Politecnico di Torino

Laboratory 2(Prediction using ARIMA model)

Group 07

Author

Ali Mohammad Alizadeh 308885

Mohammad Eftekhari Pour 307774

Supervisors:

Prof. Marco Mellia

Prof. Luca Vassio

Academic year 2023/24

LAB 2 – Prediction using ARIMA models

Introduction

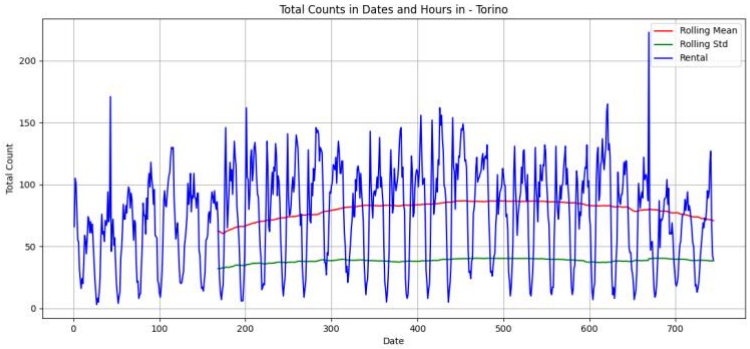
The goal of this lab is to use ARIMA model to predict the future behavior of the system using historical data and timeseries. Firstly, the data is going to be retrieved from the database and then will be cleansed using hourly mean for the missing data. Then ACF PACF figures will be drawn to check if the timeseries is stationary or not and have a guess over p, d and q values to be able to run an Arima model. After that the ARIMA model will be ran for different values of p, d and q values to check the accuracy and make a comparison among them.

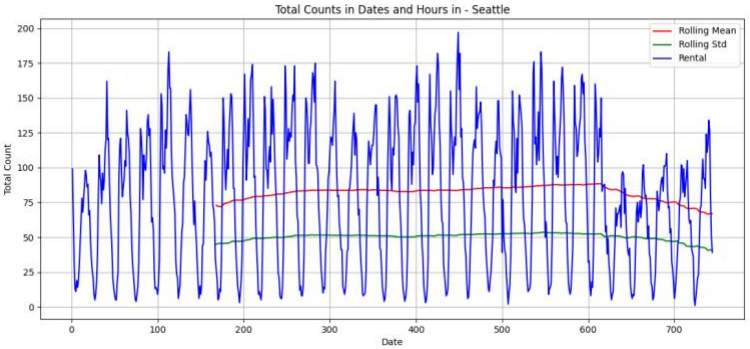
1) For each city, consider the selected period of 30 days. Extract the number of rentals recorded at each hour – after properly filtering the outliers and those bookings that are not rentals.
2) Check if there are missing samples (recall – ARIMA models assume a regular time series, with no missing data). In case of missing samples – define a policy for fitting missing data. For instance, use i) the last value, or ii) the average value, or iii) replace with zeros, or iv) replace with average value for the given time bin, etc.

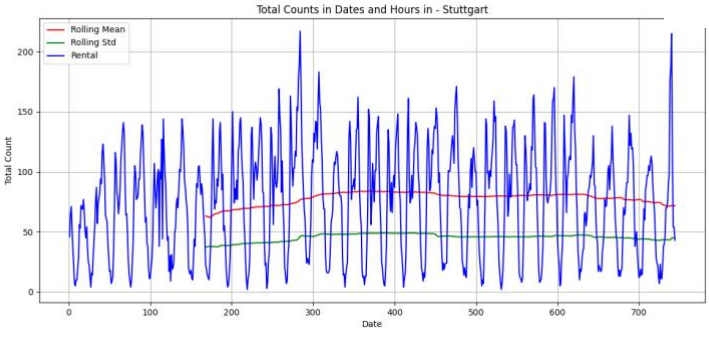
As was done in lab 1 we have a pipeline to filter the data to ignore the outliers. We check if cars are moved and the duration is between 5 minutes and 3 hours.
From the previous lab we remember that there are some missing data in the dataset and we checked the data for the January 2018 which there were some missing records that needed to be dealt with to have a complete timeseries without any gaps. For each city we calculated the hourly mean and used the corresponding hourly-mean to fill the missed data. The mean of the whole dataset did not fit as we wanted since using the mean value of the day for some hours like 3am did not seem like a good idea.

