Politecnico di Torino
ICT for smart mobility

Laboratory 2 - report
Group 2

Marika Attanasio - s311298,
Miriana Passarotto - s307735,
Cristiano Vittori - s316801

Academic Year 2023-2024

avoid using passive form
avoid using impersonal form

1 Task 1: Building the time series

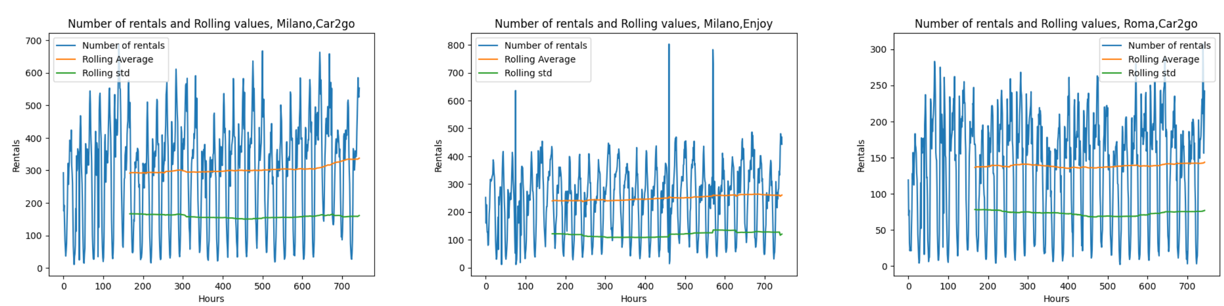
The second laboratory's first task requires selecting 30 days and collecting the recorded bookings for each hour after filtering the outliers out.
Considering periods with registered data for both services (*Car2Go* and *Enjoy*), the month having the least missing data is January. However, in the first days of the month, the behavior of the users is strongly influenced by the holidays. Instead in October, even though we have more missing data, the number of rentals has a regular behavior over the whole month. Due to these reasons, October was chosen as the month of interest.
Concerning data filtering, the same filters implemented in the first lab activity are applied in this case. This resulted in excluding bookings lasting less than 5 minutes or more than 2 hours from the analysis. In addition to those, the bookings having the same coordinates for the initial and final positions have been removed.[1] The cities which are taken into consideration are Milan (*Car2Go* and *Enjoy*) and Rome (*Car2Go*).

2 Task 2: Fitting missing data

For this second task, missing data were identified and substituted. The adopted substitution policy is to check which hours of the day are missing and then substitute those data with the average number of rentals in the same hours calculated on the other days.[1]
This step is essential for our algorithm to work efficiently since ARIMA models assume a regular time series, with no missing data.

3 Task3: Stationarity check

Once missing data were fitted, it was possible to check for stationarity and to decide whether to use differencing or not. To perform this particular analysis, the rolling average and rolling standard deviation were calculated.[2] Figure 1 depicts them.



(a) Milan, Car2Go. (b) Milan, Enjoy. (c) Rome, Car2go.

Figure 1: Stationarity check

Their behavior is mostly constant in time, meaning that the time series are stationary. Therefore, there is no need for differentiation. The ARIMA model's hyperparameter d can be set equal to 0, and the model can be represented as ARMA(p, q).

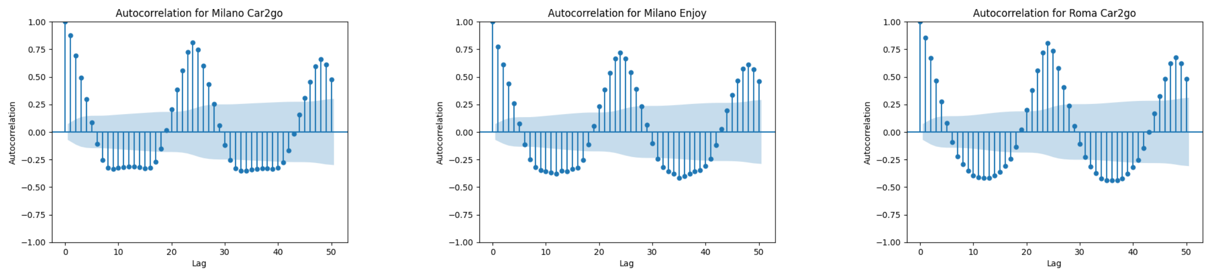
4 Task4: ACF and PACF

In this step, the ACF and PACF are computed.[3] These are useful tools for finding adequate values of the p and q parameters if the model is pure AR(p), pure MA(q), or ARMA(p,q). This technique can only be used as a first attempt at creating a good model, further analysis is needed to choose optimal values of p and q. Figures 2 and 3 display the ACF and PACF for the first 48 hours (i.e. 2 days) of the number of rentals time series for each city. The PACF provides an initial estimate of the hyper-parameter p by counting the number of significant lags (i.e. those outside the defined blue zone with a significance level less than 5%). For example, for Milano Car2Go (figure 3a), we can count 6 significant peaks; statistically important lags that follow non-significant lags are not taken into account. Consequently, p=6 is chosen for Milano Car2Go

if the process is AR only ...

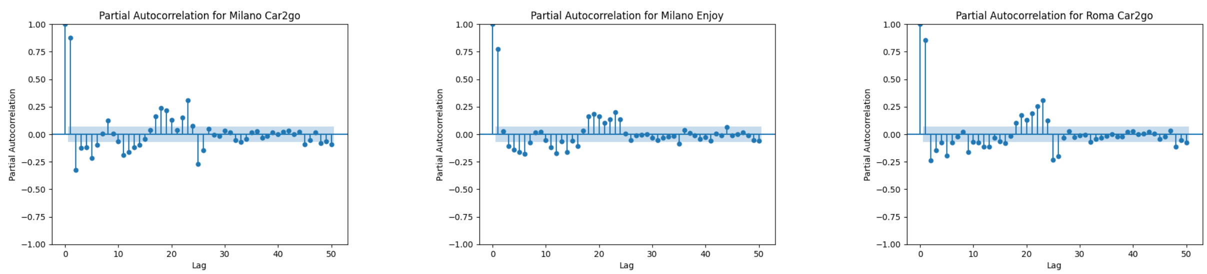
if MA only ...

and, similarly, p=1 for Milano Enjoy and p=3 for Roma Car2Go. The ACF graphs (figure 2) clearly show a periodicity: for each city, there is a regular repetition of the pattern in a 24-hour cycle. This result is consistent with the cyclical nature of the time series. A first guess of the parameter q is best done with the correlogram of the ACF; again, it is necessary to check how many significant lags exist, taking care not to overestimate. In this case, q=4 can be assumed for all cities.



