ICT in transport systems

LAB REPORT 2
Group 1

Bobonazarov Abdurasul S263771
Yu Cheng S288483
Khamidov Davron S273331

August 2022

Prediction using ARIMA models

The goal of this part is to construct and implement the best ARIMA model and predict time series data in Milano and New York

1 Determine the time period

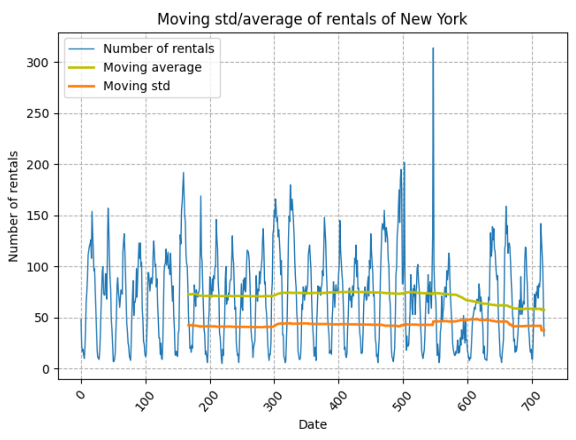
For the cities of Milano and New York, the period of 30 days is set from 16th January 2017 to 14th February 2017.

2 Check no missing data

Checking missing data for total rentals: if the first and last value are missing, 0 is filled, if the other value is missing, mean value of the previous value and the following value is filled.

3 Check time series

In order to check whether time series is stationary, check the mean value and standard deviation value of total rentals. The mean value and standard deviation of rentals for each city is close to constant. The whole and complete time series are shown in Figure 1 and Figure 2



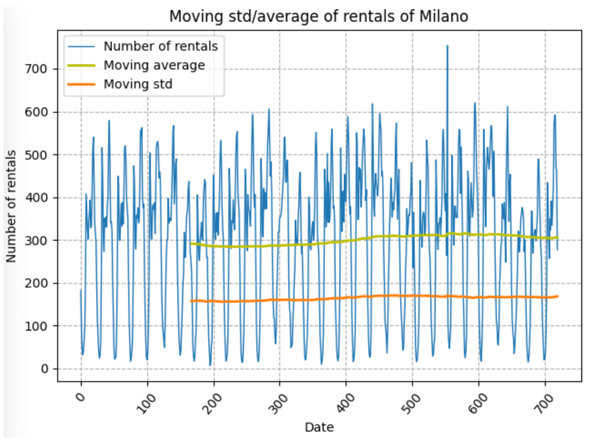


Fig 1. New York std/average

Fig 2. Milano std/average

4 Check of Stationarity Condition

To test the stationarity of time series data, we observe statistical properties such as mean, variance, autocorrelation and check if they are constant over time. In Figure 1, the rolling mean and rolling standard deviation of the two time series, calculated over a window of 168 hours (equivalent to one week), are displayed. We can see that the trends for the mean and standard deviation do not significantly change over time. We therefore considered time series having the "stationarity" property. Due to this, we put d=0, and our model becomes ARMA.

5 Compute ACF and PACF

We plot the Autocorrelation Function (ACF) and the Partial Autocorrelation Function (PACF) for the time series of the number of rentals for Milano and New York for the first 50 hours of data(Figure3 and Figure 4). They are used for determining p, q hyperparameters for the ARMA model. We decided p = 1 by looking at the PACF in Figure 3 since we can see that only one lag is highly correlated. Considering ACF in Figure 3 we observe that there are 4 lags above the confidence interval, so q = 4. The same approach used in Figure 4. So we choose p = 2 and q = 3.

