QBUS6840: Tutorial 7 – Exponentially Weighted Smoothing II

Objectives

- Implement Holt-Winters' additive method manually
- Learn how to use `statsmodels` library to call Holt-Winters' method

In the previous tutorial, we learned how to use the simple exponential smoothing and (Holt's linear) trend method to smooth a time series and to forecast the time series. However, although we could obtain a reasonable outcome for simple cases, we have not considered the influence of seasonal variations, which may cause inaccurate forecasting result in some complicated situations.

In this tutorial, we will go one-step further, taking the seasonal variations into account in the forecasting/smoothing steps. By doing this, we will learn Holt-Winters smoothing algorithm.

Generally, for Holt-Winter smoothing algorithm, you still need to refer to the smoothing equation in simple exponential smoothing to smooth each component. When doing forecasts, we rely on the forecast equation and three smoothing equations - one for the level l_t, one for the trend/slope b_t and one for the seasonal component s_t, with corresponding smoothing parameters α, β and γ.

Mathematically, for **additive** Holt-Winters' method, the forecast is calculated as:

$$\widehat{y_{t+h|1:t}} = l_t + hb_t + \boldsymbol{s_{t+h-M(k+1)}}$$

the level is:

$$l_t = \alpha(y_t - \boldsymbol{s_{t-M}}) + (1-\alpha)(l_{t-1} + b_{t-1})$$

the trend is:

$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$

and the seasonal variation is:

$$\boldsymbol{s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-M}}$$

where M is the period number, k is the integer part $(h-1)/m$, which ensures that the estimates of the seasonal indices used for forecasting come from the final year of the sample. $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$ and $0 \leq \gamma \leq 1$ defines the weight decaying.

For one-step forecast, we have $h = 1$, hence $k = \frac{h-1}{M} = \frac{0}{M} = 0$, so the forecast is

$$\widehat{y_{t+1|1:t}} = l_t + 1 \times b_t + s_{t+1-M}$$

Holt-winters' additive (or multiplicative) method has three hyper-parameters (i.e. α, β, γ). Their values have influence on the forecast accuracy. Usually we shall select the besting fitting hyper-parameters by minimizing the SSE/MSE. This results in a nonlinear optimization problem over α, β, γ:

$$SSE = \sum_{t=1}^{n} (y_t - l_{t-1} - b_{t-1} - s_{t-m})^2.$$

Also we need to find the best initial level l_0, slope/trend b_0 and M initial seasonal values $s_0, s_{-1}, \dots, s_{-M+1}$. It is suggested to do a linear regression over the data $y_1, \dots, y_T$ to find the best fitting initial level l_0, slope/trend b_0:

$$\hat{y}_t = l_0 + b_0 t$$

Then calculate seasonal series as the residual, $\hat{s}_t = y_t - \hat{y}_t$, and then take the seasonal averages of $\hat{s}_t$ as $s_0, \dots, s_{1-M}$

For Holt-Winters multiplicative smoothing, please refer to the lecture handout and the guide book: https://otexts.com/fpp2/holt-winters.html

1. Holt-Winters smoothing for forecasting manually

Let's starting with manually implement the Holt-Winters additive smoothing in order to gain a better understand.

Step 1: Importing the libraries and dataset:

Load the libraries and datasets. Download the "`AustralianVisitors.csv`" file from the QBUS6840 Canvas site.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

visitors = pd.read_csv('AustralianVisitors.csv')
y = visitors['No of Visitors']
```

Step 2: Define a new addSeasonal() function:

In order to apply a Holt-Winters additive smoothing to the given time-series data, we need to obtain many existing information. Below is a check list. Make sure you obtained all these information:

Time series data	Observation y and time index t
Period number	M
Hyper parameters	α, β, and γ
Initial values	l_0, b_0, and $s = s_0, \dots, s_{1-M}$
Number of forecasting	fc

Calling this function, we hope to get the final smoothing results $\hat{y}$, the level l, slope b, the seasonal variations s and performance evaluation (such as SSE,RMSE, etc…) as the returning values.

Read the code below and pay attention to the comments to understand the logic.

```
from math import sqrt
from sklearn.linear_model import LinearRegression

# additive Holt Winters
def addSeasonal(x, m, fc, alpha = None, beta = None,
gamma = None, \
                l0 = None, b0 = None, s = None):
    # initial some temp variables
     Y = x[:]
     l = []
     b = []
     s = []
     # if hyper paras have not defined
     if (alpha == None or beta == None or gamma == None):
          alpha, beta, gamma = 0.1, 0.1, 0.1
     # if l0 b0 and s have not initialized
     if (l0 == None or b0 == None or s == None):
          l0,b0,s = linearOptimization(Y,m)
          l.append(l0)
          b.append(b0)

     else:
          l = l0
          b = b0
          s = s

     forecasted = []
     rmse = 0

     # forecast from T = 1:t+fc
     for i in range(len(x) + fc):
          # when we reach the end of the original data
          # then we need to forecast the t = T:T+12
          if i == len(Y) :
               Y.append(l[-1] + b[-1] + s[-m])
          # update the l,b,s and y
          # lt = alpha * (yt - St-m) + (1-alpha) * (lt-1
+ bt-1)
          l.append(alpha * (Y[i] - s[i-m]) + (1 - alpha)
* (l[i] + b[i]))
          # bt = beta * (lt - lt-1) + (1-beta) * bt-1
          b.append(beta * (l[i + 1] - l[i]) + (1 - beta)
* b[i])
          # St = gamma * (yt - lt-1 - bt-1) + (1-gamma) *
St-m
          s.append(gamma * (Y[i] - l[i] - b[i]) + (1 -
gamma) * s[i-m])
          # for forecasting, Yt+1 = lt +bt + st+1-m
          forecasted.append(l[i] + b[i] + s[i-m])

     # calculate the rmse
     # The zip() function take iterables (can be zero or
```

```
more),
     # makes iterator that aggregates elements based on
the iterables passed,
     # and returns an iterator of tuples.
     rmse = sqrt(sum([(m - n + 0.) ** 2 for m, n in
zip(Y[:-fc], y[:-fc - 1])]) / len(Y[:-fc]))
     return forecasted, Y[-fc:], l, b, s, rmse
```

You may notice that we haven't define the `linearOptimization()` yet. Don't worry, we will come to this in the next step.

Firstly, let's understand the logic in this function. Below are some tips:

1. You may notice some input are set to `None`, i.e. `alpha = None`. By default, the function requires `x, m fc, alpha, beta, gamma, l0, b0,` and `s`, which is 9 variables as input. If you do not define/input the value of `alpha, beta, gamma, l0, b0,` and `s,` this `addSeasonal()` function will set them as `None`.
2. In the beginning, we create some empty list variable, such as `l, s,` and `b`. These variables are used as containers for saving the calculation results.
3. If we don't define the value of `alpha, beta, gamma,` we will set them equal to 0.1. **However, it is still suggested to build a regression model to find the best fitting values based on the observations.**
4. If we don't define the `l0, b0,` and `s,` we will call the `linearOptimization()` to find the best fitting settings. Otherwise, we directly assign the predefined value to `l, b,` and `s.`
5. For each time index t = 1:T+fc, we calculate the `l, b,` and `s.` When t = T, this means we reach the end of the last observations. Therefore, we need to forecast $\widehat{y_T}$:$\widehat{y_{T+12}}$. You can find the corresponding condition and actions in the if condition: "`if i == len(Y):`"

Step 3: Fit the l_0, b_0 and $s_0, \dots, s_{1-M}$ by training a linear regression:

Continue with the previous step, we need to define a `linearOptimization()` function to find the best fitting l_0, b_0 **and** $s_0, \dots, s_{1-M}$. Here, we build a regression model to find the best fitting $l_0 = coef[0]$ and $b_0 = intercept$.

Read the following code carefully and pay attention to the comments:

```
def linearOptimization(X, m):
     # define X: 1~t
     x = np.linspace(1, len(X), len(X))
     x = np.reshape(x, (len(x), 1))
     y = np.reshape(X, (len(X), 1))
     # train a linear regression to get l0 (intercept_)
and b0 (coef_[0])
     lm = LinearRegression().fit(x, y)
```

```
    l0 = lm.intercept_
    b0 = lm.coef_[0]

    # use trained linear regression model to get y^t
    # then s^ = y-y^
    # finally average s^ to get s
    # The following statement is equal to:
    #    res = y - lm.predict(x)+0.
    #    res = np.reshape(res,(m,int(len(X)/m)))
    #    s = np.mean(res,axis=0)
    s = np.mean(np.reshape(y + 0. -
lm.predict(x),(int(len(X)/m),m)), axis=0)
    # in the above statement, 0. is used to transfer an
int to float

    # check the datatype of l0, b, and s
    return l0[0],b0, s.tolist()
```

What is the data type of `l0, b0,` and `s`?

Step 4: Plot your smoothing results:

Finally, we call `matplotlib` library to plot the smoothing results and the corresponding level, slope/trend and seasonal.

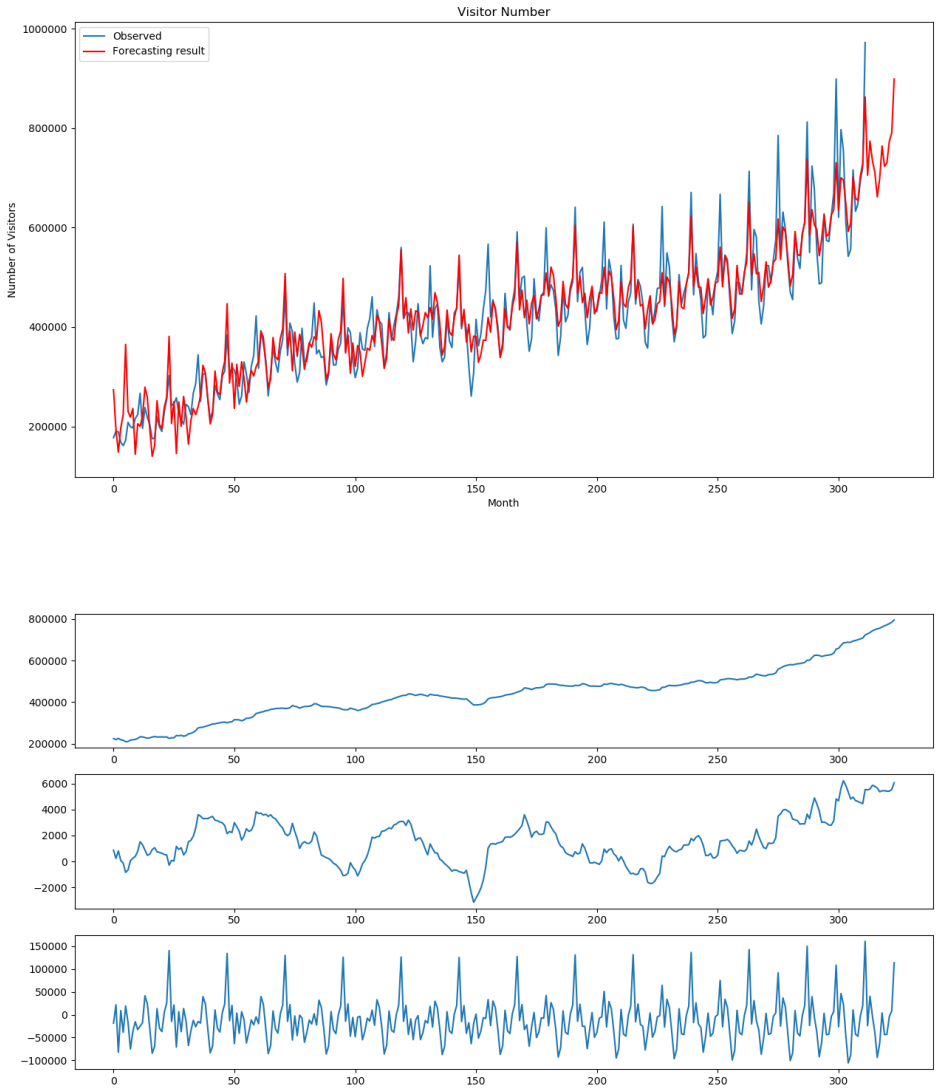
Remember to call the `addSeasonal()` function to get the forecasting results.

```
#%%
fc = 12   # One year forecasting
M = 12    # Seasonal number

# check out the y datatype here
s_smoothed, Ys, l, b, s, rmse = addSeasonal(x =
y.tolist(), m = M, fc = fc)

#%%
plt.figure(figsize=(15,8))
plt.plot(y, label='Observed')
plt.plot(s_smoothed[:], '-r', label='Smoothed')
plt.title('Visitor Number')
plt.xlabel('Month')
plt.ylabel('Number of Visitors')
plt.legend(loc=2)

#%%
# remove l0, b0,s-12
states = pd.DataFrame(np.c_[l[1:], b[1:], s[12:]], \

columns=['level','slope','seasonal'])
fig, [ax1,ax2,ax3] = plt.subplots(3, 1, figsize=(15,8))
```

```
states['level'].plot(ax=ax1)
states['slope'].plot(ax=ax2)
states['seasonal'].plot(ax=ax3)
plt.show()
```

What is the size of l, b and s in the above code?

Check the outcome of the above program and compare with this:



From the first figure, we can observe that the smoothed curve does not fit the observation very closely. Can you suggest some ways in order to obtain a more accurate result?

Hint: think about the hyper parameters `alpha, beta, gamma`.

2. Holt-Winters smoothing in statsmodels

In this task we will use the build-in functions in `statsmodels` to smooth the data with Holt-Winters's method. This implementation is much better than our manual implementation in task 1 where we did not look for the best hyperparameters `alpha, beta, gamma`.

Step 1: Importing the libraries and dataset:

From the `statsmodels.tsa.holtwinters` library import the `ExponentialSmoothing` function:

```
from statsmodels.tsa.holtwinters import
ExponentialSmoothing
```

Step 2: Call the `ExponentialSmoothing` function to smooth the data:

Holt-Winter smoothing method has two smoothing methods for seasonal variations: additive and multiplicative. For some simple cases, you can choose the suitable method by eye-looking. However, it is still suggested to do both of the decomposition and select the one with smaller residual for the forecasting.

Therefore, in this step we call the `ExponentialSmoothing()` function twice to smooth the time-series data with defining the additive and multiplicative seasonal variations decomposition respectively.

```
fit1 = ExponentialSmoothing(y, seasonal_periods=12,
trend='add', seasonal='add').fit()
fit2 = ExponentialSmoothing(y, seasonal_periods=12,
trend='add', seasonal='mul').fit()
```

Note that the `fit1` and `fit2` are object type variables. You may not be able to see the values of these variables in the Variable Explorer. If you want to know the fitting/smoothing results, you can print `.params[]` attributes, i.e. call `fit1.params['smoothing_level']` to get the best fitting α value. Similarly, you can get the besting fitting β, γ, ρ, l_0, and b_0 by calling different parameter settings. Below is an example:

```
# symbol r $ and \ in the results variable are the latex
symbols for visualization in notebook
results = pd.DataFrame(index=[r"$\alpha$",\
                              r"$\beta$",\
                              r"$\phi$",\
                              r"$\gamma$",\
                              r"$l_0$",\
                              "$b_0$",\
                              "SSE"])
# ExponentialSmoothing() object has following attributes
params = ['smoothing_level', \
```

```
          'smoothing_slope', \
          'damping_slope', \
          'smoothing_seasonal', \
          'initial_level', \
          'initial_slope']

# check out the performance of additive and
multiplicative
results["Additive"]       = [fit1.params[p] for p in
params] + [fit1.sse]
results["Multiplicative"] = [fit2.params[p] for p in
params] + [fit2.sse]
```

Execute the above code and check the outcome carefully:

1. What is the size of results variable?
2. What is the `sse` outcome for both decomposition? Based on this, which decomposition we should choose?

For more information of `holtwinter` function in `statsmodels` library, you can refer to the following two links:
https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.HoltWintersResults.html
https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html

Step 3: Plot the smoothing results:

After finding the best fitting parameters, we then need to plot the smoothing results. Here, you can call `.fitvalues` statement to check the smoothing results.

```
plt.figure()
ax = y.plot(figsize=(15,6), color='black',
title="Forecasts from Holt-Winters' multiplicative
method" )
ax.set_ylabel("Number of Australia Vistors")
ax.set_xlabel("Year")

# transfer the datatype to values
smooth_add = fit1.fittedvalues
smooth_mul = fit2.fittedvalues
smooth_add.plot(ax=ax, style='-', color='red')
smooth_mul.plot(ax=ax, style='--', color='green')
```

Can you also plot the level, slope, and seasonal?
Please check the `HoltWintersResults` class attributes by clicking the hyperlink in step 2!

Step 4: Forecast the future data:

Finally, based on the existing observations and best fitting parameters, let's forecast one more year. Here, you can call `.forecast()` function to forecast the T = t+1:t+N ,where N=12 in this case.

```
# forecast 12 more data and plot
forecast1 = fit1.forecast(12).rename('Holt-Winters (add-
add-seasonal)')
forecast1.plot(ax=ax, style='-', color='red',
legend=True)
fit2.forecast(12).rename('Holt-Winters (add-mul-
seasonal)').plot(ax=ax, style='--', marker='o',
color='green', legend=True)

# sometimes you may need to call .show() function to
display the plots
plt.show()
```

What is the data type and size of variable `forecast1`?

Step 5: Further comparison of additive vs multiplicative:

In many situations, we can use SSE or MSE for the quantitative performance analysis. However, sometimes we still need to plot the further information such as level, slope/trend and seasonal component for the qualitative analysis. In this step, we aim to extract the smoothing results and plot them in one window for comparison.

Extract the smoothing information and use DataFrame variable to save these them.

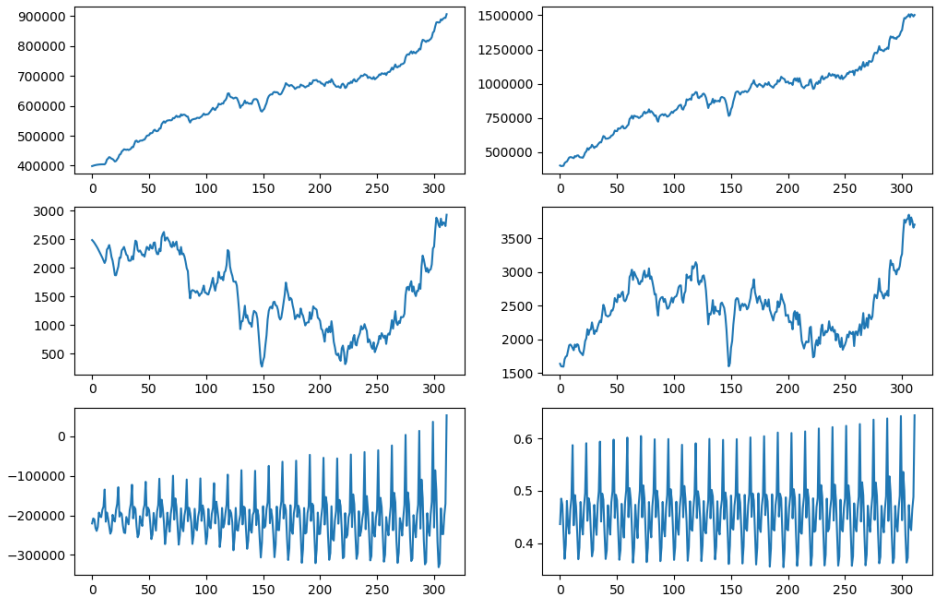
```
# compare the forecasting outcomes of additive and
multiplicative
# np.c_: Translates slice objects to concatenation along
the second axis.
df1 = pd.DataFrame(np.c_[y, fit1.level, fit1.slope,
fit1.season, fit1.fittedvalues],

columns=[r'$y_t$',r'$l_t$',r'$b_t$',r'$s_t$',r'$\hat{y}_t
$'],index=y.index)
df1.append(fit1.forecast(24).rename(r'$\hat{y}_t$').to_fr
ame())
#%%
df2 = pd.DataFrame(np.c_[y, fit2.level, fit2.slope,
fit2.season, fit2.fittedvalues],

columns=[r'$y_t$',r'$l_t$',r'$b_t$',r'$s_t$',r'$\hat{y}_t
$'],index=y.index)
df2.append(fit2.forecast(24).rename(r'$\hat{y}_t$').to_fr
```

```
ame())
```

Create a new fig window for plotting the level, slope and seasonal component results of additive and multiplicative decomposition. In this window, we have 3 rows and 2 columns, and the corresponding position is defined as `ax1, ax2, …, ax6`.

```
# Ploting the level, trend and season for fit1 and fit2
# define 2 states variable for conveinence
states_add = pd.DataFrame(np.c_[fit1.level, fit1.slope,
fit1.season], \

columns=['level','slope','seasonal'], \
                        index=y.index)
states_mul = pd.DataFrame(np.c_[fit2.level, fit2.slope,
fit2.season], \

columns=['level','slope','seasonal'], \
                        index=y.index)

# define subplots windows
fig, [[ax1, ax4],[ax2, ax5], [ax3, ax6]] =
plt.subplots(3, 2, figsize=(12,8))
states_add['level'].plot(ax=ax1)
states_add['slope'].plot(ax=ax2)
states_add['seasonal'].plot(ax=ax3)
states_mul['level'].plot(ax=ax4)
states_mul['slope'].plot(ax=ax5)
states_mul['seasonal'].plot(ax=ax6)
plt.show()
```

Run the above code and compare your outcome with the figure below:



Then carefully exam the outcome by answering the following questions:
1. What is the level range for both methods?
2. What is the slope range for both methods?
3. What is the seasonal range for both methods?
4. Let's put the residual aside and only consider the level, slope and seasonal. Based on these information, which decomposition method should you choose? Why?

Updates for Tutorial_06.pdf:

We notice there are some indexing error In the Tutorial_06.pdf:

1. Distinguish the difference of forecasting and smoothing for Holt's linear trend method.

   For forecasting, we will combine the level and slope at time t as the forecasting results:

   $$\widehat{Y_{T+1}} = l_t + hb_t$$

   Whereas for smoothing, we can directly use the level at time t:

   $$\widehat{Y_T} = l_t$$

   Therefore the code for forecasting should be:

```
#%%
# forecast the data

holtsforecast_manual = []
Y = y.tolist()
```

```
# Here, we are going to forecast 12 more months after
the last observation (t = 312).
for i in range(len(y)+12):
    # when we reach the end of the original data
    # then we need to forecast the t = T:T+12
    if i == len(Y):
        Y.append(l[-1] + b[-1])
    print(i)
    l.append(alpha * Y[i] + (1 - alpha) * (l[i] +
b[i]))
    b.append(beta * (l[i+1] - l[i]) + (1 - beta) *
b[i])
    # for forecasting,  Y^2 = l1 + b1
    #                   Y^3 = l2 + b2
    #                   Y^4 = l3 + b3
    #                       ...
    #                   Y^t+1 = lt+bt
    holtsforecast_manual.append(l[i] + b[i])
```

The code for smoothing should be:

```
#%% if you are focusing on smoothing, then use the
below code:
# Updated 12/April
holtsmoothed_manual = []
Y = y.tolist()

# Updated 12/April
# forecast the data
for i in range(len(y)):
  l.append(alpha * Y[i] + (1 - alpha) * (l[i] + b[i]))
# Calculating l[1], l[2], etc.
  b.append(beta * (l[i+1] - l[i]) + (1 - beta) * b[i])
# Calculating b[1], b[2], etc.
    # for smoothing,    Y^1 = l1
    #                   Y^2 = l2
    #                       ...
    #                   Y^t = lt
  holtsmoothed_manual.append(l[i+1])
```

We also update the tutorial document and sample code for week06, please get a latest copy on Canvas.