(a) Milan, Car2Go. (b) Milan, Enjoy. (c) Rome, Car2go.

Figure 2: ACF plots



(a) Milan, Car2Go. (b) Milan, Enjoy. (c) Rome, Car2go.

Figure 3: PACF plots

Since neither the ACF nor the PACF reaches zero after a certain value, all three models will be of ARMA(p,q) type.

5 Task5: Training the model

A total of N=336 training samples (a period of two weeks) are used, along with 120 hours of testing, to evaluate the performance of a model. To ensure that the model can best generalize from the provided samples, various hyper-parameters will be investigated. In the first set of tasks, an expansion window learning approach will be used to adapt the model to evolving patterns in the data during testing.

6 Task6: Model training and testing

To assess the performance of an ARIMA model on the three different cities, we define a single set of parameters (listed in Table 1) and use an expanding window learning strategy. The number of training and test samples coincides with those mentioned in the previous section.

p	**d**	**q**
2	0	4

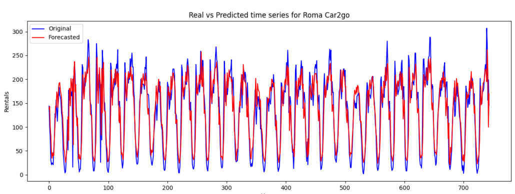
Table 1: ARIMA model parameters: q=4 as guessed in the previous section; p, instead, is chosen as "average" among the p values previously supposed, in order to determine a single set.

Table 2 displays the prediction errors for each city. [4] These errors are calculated by comparing the expected values to the actual ones. From the results, we can deduce that the model works better for Roma Car2Go.

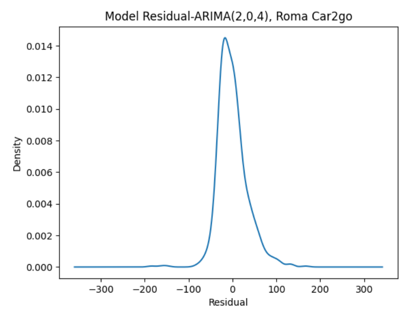
Case	MAE	MAPE	MSE	R2
Milan, Car2Go	51.310	17.887	3818.902	0.822
Milan, Enjoy	36.568	15.234	2043.024	0.820
Rome, Car2Go	24.223	18.314	952.847	0.788

Table 2: Error evaluation.

Moreover, when evaluating an ARIMA model, it is crucial to analyze the residuals, as they provide insight into the goodness of fit and may indicate the need for model adjustments. Figure 4a shows the resulting fitted model for Rome. The expected values appear to be fairly consistent with the actual values; no systematic overestimation or underestimation is visible, but the presence of discrepancies, although not too significant, certainly indicates that the model can be improved. The residual density curve in Figure 4b follows a broadly normal distribution with almost ~~no~~ 0 mean, as ideally should be. [5]
Note that the model described bases the prediction of a value only on the previous two hours (since p=2); higher p values could potentially provide more accurate results.



(a) Real vs Predicted time series.



(b) Model Residual.

Figure 4: Fitted model and residuals for Roma Car2Go.

7 Task7: Parameters definition

7a. Grid search: tuning p,q. [6]
The objective of this task is to search for the optimal values of hyperparameters p and q and study the performance of the ARIMA model for different combinations of these. We use an expansion learning strategy and maintain the same training and test dimensions as in the previous sections (N=336, test=120). To optimize the performance of the ARIMA model and ensure that it captures the characteristics of the time series at its best, we adopt the "grid search" parameter adjustment approach. The chosen ranges for the parameters are p:[0,6] and q:[0,4]. This allows us to observe if the values supposed in paragraph 4 can have some kind of match. The best model is identified based on performance evaluation metrics such as MAE and MAPE. Figure 5 shows in an easy-to-read heatmap the mean absolute percentage errors obtained for Roma Car2Go. The best combination for Roma Car2Go is $p = 2$ and $q = 4$, which is given by the lower MAPE. Similarly (see figures 11, 12 in the *References*), the best models for Milano Car2Go and Milano Enjoy are respectively ARIMA(3,0,4) and ARIMA(6,0,2). The results obtained differ from the hypothesis in section 4, which proves that a first intuition is not reliable.

7b. Tuning N and learning strategy. [7]
After selecting the optimal configuration of parameters, it's necessary to check how the model performance is affected by the size of the training set and how it changes according to different learning strategies. To

is the test set the same ?

accomplish this, the number of test samples and the parameters p and q remain constant while N varies from 24 to 504 (from 1 to 21 days). We calculate errors using both sliding window and expanding learning approaches. Table 3 displays the results obtained for the city of Rome, and the plot in figure 6 shows the behavior of the errors as N increases. The model achieves the lowest errors with a training set size of 432 samples in combination with the sliding window learning strategy.

Font too small

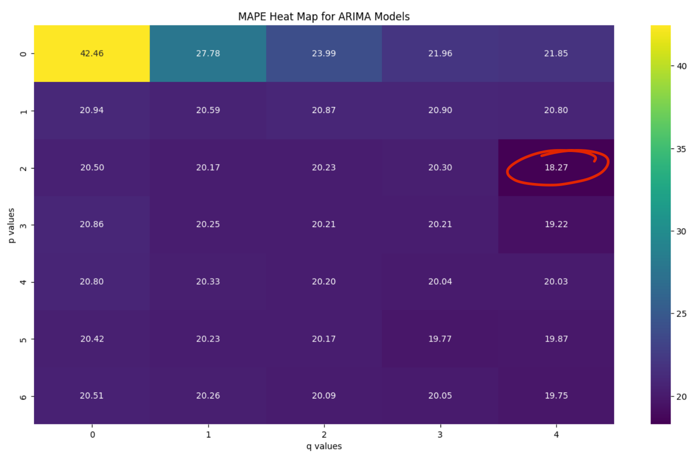


Figure 5: Roma (Car2Go), Heat map - MAPE for different combinations of parameters.

N	MAE_sl	MAE_exp	MAPE_sl	MAPE_exp
24	50.624	41.253	35.725	29.112
48	43.519	36.889	29.650	25.133
72	35.992	33.890	26.070	24.547
96	28.392	27.859	20.699	20.310
120	26.012	26.368	18.994	19.254
144	25.706	25.827	19.084	19.173
168	25.851	26.096	18.903	19.082
192	26.641	27.136	18.351	18.692
216	27.704	26.550	18.915	18.127
240	26.773	25.474	19.355	18.416
264	25.295	23.756	19.113	17.951
288	24.863	24.150	19.247	18.696
312	24.412	23.805	18.901	18.431
336	25.389	24.170	19.196	18.273
360	25.405	24.734	17.758	17.290
384	25.280	24.908	17.146	16.893
408	21.970	22.808	15.454	16.043
432	21.435	21.573	15.665	15.766
456	22.251	21.808	16.562	16.232
480	21.946	22.145	16.544	16.694
504	23.884	23.174	17.653	17.128

Table 3: Keeping p=2, d=0 and q=4 fixed, varying N for Roma Car2Go.

