submitted_assignment3_part1

January 7, 2021

[1]:
```
version = "v1.8.100820"
```

---

1 Assignment 3 Part 1: Single Time Series Forecasting (50 pts)

In this assignment, we're going to practise forecasting a single time series.

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

# Suppress warnings
import warnings
from statsmodels.tools.sm_exceptions import ValueWarning
warnings.simplefilter("ignore", ValueWarning)
```

We will explore the same time series about **daily new COVID-19 cases globally** as we had in **Assignment 2 Part 1**. In order not to reinvent the wheel, let's import the `load_data` function you wrote previously.

[3]:
```
# Copy and paste the function you wrote in Assignment 2 Part 1 here and import␣
↪any libraries necessary
# We have tried a more elegant solution by using
# from ipynb.fs.defs.assignment2_part1 import load_data
# but it doesn't work with the autograder...

def load_data():
    daily_new_cases = None

    covid_df = pd.read_csv('assets/time_series_covid19_confirmed_global.csv')
#     print(covid_df.head())

    #try melting
```

```
    covid_melty = pd.melt(covid_df, id_vars = ['Country/Region', 'Province/
↪State'], value_vars = covid_df.columns[4:],
                        var_name = 'Date', value_name = 'Cumulative_Cases')
#     print(covid_melty.tail(5))


    #try to group by date, then later take the diff of each column before...
    covid_groupdate = covid_melty.groupby('Date').sum().reset_index()
#     print(covid_groupdate.head())

    #Create a column that converts every Date string into a pd.DatetimeIndex,␣
↪then set this as the index and drop the old
    #Date column
    covid_groupdate['Date_Time'] = pd.to_datetime(covid_groupdate['Date'])
    covid_groupdate.sort_values(by='Date_Time',inplace = True)
    covid_groupdate.drop('Date', axis = 1, inplace = True)
    covid_groupdate.set_index('Date_Time', inplace = True)
#     covid_groupdate.head()

    #Take the difference between every day its respective next day.
    #Rename the column to New cases as calculated be .diff()
    #Drop NA rows. The top row after running .diff() is always NA as there was␣
↪no day before it
    covid_new_cases = covid_groupdate.diff()
    covid_new_cases.rename(columns = {'Cumulative_Cases':'New_Cases'},␣
↪inplace=True)
    covid_new_cases.dropna(inplace=True)
#     covid_new_cases


    daily_new_cases = covid_new_cases['New_Cases']


    return daily_new_cases
```

[4]:
```
# Sanity checks to make sure you have imported the correct function - no points␣
↪awarded

stu_ans = load_data()

assert isinstance(stu_ans, pd.Series), "Q0: Your function should return a pd.
↪Series. "
assert len(stu_ans) == 212, "Q0: The length of the series returned is incorrect.
↪ "
assert isinstance(stu_ans.index, pd.DatetimeIndex), "Q0: The index of your␣
↪series must be a pd.DatetimeIndex. "
```

```
assert (("2020-01-23" <= stu_ans.index) & (stu_ans.index <= "2020-08-21")).
↪all(), "Q0: The index of your series contains an incorrect time range. "
assert not stu_ans.isna().any(), "Q0: Your series contains NaN values. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q0: Your series should have␣
↪a float dtype. "

del stu_ans
```

1.1 Question 1: Stationarity Tests (20 pts)

Let's first try to understand whether our time series is stationary or not. Recall that a stationary time series has stable statistics, such as constant means and variances, over time. A non-stationary time series would not be very interesting to study, as it is essentially equivalent to a white noise, carrying little information.

1.1.1 Question 1a (15 pts)

One way of determining stationarity is to calculate some summary statistics. A rolling mean and a rolling standard deviation are the mean and the standard deviation over a rolling window of values. They both have the same length as the original time series. For a rolling window of size k, the j-th component of the rolling mean μ and the rolling standard deviation σ is precisely defined as:

$$\mu_j = \frac{1}{\min(k,j)} \sum_{i=\max(1,j-k+1)}^{j} x_i = \begin{cases} \frac{1}{j}\sum_{i=1}^{j} x_i & \text{if } j \leq k \\ \frac{1}{k}\sum_{i=j-k+1}^{j} x_i & \text{if } j > k \end{cases} \tag{1}$$

$$\sigma_j = \sqrt{\frac{1}{\min(k,j)} \sum_{i=\max(1,j-k+1)}^{j} \left(x_i - \mu_j\right)^2} = \begin{cases} \sqrt{\frac{1}{j}\sum_{i=1}^{j} \left(x_i - \mu_j\right)^2} & \text{if } j \leq k \\ \sqrt{\frac{1}{k}\sum_{i=j-k+1}^{j} \left(x_i - \mu_j\right)^2} & \text{if } j > k \end{cases} \tag{2}$$

where $j \geq 1$.

Complete the function below that takes as input a time series and that calculates the rolling mean and the rolling standard deviation of the input time series. The size of the rolling window is governed by the argument `wd_size`.

This function should return a `tuple` of length 2, whose first component is the rolling mean as a `np.ndarray` and whose last component is the rolling standard deviation as a `np.ndarray`.

[5]:
```
import math

def calc_rolling_stats(ser, wd_size=7):
    """
    Takes in a series and returns the rolling mean and the rolling std for a␣
↪window of size wd_size
    """
    # YOUR CODE HERE

    rolling_mean = []
    rolling_std = []
```

```python
for j in range(1,len(ser)+1):
    if j < wd_size:
        short_wd = ser[:j]