3) Check if the time series is stationary or not. Decide accordingly whether to use differencing or not (d=0 or not).

In order to check if the timeseries is stationary we need to use some statistical properties like Mean, Variance and Autocorrelation and … . We need the timeseries to be stationary since to be able to predict the future samples we need to make sure that the behavior will be the same as before, so we need the Mean and Standard deviation to be constant during the time. As it can be seen these two properties are almost stationary during the time, so we can say that the timeseries is stationary, this means that we will not use differencing and consider d to be 0. So, in the ARIMA model we will end up with ARMA. Chosen window for the rolling mean and rolling std was 48 hours.



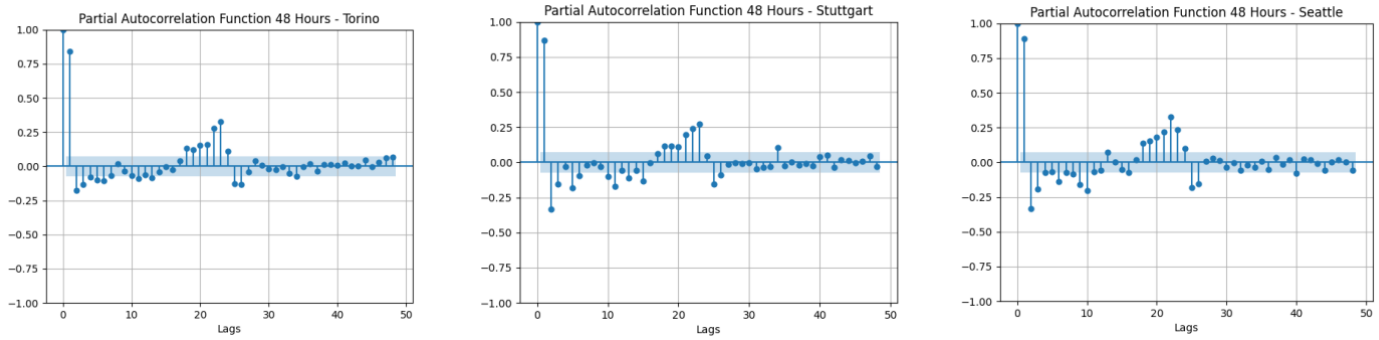


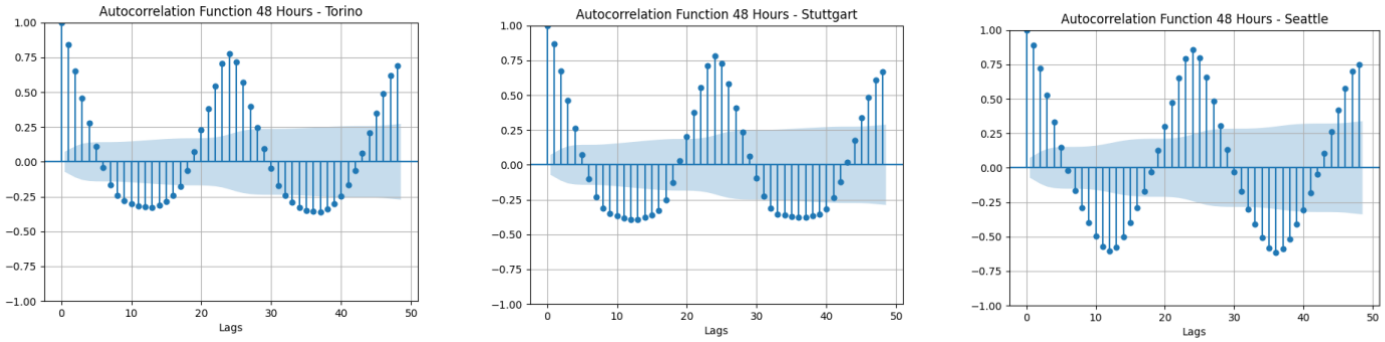


4) Compute the ACF and PACF to observe how they vanish. This is instrumental to guess possible good values of the p and q parameters, assuming the model is pure AR or pure MA. What if your model is not pure AR or pure MA, i.e., it is an ARMA process?