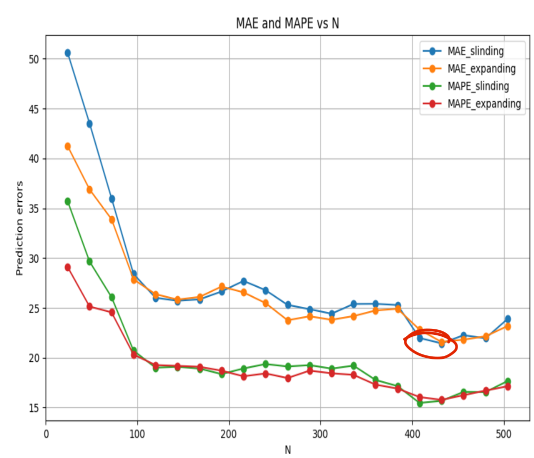


Figure 6: Errors vs N for Roma Car2Go.

Similar results are observed for the other cities:

- Milano Car2Go: $N = 480$, learning strategy: sliding window; (Table 5, figure 13)
- Milano Enjoy: N= 240, learning strategy: sliding window;(Table 6, figure 14)

7c. Comparison between cities.
After defining the best models for all three cities, we compare their performance (see Table 4). Milan Enjoy reveals the best prediction accuracy (lowest MAPE and moderate MAE) with fewer training samples. However, choosing the "best" model may depend on multiple factors such as computational efficiency, number

does it make sense to choose the city? or rather the city is given ?

of training samples, and accuracy metrics. If efficiency is a top priority, since MAE and MAPE performance are acceptable, Rome Car2Go with ARIMA(2,0,4) may be the most suitable model. If accuracy is paramount, Milan Enjoy may be preferred. In the *References* there are the plots of the "fitted" models for the three cities. [8]

	N	Model	MAPE_Sliding	MAE_Sliding
Roma Car2Go	432	ARIMA(2,0,4)	15.662	21.435
Milano Car2Go	480	ARIMA(3,0,4)	15.667	45.876
Milano Enjoy	240	ARIMA(6,0,2)	13.110	33.191

Table 4: Results summary.

8 Task8: Impact of the time horizon

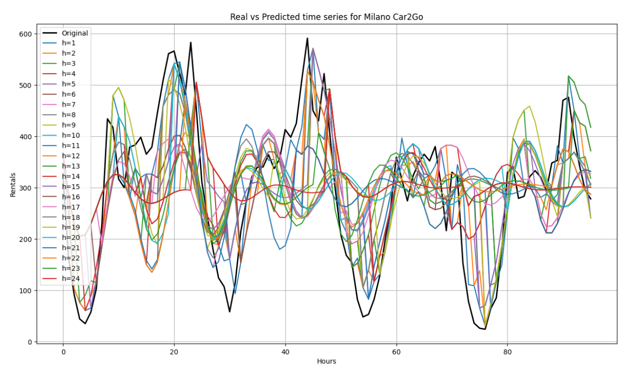
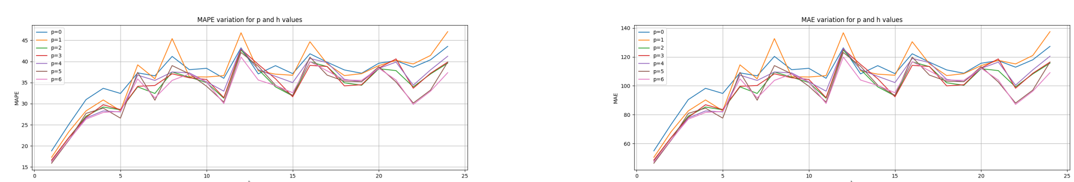


Figure 7: Forecast data for different time horizon values.

For this last task, the impact of the time horizon h on the performance of the ARIMA model was investigated. Therefore, instead of only predicting one single value per time step, we increased the value of h, forecasting all the values between t and *t+1*. Indeed, an ARIMA model ($p = 3, d = 0, q = 4$) was trained on a **train set** of 288 samples and tested on a **test set** of 96 samples (i.e. four days), all the used data were referred to **Milan, Car2Go**.[9] Figure 7 shows the results achieved for $h \in [1, 24]$.
Furthermore, both the MAPE and the MAE were evaluated for different values of h and p.[10] Figure 8 depicts the results.



(a) MAPE. (b) MAE.

Figure 8: MAPE and MAE for different values of p and h.

In general, the forecast values are closer to the original ones for lower values of h, as it can be seen in figure 8. Furthermore, it is clear that the best performance for the model is obtained for $p = 3$, which goes to prove the results stated in task 7.

4.5/5

- complete and well done
- is the test set the same when varying h ?
- no comparison with a baseline