        #Get rolling mean for window size short_wd
        accum_vals_mean = []

        for i in range(len(short_wd)):
            accum_vals_mean.append(short_wd[i])

        new_val_mean = sum(accum_vals_mean)/len(short_wd)

        rolling_mean.append(new_val_mean)

        #Get rolling std for windown size short_wd
        accum_vals_std = []

        for i in range(len(short_wd)):
            accum_vals_std.append((short_wd[i]-new_val_mean)**2)

        new_val_std = math.sqrt(sum(accum_vals_std)/len(short_wd))

        rolling_std.append(new_val_std)

    else:
        full_wd = ser[j-wd_size:j]

        #Get rolling mean for full window sieze full_wd
        accum_vals_mean = []

        for i in range(len(full_wd)):
            accum_vals_mean.append(full_wd[i])

        new_val_mean = sum(accum_vals_mean)/len(full_wd)

        rolling_mean.append(new_val_mean)

        #Get rolling std for windown size full_wd
        accum_vals_std = []

        for i in range(len(full_wd)):
            accum_vals_std.append((full_wd[i]-new_val_mean)**2)

        new_val_std = math.sqrt(sum(accum_vals_std)/len(full_wd))

        rolling_std.append(new_val_std)
```

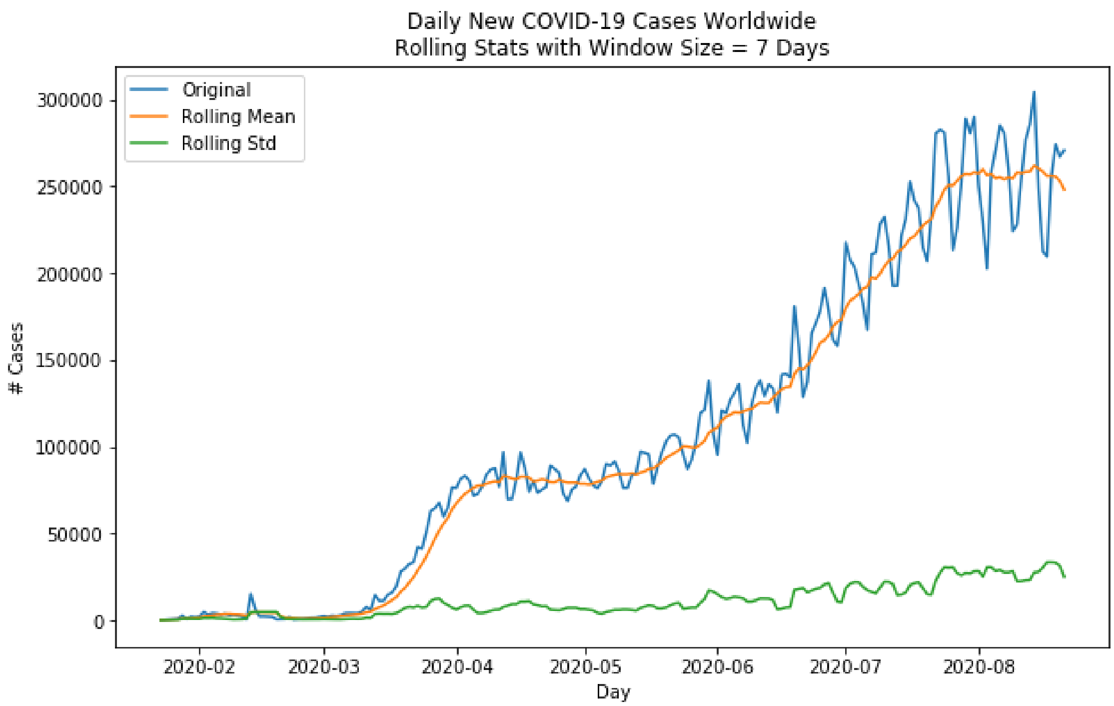
```
    rolling_mean, rolling_std = np.asarray(rolling_mean), np.
 ↪asarray(rolling_std)






    return rolling_mean, rolling_std
```

[6]:
```
# Autograder tests

stu_ser, wd_size = load_data(), 7
stu_ans = calc_rolling_stats(stu_ser, wd_size)

assert isinstance(stu_ans, tuple), "Q1a: Your function should return a tuple. "
assert len(stu_ans) == 2, "Q1a: The length of the tuple returned is incorrect. "
assert isinstance(stu_ans[0], np.ndarray), "Q1a: Please return the rolling mean␣
 ↪as np.ndarray. "
assert isinstance(stu_ans[1], np.ndarray), "Q1a: Please return the rolling std␣
 ↪as np.ndarray. "
assert len(stu_ans[0]) == len(stu_ser), "Q1a: Your rolling mean should be of␣
 ↪the same length as your data. "
assert len(stu_ans[1]) == len(stu_ser), "Q1a: Your rolling std should be of the␣
 ↪same length as your data. "
assert np.issubdtype(stu_ans[0].dtype, np.floating), "Q1a: Your rolling mean␣
 ↪should have a float dtype. "
assert np.issubdtype(stu_ans[1].dtype, np.floating), "Q1a: Your rolling std␣
 ↪should have a float dtype. "