Using ACF and PACF of timeseries we can determine the hyperparameters (p, q) of the ARMA model. In Autocorrelation Function will be negligible after q lags and in Partial Autocorrelation Function will be negligible after p lags. As it can be seen in the figures, for all the cities we decided to use p=2 using PACF and q=4 using ACF, which is not necessarily the best values but a good guess to start with.

ACF Of Different Cities





PACF Of Different Cities

5) Decide the number of past samples N to use for training, and how many to use for testing. Given you have 30 days of data (each with 24 hours) you can consider for instance training during the first week of data and test the prediction in the second week of data.

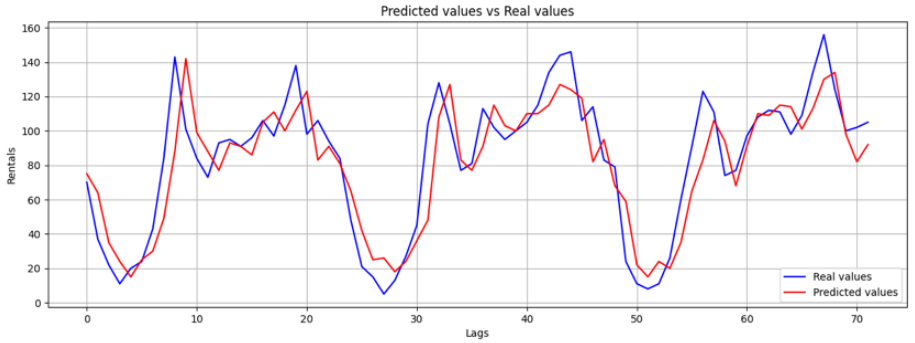
6) Given N, (p,d,q), train a model, compute the error. Consider the MPE and/or MAPE and/or other metrics – so that you can compare results for different cities (use percentage errors and not absolute errors for this comparison. Absolute errors would obviously be not directly comparable).

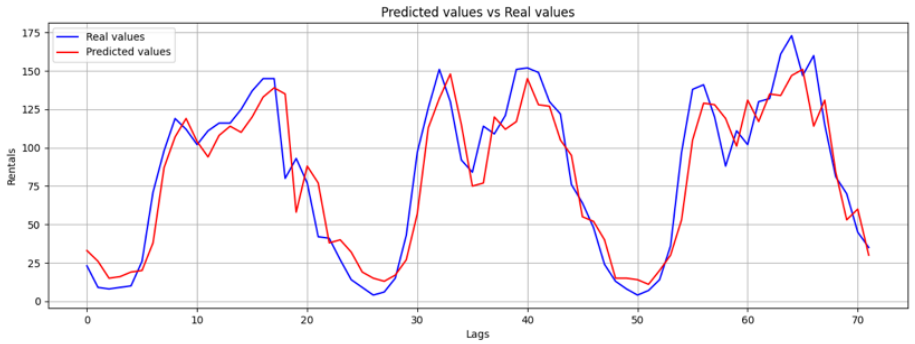
For Training size and test size we used different values from too short (train=1week, test=2days) to too long (train=3weeks, test=1week) and ended up using 14days (2Weeks) for training size and 72hours (3days) for test size which was computationally less demanding for the next step. We used the rolling window strategy for training. Rolling window strategy is better for short-term fluctuations and adapting quickly to changes and the Expanding window strategy is better when we deal with long-term trends or evolving patterns, in our case rolling window strategy was used. Values for p, d and q were correspondingly 2, 0 and 4.

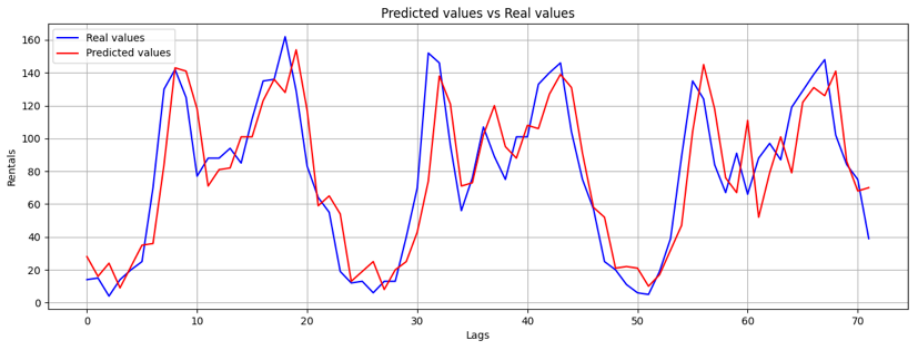
Finally, we plot the test data and predicted data which was can be considered as a good prediction, then calculated the R2_Score and MAPE and RMSE for each city and the result is as below:

CITY	R2_Score	RMSE	MAPE	
TORINO- Enjoy	*0.77*	*19.13*	*0.32*	
SEATTLE	*0.86*	*19.32*	*0.38*	
STUTTGART	*0.75*	*22.94*	*0.43*	

Considering RMSE and MAPE values, Torino had the best performance and if we consider R2 values Seattle had a slightly better results, but in this case, we are using MAPE (Mean Absolute Percentage Error).

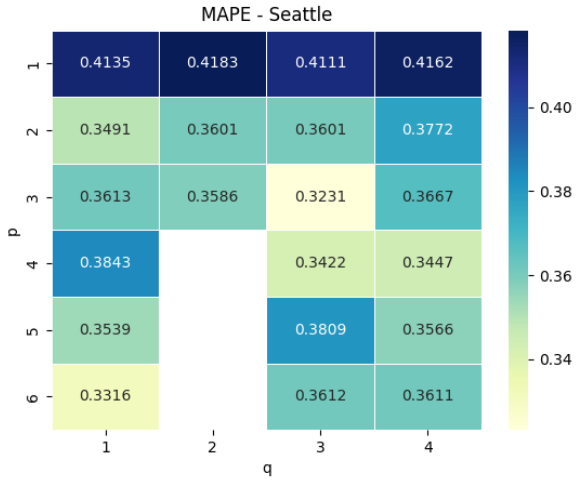


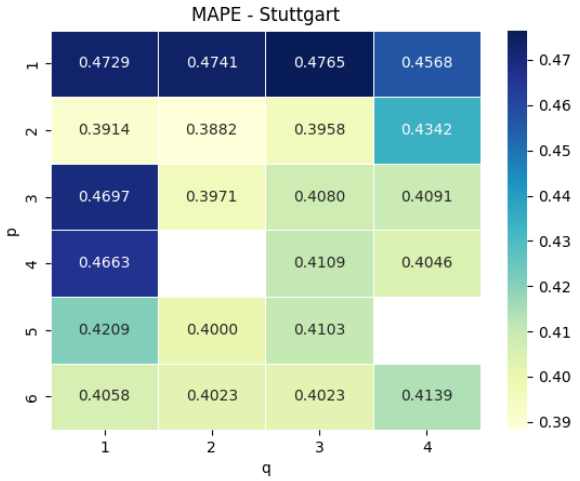


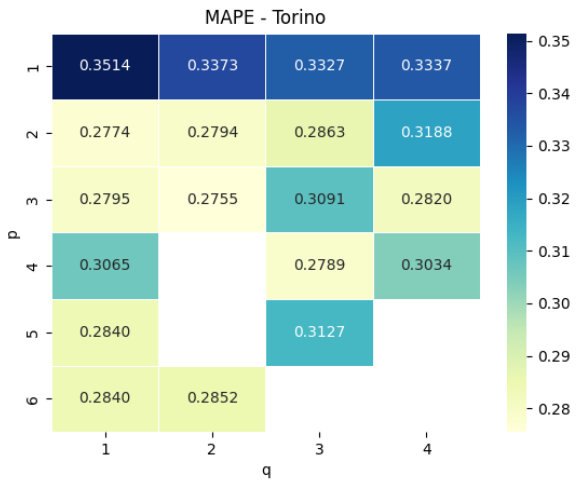


7) Now check the impact of parameters:
   a. Keep N fixed, and do a grid search varying (p,d,q) and observe how the error changes. Choose the best (p,d,q) parameter tuple for each city. Justify your choice.
   b. Given the best parameter configuration, change N and the learning strategy (expanding versus sliding window). Always keep the testing set on the same portion of the data. For instance, if you use 1 week for testing, you can use from 1 day to 3 weeks for training.
   c. Compare results for the different cities. How the relative error changes w.r.t. the absolute number of rentals?

