original_assignment2_part1

January 7, 2021

[1]:
```
version = "v1.6.092820"
```

---

1 Assignment 2 Part 1: Time Series Patterns (50 pts)

In this assignment, we're going to practise some techniques that are useful for discerning patterns in a 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 all warnings
import warnings
warnings.filterwarnings("ignore")
```

[3]:
```
import datetime as dt
```

1.1 Question 1: Load Data (5 pts)

At the time of writing this assignment, August 2020, COVID-19 is still the most topical public-health crisis globally with nearly 300,000 new cases reported worldwide every day. **The number of daily new cases worldwide** is a time series that arises naturally from this topical event, and in this assignment we'll apply some of the techniques we learned in class to this very time series to discern any patterns it may contain.

You are provided with a csv file, `assets/time_series_covid19_confirmed_global.csv`, which is part of the Johns Hopkins University CSSE COVID-19 dataset. As the name suggests, it contains the number of *cumulative* confirmed cases globally as of certain dates. However, we are interested in the number of *new* cases worldwide every day.

Create a function called `load_data` that reads in the csv file and produces a `pd.Series` that looks like:

```
2020-01-23        99.0
2020-01-24       287.0
2020-01-25       493.0
2020-01-26       684.0
2020-01-27       809.0
                 ...
2020-08-17    209672.0
2020-08-18    255096.0
2020-08-19    274346.0
2020-08-20    267183.0
2020-08-21    270751.0
Length: 212, dtype: float64
```

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

This function should return a `pd.Series` of length 212, whose index is a `pd.DatetimeIndex`.

[4]:
```
covid_df = pd.read_csv('assets/time_series_covid19_confirmed_global.csv')
covid_df.head()
```

[4]:
```
  Province/State Country/Region       Lat       Long  1/22/20  1/23/20  \
0            NaN    Afghanistan  33.93911  67.709953        0        0
1            NaN        Albania  41.15330  20.168300        0        0
2            NaN        Algeria  28.03390   1.659600        0        0
3            NaN        Andorra  42.50630   1.521800        0        0
4            NaN         Angola -11.20270  17.873900        0        0

   1/24/20  1/25/20  1/26/20  1/27/20  ...  8/12/20  8/13/20  8/14/20  \
0        0        0        0        0  ...    37345    37424    37431
1        0        0        0        0  ...     6817     6971     7117
2        0        0        0        0  ...    36699    37187    37664
3        0        0        0        0  ...      977      981      989
4        0        0        0        0  ...     1762     1815     1852

   8/15/20  8/16/20  8/17/20  8/18/20  8/19/20  8/20/20  8/21/20
0    37551    37596    37599    37599    37599    37856    37894
1     7260     7380     7499     7654     7812     7967     8119
2    38133    38583    39025    39444    39847    40258    40667
3      989      989     1005     1005     1024     1024     1045
4     1879     1906     1935     1966     2015     2044     2068