# Some hidden tests


del stu_ans, stu_ser, wd_size
```

Let's plot and see the rolling statistics together with the original time series. Is our time series stationary? Why or why not?

[7]:
```
# Let's plot and see the rolling statistics

ser, wd_size = load_data(), 7
rolling_mean, rolling_std = calc_rolling_stats(ser, wd_size)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(ser, label="Original")
ax.plot(pd.Series(rolling_mean, index=ser.index), label="Rolling Mean")
```

```
ax.plot(pd.Series(rolling_std, index=ser.index), label="Rolling Std")

ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide\n" + f"Rolling Stats with␣
↪Window Size = {wd_size} Days")
ax.legend()

del fig, ax, ser, wd_size, rolling_mean, rolling_std
```



1.1.2 Question 1b (5 pts)

Now let's see whether the *log return* of our time series is stationary. Complete the function below that computes the log return of a given time series and that returns the result as a `pd.Series` like the following:

```
2020-01-24     1.064362
2020-01-25     0.541027
2020-01-26     0.327449
2020-01-27     0.167841
2020-01-28     1.186893
                  ...
2020-08-17    -0.013336
2020-08-18     0.196096
```

```
2020-08-19    0.072750
2020-08-20   -0.026456
2020-08-21    0.013266
Length: 211, dtype: float64
```

where * the index of the series is a `pd.DatetimeIndex`; * the values of the series are the log returns; and * the series doesn't contain any `NaN` values.

This function should return a `pd.Series`, whose index is a `pd.DatetimeIndex`.

[8]:
```
def calc_log_ret(ser):
    """
    Takes in a series and computes the log return
    """

#     log_ret = ser.apply(lambda x:(np.log(x) - np.log(x - 1)) if x != ser[0])
    log_ret = pd.Series([np.log(ser[i]) - np.log(ser[i-1]) for i in␣
 ↪range(1,len(ser))], index = ser.index[1:])
    print(log_ret)
    # YOUR CODE HERE

    return log_ret
```

[9]:
```
# Autograder tests

stu_ser = load_data()
stu_ans = calc_log_ret(stu_ser)

assert isinstance(stu_ans, pd.Series), "Q1b: Your function should return a pd.
 ↪Series. "
assert len(stu_ans) == len(stu_ser) - 1, "Q1b: The length of the series␣
 ↪returned should be one less than that of your data. "
assert isinstance(stu_ans.index, pd.DatetimeIndex), "Q1b: The index of your␣
 ↪series must be a pd.DatetimeIndex. "
assert (("2020-01-24" <= stu_ans.index) & (stu_ans.index <= "2020-08-21")).
 ↪all(), "Q1b: The index of your series contains an incorrect time range. "
assert not stu_ans.isna().any(), "Q1b: Your series contains NaN values. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q1b: Your series should have␣
 ↪a float dtype. "

# Some hidden tests


del stu_ans, stu_ser
```

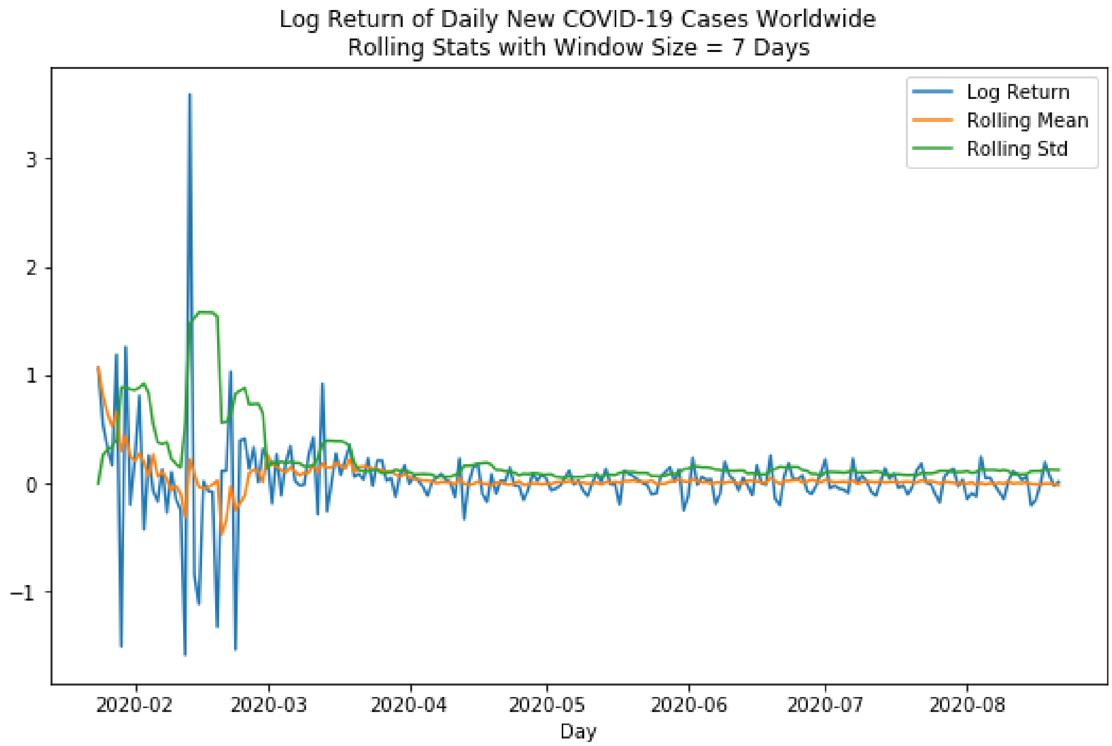
```
Date_Time
2020-01-24    1.064362
2020-01-25    0.541027
2020-01-26    0.327449
2020-01-27    0.167841
```