Note: values that are missed are caused by an error during predication phase, so the result for metrics where set to None to keep the heatmap consistent.







For variant values of p and q we have chosen the [1,2,3,4,5,6] for p values and [1,2,3,4] for q values. We ran the model for different combinations of p and q for each city. For each iteration we kept the original values and predictions and calculated the MAPE metric and the result is as shown.

There is a common pattern for all cities, if we keep the q value constant and increase the p value gradually, we will get a slightly better result, while for increasing the q value and keeping the p value constant we observe different patterns in different cities, but it can be said that the result gets worst or doesn't change or the improvement of the result might be due to increasing the p value. Also it can be understood that p=1 with any value for q has the worst results.

To choose the best values for p and q, we need to select the best result with the lowest p and q values since with increasing the values the ARIMA model considers more parameters, leading to increased computational complexity, model fitting will take more time and overall, in using grid search for optimal p and q will result in higher computation time as values change.

For Torino, the MAPE range in between 0.27 and 0.35, the best value achieved with the given range of p and q is 0.2755 and we chose 3 and 2 for p and q. For Seattle the best value is p=3 and q=3 which has the lowest MAPE value 0.3231, but overall combination of p= [3,4] q= [3,4] has the better results after the best choice. For Stuttgart the best combination is p=2 and q=2 with the least value of 0.3882 for MAPE metric, and overall p=2 and q ranges of [1,2,3] can produce best results.

Accuracy in Torino was significantly better than other two cities, and even Seattle better than Stuttgart, this might be due to accurate dataset recorded for Torino in Enjoy collection which results in better patterns and predictions.

In [ ]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pymongo as pm
import pprint
from enum import Enum
from datetime import datetime, timedelta
import pytz

#------------------------------------------------------------ Connect to MongoDB
client = pm.MongoClient('bigdatadb.polito.it',
                        ssl=True,
                        authSource = 'carsharing',
                        username = 'ictts',
                        password ='Ict4SM22!',
                        tlsAllowInvalidCertificates=True)
db = client['carsharing']
#Choose the DB to use
permenant_booking = db['PermanentBookings']
permenant_parking = db['PermanentParkings']
enjoy_permenant_booking = db['enjoy_PermanentBookings']
enjoy_permenant_parking = db['enjoy_PermanentParkings']
#ENUM of cities
class CITY_ENUM(Enum):
    TO = 'Torino'
    SEA = 'Seattle'
    STU = 'Stuttgart'
class CITY_TIMEZONES(Enum):
    TO = 'Europe/Rome'
    SEA = 'America/Los_Angeles'
    STU = 'Europe/Berlin'

# def get_start_end_unix_zone(timezone):
#     # start_timestamp = datetime(2018, 1, 1,0,0,0,0, pytz.timezone(timezone)).timestamp()
#     # end_timestamp  = datetime(2018, 1, 31,23,59,59,0, pytz.timezone(timezone)).timestamp()
#     return start_timestamp,end_timestamp
start_unix_time = datetime.strptime("01/01/2018", "%d/%m/%Y").timestamp()
end_unix_time = datetime.strptime("31/01/2018", "%d/%m/%Y").timestamp()

#pipeline for getting the data for the rentals with the filteration of the data
#too short and too long rentals are filtered out
#considered if car is moved
def filter_pipeline(city,start_unix_time,end_unix_time):
    return [
    {
        '$match': {
            'city': city,
            'init_time': {
                '$gte': start_unix_time,
                '$lt': end_unix_time
            },
            'final_time': {
                '$gte': start_unix_time,
                '$lt': end_unix_time
            }
        }
    },
    {
        '$project': {
            '_id': 0,
            'duration': {
                '$divide': [
                    { '$subtract': ['$final_time', '$init_time'] },
                    60  # Divide by 60 to convert seconds to minutes
                ]
            },
            'day': {'$dayOfMonth': '$init_date'},
            'hour': {'$hour': '$init_date'},
            'date': {
                '$dateToString': {
                    'format': '%Y-%m-%d',
                    'date': '$init_date'
                }
            },
            'moved': {
                '$ne':[
                    {"$arrayElemAt": [ "$origin_destination.coordinates", 0]},
                    {"$arrayElemAt": [ "$origin_destination.coordinates", 1]}
                 ]
            }
        }
    },
    {
        '$match': {
            'moved': True,
            'duration':{'$gt':5, '$lt':180},