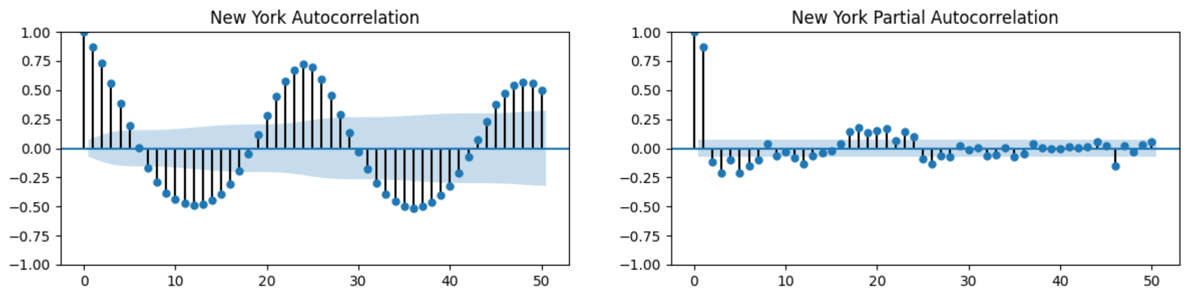


Fig 3 ACF and PACF for New York

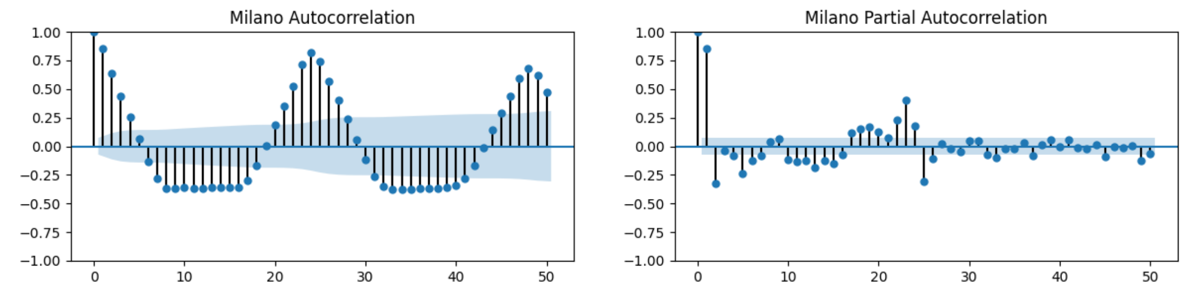


Fig 4 ACF and PACF for Milano

6 Dividing train and test data

We chose to use the first week data for training and the second week data as test data for prediction. So there are 168 samples in the training data set and the same amount of samples in the testing data set. And an expanding window strategy was chosen as a training methodology.

7 Train the model

We trained a model for each city using the configuration described in the previous sections. Figure 5 shows the final fitted model. We analysed various error types as well as the coefficient of determination R2 while calculating their performance. Tab. 1 displays the results for each city. The Mean Absolute Percentage Error is the most important error to consider when comparing findings from 2 different time series (MAPE). So, we decide to use the MAPE for comparing results.

	Hyperparameters	MSE	MAE	MAPE	R2
Milano	p=2, d=0, q=3	4558.44	53.17	0.38	0.82
New York	p=1, d=0, q=4	305.00	13.29	0.30	0.84

Table 1: Performances of the models

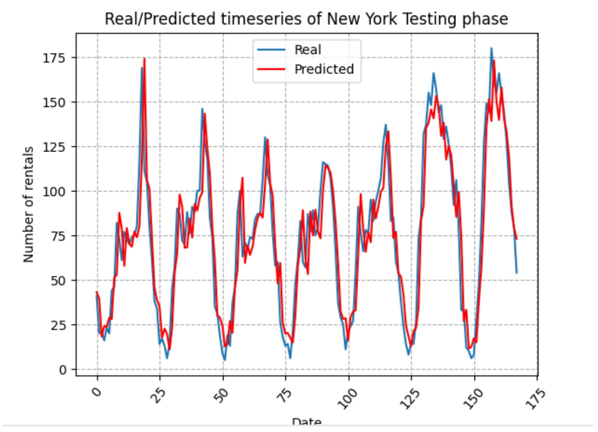


Fig 5. Test data of New York

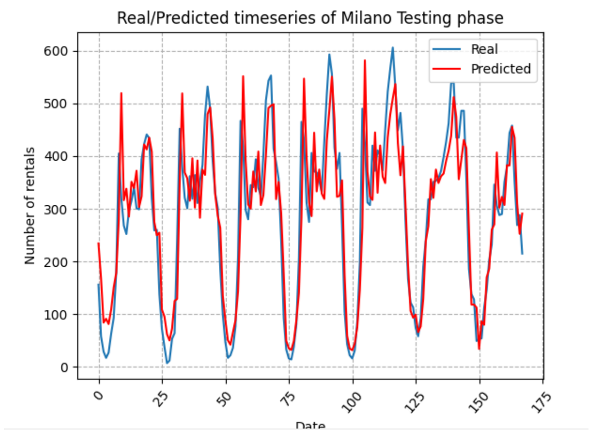
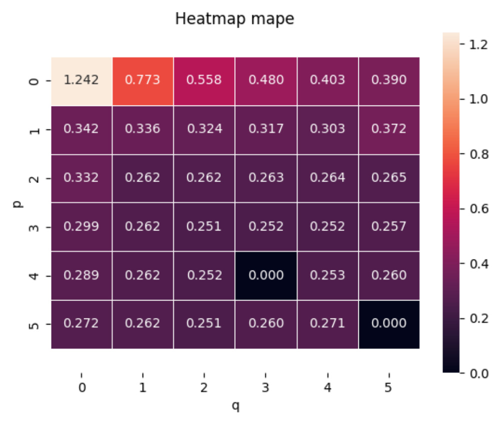