```
2020-01-28    1.186893
                ...
2020-08-17   -0.013336
2020-08-18    0.196096
2020-08-19    0.072750
2020-08-20   -0.026456
2020-08-21    0.013266
Length: 211, dtype: float64
```

This time let's plot and see the rolling statistics together with the log returns. Are the log returns of our time series stationary? Why or why not?

[10]:
```python
# Let's plot and see the rolling statistics

log_ret, wd_size = calc_log_ret(load_data()), 7
rolling_mean, rolling_std = calc_rolling_stats(log_ret, wd_size)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(log_ret, label="Log Return")
ax.plot(pd.Series(rolling_mean, index=log_ret.index), label="Rolling Mean")
ax.plot(pd.Series(rolling_std, index=log_ret.index), label="Rolling Std")

ax.set_xlabel("Day")
ax.set_title("Log Return of Daily New COVID-19 Cases Worldwide\n" + f"Rolling␣
↪Stats with Window Size = {wd_size} Days")
ax.legend()

del fig, ax, log_ret, wd_size, rolling_mean, rolling_std
```

```
Date_Time
2020-01-24    1.064362
2020-01-25    0.541027
2020-01-26    0.327449
2020-01-27    0.167841
2020-01-28    1.186893
                ...
2020-08-17   -0.013336
2020-08-18    0.196096
2020-08-19    0.072750
2020-08-20   -0.026456
2020-08-21    0.013266
Length: 211, dtype: float64
```



Yet another way of determining stationarity would be to use a statistical test, such as the Augmented Dickey-Fuller unit root test. The null hypothesis is usually that the time series is non-stationary. A p-value less than 0.05 would lead to the conclusion that the time series is stationary, although some scientists have rised up against this magic numer!

[11]:
```python
# An example of performing an Augmented Dickey-Fuller unit root test

from statsmodels.tsa.stattools import adfuller

_, pval, *_ = adfuller(load_data())
print(f"p-value: {pval}")

del adfuller, pval
```

```
p-value: 0.67658525115441
```

1.2 Question 2: Autocorrelations (10 pts)

Observations in a time series are often not isolated but rather correlated. That is, there might be a correlation between an observation y_t and another observation y_{t-k} that is k time steps (or *lags*) earlier. (Partial) autocorrelations precisely capture this idea.

1.2.1 Question 2a (5 pts)

Complete the function below to calculate the **Autocorrelation Function (ACF)** of the input time series, with the maximum number of lags to consider specified by the parameter `max_lag`. You may use the `acf` function from the `statsmodels` library.

This function should return a `np.ndarray` of length `max_lag + 1`.

[12]:
```
from statsmodels.tsa.stattools import acf, pacf

def calc_acf(ser, max_lag):
    """
    Takes a series and calculates the ACF
    """
    # YOUR CODE HERE
    ans_acf = acf(ser, nlags = max_lag)
    print(len(ans_acf))
    print(ans_acf)

    return ans_acf
```

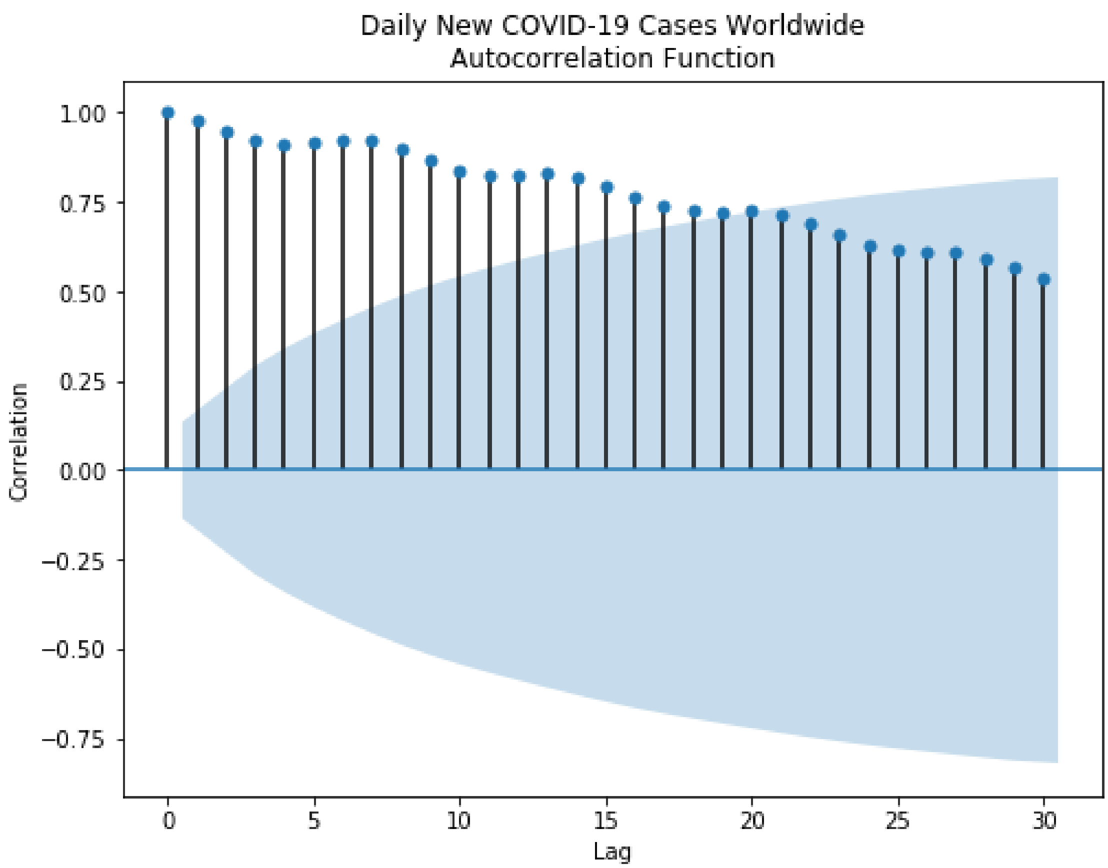
[13]:
```
# Autograder tests