References

[1] **Task1 and Task2**

```
import pymongo as pm
import matplotlib.pyplot as plt
from datetime import datetime
import pymongo as pm
import pandas as pd
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, acf, pacf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
import seaborn as sns

def bookings_per_hour(db, service, city,time_start, time_end):
    if service == 'Car2go':
        Bookings = db['PermanentBookings']
    elif service == 'Enjoy':
        Bookings = db['enjoy_PermanentBookings']
    else:
        raise Exception('No service found!')

    start_unix = time_start.timestamp()
    end_unix = time_end.timestamp()

    res = Bookings.aggregate([
    # filtro citt  e periodo
    {'$match': {'city': city, 'init_time': {'$gte': start_unix, '$lte': end_unix}}},
    #nuovo format con data e duration
     {"$project": {
      "_id": 0,
      "date": {"$dateToString": {"format": "%Y-%m-%d %H:00:00", "date":"$init_date"}},
      "duration": {"$divide": [{"$subtract": ["$final_time",
      "$init_time"]}, 60]}
      }
    #filtri lab 1
      },
    {"$match": {"duration": {"$gte": 5, "$lte": 120},
                "final_address": {"$ne": "$init_address"}}},
    #nuovo format
    {"$group": {"_id": { "date": "$date"},
    "number_cars": {"$sum": 1}}
  },
  {"$project": {
    "_id": 1,
    "number_cars": 1
}},
{"$sort": {"_id.date": 1}},
])

    df = pd.DataFrame(list(res))
    #plt.figure()
    #df.number_cars.plot(title="Rentals in {}, {} before fitting".format(city,service))
    #t_start = pd.to_datetime("2017-10-01 00:00:00")
    #t_end = pd.to_datetime("2017-10-31 23:00:00")

    return df

def replace_with_hourly_average(df, complete_time_range):
    df["_id.date"] = pd.to_datetime(df["_id.date"])
    df = df.set_index("_id.date")
    #calcolo media per ogni orario del giorno
```

```
    hourly_avg_df = df.groupby(df.index.hour)["number_cars"].mean().reset_index()
    for time in complete_time_range:
      #cerco i missing e sostituisco con la media
        if time not in df.index:
            hour_avg = hourly_avg_df[hourly_avg_df["_id.date"] == time.hour]["number_cars"].values[0]
            df.loc[time] = hour_avg
    df = df.sort_index()
    return df

def handle_missing_samples(service, city, df, start_date, end_date, freq="1H"):
    df["_id.date"] = pd.to_datetime(df["_id"].apply(lambda x: x["date"]))
    complete_time_range = pd.date_range(start=start_date, end=end_date, freq=freq)
    days = int(end_date.split("-")[2][:2])

    if len(df)!=days*24:
        df = replace_with_hourly_average(df, complete_time_range)

    return df

if __name__ == '__main__':
    client = pm.MongoClient('bigdatadb.polito.it',
                            ssl=True,
                            authSource='carsharing',
                            username='ictts',
                            password='Ict4SM22!',
                            tlsAllowInvalidCertificates=True)
    db = client['carsharing']
    data_types = ['bookings']
    city_names = ['Milano', 'Roma']
    service_names = ['Car2go', 'Enjoy']

    time_start = datetime(2017, 10, 1, 0, 0, 0)  # ottobre 1
    time_end = datetime(2017, 10, 31, 23, 00, 00)  # ottobre 30
    for city in city_names:
        for service in service_names:
            df = bookings_per_hour(db, service, city, time_start, time_end)
            new_df = handle_missing_samples(service, city, df,\
                                            time_start.strftime("%Y-%m-%d %H:%M:%S"), time_end.strftime("%Y-%m-%d %H:%M:%S"))
```

Task3

```
def check_stationarity_mean(fitted_df, window_size):
    fitted_df = fitted_df.set_index("_id")
    fitted_df['number_cars'] = pd.to_numeric(fitted_df['number_cars'])
    fitted_df['rolling_avg'] = fitted_df['number_cars'].rolling(window=window_size).mean()
    f_df = fitted_df.dropna()
    dropped_values = len(fitted_df) - len(f_df)

    return f_df, dropped_values

def check_stationarity_std(fitted_df, window_size):
    fitted_df = fitted_df.set_index("_id")
    fitted_df['number_cars'] = pd.to_numeric(fitted_df['number_cars'])
    fitted_df['rolling_std'] = fitted_df['number_cars'].rolling(window=window_size).std()
    f_df = fitted_df.dropna()
    dropped_values = len(fitted_df) - len(f_df)

    return f_df,dropped_values

if __name__ == '__main__':
    client = pm.MongoClient('bigdatadb.polito.it',
                            ssl=True,
```

```
                             authSource='carsharing',
                             username='ictts',
                             password='Ict4SM22!',
                             tlsAllowInvalidCertificates=True)
    db = client['carsharing']
    data_types = ['bookings']
    city_names = ['Milano', 'Roma']
    service_names = ['Car2go', 'Enjoy']

    time_start = datetime(2017, 10, 1, 0, 0, 0)  # October 1
    time_end = datetime(2017, 10, 31, 23, 00, 00)  # October 30
    window_size=168
    city_df={}

    for city in city_names:
      for service in service_names:
        df = bookings_per_hour(db, service, city, time_start, time_end)
        new_df = handle_missing_samples(service, city, df,
                                        time_start.strftime("%Y-%m-%d %H:%M:%S"),
                                        time_end.strftime("%Y-%m-%d %H:%M:%S"))
        rolling_mean_df, dropped_values_1 = check_stationarity_mean(new_df, window_size)
        rolling_mean_df['index_column'] = range(dropped_values_1, dropped_values_1 + len(rolling_mean_df))
        rolling_std_df, dropped_values_2 = check_stationarity_std(new_df, window_size)
        rolling_std_df['index_column'] = range(int(dropped_values_2), int(dropped_values_2) + len(rolling_std_df))


        # Reset the index for new_df just for plotting (old index was whole date )
        new_df_reset = new_df.reset_index()
        new_df_reset['index_column'] = range(len(new_df_reset))
        city_df[(city, service)] = new_df_reset
        plt.figure()
        plt.plot(new_df_reset['index_column'], new_df_reset['number_cars'], label='Number of rentals')
        plt.plot(rolling_mean_df['index_column'], rolling_mean_df['rolling_avg'], label='Rolling Average')
        plt.plot(rolling_std_df['index_column'], rolling_std_df['rolling_std'], label='Rolling std')
        plt.xlabel('Hours')
        plt.ylabel('Rentals')
        plt.title('Number of rentals and Rolling values, {},{}'.format(city, service))
        plt.legend()
        plt.show()
```

Task4

[3]
```
def acf_plot(df,n_lags,city,service):
    plot_acf(df,lags=n_lags)
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')
    plt.title(f'Autocorrelation for {city} {service}')
    plt.show()

def pacf_plot(df,n_lags,city,service):
    plot_pacf(df, lags=n_lags, method='ols')
    plt.xlabel('Lag')
    plt.ylabel('Partial Autocorrelation')
    plt.title(f'Partial Autocorrelation for {city} {service}')
    plt.show()

#-------------Alternative-------------------
def alternative_acf_plot(df,nlags,city,service):
    acf_lags=acf(df,nlags = nlags) # hp) nlags= len(df)
    plt.plot(acf_lags)
```

```python
    plt.axis([0,nlags,-1,1])
    plt.axhline(y= 0,linestyle="--",color = 'gray')
    plt.axhline(y = -1.96/np.sqrt(len(df)),linestyle = '--',color = 'gray') #95% confidentiality
    plt.axhline(y = 1.96/np.sqrt(len(df)),linestyle = '--',color = 'gray')
    plt.title(f'Autocorrelation for {city} {service}')
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')
    plt.show()

def alternative_pacf_plot(df, nlags,city,service):
    pacf_lags=pacf(df,nlags = nlags) # hp) nlags= len(df)
    plt.plot(pacf_lags)
    plt.axis([0,nlags,-.5,1])
    plt.axhline(y= 0,linestyle="--",color = 'gray')
    plt.axhline(y = -1.96/np.sqrt(len(df)),linestyle = '--',color = 'gray')
    plt.axhline(y = 1.96/np.sqrt(len(df)),linestyle = '--',color = 'gray')
    plt.title(f'Partial Autocorrelation for {city} {service}')
    plt.xlabel('Lag')
    plt.ylabel('Partial Autocorrelation')
    plt.show()

for (city, service), df_reset in city_df.items():
        if city=='Roma' and service=='Enjoy':
            pass
        else:
            # df should represent a time series with a time-type index
            df= df_reset[['index_column','number_cars']] #consider just these 2 columns (time=index +rentals)
            df = df.set_index('index_column') # time=index
            #print(f"DataFrame for {city}, {service}:\n{df}")
            acf_plot(df,48,city,service)
            alternative_acf_plot(df,48,city,service)
            pacf_plot(df,48, city, service)
            alternative_pacf_plot(df,48,city,service)
```

Task6

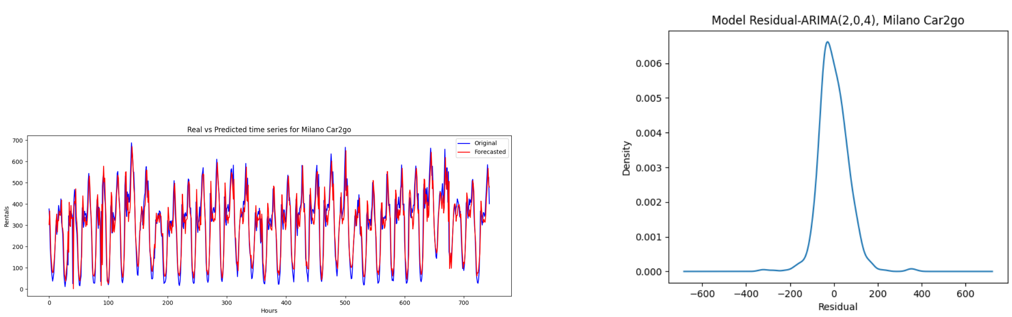
[4]

```python
p=2
d=0
q=4
for (city, service), df_reset in city_df.items():
        key = f'{city}_{service}'
        if city=='Roma' and service=='Enjoy':
            pass
        else:
            df= df_reset[['index_column','number_cars']]
            df = df.set_index('index_column')
            X = df.values.astype(float)
            train_len = 216
            print("Training on ", train_len)
            test_len = 120
            predictions = np.zeros( (test_len) )
            print('Testing ARIMA order (%i,%i,%i) for %s' % (p,d,q,key))
            train, test = X[0:train_len], X[train_len:(train_len+test_len)] # split the train and test - we use an expanding window
                                                                # approach - where data from 0 to now is used to train
                                                                # and then predict now+1. We then shift the time and repeat
            history = [x for x in train] # this is the past data used for training
            for t in range(0,test_len): # repeat for all tests we want to do
                model=ARIMA(history, order=(p,d,q))
                model_fit = model.fit(method_kwargs={'maxiter':300})#
                output = model_fit.forecast() # here we get the forecasted data
```

```
                yhat = output[0] # get now+1
                predictions[t]=yhat #and store it
                obs = test[t] # slide over time, by putting now+1 into past
                history.append(obs)

            print('(%i,%i,%i) model => MAE: %.3f -- MAPE: %.3f -- MSE: %.3f -- R2: %.3f' % (p,d,q, mean_absolute_error(test, predictions),
                                        mean_absolute_error(test, predictions) / test.mean()*100,
                                        mean_squared_error(test, predictions),
                                        r2_score(test, predictions)))
```

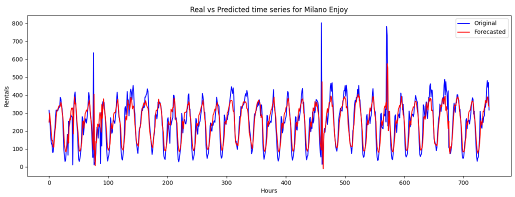
```
for (city, service), df_reset in city_df.items():
        key = f'{city}_{service}'
        if city=='Roma' and service=='Enjoy':
            pass
        else:
            df= df_reset[['index_column','number_cars']]
            df = df.set_index('index_column')
            model=ARIMA(df, order=(p,d,q))
            model_fit = model.fit(method_kwargs={'maxiter':300}) # "statespace","innovations_mle","hannan_rissanen"
            #plot results
            fig = plt.figure(figsize=(15,5))
            plt.plot(df, color='blue', label='Original')
            plt.plot(model_fit.fittedvalues, color='red', label='Forecasted')
            plt.xlabel('Hours')
            plt.ylabel('Rentals')
            plt.title('Real vs Predicted time series for {} {}'.format(city,service))
            plt.legend()
            plt.show()
            residuals = pd.DataFrame(model_fit.resid)
            residuals.plot(kind='kde',  legend=False)
            plt.xlabel('Residual')
            plt.ylabel('Density')
            plt.title('Model Residual-ARIMA({},{},{}), {} {}'. format(p,d,q,city,service))
            plt.show()
```



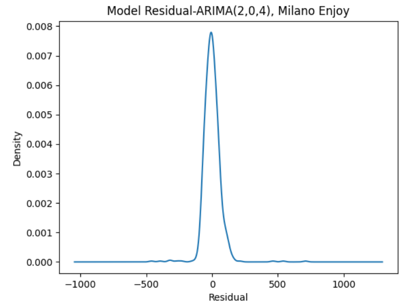
(a) Real vs Predicted time series.

(b) Model Residual.

Figure 9: Fitted model and residuals for Milano Car2Go.



(a) Real vs Predicted time series.



(b) Model Residual.

Figure 10: Fitted model and residuals for Milano Enjoy.

