b it's possible to determine q=3 Instead looking at figure 4c it's possible to determine p=1

The choosen method to select the values for p an q is based on using the ACF and the PACF's plots. The method to select the best value for P follows these steps: Consider the PACF's plot,ignore the 0-th one and count the first n-elements until the closer lag to the dense part is met. Once that lag is met, the $n-1$ formula is used, where n represents the counted elements;

The method to choice the best value for Q is the basically the same but it must be used the ACF plot so the steps are:

Consider the ACF's plot,ignore the 0-th one,count the first n-elements until the closer lag to the dense part is met. Once that lag is met, the $n-1$ formula is used, where n represents the counted elements;

Therefore the choice of parameters p and q are:

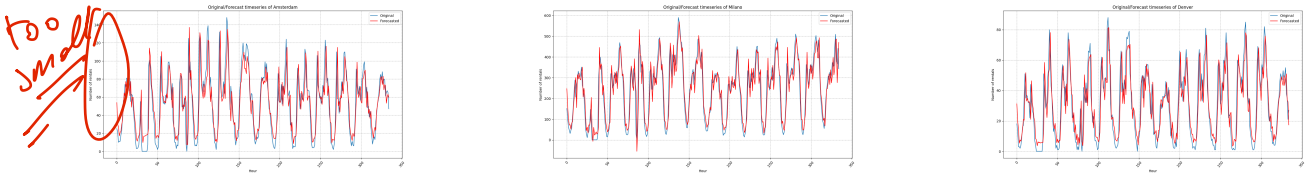
- **Milano(1,0,3)**
- **Amsterdam (2,0,3)**
- **Denver (2,0,4)**

5 TASK 5: Data splitting

After the choice of the initial parameter p,d and q we split the dataset in order to consider a time interval of one week out of 30 days as unit of analysis. In fact we use the first week to train model and the second week to test the model. Since our samples are aggregated per hour, we consider as inital guess N=168 samples for training. Moreover we decide to use an expanding window learning strategy.

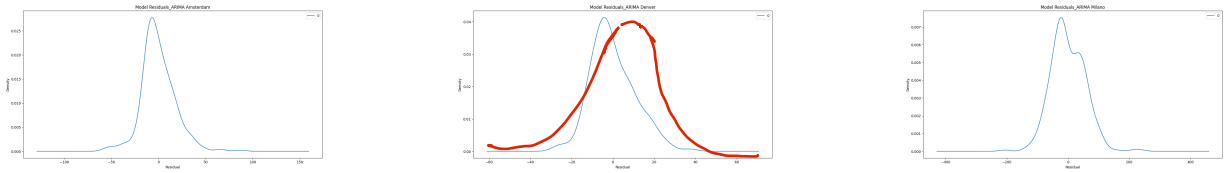
6 TASK 6: Model training

After set the configuration explained in the preceding sections, models have been trained individually for each city. The initial fitted model can be seen in figures 5 and from figures 6 is possible to visualize the error distribution of the residuals. We can observe the existence of residual side-lobes, indicating the persistence of biases. Nevertheless, the observed error exhibits characteristics akin to White Gaussian Noise, aligning with our assumed Autoregressive Moving Average (ARMA) model.



(a) Amsterdam (b) Milano (c) Denver

Figure 5: Starting model fitting



(a) Residual Amsterdam (b) Residual Denver (c) Residual Milano

Figure 6: Residuals for each city

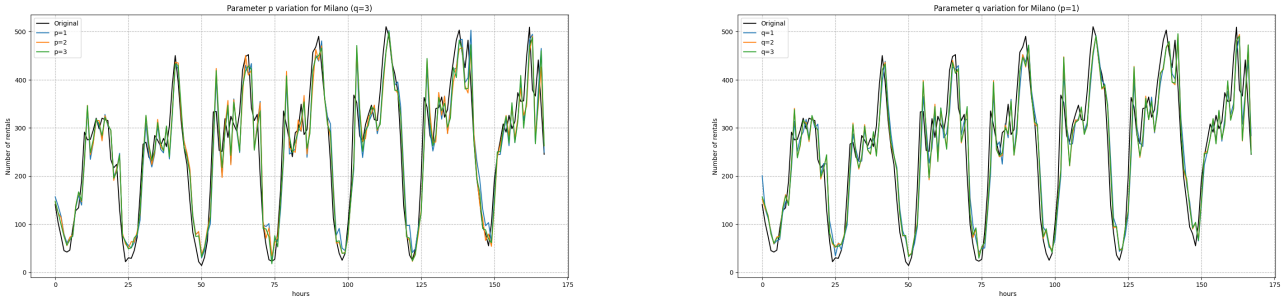
In Table 1, it is possible to observe the calculated **Mean Absolute Percentage Error** (MAPE), which is percentage-based and useful for expressing errors relative to the actual values; the **Mean Square Error** (MSE), which, by squaring the errors, penalizes larger errors more significantly. These are important metrics that define the accuracy of a forecasting method; and the **Mean Absolute Error** (MAE), easy to interpret as it is in the same unit as the data but does not account for the direction of errors; and

City	Amsterdam	Milano	Denver
MAPE	23.010	16.104	25.522
MSE	400.816	3470.423	134.044
MAE	14.378	46.723	8.965

Table 1: Error metrics for the initial model

7 TASK 7: Parameters tuning

Tuning a model involves adjusting a parameter's value to enhance its performance. In this case, a grid search was conducted for parameters p and q, keeping the value of N constant and utilizing only the sliding-window training approach. Grid search is a technique for fine-tuning a model by exploring all possible combinations of parameter values and assessing the model's performance for each combination. For each pair of values (p, q) within the range of 1-3, various ARIMA models were trained, with N set to 1 week of training samples and 1 week of test samples.