[5 rows x 217 columns]
```

[5]:
```
def load_data():
    daily_new_cases = None

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

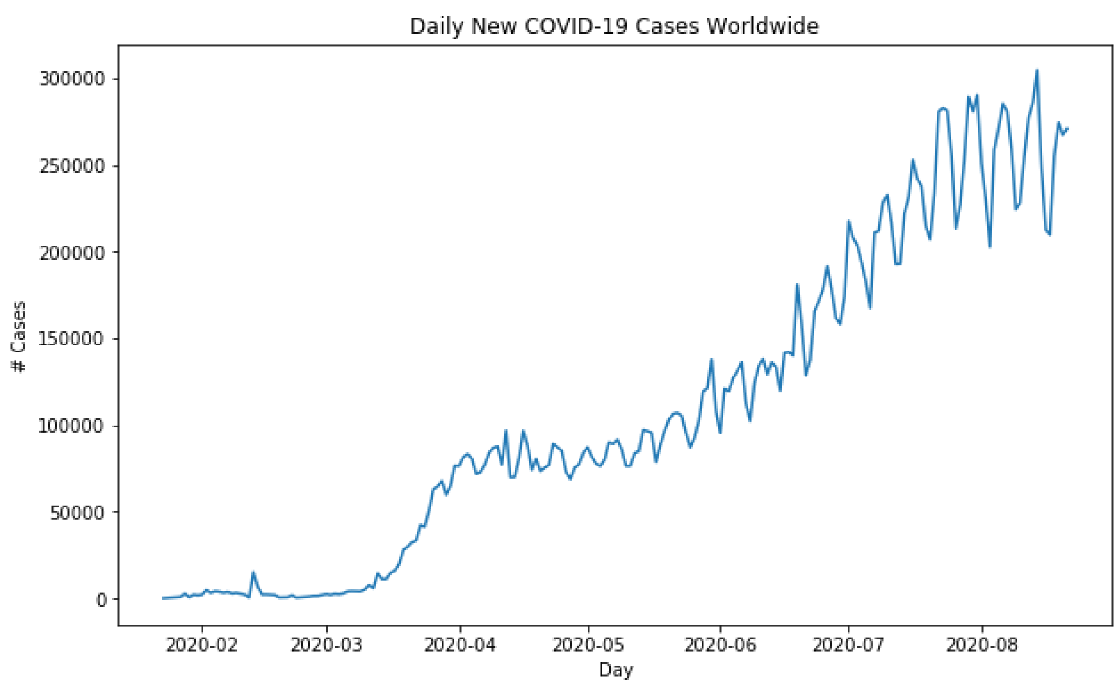
```
    #try melting
    covid_melty = pd.melt(covid_df, id_vars = ['Country/Region', 'Province/
 ↪State'], value_vars = covid_df.columns[4:],
                      var_name = 'Date', value_name = 'Cumulative_Cases')
#     print(covid_melty.tail(5))


    #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
```

[6]:

```
# Autograder tests

stu_ans = load_data()

assert isinstance(stu_ans, pd.Series), "Q1: Your function should return a pd.
 ↪Series. "
assert len(stu_ans) == 212, "Q1: The length of the series returned is incorrect.
 ↪ "
assert isinstance(stu_ans.index, pd.DatetimeIndex), "Q1: The index of your␣
 ↪series must be a pd.DatetimeIndex. "
```

```
assert (("2020-01-23" <= stu_ans.index) & (stu_ans.index <= "2020-08-21")).
 ↪all(), "Q1: The index of your series contains an incorrect time range. "
assert not stu_ans.isna().any(), "Q1: Your series contains NaN values. "

# Some hidden tests


del stu_ans
```

[7]:
```
# Let's plot and see the time series

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(load_data())
ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide")

del fig, ax
```



1.2 Question 2: Perform a Seasonal Decomposition (5 pts)

With the time series ready, let's first perform a seasonal decomposition using tools from the `statsmodels` library to get a sense of what the possible patterns are hidden in the data. Complete the function below that takes a time series and an argument `model`, which indicates whether an additive or multiplicative seasonal decomposition should be performed, and that returns a

DecomposeResult as produced by the seasonal_decompose function from the statsmodels library.

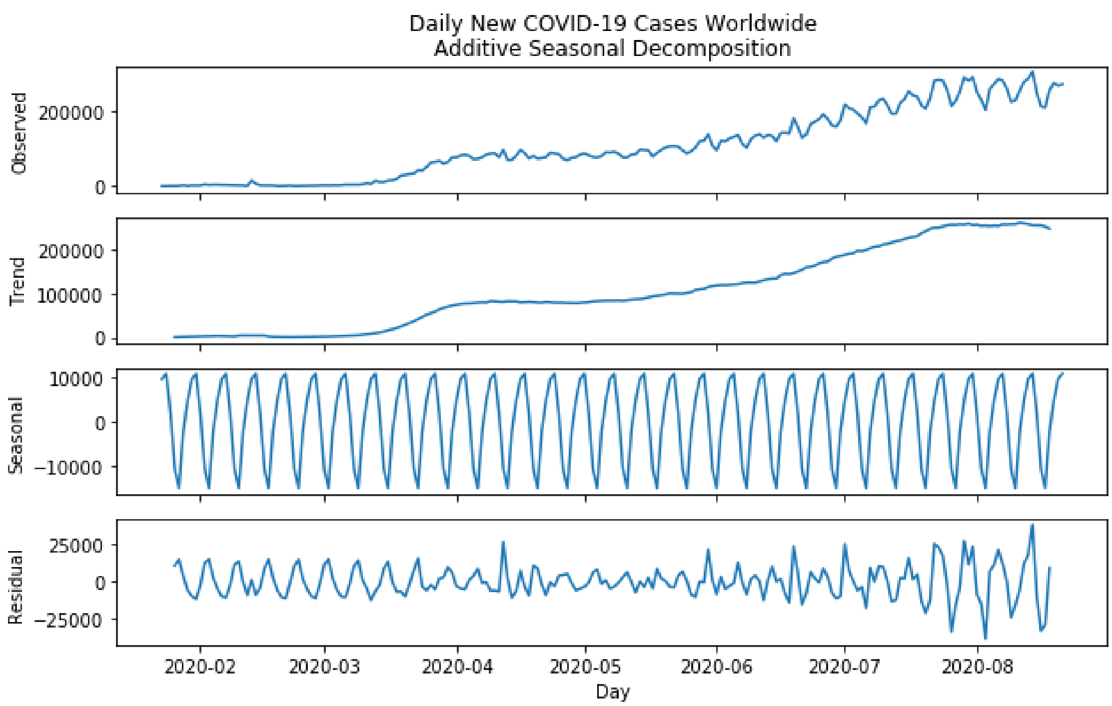
This function should return a statsmodels.tsa.seasonal.DecomposeResult.

[8]:
```
from statsmodels.tsa.seasonal import seasonal_decompose, DecomposeResult

def sea_decomp(ser, model="additive"):
    """
    Takes in a series and a "model" parameter indicating which seasonal decomp␣
 ↪to perform
    """
    result = None

    # YOUR CODE HERE
    result = seasonal_decompose(ser,model = model)

    return result
```

[9]:
```
# Autograder tests

stu_ser = load_data()
stu_ans = sea_decomp(stu_ser, model="additive")

assert isinstance(stu_ans, DecomposeResult), "Q2: Your function should return a␣
 ↪DecomposeResult. "

# Some hidden tests


del stu_ser, stu_ans
```

[10]:
```
# Let's plot and see the seasonal decomposition

fig, axes = plt.subplots(4, 1, figsize=(10, 6), sharex=True)
res = sea_decomp(load_data(), model="additive")

axes[0].set_title("Additive Seasonal Decomposition")
axes[0].plot(res.observed)
axes[0].set_ylabel("Observed")

axes[1].plot(res.trend)
axes[1].set_ylabel("Trend")

axes[2].plot(res.seasonal)
axes[2].set_ylabel("Seasonal")

axes[3].plot(res.resid)
axes[3].set_ylabel("Residual")
```

```
axes[3].set_xlabel("Day")
fig.suptitle("Daily New COVID-19 Cases Worldwide", x=0.513, y=0.95)

del fig, axes, res
```



1.3 Question 3: Fit a Trend Curve (15 pts)

The plot above suggests that there is a non-linear trend hidden in the time series. One approach to discover such a trend is to fit a regression model to the time series and ask the regression model to make predictions at each timestamp. When connected, these chronological predictions form a "trend curve". In the problem, we will explore how to fit a trend curve to our time series.

Complete the function below that fits an `n`-th order polynomial to the input time series and that returns the predictions as a `np.ndarray` of the same length. An n-th order polynomial regression model assumes that each dependent variable y_i is an n-th order polynomial function of the corresponding independent variable x_i:

$$y_i = c_0 + c_1 x_i + c_2 x_i^2 + \cdots + c_n x_i^n$$

Now, the most interesting and important question to think about is, "**what are x_i's and y_i's in the problem?**". The y_i's are the daily new cases worldwide at timestamps x_i's, but **how should we represent the timestamps x_i's in such a regression model?** There are many choices you may explore. In the function below, you are already given the code for training a polynomial regression model, but you have to figure out what `train_X` (x_i's) and `train_y` (y_i's) are. Since it's possible that everyone has a different design, this question is graded on the R^2 score of your predictions.

For a 10-th order polynomial regression model, at least one choice of x_i's leads to an R^2 score ≥ 0.95.

This function should return a `np.ndarray` of shape `(len(ser), )`, which represents the predictions of your polynomial regression model on the input time series. The predictions form the "trend curve" we are looking for.

[11]:
```
test = load_data()
test
```

[11]:
```
Date_Time
2020-01-23        99.0
2020-01-24       287.0
2020-01-25       493.0
2020-01-26       684.0
2020-01-27       809.0
                 ...
2020-08-17    209672.0
2020-08-18    255096.0
2020-08-19    274346.0
2020-08-20    267183.0
2020-08-21    270751.0
Name: New_Cases, Length: 212, dtype: float64
```

[12]:
```
# ser = test.reset_index()
# ser
```

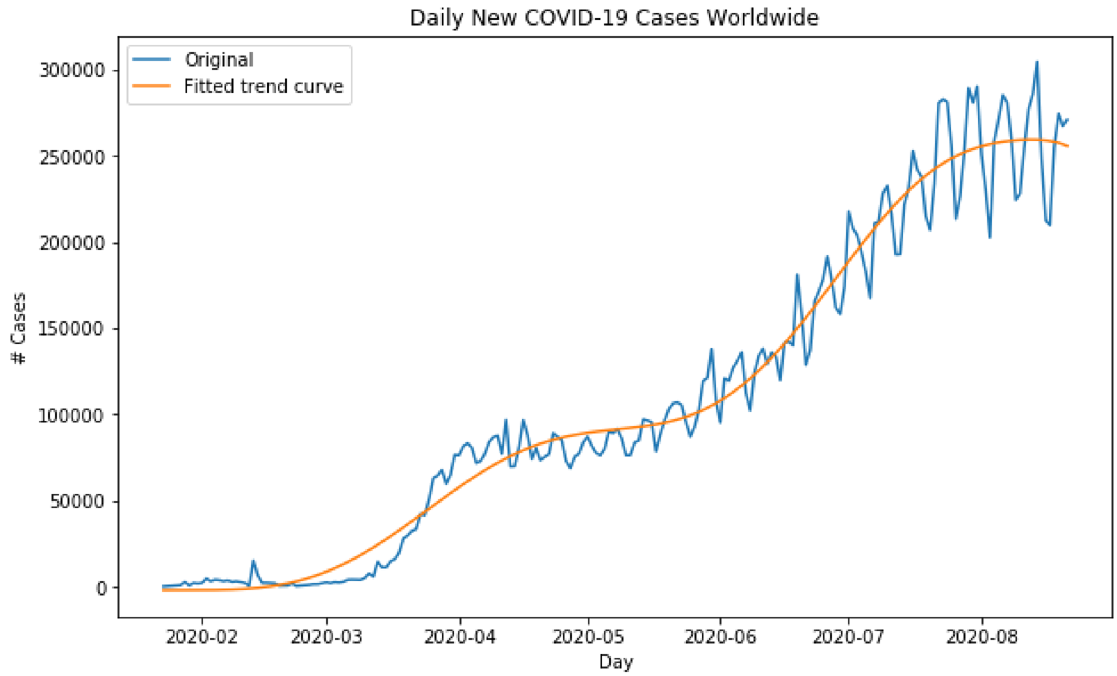
[13]:
```
# ser['datetime_ordinal'] = [i+1 for i in range(len(ser))]
# ser.head()
```

[14]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

def fit_trend(ser, n):
    """
    Takes a series and fits an n-th order polynomial to the series.
    Returns the predictions.
    """
    #Generate ordinal values to represent timestamps for use in regression
    # Create train_X and train_y
#     train_X, train_y = np.asarray(ser.index.map(dt.datetime.toordinal)), np.
 ↪asarray(ser) # xi's and yi's
    train_X, train_y = np.asarray([i+1 for i in range(len(ser.index))]), np.
 ↪asarray(ser)

    # Fit a polynomial regression model - code given to you

    train_X = PolynomialFeatures(n).fit_transform(train_X.reshape(-1, 1))

    lin_reg = LinearRegression().fit(train_X, train_y.reshape(-1))
```

```
    # Make predictions to create the trend curve
    # YOUR CODE HERE
    y_pred = np.asarray(lin_reg.predict(train_X))

    trend_curve = y_pred

    return trend_curve
```

[15]:
```
# Autograder tests

stu_ser = load_data()
stu_ans = fit_trend(stu_ser, 10)

assert isinstance(stu_ans, np.ndarray), "Q3: Your function should return a np.
 ↪ndarray. "
assert stu_ans.shape == (len(stu_ser), ), "Q3: The shape of your np.ndarray is␣
 ↪not correct. "

# Some hidden tests


del stu_ser, stu_ans
```

[16]:
```
# Let's plot and see your regression line

fig, ax = plt.subplots(figsize=(10, 6))
ser = load_data()
preds = fit_trend(ser, 10)
ax.plot(ser.index, ser.values, label="Original")
ax.plot(ser.index, preds, label="Fitted trend curve")
ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide")
ax.legend()

del fig, ax, ser, preds
```



It's worth mentioning that the `seaborn` library provides a function `regplot` that can plot both the data and the regression line in a few lines of code, thus saving you the trouble of fitting a regression model.

1.4 Question 4: Calculate Weighted Moving Average (WMA) (15 pts)

The regression method seems to give a fairly accurate description of the trend hidden in the time series. In this problem and the next, we will explore an alternative method for discovering trends that is based on moving averages.

Recall from the lectures that a Weighted Moving Average (WMA) method applies the following transformation to each data point x_j:

$$x'_j = \frac{w_k x_j + w_{k-1} x_{j-1} + \cdots + w_1 x_{j-k+1}}{w_k + w_{k-1} + \cdots + w_1}$$

for a window of size k. Complete the function below that calculates the WMA for an input time series.

This function should return a `np.ndarray` of shape `(len(ser), )` that represents the WMA values for the input time series.

[17]:
```
load_data()
```

[17]:
```
Date_Time
2020-01-23       99.0
2020-01-24      287.0
2020-01-25      493.0
2020-01-26      684.0
2020-01-27      809.0
```

```
                ...
2020-08-17    209672.0
2020-08-18    255096.0
2020-08-19    274346.0
2020-08-20    267183.0
2020-08-21    270751.0
Name: New_Cases, Length: 212, dtype: float64
```

[18]:
```
def calc_wma(ser, wd_size, weights=1):
    """
    Takes in a series and calculates the WMA with a window size of wd_size
    """
    wma = []
    #Set the number of weights to the window size
#     weights = np.arange(1, wd_size+1, 1)
    weights = np.arange(wd_size, 0, -1)
    print(weights)


    if isinstance(weights, int):
        weights = np.full(wd_size, weights)

    assert len(weights) == wd_size, "Q4: The size of the weights must be the␣
 ↪same as the window size. "

    # YOUR CODE HERE

    ser_len = len(ser)
    for i in range(1, ser_len+1):
        if i >= wd_size:
#             print("i", i)
            temp_window = ser[i-wd_size:i]
            #resort the weights so they line up in the j loop such that the␣
 ↪greatest weight goes with the most recent observ.
            new_weights = np.sort(weights)
#             print(new_weights)
#             print(temp_window)

            adjusted_vals = []
            for j in range(len(temp_window)):
                weighted_val = temp_window[j] * new_weights[j]
                adjusted_vals.append(weighted_val)

            new_val = sum(adjusted_vals)/sum(weights)

            wma.append(new_val)
```

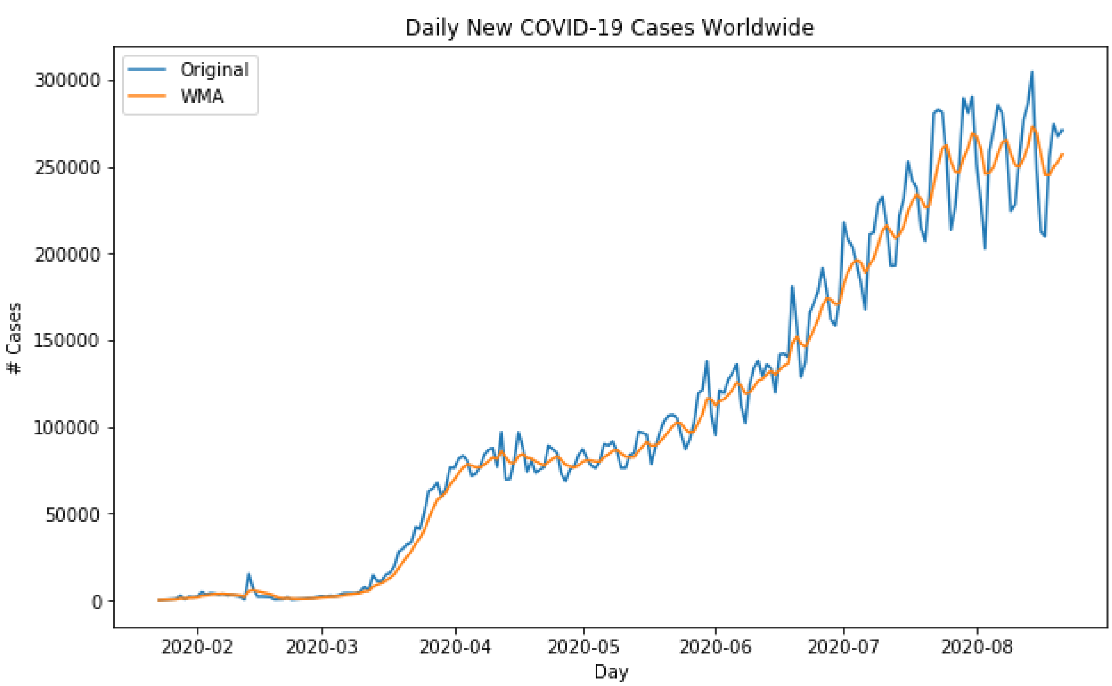
```
        else:
            print("i", i)
            new_weights = weights[:i]
            #resort the weights so they line up in the j loop such that the␣
 ↪greatest weight goes with the most recent observ.
            new_weights = np.sort(new_weights)
#             print(new_weights)
            temp_window = ser[:i]
#             print(temp_window)

            adjusted_vals = []
            for j in range(len(temp_window)):
                weighted_val = temp_window[j] * new_weights[j]
                adjusted_vals.append(weighted_val)

            new_val = sum(adjusted_vals)/sum(new_weights)

            wma.append(new_val)

    wma = np.asarray(wma)
#     print(len(wma))
#     print(len(ser))
    return wma
```

[19]:
```
# Autograder tests

wd_size = 7
weights = np.arange(1, wd_size + 1) # linear weighting
stu_ser = load_data()
stu_ans = calc_wma(stu_ser, wd_size, weights)

assert isinstance(stu_ans, np.ndarray), "Q4: Your function should return a np.
 ↪ndarray. "
assert stu_ans.shape == (len(stu_ser), ), "Q4: The np.ndarray returned is of an␣
 ↪incorrect shape. "

# Some hidden tests


del wd_size, weights, stu_ser, stu_ans
```

```
[7 6 5 4 3 2 1]
i 1
i 2
i 3
i 4
i 5
i 6
```

[20]:
```
# Let's plot and see your WMA

fig, ax = plt.subplots(figsize=(10, 6))
wd_size = 7
weights = np.arange(1, wd_size + 1)
ser = load_data()
wma = calc_wma(ser, wd_size, weights=weights)

ax.plot(ser.index, ser.values, label="Original")
ax.plot(ser.index, wma, label="WMA")
ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide")
ax.legend()

del fig, ax, wd_size, weights, ser, wma
```

```
[7 6 5 4 3 2 1]
i 1
i 2
i 3
i 4
i 5
i 6
```



1.5 Question 5: Calculate "Time" Exponential Moving Average (EMA) (10 pts)

WMA usually works well if each data point is sampled at regular time intervals (which is the case for our time series). "Time" Exponential Moving Average (EMA), on the other hand, works well on both regular and irregular time series. Let's now explore how to apply EMA to our time series.

Recall from the lectures that a "time" EMA method applies the following transformation to each data point x_j:

$$x'_j = \frac{\sum_{i=1}^{j} \exp\left[-\lambda\left(t_j - t_i\right)\right] x_i}{\sum_{i=1}^{j} \exp\left[-\lambda\left(t_j - t_i\right)\right]}$$

where $0 \leq \lambda \leq 1$ is the "decay rate". Also note that, when $\lambda = 0$, this is equivalent to a cumulative moving average (CMA). Complete the function below that calculates the "time" EMA for an input time series, **assuming the time intervals are days**.

This function should return a `np.ndarray` of shape `(len(ser), )`, which represents the "time" EMA for the input time series.

[21]:
```
def calc_time_ema(ser, lmbd=0.0):
    """
    Takes in a series and calculates EMA with the lambda provided
    """

    time_ema = None

    # YOUR CODE HERE
    ser = np.array(ser)

    return pd.ewm(ser, span = lmbd)[-1]
```

[22]:
```
# Autograder tests

stu_ser = load_data()

# Sanity checks for a trivial case - CMA
stu_ans = calc_time_ema(stu_ser, lmbd=0.0)

assert isinstance(stu_ans, np.ndarray), "Q5: Your function should return a np.
 ↪ndarray. "
assert stu_ans.shape == (len(stu_ser), ), "Q5: The np.ndarray returned is of an␣
 ↪incorrect shape. "
assert np.isclose(stu_ans, np.cumsum(stu_ser) / np.arange(1, len(stu_ser) + 1)).
 ↪all(), "Q5: When lmbd = 0 your function should calculate CMA. "


# Redefine the variable for hidden tests - lmbd=0.5
stu_ans = calc_time_ema(stu_ser, lmbd=0.5)


# Some hidden tests
```

```
del stu_ser, stu_ans
```

```
                                                                                
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-22-1a89577b600f> in <module>
         4
         5 # Sanity checks for a trivial case - CMA
   ----> 6 stu_ans = calc_time_ema(stu_ser, lmbd=0.0)
         7
         8 assert isinstance(stu_ans, np.ndarray), "Q5: Your function should␣
↪return a np.ndarray. "


       <ipython-input-21-2d2977816153> in calc_time_ema(ser, lmbd)
         9     ser = np.array(ser)
        10
   ---> 11     return pd.ewm(ser, span = lmbd)[-1]


       /opt/conda/lib/python3.7/site-packages/pandas/__init__.py in␣
↪__getattr__(name)
       256             return _SparseArray
       257
   --> 258         raise AttributeError(f"module 'pandas' has no attribute␣
↪'{name}'")
       259
       260


       AttributeError: module 'pandas' has no attribute 'ewm'
```

[ ]:
```
# Let's plot and see your time EMA

fig, ax = plt.subplots(figsize=(10, 6))
ser = load_data()
ema = calc_time_ema(ser, lmbd=0.5)

ax.plot(ser.index, ser.to_numpy(), label="Original")
ax.plot(ser.index, ema, label="Time EMA")
```

```python
ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide")
ax.legend()

del fig, ax, ser, ema
```

The SimpleExpSmoothing class from the `statsmodels` library is a handy tool for EMA. See an example below.

[ ]:
```python
from statsmodels.tsa.api import SimpleExpSmoothing

fig, ax = plt.subplots(figsize=(10, 6))

ser = load_data()
ema = SimpleExpSmoothing(ser, initialization_method=None).fit(smoothing_level=0.
↪5, optimized=False)

ax.plot(ser, label="Original")
ax.plot(ema.fittedvalues, label="EMA")
ax.set_xlabel("Day")
ax.set_ylabel("# Cases")
ax.set_title("Daily New COVID-19 Cases Worldwide")
ax.legend()

del ser, ema, fig, ax
```