stu_ser, max_lag = load_data(), 30
stu_ans = calc_acf(stu_ser, max_lag)

assert isinstance(stu_ans, np.ndarray), "Q2a: Your function should return a np.
↪ndarray. "
assert len(stu_ans) == max_lag + 1, "Q2a: The length of the ACF returned is␣
↪incorrect. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q2a: Your np.ndarray should␣
↪have a float dtype. "


# Some hidden tests


del stu_ans, stu_ser, max_lag
```

```
31
[1.         0.97535964 0.94724559 0.92385736 0.91386313 0.9167115
 0.92474083 0.92029816 0.89570334 0.86531167 0.83986903 0.82678187
 0.82650348 0.82935306 0.82125569 0.79719604 0.76575177 0.74163109
 0.72568749 0.72337754 0.72382142 0.71434956 0.68860387 0.65799162
 0.63121293 0.61553004 0.61048868 0.60786787 0.59472779 0.56872414
 0.53846277]
```

Let's see a plot of the ACF.

[14]:
```
from statsmodels.graphics.tsaplots import plot_acf
```

```
ser, max_lag = load_data(), 30

fig, ax = plt.subplots(1, 1, figsize=(8, 6))

plot_acf(ser, ax, lags=max_lag, title="Daily New COVID-19 Cases␣
 ↪Worldwide\nAutocorrelation Function")
ax.set_xlabel(r"Lag")
ax.set_ylabel(r"Correlation")

del fig, ax, ser, max_lag, plot_acf
```



1.2.2 Question 2b (5 pts)

Complete the function below to calculate the **Partial Autocorrelation Function (PACF)** of the input time series, with the maximum number of lags to consider specified by the parameter `max_lag`. You may use the `pacf` function from the `statsmodels` library.

This function should return a `np.ndarray` of length `max_lag + 1`.

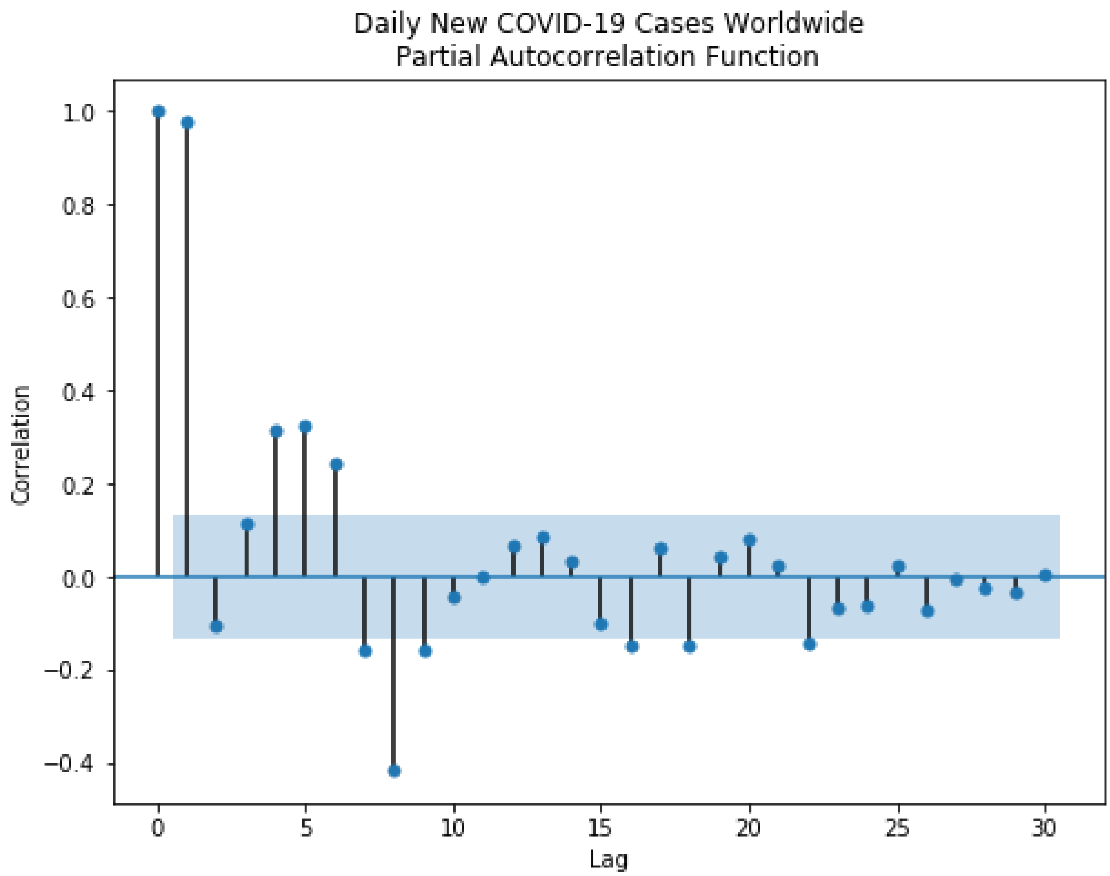
[15]:
```
def calc_pacf(ser, max_lag):
    """
```