Fig 6. Test data for Milano

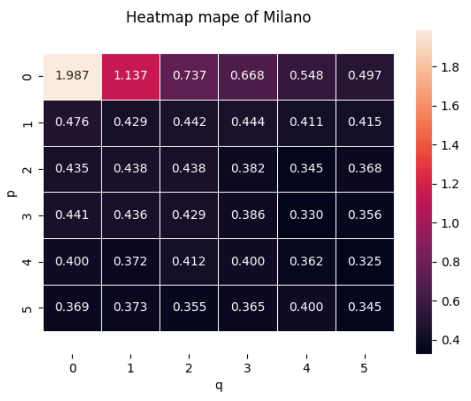
8 Grid search for tuning p and q hyperparameters

We selected the range of p and q parameters based on PACF and ACF trends. We chose p=(0,5) and q=(0,5) for both cities. MAPE was calculated for each model and results for each city were displayed on heatmap as shown in Figure 7. There are some values with MAPE=0. The Arima model failed in these p, q parameters and it was not possible to calculate the MAPE so, they were set to 0. It is possible to notice that for New York the model ARIMA(3,0,2) is the best because it presents the lowest MAPE value, which differs from our initial guess (ARIMA(1,0,4)). Next figure(Figure 8) shows ARIMA(4,0,5) is the best for Milano. While executing the following task, we encountered an error with these parameters. In order to avoid the error we choose the ARIMA(3,0,4) model which has the closest MAPE to the ARIMA(4,0,5) model.



p \ q	0	1	2	3	4	5
0	1.242	0.773	0.558	0.480	0.403	0.390
1	0.342	0.336	0.324	0.317	0.303	0.372
2	0.332	0.262	0.262	0.263	0.264	0.265
3	0.299	0.262	0.251	0.252	0.252	0.257
4	0.289	0.262	0.252	0.000	0.253	0.260
5	0.272	0.262	0.251	0.260	0.271	0.000

Fig 7. Heatmap of MAPE for New York



p \ q	0	1	2	3	4	5
0	1.987	1.137	0.737	0.668	0.548	0.497
1	0.476	0.429	0.442	0.444	0.411	0.415
2	0.435	0.438	0.438	0.382	0.345	0.368
3	0.441	0.436	0.429	0.386	0.330	0.356
4	0.400	0.372	0.412	0.400	0.362	0.325
5	0.369	0.373	0.355	0.365	0.400	0.345

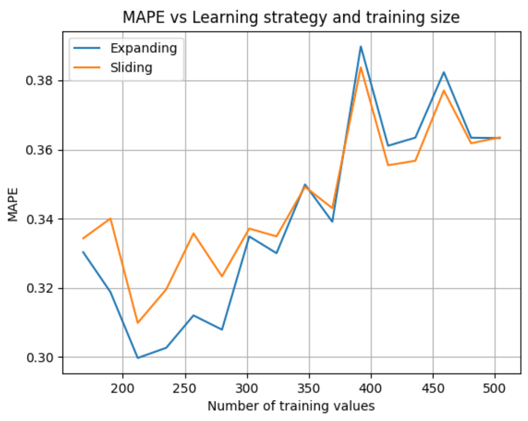
Fig 8. Heatmap of MAPE for Milano

9 Training size and learning strategies

After finding the best p and q parameters, we prepared training samples in order to find optimal training window size and learning strategies. There are 16 samples, from 7 days upto 21 days. Test data was kept fixed and contains 7 days data. We calculated MAPE for Milano and New York with a different number of training size and learning strategies. While observing results (Figure 11,12) we can see that, trend is increasing as the number of training sizes increases. In Milano, a training window size of 235 gives the lowest MAPE while in New York it is equal to 280. We also see that in New York, both learning strategies show approximately the same MAPE. Table 2 presents final results of the model.

City	Milano	New York
Learning strategy	Expanding window	Sliding window
Training window size	235	324
ARIMA model hyperparameters (p,d,q)	(3, 0, 4)	(3, 0, 2)
MAPE	0,3	0.22

Table 2: Final model with learning strategy and training window



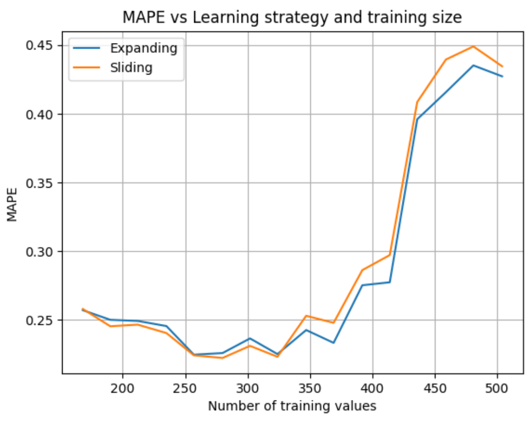
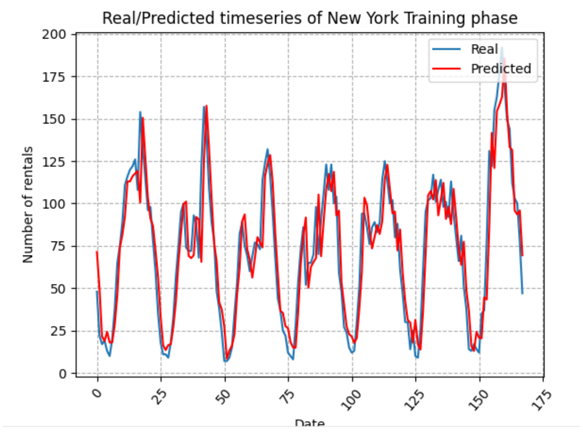


Fig 11. Mape sliding vs expanding window for Milano

Fig 12. Mape sliding vs expanding window for New York

APPENDIX

Figures



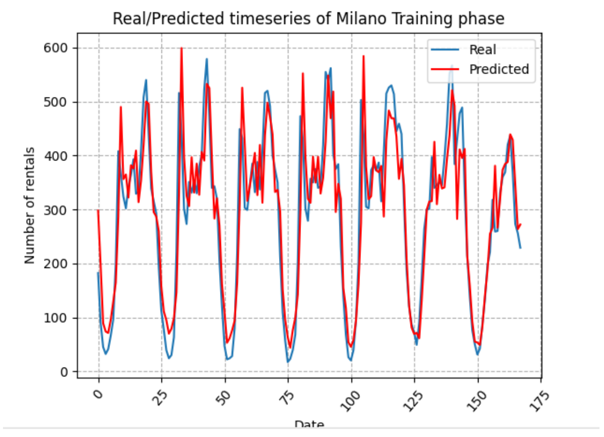


Fig 13. Training data of New York

Fig 14. Training data for Milano

Code

A.1 Connection to Database

```
def setup_connection():
  client = pm.MongoClient('bigdatadb.polito.it',
                   ssl=True,
                   authSource='carsharing',
                   tlsAllowInvalidCertificates=True)
  db_car_sharing = client['carsharing']
  db_car_sharing.authenticate('ictts', 'Ictts16!')
  return db_car_sharing
```

A. 2 Data extraction

```
def data_extraction(db):
  start_date = datetime(2017, 1, 16, 0, 0, 0)
  end_date = datetime(2017, 2, 14, 23, 59, 59)

  # collection_car2go[0] = "PermanentBookings"
  car_per_hour_filtered_new_york = pipline_cars_per_hour_filtered(db, collection_car2go[0],
'New York City', start_date, end_date)
  car_per_hour_filtered_milano = pipline_cars_per_hour_filtered(db, collection_car2go[0],
'Milano', start_date, end_date)
  return car_per_hour_filtered_new_york, car_per_hour_filtered_milano

def pipline_cars_per_hour_filtered(db, collection, city, start_date, end_date):
  return list(db.get_collection(collection).aggregate(
     [
        {  # filter data for a given city in a given period
            '$match': {
                '$and': [
                    {'city': city},
                    {'init_date': {'$gte': start_date}},
                    {'final_date': {'$lte': end_date}}
                ]}
        },
        {  # convert init_date to part of date like:
            # date_parts: [{year: 2017}, {month: 1}, {day: 1}, {hour: 11}, {minute: 20}]
            # moved status
            # calculate duration of booking and parking: duration = (final_time-init_time)/60 |
added weekday
            '$project': {
                '_id': 1,
                'city': 1,
                'moved': [
                    {'$ne': ['$origin_destination', 'undefined']},
                    {'$ne': [  # match the coordinates
                        {'$arrayElemAt': ['$origin_destination.coordinates', 0]},
                        {'$arrayElemAt': ['$origin_destination.coordinates', 1]}
                    ]},
                    True
                ],
                'duration': {
                    '$divide': [
                        {
                            '$subtract': ['$final_time', '$init_time']
                        },
                        60
                    ]
                },
                'date_parts': {'$dateToParts': {'date': '$init_date'}}
            }
        },
        {  # filter the data
            '$match': {
                '$and': [
                    {'duration': {'$gte': 3}},
                    {'duration': {'$lte': 180}},
                    {'moved': True}
                ]
            }
        },
        {  # calculate sum of rentals per hour on filtered data
            '$group': {
                '_id': {
                    'month': '$date_parts.month',
                    'day': '$date_parts.day',
                    'hour': '$date_parts.hour'
                },
                'total_rentals': {'$sum': 1}
            }
        },
        {  # sort by id(month,day,hour)
            '$sort': {'_id': 1}
        }
    ]
))
```

A. 3 Check missing and cleaning data

```
def clean_missing_values(new_york, milano):
    '''
    print(new_york[0])
    print(len(new_york))
    print(milano[0])
    print(len(milano))
    '''

    # checking list if there are less then 720 items, add them
    milano = add_missing_item_to_list(milano)
    new_york = add_missing_item_to_list(new_york)

    # checking each key and values of dictionary
    count = 0
    for i in range(720):
        lineNewYork = new_york[i]
        lineMilano = milano[i]
        # total rentals part:
        if i < 1:
            if lineNewYork.get('total_rentals', -1) == -1:
                lineNewYork['total_rentals'] = milano[i + 1]['total_rentals']
            if lineMilano.get('total_rentals', -1) == -1:
                lineMilano['total_rentals'] = new_york[i]['total_rentals']
        elif i == 720 - 1:
            if lineNewYork.get('total_rentals', -1) == -1:
                lineNewYork['total_rentals'] = milano[i - 1]['total_rentals']
            if lineMilano.get('total_rentals', -1) == -1:
                lineMilano['total_rentals'] = new_york[i - 1]['total_rentals']
        else:
            if lineNewYork.get('total_rentals', -1) == -1:
                lineNewYork['total_rentals'] = (new_york[i - 1]['total_rentals'] + new_york[i +
1]['total_rentals']) / 2
            if lineMilano.get('total_rentals', -1) == -1:
```

```
            lineMilano['total_rentals'] = (milano[i - 1]['total_rentals'] + milano[i +
1]['total_rentals']) / 2
    # datetime part:
    if i < 1:  # the first value is missing
        if lineNewYork["_id"].get('day', -1) == -1:
            lineNewYork["_id"]["day"] = 16
        if lineNewYork["_id"].get('month', -1) == -1:
            lineNewYork["_id"]["month"] = 1
        if lineNewYork["_id"].get('hour', -1) == -1:
            lineNewYork["_id"]["hour"] = 0
        # milano part
        if lineMilano["_id"].get('day', -1) == -1:
            lineMilano["_id"]["day"] = 16
        if lineMilano["_id"].get('month', -1) == -1:
            lineMilano["_id"]["month"] = 1
        if lineMilano["_id"].get('hour', -1) == -1:
            lineMilano["_id"]["hour"] = 0
    else:  # the other value except the first value, so the value i-1 is worked.
        if lineNewYork["_id"].get('hour', -1) == -1:
            if new_york[i - 1]["_id"]["hour"] == 23:  # next day
                lineNewYork["_id"]["hour"] = 0
            else:
                lineNewYork["_id"]["hour"] = new_york[i - 1]["_id"]["hour"] + 1

        if lineNewYork["_id"].get('day', -1) == -1:
            if new_york[i - 1]["_id"]["day"] == 31:  # next month
                # countNewYork = countNewYork + 1
                lineNewYork["_id"]["day"] = 1
                # lineNewYork["_id"]["month"] = 2
            else:
                lineNewYork["_id"]["day"] = new_york[i - 1]["_id"]["day"] + 1
        if lineNewYork["_id"].get('month', -1) == -1:
            if i > 384:
                count = 1
                if count == 1:
                    lineNewYork["_id"]["month"] = 2
                else:
                    lineNewYork["_id"]["month"] = 1
        # milano part
        if lineMilano["_id"].get('hour', -1) == -1:
            if milano[i - 1]["_id"]["hour"] == 23:
                lineMilano["_id"]["hour"] = 0
            else:
                lineMilano["_id"]["hour"] = milano[i - 1]["_id"]["hour"] + 1
        if lineMilano["_id"].get('day', -1) == -1:
            if milano[i - 1]["_id"]["day"] == 31:  # next month
                # countMilano = countMilano + 1
                lineMilano["_id"]["day"] = 1
                # lineMilano["_id"]["month"] = 2
            else:
                lineMilano["_id"]["day"] = milano[i - 1]["_id"]["day"] + 1
        if lineMilano["_id"].get('month', -1) == -1:
            if i > 383:
                count = 1
                if count == 1:
                    lineMilano["_id"]["month"] = 2
                else:
                    lineMilano["_id"]["month"] = 1

    # change dictionary format to [{'timestamp': timestamp, 'total_rentals': 123}, ... ]
    new_milano = change_format(milano)
    new_new_york = change_format(new_york)
    # return new formated city data
    return new_new_york, new_milano

def add_missing_item_to_list(city_data):
    days_list = list(range(16, 32)) + list(range(1,
                                               15))  # gives list of days [16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
    hours_list = list(range(0, 24))

    length_city_data = len(city_data)
    if length_city_data < 720:
        index = 0
        for day in days_list:
            for hour in hours_list:
                item = city_data[index].get('_id', -1)

                item_month = item.get('month', -1)
                item_day = item.get('day', -1)
                item_hour = item.get('hour', -1)

                if item_day == day and item_hour == hour:
                    index += 1
                else:
                    data = {
                        '_id': {
                            'month': 1,
                            'day': day,
                            'hour': hour
                        },
                        'total_rentals': 0
                    }
                    city_data.insert(index, data)
                    index += 1
    return city_data
```

```
def change_format(city):
  new_city = []
  for item in city:
     data = {
        #'timestamp': datetime(2017, item['_id']['month'], item['_id']['day'], item['_id']['hour']),
        'timestamp': str(date(2017, item['_id']['month'], item['_id']['day'])),
        'total_rentals': item['total_rentals']
     }
     new_city.append(data)

  return new_city
```

A. 4 Check stationarity

```
def check_stationarity(new_york, milano):

  df_new_york = pd.DataFrame(new_york)
  df_new_york.set_index('timestamp')

  df_milano = pd.DataFrame(milano)
  df_milano.set_index('timestamp')

  print_moving_avg_std(df_new_york, 'New York')
  print_moving_avg_std(df_milano, 'Milano')

def print_moving_avg_std(df_city, city_name):
  roll_days = 7
  df_city_mean = df_city['total_rentals'].rolling(24 * roll_days).mean()
  df_city_std = df_city['total_rentals'].rolling(24 * roll_days).std()

  plt.figure(constrained_layout=True)
  plt.plot(df_city['total_rentals'], linewidth=1, label='Number of rentals')
  plt.plot(df_city_mean, linewidth=2, color='y', label='Moving average')
  plt.plot(df_city_std, linewidth=2, label='Moving std')
  plt.title(f'Moving std/average of rentals of {city_name}')
  plt.xlabel('Date')
  plt.ylabel('Number of rentals')
  plt.xticks(rotation=50)
  plt.grid(linestyle='--', linewidth=0.8)
  plt.legend()
  plt.show()
```

A. 5 ACF and PACF

```
def acf_and_pacf(new_york, milano):
  df_new_york = pd.DataFrame(new_york)
  df_new_york.set_index('timestamp')
  fig, axes = plt.subplots(1, 2, figsize=(16, 3), dpi=100)

  plt.title("New York")
  plot_acf(df_new_york['total_rentals'], lags=50, ax=axes[0], title='New York Autocorrelation')
  plot_pacf(df_new_york['total_rentals'], lags=50, ax=axes[1], title='New York Partial
Autocorrelation')

  df_milano = pd.DataFrame(milano)
  df_milano.set_index('timestamp')

  fig, axes = plt.subplots(1, 2, figsize=(16, 3), dpi=100)
  plt.title("Milano")
  plot_acf(df_milano['total_rentals'], lags=50, ax=axes[0], title='Milano Autocorrelation')
  plot_pacf(df_milano['total_rentals'], lags=50, ax=axes[1], title='Milano Partial
Autocorrelation')

  plt.show()
```