Task7

[6]
```
#--------------------Task 7a--------------------------
#Heatmap Plot Function
def heatmap_forARIMA(matrix,matrix_name):
    plt.figure(figsize=(12, 8))
    sns.heatmap(matrix, annot=True, fmt=".2f", cmap="viridis",
                xticklabels=diff_degrees, yticklabels=lag_orders)
    plt.title(f'{matrix_name} Heat Map for ARIMA Models')
    plt.xlabel('q values')
    plt.ylabel('p values')
    plt.show()
#--------------------------------------------------
train_len = 24*7*2 #2 weeks
test_len = 24*5 #5 days
d=0
lag_orders = (0,1,2,3,4,5,6) #values of p
diff_degrees = (0,1,2,3,4) #values of q

for (city, service), df_reset in city_df.items():
    key = f'{city}_{service}'
    if city=='Roma' and service=='Enjoy':
        pass
    else:
        df= df_reset[['index_column','number_cars']]
        df = df.set_index('index_column')
        X = df['number_cars'].values.astype(float)
        predictions = np.zeros((len(lag_orders), len(diff_degrees), test_len))
        train, test = X[0:train_len], X[train_len:(train_len + test_len)]
        mape_matrix = np.zeros((len(lag_orders), len(diff_degrees)))
        mae_matrix = np.zeros((len(lag_orders), len(diff_degrees)))
        for i, p in enumerate(lag_orders):
            for j, q in enumerate(diff_degrees):
                print(f'Case {key}')
                print(f'Testing ARIMA order ({p}, {d}, {q})')
                history = [x for x in train]
                for t in range(0, test_len):
                    model = ARIMA(history, order=(p, d, q))
                    model.initialize_approximate_diffuse()
                    model_fit = model.fit()
                    output = model_fit.forecast()
                    yhat = output[0]
                    predictions[i][j][t] = yhat
                    obs = test[t]
                    history.append(obs)

                # Calcolo del MAPE
```

```python
                mape = mean_absolute_error(test, predictions[i][j]) / test.mean() * 100
                mape_matrix[i, j] = mape
                mae = mean_absolute_error(test, predictions[i][j])
                mae_matrix[i,j]=mae

                print('(%i,%i,%i) model => MAE: %.3f -- MAPE: %.3f' % (p, d, q, mae, mape
    ))

        heatmap_forARIMA(mape_matrix, 'MAPE')
        heatmap_forARIMA(mae_matrix, 'MAE')
```

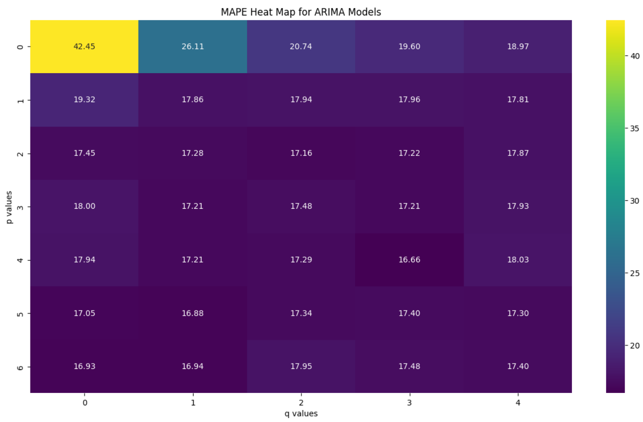


Figure 11: Milano (Car2Go),Heat map - MAPE for different combinations of parameters.

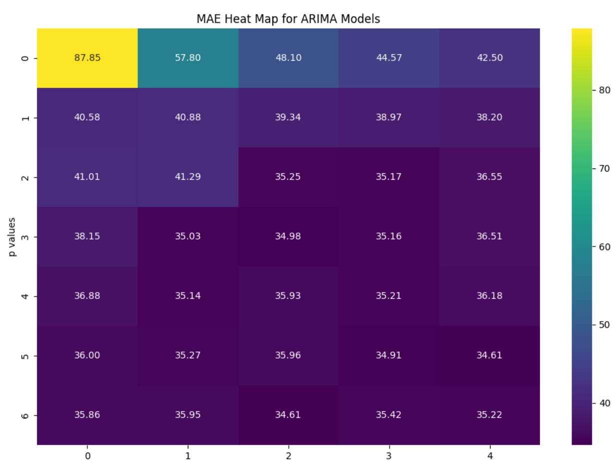


Figure 12: Milano (Enjoy),Heat map - MAPE for different combinations of parameters.

[7]
```python
#-------------------Task 7b--------------------------------
test_len=24*5 #5 days
train_len_list= 24 * np.linspace(1, 21, 21) # from 1 to 21 days
```

```
lag_orders=(0,1,2,3,4,5,6)
d=0
# ###############Roma Car2Go parameters##################
p=2
q=4
df=city_df[("Roma","Car2Go")]
# ###############Milano Car2Go parameters################
# p=3                                                     #
# q=4                                                     #
# df=city_df[("Milano","Car2Go")]                         #
###########################################################

# ###############Milano Enjoy parameters#################
# p=6                                                     #
# q=2                                                     #
# df=city_df[("Milano","Enjoy")]                          #
###########################################################

predictions_sl=np.zeros((len(lag_orders), test_len))
predictions_exp=np.zeros((len(lag_orders), test_len))
df = df.set_index('index_column')
tot_stats=pd.DataFrame()
X = df['number_cars'].values.astype(float) # extract the time series
for N in train_len_list:
    N=int(N)
    train, test = X[0:N], X[N:(N + test_len)]
    print(f'Testing N equal to {N}')
    history_sl = [x for x in train]
    history_exp = [x for x in train]
    for t in range(0, test_len):
        # SLIDING
        model_sl= ARIMA(history_sl, order=(p, d, q))
        model_sl.initialize_approximate_diffuse()
        model_fit_sl = model_sl.fit()
        output_sl = model_fit_sl.forecast()
        yhat_sl = output_sl[0]
        predictions_sl[lag_orders.index(p)][t] = yhat_sl
        # EXPANDING
        model_exp = ARIMA(history_exp, order=(p, d, q))
        model_exp.initialize_approximate_diffuse()
        model_fit_exp = model_exp.fit()
        output_exp = model_fit_exp.forecast()
        yhat_exp = output_exp[0]
        predictions_exp[lag_orders.index(p)][t] = yhat_exp

        obs = test[t]
        history_sl.append(obs)
        history_sl = history_sl[1:]
        history_exp.append(obs)

    mae_sl=mean_absolute_error(test, predictions_sl[lag_orders.index(p)])
    mape_sl=mean_absolute_error(test, predictions_sl[lag_orders.index(p)]) / test.mean()*100
    mse_sl=mean_squared_error(test, predictions_sl[lag_orders.index(p)])
    r2_sl=r2_score(test, predictions_sl[lag_orders.index(p)])
    print('(%i,%i,%i) model_sliding => MAE: %.3f -- MAPE: %.3f -- MSE: %.3f -- R2: %.3f' % (p,d,q, mae_sl,mape_sl,mse_sl,r2_sl))


    mae_exp=mean_absolute_error(test, predictions_exp[lag_orders.index(p)])
    mape_exp=mean_absolute_error(test, predictions_exp[lag_orders.index(p)]) / test.mean()*100
    mse_exp=mean_squared_error(test, predictions_exp[lag_orders.index(p)])
    r2_exp=r2_score(test, predictions_exp[lag_orders.index(p)])
    print('(%i,%i,%i) model_expanding => MAE: %.3f -- MAPE: %.3f -- MSE: %.3f -- R2: %.3f
```

```python
    ' % (p,d,q, mae_exp,mape_exp,mse_exp,r2_exp))

    statistics={"N":N, "MAE_sl":mae_sl, "MAE_exp":mae_exp, "MAPE_sl":mape_sl,"MAPE_exp":
    mape_exp,"MSE_sl":mse_sl, "MSE_exp": mse_exp, "R2_sl":r2_sl,"R2_exp":r2_exp }
    statistics_df=pd.DataFrame(statistics, index=[0])
    tot_stats=pd.concat([tot_stats, statistics_df], ignore_index=True)

print(tot_stats)
#-----------------------------------------------------------
#Plot prediction errors vs N
N=tot_stats['N'].to_list()
MAE_sl=tot_stats['MAE_sl'].to_list()
MAE_exp=tot_stats['MAE_exp'].to_list()
MAPE_sl=tot_stats['MAPE_sl'].to_list()
MAPE_exp=tot_stats['MAPE_exp'].to_list()
plt.figure(figsize=(10, 6))
plt.plot(N, MAE_sl, marker='o', label='MAE_slinding')
plt.plot(N, MAE_exp, marker='o', label='MAE_expanding')
plt.plot(N, MAPE_sl, marker='o', label='MAPE_slinding')
plt.plot(N, MAPE_exp, marker='o', label='MAPE_expanding')
plt.xlabel('N')
plt.ylabel('Prediction errors')
plt.title('MAE and MAPE vs N')
plt.legend()
plt.grid(True)
plt.show()
```