        }
    },
    {
        '$group':{
            '_id': {'day': '$day', 'hour': '$hour', 'date': '$date'},
            'total_count': {'$sum': 1},
        }
```

```
    },
    {
        '$sort': {
            '_id': 1,
        }
    },
]
#------------------------------------------------------------ Get the data from MongoDB
TO_Data = list(enjoy_permenant_booking.aggregate(filter_pipeline(CITY_ENUM.TO.value,
          start_unix_time,end_unix_time)))
SEA_Data = list(permenant_booking.aggregate(filter_pipeline(CITY_ENUM.SEA.value,
          start_unix_time,end_unix_time)))
STU_Data = list(permenant_booking.aggregate(filter_pipeline(CITY_ENUM.STU.value,
          start_unix_time,end_unix_time)))
cities_data_array = [(CITY_ENUM.TO.value,TO_Data),(CITY_ENUM.SEA.value,SEA_Data),(CITY_ENUM.STU.value,STU_Data)]
#------------------------------------------------------------ check for missing data
print("TO_Data",len(TO_Data))
print("SEA_Data",len(SEA_Data))
print("STU_Data",len(STU_Data))
#------------------------------------------------------------ Dropping _id and Flattening the data
def dfModifier(city_list):
  df = pd.DataFrame(city_list, columns =['_id', 'total_count'])
  df['date'] = df['_id'].apply(lambda x: x['date'])
  df['day'] = df['_id'].apply(lambda x: x['day'])
  df['hour'] = df['_id'].apply(lambda x: x['hour'])
  df['myIndex'] = (df['day']-1)*24 + (df['hour']+1)
  df.drop(['_id'], axis=1, inplace=True)
  return df
#day | hour
#1   | 0 -> day*24 + hour => 1*24 + 0 = 24
#1   | 1 -> day*24 + hour => 1*24 + 1 = 25
#1   | 2 -> day*24 + hour => 1*24 + 2 = 26
#day | hour
#0   | 1 -> day*24 + hour => 0*24 + 1 = 1
#0   | 2 -> day*24 + hour => 0*24 + 2 = 2
#0   | 3 -> day*24 + hour => 0*24 + 3 = 3
TO_df = dfModifier(TO_Data)
SEA_df = dfModifier(SEA_Data)
STU_df = dfModifier(STU_Data)
cities_df_array = [(CITY_ENUM.TO.value,TO_df),(CITY_ENUM.SEA.value,SEA_df),(CITY_ENUM.STU.value,STU_df)]
#------------------------------------------------------------ Calculating hourly mean
# calculating the avg for each hour of the day
TO_hourly_avg = TO_df.groupby('hour')['total_count'].mean().round().reset_index().astype(int)['total_count'].tolist()
SEA_hourly_avg = SEA_df.groupby('hour')['total_count'].mean().round().reset_index().astype(int)['total_count'].tolist()
STU_hourly_avg = STU_df.groupby('hour')['total_count'].mean().round().reset_index().astype(int)['total_count'].tolist()
#------------------------------------------------------------ Filling the missing data with the mean
def fillMissingValues(df:pd.DataFrame, avg_df):
  missingValues=set(np.arange(1,31*24+1)).difference(set(df['myIndex']))
  # dfMean = round(np.mean(df['total_count']))
  print("Missing values are:", len(missingValues), missingValues)
  df2 = df
  for value in missingValues:
    dayOfValue = int((value-1)/24)+1
    hourOfValue = (value-1)%24
    new_row = pd.DataFrame({'total_count':avg_df[hourOfValue],'date':f'2018-01-{dayOfValue:02d}',
                            'day':dayOfValue,'hour':hourOfValue,'myIndex':value}, index =[0])
    df2 = pd.concat([new_row,df2.loc[:]]).reset_index(drop = True)
  df2.sort_values(by=['myIndex'], inplace=True)
  return df2

To_FilledValues = fillMissingValues(TO_df, TO_hourly_avg)
SEA_FilledValues = fillMissingValues(SEA_df,SEA_hourly_avg)
STU_FilledValues = fillMissingValues(STU_df,SEA_hourly_avg)
#------------------------------------------------------------ Plotting the data with Rolling mean and checking for stationarity
def plotter(plotTitle, df:pd.DataFrame):
    mean = df['total_count'].rolling(window=24*7).mean()
    std = df['total_count'].rolling(window=24*7).std()
    plt.figure(figsize=(14, 6))
    plt.plot(df['myIndex'], mean, label='Rolling Mean', color='red')
    plt.plot(df['myIndex'], std, label='Rolling Std', color='green')
    plt.plot()
    plt.plot(df['myIndex'], df['total_count'], label='Rental', color='blue')
    plt.xlabel('Date')
    plt.ylabel('Total Count')
    plt.legend()
    plt.grid(True)
    plt.title(f'Total Counts in Dates and Hours in - {plotTitle}')
    plt.grid(True)
    plt.savefig(f'{plotTitle}-Roolings-mean-std')
    plt.clf()
plotter('Torino',To_FilledValues)
plotter('Seattle',SEA_FilledValues)
plotter('Stuttgart',STU_FilledValues)
#------------------------------------------------------------ making a clean data
cleanFilledCities = [(CITY_ENUM.TO.value,To_FilledValues),(CITY_ENUM.SEA.value,SEA_FilledValues),(CITY_ENUM.STU.value,STU_Fi
#------------------------------------------------------------ Computing ACF and PACF and Plotting them
from statsmodels.tsa.stattools import acf,pacf
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf

# Use ACF to find q.
# Use PACF to find p.

def ACF_PACF(city_data):
  # plot acf
  plt.figure(figsize=(6,4))
  plot_acf(city_data[1]["total_count"], lags=48)
```

```
  plt.title(f'Autocorrelation Function 48 Hours - {city_data[0]}')
  plt.xlabel('Lags')
  plt.grid(True)
  # plt.show()
  plt.savefig(f'{city_data[0]}-ACF')

  # plot pacf
  plt.figure(figsize=(6,4))
  plot_pacf(city_data[1]["total_count"], lags=48)
  plt.title(f'Partial Autocorrelation Function 48 Hours - {city_data[0]}')
  plt.xlabel('Lags')
  plt.grid(True)
  # plt.show()
  plt.savefig(f'{city_data[0]}-PACF')

for city_data in cities_df_array:
  ACF_PACF(city_data)

#------------------------------------------------------------ ARIMA model and prediction
from statsmodels.tsa.arima.model import ARIMA
import warnings
from statsmodels.tools.sm_exceptions import ConvergenceWarning
warnings.simplefilter('ignore', ConvergenceWarning)

q = 4
p = 2
d = 0
train_size = 24 * 7 * 2 # 24 * 7 * 3 # 3 weeks -> we will change this to 14 days
test_size = 24 * 3 # 10 days -> this takes too long to run so we will use 72 hours
myModel = None
def Predictor(cleanCity):
  originalData = list(cleanCity[1]['total_count'][:train_size])#.tolist()
  y_hat = [None for _ in range(train_size)] # should it be a list or pandas array?
  for record in range(train_size,train_size+test_size):
    model = ARIMA(originalData, order=(p,d,q))
    model_fit = model.fit()
    prediction = int(model_fit.forecast()[0])
    y_hat.append(prediction)
    originalData.append(cleanCity[1]['total_count'][record])
    originalData = originalData[1:]
    myModel = model_fit