A. 6 Model training and testing with p and q

```
def model_evaluation(new_york, milano):
  # Data Splitting
  N = 7 * 24

  # New York
  data_NY = [float(item['total_rentals']) for item in new_york]
  train_NY, test_NY = data_NY[0:N], data_NY[N:2*N]

  # Milano
  data_milano = [float(item['total_rentals']) for item in milano]
  train_Milano, test_Milano = data_milano[0:N], data_milano[N:2*N]

  # Model training, plotting and printing error metrics
  model_training(train_NY, 1, 0, 4, 'New York')  # For NY p=2 d=0 q=3
  model_training(train_Milano, 2, 0, 4, 'Milano')  # For Milano p=1 d=0 q=4

  # Test model
  model_testing(train_NY, test_NY, 1, 0, 4, 'New York')
  model_testing(train_Milano, test_Milano, 2, 0, 4, 'Milano')

def model_training(train, p, d, q, city):
  # Defining model
```

```
    model = ARIMA(train, order=(p, d, q))
    model_fit = model.fit()

    # ploting model
    plot_model_training(train, model_fit, city)

    # printing error metrics
    print(f"Train Error metrics for {city} parametrs p={p}, d={d}, q={q}")
    print_error_metrics(train[0:len(model_fit.fittedvalues)], model_fit.fittedvalues)

def plot_model_training(train_data, model_fit, city):
    plt.plot(train_data, label='Real')  # change
    plt.plot(model_fit.fittedvalues, color='red', label="Predicted")  # change
    plt.title(f'Real/Predicted timeseries of {city} Training phase')  # change
    plt.xlabel('Date')
    plt.ylabel('Number of rentals')
    plt.xticks(rotation=50)
    plt.legend(loc='upper right')
    plt.grid(linestyle='--', linewidth=0.8)
    plt.show()

def model_testing(train, test, p, d, q, city):
    # Defining model with expanding window
    history = train
    predictions = []
    for i in range(0, len(test)):
        model = ARIMA(history, order=(p, d, q))
        model_fit = model.fit()
        output = model_fit.forecast()

        y_hat = output[0]
        predictions.append(y_hat)

        obs = test[i]
        history.append(obs)

    # ploting model
    plot_model_testing(test, predictions, city)

    # prining error metrics
    print(f"Testing error metrics for {city} parametrs p={p}, d={d}, q={q}")
    print_error_metrics(test, predictions)

def plot_model_testing(test, predictions, city):
    plt.plot(test, label='Real')  # change
    plt.plot(predictions, color='red', label="Predicted")  # change
    plt.title(f'Real/Predicted timeseries of {city} Testing phase')  # change
    plt.xlabel('Date')
    plt.ylabel('Number of rentals')
    plt.xticks(rotation=50)
    plt.legend(loc='best')
    plt.grid(linestyle='--', linewidth=0.8)
    plt.show()

def print_error_metrics(original, predicted):
    mae = mean_absolute_error(original, predicted)
    mape = mean_absolute_percentage_error(original, predicted)
    mse = mean_squared_error(original, predicted)
    r2 = r2_score(original, predicted)
    print(f"MAE: {mae}, MSE: {mse}, R2: {r2}, MAPE: {mape}")
    return mae, mse, r2, mape
```