N	MAE_sl	MAE_exp	MAPE_sl	MAPE_exp
24	91.730	76.069	31.129	25.814
48	71.462	67.988	22.778	21.671
72	61.142	63.385	20.076	20.812
96	52.672	54.667	17.604	18.271
120	52.720	54.633	17.824	18.471
144	50.482	49.449	17.990	17.622
168	48.600	48.254	17.385	17.261
192	51.764	52.188	17.315	17.457
216	51.247	52.097	16.281	16.551
240	50.846	50.454	16.497	16.370
264	48.393	48.696	16.191	16.292
288	47.714	48.545	16.337	16.621
312	47.679	48.466	16.671	16.947
336	48.732	51.432	16.988	17.929
360	52.038	54.468	17.072	17.869
384	53.857	56.172	16.773	17.494
408	53.293	54.443	16.932	17.298
432	50.842	51.771	16.589	16.892
456	47.667	48.363	16.047	16.281
480	45.876	47.577	15.667	16.247
504	47.565	47.610	16.326	16.341

Table 5: Keeping p=3, d=0 and q=4 fixed, varying N for Milano Car2Go.

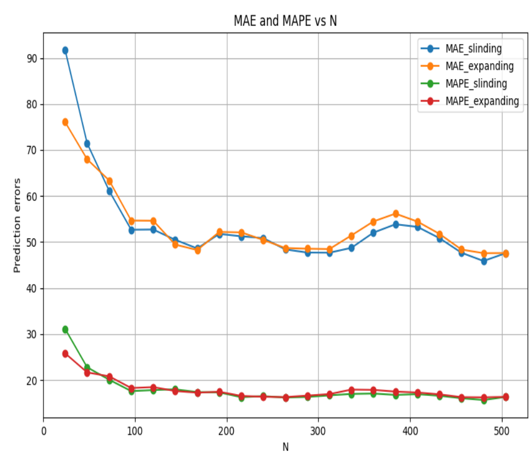


Figure 13: Errors vs N for Milano Car2Go.

N	MAE_sl	MAE_exp	MAPE_sl	MAPE_exp
24	106.138	72.211	46.206	31.436
48	67.288	64.240	27.659	26.407
72	60.424	61.958	24.243	24.859
96	48.794	47.968	19.469	19.139
120	46.424	45.649	18.506	18.197
144	37.154	37.749	15.568	15.817
168	34.124	35.999	14.970	15.793
192	36.737	39.914	15.941	17.319
216	34.826	38.506	14.252	15.758
240	33.191	35.829	13.110	14.152
264	34.383	36.078	13.517	14.183
288	34.444	37.069	13.530	14.562
312	34.969	35.664	13.966	14.243
336	33.744	34.606	14.057	14.417
360	56.894	50.735	23.630	21.071
384	52.132	50.404	20.651	19.967
408	49.247	49.143	18.643	18.603
432	48.925	48.544	18.522	18.378
456	57.380	58.503	21.426	21.846
480	43.202	43.453	16.535	16.631
504	43.451	43.482	17.153	17.166

Table 6: Keeping p=6, d=0 and q=2 fixed, varying N for Milano Enjoy.

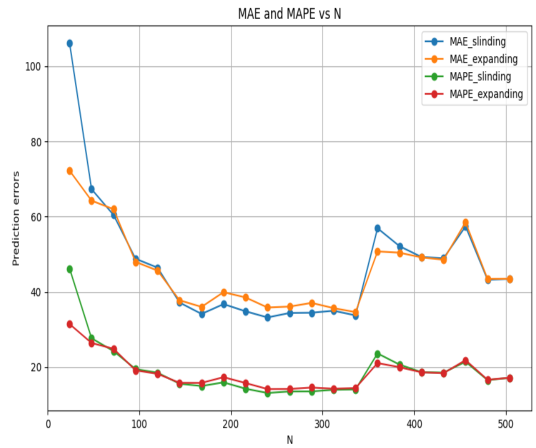


Figure 14: Errors vs N for Milano Enjoy.

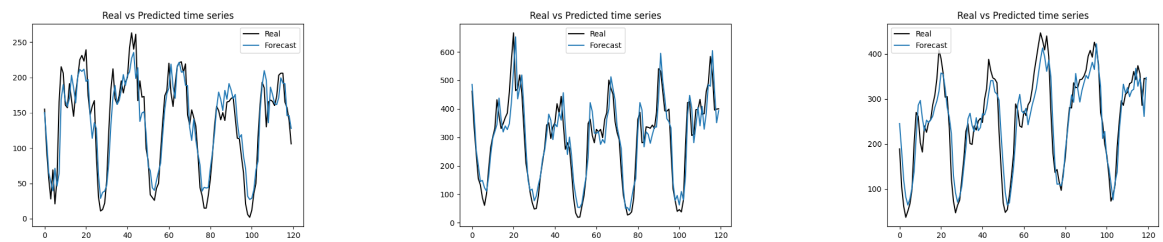
[8]
```python
# -------------------------- Task 7c ----------------------------
def fit_plot(test,predictions):
    plt.figure()
    plt.plot(test, color='black', label='Real') # plot the real time series
    plt.plot(predictions, label='Forecast')
    plt.title("Real vs Predicted time series")
    plt.legend()
    plt.show()

def buildARIMA_model_sl(p,d,q,N,X,predictions):
    train, test = X[0:N], X[N:(N+ test_len)]
    print(f'Testing N equal to {N}')
    history_sl = [x for x in train]
    for t in range(0, test_len):
        # SLIDING
        model_sl= ARIMA(history_sl, order=(p, d, q))
        model_sl.initialize_approximate_diffuse()
        model_fit_sl = model_sl.fit()
        output_sl = model_fit_sl.forecast()
        yhat_sl = output_sl[0]
        predictions[t] = yhat_sl
        obs = test[t]
        history_sl.append(obs)
        history_sl = history_sl[1:]

    fit_plot(test,predictions)

test_len = 24 * 5
d = 0
#Roma Car2Go
p_Roma = 2
q_Roma = 4
N_Roma = 432
strategy_Roma= "sliding"
df_Roma=city_df[("Roma","Car2go")]
X_Roma = df_Roma['number_cars'].values.astype(float)
predictions_Roma = np.zeros(test_len)
buildARIMA_model_sl(p_Roma,d,q_Roma,N_Roma,X_Roma,predictions_Roma)
#Milano Car2Go
```

```
p_MilanoC = 3
q_MilanoC = 4
N_MilanoC = 480
strategy_MilanoC= "sliding"
df_MilanoC=city_df[("Milano","Car2go")]
X_MilanoC = df_MilanoC['number_cars'].values.astype(float)
predictions_MilanoC = np.zeros(test_len)
buildARIMA_model_sl(p_MilanoC,d,q_MilanoC,N_MilanoC,X_MilanoC,predictions_MilanoC)
#Milano Enjoy
p_MilanoE = 6
q_MilanoE = 2
N_MilanoE = 240
strategy_MilanoE= "sliding"
df_MilanoE=city_df[("Milano","Enjoy")]
X_MilanoE = df_MilanoE['number_cars'].values.astype(float)
predictions_MilanoE = np.zeros(test_len)
buildARIMA_model_sl(p_MilanoE,d,q_MilanoE,N_MilanoE,X_MilanoE,predictions_MilanoE)
```



(a) Roma, Car2Go. (b) Milano, Car2Go. (c) Milano, Enjoy

Figure 15: Real vs Forecast time series.

Optional task

[9]

```
city = 'Milano'
service = 'Car2Go'
# Best parameters for Milano Car2Go -> Found after a grid search
p = 3
q = 4
d = 0

train_len = 288
test_len = 96

# Forecast step values
h_list = list(range(1,25))
final_res = []
df_MilanoC=city_df[("Milano","Car2go")]
df_MilanoC = df_MilanoC.set_index('index_column')
X = df_MilanoC['number_cars'].values.astype(float)

print(f'Testing ARIMA order ({p}, {d}, {q}) on {city}, {service}.')
train, test = X[0:train_len], X[train_len:(train_len + test_len)]

for h in h_list:
    predictions = np.zeros(test_len)
    history = [x for x in train]
    t = 0
    while t < test_len:
        model = ARIMA(history, order=(p,d,q))
        model_fit = model.fit()
        # Various values for step h
```

```
        output = model_fit.forecast(steps=h)
        yhat = output[0:h]
        predictions[t:t+h] = yhat[:len(predictions[t:t+h])]
        obs = test[t:t+h]
        history = history[h:]
        history = history + list(obs)
        t += h

    final_res.append(predictions)

# plot results
fig = plt.figure(figsize=(15,5))
plt.plot(range(test_len), test, color='black', label='Original', linewidth=2)
for h in range(1,25):
    plt.plot(range(test_len), np.array(final_res[h-1]).flatten(), label=f'h={h}')
plt.xlabel('Hours')
plt.ylabel('Rentals')
plt.title('Real vs Predicted time series for {} {}'.format(city,service))
plt.legend()
plt.grid()
plt.show()
```

[10]
```
# Compute MAPE and MAE for different values of p and h
city = 'Milano'
service = 'Car2Go'
# Best parameters for Milano Car2Go -> Found after a grid search
p = 3
q = 4
d = 0
# Range of p values
Np = 7

train_len = 288
test_len = 96

# Forecast step values
h_list = list(range(1,25))

mape_p = []
mae_p = []

df_MilanoC=city_df[("Milano","Car2go")]
df_MilanoC = df_MilanoC.set_index('index_column')
X = df_MilanoC['number_cars'].values.astype(float)

train, test = X[0:train_len], X[train_len:(train_len + test_len)]

for p in range(Np):
  mape_h = []
  mae_h = []
  print(f'Testing ARIMA order ({p}, {d}, {q}) on {city}, {service}.')

  for h in h_list:
    predictions = np.zeros(test_len)
    history = [x for x in train]
    t = 0
    while t < test_len:
      model = ARIMA(history, order=(p,d,q))
      model_fit = model.fit()
      # Various values for step h
      output = model_fit.forecast(steps=h)
      yhat = output[0:h]
      predictions[t:t+h] = yhat[:len(predictions[t:t+h])]
      obs = test[t:t+h]
```

```
      history = history[h:]
      history = history + list(obs)
      t += h

    mape = mean_absolute_error(test, predictions) / test.mean() * 100
    mae = mean_absolute_error(test, predictions)
    mape_h.append(mape)
    mae_h.append(mae)

  mape_p.append(mape_h)
  mae_p.append(mae_h)

# Plot MAPE values
fig = plt.figure(figsize=(15,5))
for p in range(Np):
  plt.plot(range(1,25), mape_p[p], label=f'p={p}')
plt.xlabel('h')
plt.ylabel('MAPE')
plt.title('MAPE variation for p and h values')
plt.legend()
plt.grid()
plt.show()

# Plot MAE values
fig = plt.figure(figsize=(15,5))
for p in range(Np):
  plt.plot(range(1,25), mae_p[p], label=f'p={p}')
plt.xlabel('h')
plt.ylabel('MAE')
plt.title('MAE variation for p and h values')
plt.legend()
plt.grid()
plt.show()
```