(a) Variation p for Milano

(b) Variation q for Milano

Figure 7: Comparison of predictions with different p and q values - MILANO

Various types of errors, including mean absolute percentage error (MAPE), and mean square error (MPE), are calculated. However, MAPE is specifically employed for selecting optimal parameters as it enables a direct comparison of results across the three cities. The heatmaps in figures 8 illustrates MAPE and MPE for all parameter combinations. Notably, tuning for these cities involved different ranges for p and q. Some combinations were excluded from the heatmaps due to the occurrence of the error "LinAlgError: LU decomposition error". Heatmaps for Amsterdam and Denver can be found in the appendix (Fig: 14, 15).

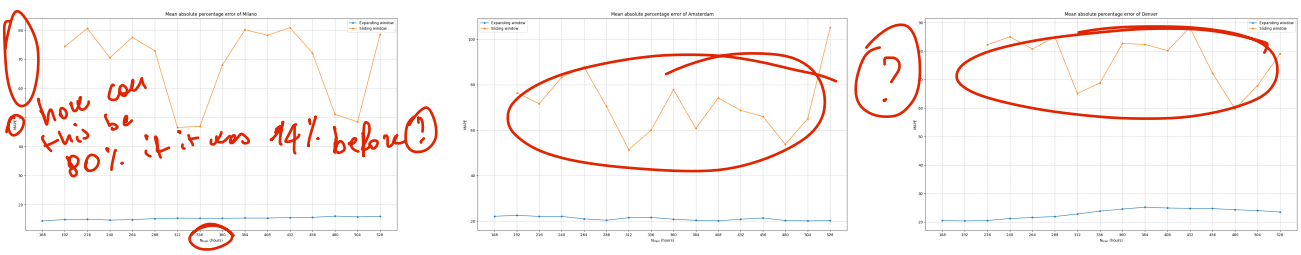
(a) MAPE HEATMAP of MILANO

(b) MPE HEATMAP of MILANO

Figure 8: HEATMAP - different combinations of parameters

The most effective parameter combination for Milano is identified as p = 5 and q = 4, determined by the lower MAPE. The best combinations for Amsterdam and Denver are ARIMA(2,0,5) and ARIMA(3,0,5) respectively.

Once the optimal values for p and q in the ARIMA models were determined, the investigation focuses on assessing the impact of Training Size (N) and the learning strategy by testing both the Sliding Window and Expanding Window approaches. Unlike the sliding window technique, which utilizes overlapping subsets of a fixed size, the expanding window strategy involve training a model on progressively larger subsets of data. To achieve this objective, a grid search is conducted by training several models, each corresponding to a unique combination of Training Size and Learning Strategy. For each of these models, the MAPE is evaluated, and the parameters leading to the best result, i.e., the lowest MAPE, are selected.



(a) Milano (5,0,4)

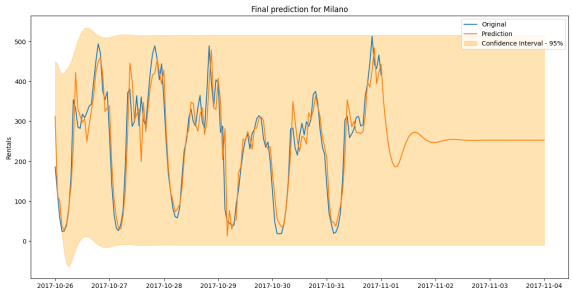
(b) Amsterdam (2,0,5)

(c) Denver (3,0,5)

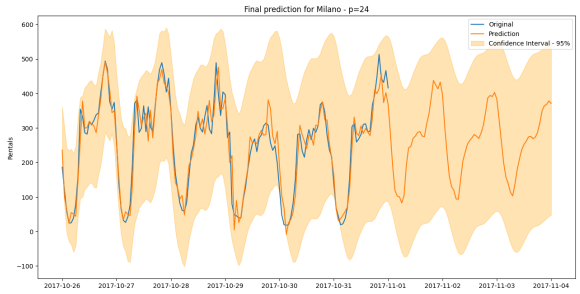
Figure 9: variation of N

From figures 9 it is possible to see that the best N, found with expanding window for all the three cities,

are: for **Milano N=168**, for **Amsterdam N=504** and for **Denver N=192**.
Once the optimal ARIMA model, best N and the choosen windows tecnique are selected, the final model were implemented and their performance were tested as it is possible to see from figures 10.



(a) Final ARIMA prediction of Milano



(b) Final ARIMA prediction of Milano - p=24

Figure 10: Final ARIMA predictions of MILANO

City	Amsterdam	Milano	Denver
MAPE	20.048	14.428	20.433
MSE	183.465	2213.271	79.004
MAE	10.510	36.906	6.528

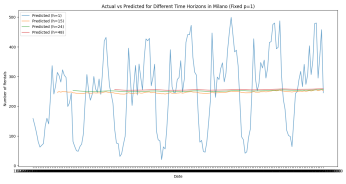
Table 2: Error metrics for final model

As it is possible to see from table 2 the error metrics decrease after the setting of the best parameters get from the task 7.