A. 7 Grid search

```
def grid_search(new_york, city):
    N = 7 * 24
    # Splitting data
    data = [float(item['total_rentals']) for item in new_york]
    train, test = data[0:N], data[N:2 * N]
    test_len = len(test)

    p_list = (0, 1, 2, 3, 4, 5)
    q_list = (0, 1, 2, 3, 4, 5)
    # 4,3 5,5 error
    d = 0

    predictions = np.zeros(((len(p_list)*len(q_list)), test_len))

    results = {'p': [], 'd': [], 'q': [], 'mse': [], 'mae': [], 'mape': []}
    combinations = range(0, (len(p_list)*len(q_list)))
    comb = 0
    for p in p_list:
        for q in q_list:
            print(f'Testing {p}, {d}, {q}')
            # if p == 4 and q == 3:
            #     results['p'].append(p)
            #     results['d'].append(d)
            #     results['q'].append(q)
            #     results['mse'].append(0)
            #     results['mape'].append(0)
            #     results['mae'].append(0)
            #     comb += 1
            # elif p == 5 and q == 5:
            #     results['p'].append(p)
            #     results['d'].append(d)
            #     results['q'].append(q)
            #     results['mse'].append(0)
            #     results['mape'].append(0)
            #     results['mae'].append(0)
            #     comb += 1
            # else:
            history = [x for x in train]

            for t in range(0, test_len):
                output = arima_model(history, p, d, q)

                y_hat = output[0]
                predictions[comb][t] = y_hat

                obs = test[t]
                history.append(obs)  # expanding window
            print(f'{p}, {d}, {q}')
            mae, mse, r2, mape = print_error_metrics(test, predictions[comb])

            results['p'].append(p)
            results['d'].append(d)
            results['q'].append(q)
            results['mse'].append(mse)
            results['mape'].append(mape)
            results['mae'].append(mae)
            comb += 1

    df_results = pd.DataFrame(results)

    fig = plt.figure()
    heat_df_mape = df_results.pivot(index='p', columns='q', values='mape')
    ax = sns.heatmap(heat_df_mape, annot=True, linewidths=.5, fmt='.3f')
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom+0.5, top-0.5)
    plt.title(f'Heatmap mape of {city}')
    plt.show()

    best = df_results['mape'].idxmin()
    p = df_results.loc[best]['p'].astype(int)
    d = 0
    q = df_results.loc[best]['q'].astype(int)
    return (p, d, q)

def arima_model(history, p, d, q):
    model = ARIMA(history, order=(p, d, q))
    model_fit = model.fit()
    output = model_fit.forecast()
    return output
```

A. 8 Learning strategy and window size

```
def learning_strategy(city, p, d, q):
  methods = [0, 1]
  tr_size = 21*24
  ts_size = 168

  N_values = list(np.linspace(7*24, 21*24, num=16, dtype=int))
  predictions = np.zeros((len(N_values), len(methods), ts_size))
  mae = np.zeros((len(N_values), len(methods)))
  mse = np.zeros((len(N_values), len(methods)))
  mape = np.zeros((len(N_values), len(methods)))
  r2 = np.zeros((len(N_values), len(methods)))
  warnings.filterwarnings('ignore')
  for m in methods:
     for j in N_values:
        print(f'method {m}, tr size {j}')
        data = [float(item['total_rentals']) for item in city]
        tr, ts = data[0:j], data[j:(j+ts_size)]
        history = [x for x in tr]
        for t in range(0, ts_size):
           output = arima_model(history, p, d, q)

           y_hat = output[0]
           predictions[N_values.index(j)][methods.index(m)][t] = y_hat

           obs = ts[t]
           history.append(obs)
           history = history[m:]

        mae[N_values.index(j)][methods.index(m)] = mean_absolute_error(ts,
predictions[N_values.index(j)][methods.index(m)])
        mse[N_values.index(j)][methods.index(m)] = mean_squared_error(ts,
predictions[N_values.index(j)][methods.index(m)])
        r2[N_values.index(j)][methods.index(m)] = r2_score(ts,
predictions[N_values.index(j)][methods.index(m)])
        mape[N_values.index(j)][methods.index(m)] = mean_absolute_percentage_error(ts,
predictions[N_values.index(j)][methods.index(m)])

  plt.figure()
  heatmap = sns.heatmap(mape, xticklabels=methods, yticklabels=N_values, annot=True,
cmap="Blues")
  heatmap.set_xticklabels(['Expanding', 'Sliding'])
  heatmap.set_ylabel('Number of training values')
  plt.suptitle('Mape for different size of training')
  plt.show()

  plt.figure()
```

```
    plt.plot(N_values, mape[:, 0], label='Expanding')
    plt.plot(N_values, mape[:, 1], label='Sliding')
    plt.xlabel('Number of training values')
    plt.ylabel('MAPE')
    plt.title('MAPE vs Learning strategy and training size')
    plt.legend()
    plt.grid()
    plt.show()
```

A. 9 Main code

```
if __name__ == '__main__':
    db = setup_connection()

    # Task1
    new_york, milano = data_extraction(db)

    # Task2
    new_york, milano = clean_missing_values(new_york, milano)

    # Task3
    check_stationarity(new_york, milano)

    # Task4
    acf_and_pacf(new_york, milano)

    #  Task 5 and 6
    model_evaluation(new_york, milano)

    # Task 7
    best_order_new_york = grid_search(new_york, 'New York')
    # print(f'Best NY {best_order_new_york}')
    best_order_milano = grid_search(milano, 'Milano')
    # print(f'Best Milano {best_order_milano}')

    learning_strategy(milano, 3, 0, 4)
    learning_strategy(new_york, 3, 0, 2)
```