```
    Takes a series and calculates the PACF
    """

    # YOUR CODE HERE
    ans_pacf = pacf(ser, nlags = max_lag)

    return ans_pacf
```

[16]:
```
# Autograder tests

stu_ser, max_lag = load_data(), 30
stu_ans = calc_pacf(stu_ser, max_lag)

assert isinstance(stu_ans, np.ndarray), "Q2b: Your function should return a np.
 ↪ndarray. "
assert len(stu_ans) == max_lag + 1, "Q2b: The length of the PACF returned is␣
 ↪incorrect. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q2b: Your np.ndarray should␣
 ↪have a float dtype. "

# Some hidden tests


del stu_ans, stu_ser, max_lag
```

Let's see a plot of the PACF.

[17]:
```
from statsmodels.graphics.tsaplots import plot_pacf

ser, max_lag = load_data(), 30

fig, ax = plt.subplots(1, 1, figsize=(8, 6))

plot_pacf(ser, ax, lags=max_lag, title="Daily New COVID-19 Cases␣
 ↪Worldwide\nPartial Autocorrelation Function")
ax.set_xlabel(r"Lag")
ax.set_ylabel(r"Correlation")

del fig, ax, ser, max_lag, plot_pacf
```



1.3 Question 3: ARMA on Log Returns (10 pts)

Complete the function below that fits an ARMA(p, q) model on the **log return** of an input series. Your function should return a multi-day forecast in the original data space (i.e., the number of daily new cases globally) starting from 2020-08-22. For example, if `num_forecasts=20`, your function should return a `pd.Series` similar to

```
2020-08-22    239936.746954
2020-08-23    237307.407386
2020-08-24    240073.408295
                  ...
2020-09-08    279778.977067
2020-09-09    307210.157343
2020-09-10    305203.431533
Freq: D, Name: predicted_mean, dtype: float64
```

where * the index of the series is a `pd.DatetimeIndex`; * the values of the series are the forecasted daily new cases; and * the series doesn't contain any `NaN` values.

This question is graded on the Root Mean Sqaure Error (RMSE) of your forecasts. You have complete freedom in how you'd like to implement the function, but one recommended API to use is the `ARIMA` class from the `statsmodels` library. Why do we recommend `ARIMA`, when the question actually asks for a ARMA(p, q) model? Hopefully you'll find it out while working on the implementation!

This function should return a `pd.Series` of length `num_forecasts`, whose index is a `pd.DatetimeIndex`.

[ ]:

```

```

[31]:

```
from statsmodels.tsa.arima.model import ARIMA

def arma_log_ret(ser, p, q, num_forecasts):
    """
    Takes a series and fits an ARMA(p, q) model on log return.
    Returns a number of forecasts as specified by num_forecasts.
    """
    # YOUR CODE HERE
    model = ARIMA(ser, order = (p, 0, q))
    fit = model.fit()
    forecasts = fit.forecast(steps = num_forecasts)

    return forecasts
```

[32]:

```
test3 = arma_log_ret(load_data(), 7, 7, 20)
test3
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  ConvergenceWarning)
```

[32]:

```
2020-08-22    241043.375514
2020-08-23    206304.507850
2020-08-24    208371.721607
2020-08-25    241506.101457
2020-08-26    260975.128654
2020-08-27    271461.938620
2020-08-28    269353.195719
2020-08-29    241718.980512
2020-08-30    205894.069500
2020-08-31    209840.019282
2020-09-01    239518.611886
2020-09-02    260496.912092
2020-09-03    270234.552301
2020-09-04    269478.643678
2020-09-05    240921.633802
2020-09-06    206845.999158
2020-09-07    210068.445739
2020-09-08    238568.245742
2020-09-09    259263.322474
```

```
2020-09-10    269587.394470
Freq: D, Name: predicted_mean, dtype: float64
```

[33]:
```
# Autograder tests

stu_ser = load_data()
p, q, num_forecasts = 7, 7, 20

stu_ans = arma_log_ret(stu_ser, p, q, num_forecasts)

assert isinstance(stu_ans, pd.Series), "Q3: Your function should return a pd.
↪Series. "
assert len(stu_ans) == num_forecasts, "Q3: The length of the series returned is␣
↪incorrect. "
assert isinstance(stu_ans.index, pd.DatetimeIndex), "Q3: The index of your␣
↪series must be a pd.DatetimeIndex. "
assert (("2020-08-22" <= stu_ans.index) & (stu_ans.index <= "2020-09-10")).
↪all(), "Q3: The index of your series contains an incorrect time range. "
assert not stu_ans.isna().any(), "Q3: Your series contains NaN values. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q3: Your series should have␣
↪a float dtype. "