  plt.figure(figsize=(15,5))
  plt.title("Predicted values vs Real values")
  plt.plot(list(cleanCity[1]['total_count'][train_size:train_size+test_size]), color='blue', label="Real values")
  plt.plot(list(y_hat[train_size: train_size+test_size]), color='red', label="Predicted values")
  plt.legend()
  plt.xlabel("Lags")
  plt.ylabel("Rentals")
  plt.grid(True)
  plt.savefig(f'2 day prediction {cleanCity[0]}')
  plt.clf()
  # plot residual errors
  residuals = pd.DataFrame(myModel.resid)
  residuals.plot()
  plt.title(f'Residuals - {cleanCity[0]}')
  plt.xlabel("Residual Error")
  plt.ylabel("Residuals")
  plt.grid(True)
  plt.savefig(f'2 day Residuals {cleanCity[0]}')
  plt.clf()
  #plot the gaussian density of the residuals
  residuals.plot(kind='kde')
  plt.title(f'Density of Residuals - {cleanCity[0]}')
  plt.xlabel("Residual Error")
  plt.ylabel("Density")
  plt.grid(True)
  plt.savefig(f'2 day Density of Residuals {cleanCity[0]}')
  plt.clf()
  return y_hat, model_fit
#------------------------------------------------------------ Prediction for 3 days
#create an array to store the results to be used for the comparison and metrics
comparisonArray = []
for city_data in cleanFilledCities:
  print(city_data[0])
  y_hat, model_fit = Predictor(city_data)
  y_hat = y_hat[train_size:train_size+test_size]
  comparisonArray.append((city_data[0],city_data[1]['total_count'][train_size:train_size+test_size], y_hat, model_fit))
#------------------------------------------------------------ Metrics
from sklearn.metrics import mean_squared_error,r2_score, mean_absolute_percentage_error
for item in comparisonArray:
  mse = mean_squared_error(item[1], item[2])
  rmse = np.sqrt(mse)
  r2 = r2_score(item[1], item[2])
  mape = mean_absolute_percentage_error(item[1], item[2])
  print(f'{item[0]} -> MSE: {mse:.2f}, RMSE: {rmse:.2f}, R2: {r2:.2f}, MAPE: {mape:.2f}')
#------------------------------------------------------------ Runninf model for different P and Q values
p = [1,2,3,4,5,6]
q = [1,2,3,4]
d = 0
finalValues = []

for cleanCity in cleanFilledCities:
  for i in p:
    for j in q:
      worked = False
```

```
      originalData = list(cleanCity[1]['total_count'][:train_size])#.tolist()
      y_hat = [None for _ in range(train_size)] # should it be a list or pandas array?
      for record in range(train_size,train_size+test_size):
        try:
          model = ARIMA(originalData, order=(i,d,j))
          model_fit = model.fit()
          prediction = int(model_fit.forecast()[0])
          # print(f'Prediction for {cleanCity[0]} at {record} is {prediction}')
          y_hat.append(prediction) #shoudl it be int(prediction) or prediction as a float
          originalData.append(cleanCity[1]['total_count'][record])
          originalData = originalData[1:]
          worked = True
        except:
          print("error")
          worked = False
          continue
      if worked:
        actual_values = cleanCity[1]['total_count'][train_size:train_size+test_size]
        prediction_values = y_hat[train_size:train_size+test_size]
        mse = mean_squared_error(actual_values, prediction_values)
        rmse = np.sqrt(mse)
        r2 = r2_score(actual_values, prediction_values)
        mape = mean_absolute_percentage_error(actual_values, prediction_values)
        finalValues.append((cleanCity[0],i,j,mse,rmse,r2,mape))
      elif not worked:
        finalValues.append((cleanCity[0],i,j,0,0,0,0))
#------------------------------------------------------------ Plotting a heatmap for the results
#get the list and  change it to a 2d array to fit the heatmap
from itertools import groupby
import seaborn as sb

# print(finalValues)
dont_touch_this = finalValues
touchthis = finalValues
touchThat = pd.DataFrame(touchthis, columns=['city','p','q','mse','rmse','r2','mape'])
#group by city
touchThat = touchThat.groupby('city')
for name, group in touchThat:
    mapeList = group['mape'].tolist()
    mapPD = pd.DataFrame(mapeList)
    MAPE2d = mapPD.values.reshape(6,4)
    print(MAPE2d)
    sb.heatmap(MAPE2d, annot=True, cmap="YlGnBu", fmt=".2f", linewidths=.5,
            xticklabels=[1,2,3,4], yticklabels=[1,2,3,4,5,6])
    plt.title(f'MAPE - {name}')
    plt.xlabel("q")
    plt.ylabel("p")
    # plt.show()
    plt.savefig(f'MAPE - {name}')
    plt.clf()
```