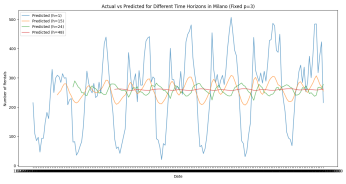
8 TASK 8: Impact of prediction's time horizon

In this task it has been used the time horizon which refers to the lenght of time into the future to consent the model to make predictions. The code implements an ARIMA model for time series forecasting, assessing performance across different time horizons. The series is split into training and test sets, with the model trained on historical data. Predictions are made for each specified time horizon, calculating errors such as MAE and MAPE. The results are visualized in a graph comparing predictions to actual data. This approach allows for evaluating the model's effectiveness across various time horizons, providing a comprehensive view of its performance.

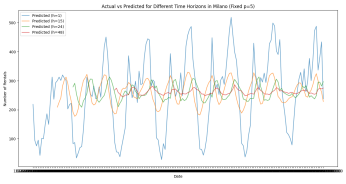
By varying the parameter p in the ARIMA model, we observe a noticeable impact on the predictions. As p increases, the lines appear to deviate less from the actual values, suggesting that higher values of p allow the model to capture and incorporate more historical information. This can lead to predictions that exhibit smoother trends and better alignment with the underlying patterns in the time series data



(a) Horizon prediction with p=1



(b) Horizon prediction with p=3



(c) Horizon prediction with p=5

Figure 11: Final Milano horizon prediction with different p

9 Appendices

```
import pymongo as pm
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from datetime import datetime, timedelta
import calendar
import gmplot
import math
import mne
import io
import json
from statsmodels.tsa.arima.model import ARIMA
import seaborn as sns
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
from sklearn.metrics import (mean_squared_error, mean_absolute_error, r2_score)
import os



date_init = datetime.strptime('2017-10-01T00:00:00', '%Y-%m-%dT%H:%M:%S')
date_finish = datetime.strptime('2017-10-31T23:59:59', '%Y-%m-%dT%H:%M:%S')
#date_init_Denver = datetime.strptime('2017-10-01T06:00:00','%Y-%m-%dT%H:%M:%S')
#date_finish_Denver = datetime.strptime('2017-10-30T07:59:59','%Y-%m-%dT%H:%M:%S')
init_unix = (date_init - datetime(1970, 1, 1)).total_seconds()
finish_unix = (date_finish - datetime(1970, 1, 1)).total_seconds()

timeSeries = {}
for city in cities:
timeSeries['Bookings{0}'.format(city)] = Bookings.aggregate([
    {"$match": {
        "city": city,
        "init_time": {"$gte": init_unix, "$lte": finish_unix}
    }
    },
    {"$project": {
        "duration": {
            "$subtract": ["$final_time", "$init_time"]
        },

        "or_de": {
            "$ne": [{
                "$arrayElemAt": ["$origin_destination.coordinates", 0]
            },
                {
                    "$arrayElemAt": ["$origin_destination.coordinates", 1]
                }
            ]
        },

        "dow": {"$dayOfYear": "$init_date"},
        "hour": {"$hour": "$init_date"}
    }},
    {"$match": {
        "duration": {
            "$gte": 5 * 60, "$lte": 150 * 60
        },
        "or_de": True
    }
    },

    {"$group": {
        "_id": {"dow": "$dow", "hour": "$hour"},
        "totOFbookings": {"$sum": 1}

    }
    },
    {"$sort": {
        "_id": 1

    }

    }])
```

9.1 TASK 1

```
def timeSeries_day(title, Bookings):
    listBookings = list(Bookings)
    DFBookings = pd.DataFrame(listBookings)
    labels = []
    ticks = []

    for i in range(DFBookings.shape[0] - 1):
        if (DFBookings["_id"][i]["hour"] == 0) or (
                (DFBookings["_id"][i + 1]["dow"] - DFBookings["_id"][i]["dow"]) != 0) and (
                (DFBookings["_id"][i + 1]["dow"] - DFBookings["_id"][i]["dow"]) != 1):
            ticks.append(i)
            formatted_date = day_of_year_to_date(DFBookings["_id"][i + 1]["dow"], 2017)
            labels.append(formatted_date)

    plt.figure(figsize=(10, 6))
    plt.xlabel("Days of October")
    plt.ylabel("N. of Bookings")
    plt.title(title)
    plt.plot(DFBookings["totOFbookings"], label="Bookings")
    plt.xticks(ticks=ticks,
               labels=labels,
               rotation=-30)
    plt.legend(loc='best')
    plt.grid(True, which="both")
    plt.show()

    DFBookings.columns = ['_id', 'totOFbookings']
    df_result = DFBookings.to_csv(index=False)  # Save to CSV without index

    return df_result


df = time_series_df = timeSeries_day(city + "_timeSeries", timeSeries['Bookings' + city])
```

9.2 TASK 2

```
def add_miss(df, title):
    dow_hour_combinations = [(dow, hour) for dow in range(275, 305) for hour in range(24)]
    new_rows = []

    #print(df)

    for dow, hour in dow_hour_combinations:
        if not any(((eval(entry['_id'])['dow'] == dow) and (eval(entry['_id'])['hour'] == hour
    )) for _, entry in
                   df.iterrows()):
            new_rows.append({'_id': {'dow': dow, 'hour': hour}, 'totOFbookings': 0})

    new_df = pd.DataFrame(new_rows)
    df = pd.concat([df, new_df], ignore_index=True)
    #print(df)

    df[['dow', 'hour']] = df['_id'].apply(
        lambda x: pd.Series([x['dow'], x['hour']] if isinstance(x, dict) else x.split(',')))

    df['dow'] = df['dow'].apply(lambda x: int(x.split(':')[1].strip('}{ ')) if isinstance(x,
    str) else int(x))
    df['hour'] = df['hour'].apply(lambda x: int(x.split(':')[1].strip('}{ ')) if isinstance(x,
     str) else int(x))

    df = df.sort_values(by=['dow', 'hour']).reset_index(drop=True)
    #print(df)

    labels = []
    ticks = []

    for i in range(df.shape[0] - 1):
        if (df["hour"][i] == 0) or (
                (df["dow"][i + 1] - df["dow"][i]) != 0) and (
                (df["dow"][i + 1] - df["dow"][i]) != 1):
            ticks.append(i)
            print(df["dow"][i + 1])
            formatted_date = day_of_year_to_date(int(df["dow"][i + 1]), 2017)
```

```
            labels.append(formatted_date)
    plt.figure(figsize=(10, 6))
    plt.xlabel("Days of October")
    plt.ylabel("N. of Bookings")
    plt.title(title)
    plt.plot(df["totOFbookings"], label="Bookings")
    plt.xticks(ticks=ticks,
            labels=labels,
            rotation=-30)
    plt.legend(loc='best')
    plt.grid(True, which="both")
    plt.show()

    return df

df_miss = add_miss(pd.read_csv(io.StringIO(df)), city + "_timeSeries with missed data")
```

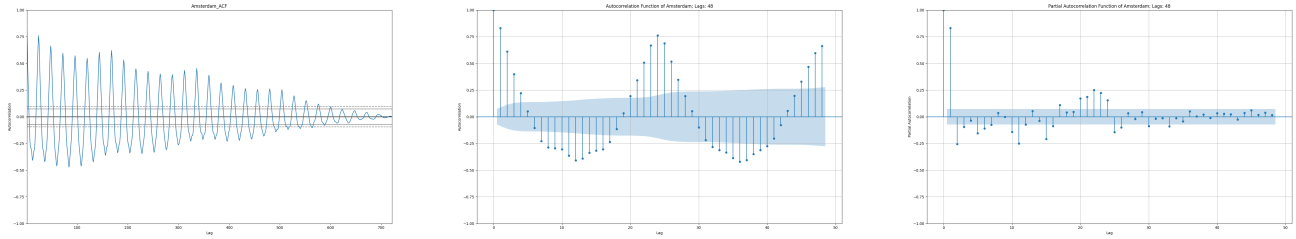
9.3 TASK 3

```
def stationarity(df, title):
    labels = []
    ticks = []
    for i in range(df.shape[0] - 1):
        if (df["hour"][i] == 0) or (
                (df["dow"][i + 1] - df["dow"][i]) != 0) and (
                (df["dow"][i + 1] - df["dow"][i]) != 1):
            ticks.append(i)
            print(df["dow"][i + 1])
            formatted_date = day_of_year_to_date(int(df["dow"][i + 1]), 2017)
            labels.append(formatted_date)

    df['MA'] = df['totOFbookings'].rolling(24 * 7).mean()  # Moving average
    df['MS'] = df['totOFbookings'].rolling(24 * 7).std()  # Moving std
    plt.figure(constrained_layout=True)
    plt.plot(df['totOFbookings'], linewidth=1, label='Number of rentals')
    plt.plot(df['MA'], linewidth=2, color='r', label='Moving Average')
    plt.plot(df['MS'], linewidth=2, label='Moving Std')
    plt.title('Moving Average & Standard deviation of bookings of ' + title)
    plt.xlabel('Date')
    plt.ylabel('Number of bookings')
    plt.xticks(ticks=ticks,
               labels=labels,
               rotation=-30)
    plt.grid(linestyle='--', linewidth=0.8)
    plt.legend()
    plt.savefig(title + 'MEAN_&_STD.png', format='png')
    plt.show()

    return df

df_stat = stationarity(df_miss, city)
```

9.4 TASK 4

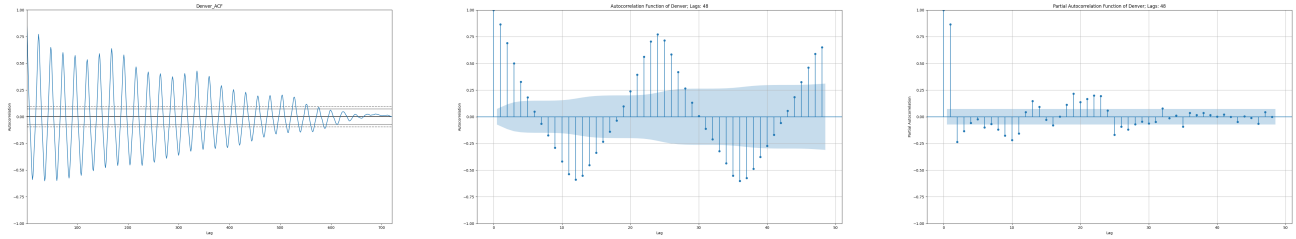
```
def autocorrelation(df, title):
    plt.figure(constrained_layout=True)
    pd.plotting.autocorrelation_plot(df["totOFbookings"])
    plt.title(title + '_ACF')
    plt.grid()
    plt.show()

    n_lags = 48
    fig, ax = plt.subplots(constrained_layout=True)
    plot_acf(df["totOFbookings"], ax=ax, lags=n_lags)
    plt.title(title + '_ACF - Lags: %d' % n_lags)
    plt.grid(which='both')
    plt.xlabel("Lag")
    plt.ylabel("Autocorrelation")
    plt.show()

    n_lags = 48
    fig, ax = plt.subplots(constrained_layout=True)
    plot_pacf(df["totOFbookings"], ax=ax, lags=n_lags)
```

```
    plt.title(title + '_PACF - Lags: %d' % n_lags)
    plt.grid(which='both')
    plt.xlabel("Lag")
    plt.ylabel("Partial Autocorrelation")
    plt.show()

autocorrelation(df_miss, city)
```



(a) Autocorrelation function (b) ACF - zoom (c) PACF - zoom

Figure 12: AMSTERDAM autocorrelation and partial autocorrelation



(a) Autocorrelation function (b) ACF - zoom (c) PACF - zoom

Figure 13: DENVER autocorrelation and partial autocorrelation

9.5 TASK 5

```
def split(df):
    data = df["totOFbookings"].values.astype(float)
    N = 7 * 24
    train_set, test_set = data[0:N], data[N:(2 * N)]

    return train_set, test_set, data

train, test, data = split(df_miss)
```

9.6 TASK 6

```
def model_training(train, test, data, title):
    p=5
    q=4
    order = (p, 0, q)
    model = ARIMA(train.astype(float), order=order)
    model_fit = model.fit(method='statespace')

    print(model_fit.summary())

    plt.plot(train, label='Original')
    plt.plot(model_fit.fittedvalues, color="red", label='Forecasted')
    plt.title('Original/Forecast timeseries of ' + title + ' - Training phase')
    plt.xlabel("Hour")
    plt.ylabel("Number of rentals")
    plt.xticks(rotation=50)
    plt.legend(loc='upper right')
    plt.grid(linestyle='--', linewidth=0.8)
    plt.show()

    mae = mean_absolute_error(train[0:len(model_fit.fittedvalues)], model_fit.fittedvalues)
```

```
    mape = mae / np.mean(train[0:len(model_fit.fittedvalues)]) * 100
    print("TRAIN DATASET : (%i,0,%i) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f -- MAPE: %.3f
    " % (1, 3, mean_absolute_error(train[0:len(model_fit.fittedvalues)], model_fit.
    fittedvalues), mean_squared_error(train[0:len(model_fit.fittedvalues)], model_fit.
    fittedvalues), r2_score(train[0:len(model_fit.fittedvalues)], model_fit.fittedvalues),mape
    ))

    history = train.astype(float)
    predictions = []
    for t in range(0, len(test)):
        model = ARIMA(history, order=order)
        model_fit = model.fit(method='statespace')
        output = model_fit.forecast()
        yhat = output[0]
        predictions.append(yhat)
        obs = test[t]
        history = np.append(history, obs)  # expanding window

    plt.plot(test, label='Original')
    plt.plot(predictions, color="red", label='Forecasted')
    plt.title('Original/Forecast timeseries of ' + title + ' - Testing phase')
    plt.xlabel("Hour")
    plt.ylabel("Number of rentals")
    plt.xticks(rotation=50)
    plt.legend(loc='best')
    plt.grid(linestyle='--', linewidth=0.8)
    plt.show()

    mae = mean_absolute_error(test, predictions)
    mape = mae / np.mean(test[0:len(model_fit.fittedvalues)]) * 100
    print(str(mae) + "      " + str(mape))

    plt.plot(data[0:len(model_fit.fittedvalues)], label='Original')
    plt.plot(model_fit.fittedvalues, color="red", label='Forecasted')
    plt.title('Original/Forecast timeseries of ' + title)
    plt.xlabel("Hour")
    plt.ylabel("Number of rentals")
    plt.xticks(rotation=50)
    plt.legend(loc='upper right')
    plt.grid(linestyle='--', linewidth=0.8)

    residuals = pd.DataFrame(model_fit.resid)
    residuals.plot()
    residuals.plot(kind='kde')
    plt.title('Model Residuals_ARIMA ' + title)
    plt.xlabel("Residual")
    plt.ylabel("Density")
    plt.show()

model_training(train, test, data, city)
```

9.7 Task 7

```
def variation(df,title,q,d):
    data=df['totOFbookings'].values.astype(float)
    train,test= data[0:7*24], data [7*24:(2*7*24)]
    len_test=len(test)
    p_var=(1,2,3)

    predictions = np.zeros((len(p_var),len_test))
    results = {"p": [], "d": [], "q": [], "mse": [], "mae": [], "mape": []}
    try:
        for p in p_var:
            print('Testing ARIMA order (%i, 0, %i)' % (p,q))
            train, test = data[0:7*24], data[7*24:(7*24+len_test)]
            history = [x for x in train]
            for t in range(0, len_test):
                model = ARIMA(history, order= (p, d, q))
                model_fit = model.fit( method='statespace')
                output = model_fit.forecast()
                yhat = output[0]
                predictions[p_var.index(p)][t] = yhat
                obs = test[t]
                history.append(obs) #expanding window
                history=history[1:]
```

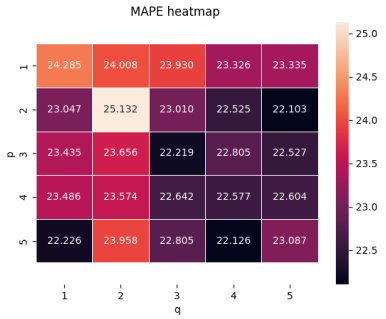
```
    except Exception as e:
        print(f"Si e' verificata un'eccezione di tipo {type(e)._name_}{str(e)}")
        pass

    plt.plot(test,color = 'black', label = "Original")
    for p in p_var:
        print("(%i,0,2) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f" %(p, mean_absolute_error(test,predictions[p_var.index(p)]), mean_squared_error(test,predictions[p_var.index(p)]), r2_score(test,predictions[p_var.index(p)])))


        mae = mean_absolute_error(test,predictions[p_var.index(p)])
        mape = mae/np.mean(test)*100
        results["p"].append(p)
        results["d"].append(0)
        results["q"].append(q)    results["mse"].append(mean_squared_error(test,predictions[p_var.index(p)]))    results["mae"].append(mean_absolute_error(test,predictions[p_var.index(p)]))
        results["mape"].append(mape)
        plt.plot(predictions[p_var.index(p)],label='p=%i' %p)

    plt.title('Parameter p variation for '+ city + ' (q=3)')
    plt.xlabel(" hours")
    plt.ylabel("Number of rentals")
    plt.legend(loc='best')
    plt.grid(linestyle = '--', linewidth=0.8)
    plt.show()


    p = 2
    MA_orders = (1, 2, 3)
    predictions = np.zeros((len(p_var), len_test))
    for q in MA_orders:
        print('Testing ARIMA order (%i, 0, %i)' % (p, q))
        train, test = data[0:7*24], data[7*24:(7*24 + len_test)]
        history = [w for w in train]
        for t in range(0, len_test):
            model = ARIMA(history, order=(p, d, q))
            model_fit = model.fit()
            output = model_fit.forecast()
            yhat = output[0]
            predictions[p_var.index(q)][t] = yhat
            obs = test[t]
            history.append(obs)

    plt.plot(test, color="black", label="Original")
    for q in MA_orders:
        print("(%i,0,%i) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f" % (p, q, mean_absolute_error(test,predictions[p_var.index(q)]), mean_squared_error(test, predictions[p_var.index(q)]), r2_score(test, predictions[p_var.index(q)])))

        mae = mean_absolute_error(test, predictions[p_var.index(p)])
        mape = mae / np.mean(test) * 100
        results["p"].append(p)
        results["d"].append(0)
        results["q"].append(q)
        results["mse"].append(mean_squared_error(test, predictions[p_var.index(q)]))
        results["mae"].append(mean_absolute_error(test, predictions[p_var.index(q)]))
        results["mape"].append(mape)
        plt.plot(predictions[p_var.index(q)], label='q=%i' % q)

    plt.title('Parameter q variation for '+ city + " (p=1)")
    plt.xlabel(" hours")
    plt.ylabel("Number of rentals")
    plt.legend(loc='best')
    plt.grid(linestyle = '--', linewidth=0.8)
    plt.show()



def variation_p_d(df):
    data = df['totOFbookings'].values.astype(float)
    train, test = data[0:7 * 24], data[7 * 24:(2 * 7 * 24)]
    test_len = len(test)

    N = 7 * 24
```

```
    train, test = data[0:N], data[N:(2*N)] test_len = len(test)
    lag_orders = (1, 2,3,4,5)
    MA_orders = (1, 2,3,4,5)
    # predictions = np.zeros((len(lag_orders), test_len))
    predictions = np.zeros(((len(lag_orders) * len(MA_orders)), test_len))
    results = {"p": [], "d": [], "q": [], "mse": [], "mae": [], "mape": [], "mpe": []}
    combinations = range(0, (len(lag_orders) * len(MA_orders)))

    comb = 0
    for p in lag_orders:
        for q in MA_orders:
            print('Testing ARIMA order (%i, %i, %i)' % (p, 0, q))
            train, test = data[0:N], data[N:(N + test_len)]
            history = [x for x in train]
            try:
                for t in range(0, test_len):
                    model = ARIMA(history, order=(p, 0, q))
                    model_fit = model.fit( method='statespace')
                    output = model_fit.forecast()
                    yhat = output[0]
                    predictions[comb][t] = yhat
                    obs = test[t]
                    history.append(obs)  make sliding window
                    # history = history[1:]

                print("(%i,%i,%i) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f" % (p, 0, q,
    mean_absolute_error(test, predictions[comb]), mean_squared_error(test, predictions[comb]),
     r2_score(test, predictions[comb])))

                mae = mean_absolute_error(test, predictions[comb])
                mape = mae / np.mean(test) * 100
                adder = [(a - b) / a for a, b in zip(test, predictions[comb])]
                mpe = (100 / test_len) * np.sum(adder)
                results["p"].append(p)
                results["d"].append(0)
                results["q"].append(q)
                results["mse"].append(mean_squared_error(test, predictions[comb]))
                results["mae"].append(mean_absolute_error(test, predictions[comb]))
                results["mape"].append(mape)
                results["mpe"].append(mpe)
                comb += 1
            except:
                pass

    return results


def heat_map (results):
    results_df = pd.DataFrame(results)
    print(results_df)

    plt.figure()
    heat_df_mpe = results_df.pivot(index='p', columns='q', values='mape')
    ax = sns.heatmap(heat_df_mpe, annot=True, linewidths=.5, fmt='.3f')
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom + 0.5, top - 0.5)
    plt.title('MAPE heatmap ')
    plt.show()

    # MPE
    plt.figure()
    heat_df_mpe = results_df.pivot(index='p', columns='q', values='mpe')
    ax = sns.heatmap(heat_df_mpe, annot=True, linewidths=.5, fmt='.2f')
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom + 0.5, top - 0.5)
    plt.title('MPE heatmap ')
    plt.show()


    best = results_df["mape"].idxmin()
    p = results_df.loc[best]['p'].astype(int)
    d = 0
    q = results_df.loc[best]['q'].astype(int)
    order = (p, d, q)
    print("BEST: ", order)

```

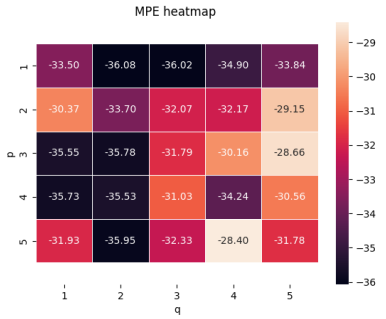
```
    return order, p,q


def N_variation (test,p,q):
    d = 0
    order = (p,d,q)
    results = {"N": [], "window": [], "mse": [], "mae": [], "mape": [], "mpe": []}
    comb = 0
    train_size = [24 * x for x in range(7, 23)]
    test_len = len(test)
    predictions = np.zeros(((len(train_size) * 2), test_len))
    window = [0, 1]
    for N in train_size:
        for w in window:
            try:
                if w == 0:
                    a = 'Expanding Window'
                else:
                    a = 'Sliding Window'
                print(f'Testing ARIMA best order, size: {N} hours and {a}')
                train, test = data[0:N], data[N:(N + test_len)]

                history = [x for x in train]

                for t in range(0, test_len):
                    model = ARIMA(history, order=order)
                    model_fit = model.fit( method='statespace')
                    output = model_fit.forecast()

                    yhat = output [0]

                    predictions[comb][t] = yhat
                    obs = test[t]
                    if w == 0:
                        history.append(obs)  # expanding window
                    else:
                        history = history[1:]  # to make sliding window
                print("(%i,%i,%i) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f" % (p, d, q,
    mean_absolute_error(test, predictions[comb]), mean_squared_error(test, predictions[comb]),
     r2_score(test, predictions[comb])))

                mae = mean_absolute_error(test, predictions[comb])
                mape = mae / np.mean(test) * 100
                adder = [(a - b) / a for a, b in zip(test, predictions[comb])]
                mpe = (100 / test_len) * np.sum(adder)
                results["N"].append(N)
                results["window"].append(w)
                results["mse"].append(mean_squared_error(test, predictions[comb]))
                results["mae"].append(mean_absolute_error(test, predictions[comb]))
                results["mape"].append(mape)
                results["mpe"].append(mpe)
                comb += 1
            except:
                pass


    results = pd.DataFrame(results)
    mape_expanding = results.pivot(index='N', columns='window', values='mape')
    print(mape_expanding)
    plt.figure(constrained_layout=True)
    plt.plot(mape_expanding, linestyle='-', marker='o', markersize=4)
    plt.title('Mean absolute percentage error of ')
    plt.xticks(train_size)
    plt.xlabel(r'N$_{\mathrm{train}}$' + ' (hours)')
    plt.ylabel("MAPE")
    plt.legend(["Expanding window", "Sliding window"])
    plt.grid(linestyle = '--', linewidth=0.8)
    plt.show()

    best = results["mape"].idxmin()
    N = results.loc[best]['N'].astype(int)
    window = results.loc[best]['window'].astype(int)
    if window == 0:
        window = 'Expanding Window'
    else:
        window = 'Sliding Window'
```

```
    print("BEST: N: %d, window: %s " % (N, window))
    return N


def testing_model(N, test, order, p, q, df, city):
    train, test = df['totOFbookings'][0:N], df['totOFbookings'][N:(N + len(test))]

    history = train.astype(float)
    predictions = []
    for t in range(0, len(test)):
        model = ARIMA(history, order=order)
        model_fit = model.fit(method='statespace')
        output = model_fit.forecast()
        yhat = output[0]
        predictions.append(yhat)
        obs = test[t]
        history = np.append(history, obs)  # expanding window
        plt.plot(predictions, color = "red", label='Forecasted')

    plt.plot(test, label='Original')
    plt.title('Original/Forecast timeseries of '+city+' - Testing phase')
    plt.xlabel("Date")
    plt.ylabel("Number of rentals")
    plt.xticks(rotation=50)
    plt.legend(loc='best')
    plt.grid(linestyle = '--', linewidth=0.8)
    plt.show()

    mae = mean_absolute_error(test, predictions)
    mape = mae / np.mean(test) * 100
    print("TEST DATASET : (%i,0,%i) model => MAE: %.3f -- MSE: %.3f -- R2: %.3f -- MAPE: %.3f"
     % (p, q, mean_absolute_error(test, predictions), mean_squared_error(test, predictions),
    r2_score(test, predictions), mape))

    print(df.columns)
    train, test = df['totOFbookings'][-N:], df['totOFbookings'][-len(test):]
    history = [x for x in train]
    predictions = np.zeros((len(train), len(test)))
    model = ARIMA(train.astype(float), order=order)
    model_fit = model.fit(method='statespace')

    t_start = pd.to_datetime("2017-11-15 00:00:00")
    t_end = pd.to_datetime("2017-10-31 00:00:00")
    past = pd.Timedelta(days=5)
    future = pd.Timedelta(days=4)

    predict = model_fit.predict(t_end - past, t_end + future)

    df.index = pd.to_datetime(df.index)
    fig, ax = plt.subplots(1, figsize=(15, 5))
    ax.plot(df['totOFbookings'].loc[t_end - past:], label='Original')
    ax.plot(predict, label='Prediction')
    confidence_interval = model_fit.get_forecast(steps=len(predict)).conf_int(alpha=0.05)
    ax.fill_between(predict.index, confidence_interval.iloc[:, 0], confidence_interval.iloc[:,
     1], color='orange', alpha=0.3, label='Confidence Interval - 95%')
    plt.title('Final prediction for ' + city)
    plt.ylabel("Rentals")
    plt.legend(loc='best')
    plt.show()

variation(df_miss,city,3,0)

results = variation_p_d(df_miss)

order, p, q = heat_map(results)

N=N_variation(test,p,q)

colonne_da_rimuovere = ['dow', 'hour', 'MA', 'MS']
df_miss = df_miss.drop(colonne_da_rimuovere, axis=1)
df_miss['_id'] = df_miss['_id'].apply(lambda x: eval(x) if isinstance(x, str) else x)
df_miss['_id'] = df_miss['_id'].apply(lambda x: day_of_year_to_date_(int(x['dow']), 2017) + (f
    " {str(x['hour']).zfill(2)}:00:00" if 'hour' in x else ''))
df_miss = df_miss.set_index('_id')

testing_model(N, test, (p, 0, q), p, q, df_miss, city)
```

```
testing_model(N, test, (p, 0, q), p, q, df_miss, city + ' - p=24')
```

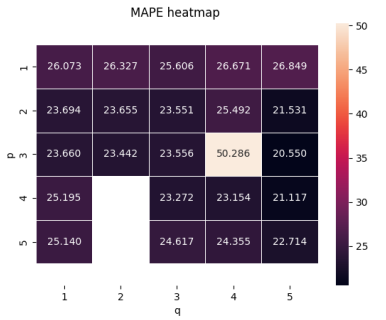


(a) MAPE HEATMAP of AMSTERDAM

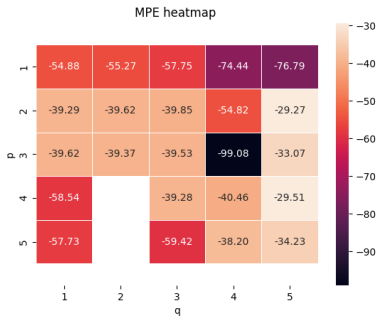


(b) MPE HEATMAP of AMSTERDAM

Figure 14: HEATMAP - different combinations of parameters



(a) MAPE HEATMAP of DENVER



(b) MPE HEATMAP of DENVER

Figure 15: HEATMAP - different combinations of parameters

9.8 TASK 8

```
def testing_model_with_time_horizon(N, test, order, varying_p_values, q, df, city,
    time_horizons):

    train, test = df['totOFbookings'][0:N], df['totOFbookings'][N:(N + len(test))]

    history = train.astype(float)

    for fixed_p in varying_p_values:
        print(f"Fixed parameter p = {fixed_p}:")

        plt.figure(figsize=(10, 6))
        for h in time_horizons:
            predictions = []

            for t in range(0, len(test) - h + 1):
                model = ARIMA(history, order=(fixed_p, 0, q))
                model_fit = model.fit(method='statespace')
                output = model_fit.get_forecast(steps=h)
                yhat = output.predicted_mean[-1]
                predictions.append(yhat)
                obs = test[t + h - 1]
                history = np.append(history, obs)

            mae = mean_absolute_error(test[:len(predictions)], predictions)
            mape = mae / np.mean(test[:len(predictions)]) * 100
            print(f"Time Horizon (h) = {h}: MAE: {mae:.3f} -- MAPE: {mape:.3f}")

            plt.plot(df.index[-len(predictions):], predictions, label=f'Predicted (h={h})',
    alpha=0.7)

```

```
        plt.title(f'Actual vs Predicted for Different Time Horizons in {city} (Fixed p={
    fixed_p})')
        plt.xlabel('Date')
        plt.ylabel('Number of Rentals')
        plt.legend()
        plt.show()

testing_model_with_time_horizon(168, test, (5,0,4), [1,3,5], 4, df_miss, city, [1,15,24,48])
```