# Some hidden tests


del stu_ser, stu_ans, p, q, num_forecasts
```

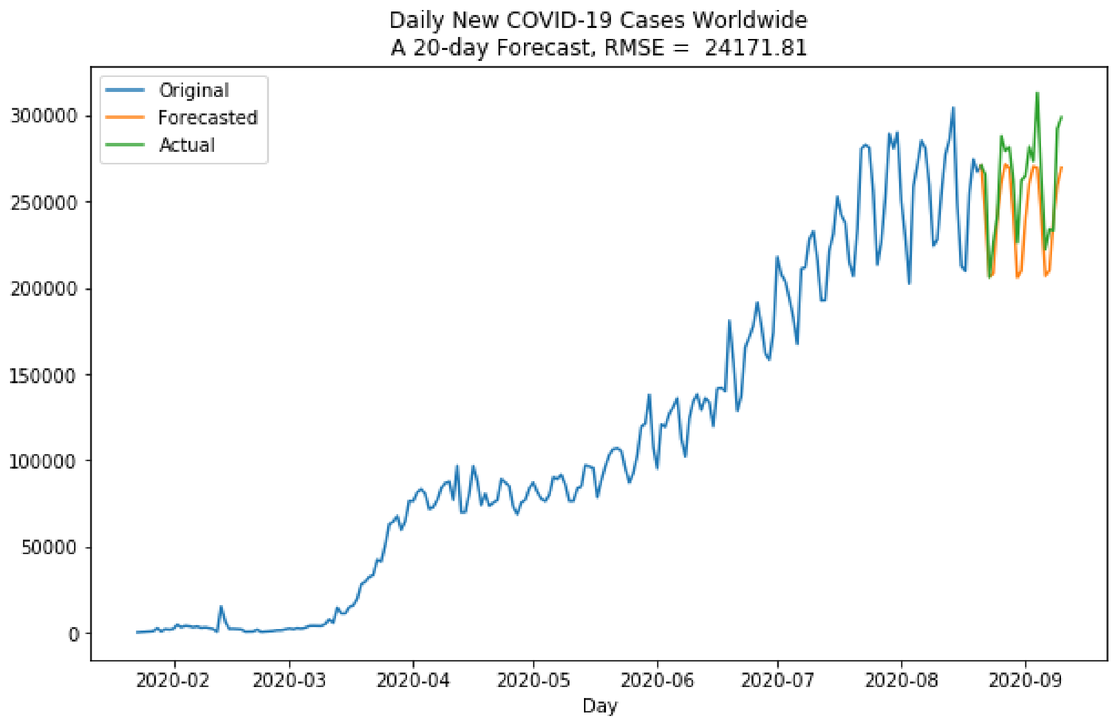
```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  ConvergenceWarning)
```

Now let's plot and compare the original time series, your forecasts and the ground-truth values of your forecasts.

[34]:
```
ser = load_data()
p, q, num_forecasts = 7, 7, 20

forecasts = arma_log_ret(ser, p, q, num_forecasts)
actual = pd.read_pickle("assets/actual.pkl")
rmse = np.sqrt(np.mean((actual - forecasts) ** 2))

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(ser, label="Original")
ax.plot(ser[-1:].append(forecasts), label="Forecasted")
ax.plot(ser[-1:].append(actual), label="Actual")

ax.set_xlabel("Day")
```

```
ax.set_title("Daily New COVID-19 Cases Worldwide\n" + f"A {len(forecasts)}-day␣
↪Forecast, RMSE = {rmse: .2f}")
ax.legend()

del fig, ax, ser, p, q, num_forecasts, forecasts, actual
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  ConvergenceWarning)
```



1.4 Question 4: ARMA on First-order Differences (10 pts)

Complete the function below that fits an ARMA(p, q) model on the **first-order differences** of an input series. Your function should return a multi-day forecast in the original data space (i.e., the number of daily new cases globally) starting from `2020-08-22`. For example, if `num_forecasts=20`, your function should return a `pd.Series` similar to

```
2020-08-22    242994.084820
2020-08-23    205194.792913
2020-08-24    201803.644029
                 ...
2020-09-08    214574.419936
```

```
2020-09-09    243506.281330
2020-09-10    253847.751339
Freq: D, Name: predicted_mean, dtype: float64
```

where * the index of the series is a `pd.DatetimeIndex`; * the values of the series are the forecasted daily new cases; and * the series doesn't contain any `NaN` values.

This question is graded on the Root Mean Sqaure Error (RMSE) of your forecasts. You have complete freedom in how you'd like to implement the function, but one recommended API to use is the `ARIMA` class from the `statsmodels` library. Why do we recommend `ARIMA`, when the question actually asks for a ARMA(p, q) model? Again, hopefully you'll find it out while working on the implementation!

This function should return a `pd.Series` of length `num_forecasts`, whose index is a `pd.DatetimeIndex`.

[35]:
```
from statsmodels.tsa.arima.model import ARIMA

def arma_first_diff(ser, p, q, num_forecasts):
    """
    Takes a series and fits an ARMA(p, q) model on first-order diff.
    Returns a number of forecasts as specified by num_forecasts.
    """

    model = ARIMA(ser, order = (p, 0, q))
    fit = model.fit()
    forecasts = fit.forecast(steps = num_forecasts)

    return forecasts
```

[36]:
```
# Autograder tests

stu_ser = load_data()
p, q, num_forecasts = 7, 7, 20

stu_ans = arma_first_diff(stu_ser, p, q, num_forecasts)

assert isinstance(stu_ans, pd.Series), "Q4: Your function should return a pd.
↪Series. "
assert len(stu_ans) == num_forecasts, "Q4: The length of the series returned is␣
↪incorrect. "
assert isinstance(stu_ans.index, pd.DatetimeIndex), "Q4: The index of your␣
↪series must be a pd.DatetimeIndex. "
assert (("2020-08-22" <= stu_ans.index) & (stu_ans.index <= "2020-09-10")).
↪all(), "Q4: The index of your series contains an incorrect time range. "
assert not stu_ans.isna().any(), "Q4: Your series contains NaN values. "
assert np.issubdtype(stu_ans.dtype, np.floating), "Q4: Your series should have␣
↪a float dtype. "

# Some hidden tests
```

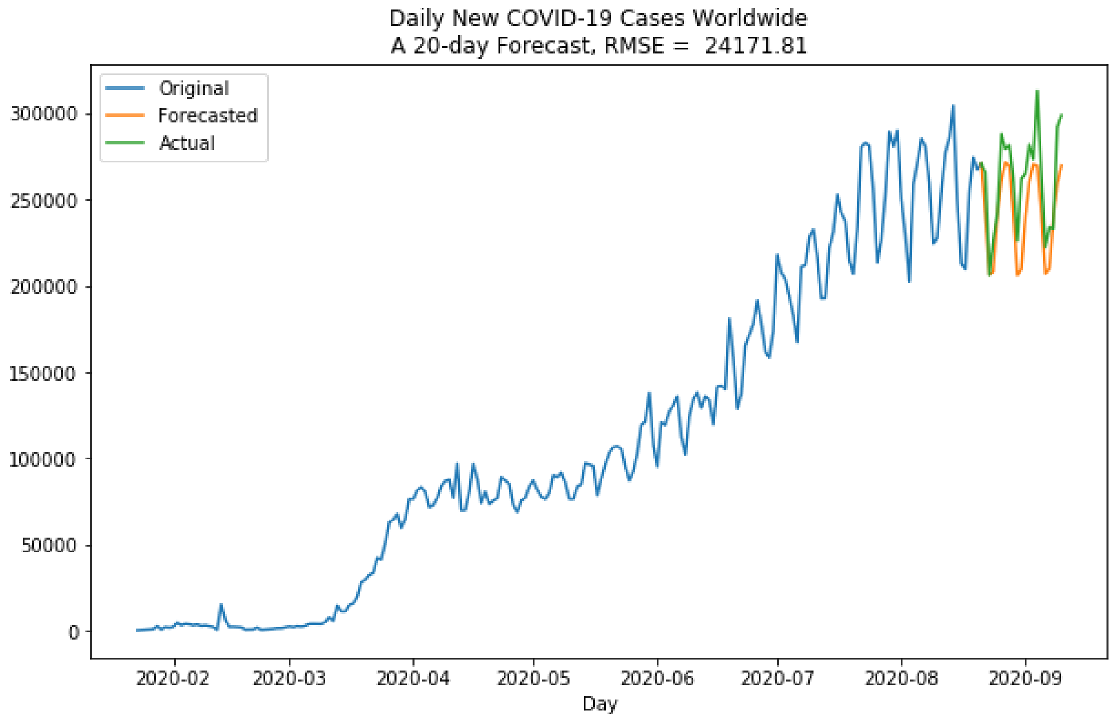
```
del stu_ser, stu_ans, p, q, num_forecasts
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  ConvergenceWarning)
```

Now let's plot and compare the original time series, your forecasts and the ground-truth values of your forecasts. How does this compare with the one trained on log returns?

[37]:
```
ser = load_data()
p, q, num_forecasts = 7, 7, 20

forecasts = arma_first_diff(ser, p, q, num_forecasts)
actual = pd.read_pickle("assets/actual.pkl")
rmse = np.sqrt(np.mean((actual - forecasts) ** 2))

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(ser, label="Original")
ax.plot(ser[-1:].append(forecasts), label="Forecasted")
ax.plot(ser[-1:].append(actual), label="Actual")

ax.set_xlabel("Day")
ax.set_title("Daily New COVID-19 Cases Worldwide\n" + f"A {len(forecasts)}-day␣
 ↪Forecast, RMSE = {rmse: .2f}")
ax.legend()

del fig, ax, ser, p, q, num_forecasts, forecasts, actual
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  ConvergenceWarning)
```


Daily New COVID-19 Cases Worldwide
A 20-day Forecast, RMSE = 24171.81
Original
Forecasted
Actual
300000
250000
200000
150000
100000
50000
0
2020-02
2020-03
2020-04
2020-05
2020-06
2020-07
2020-08
2020-09
Day

[ ]: