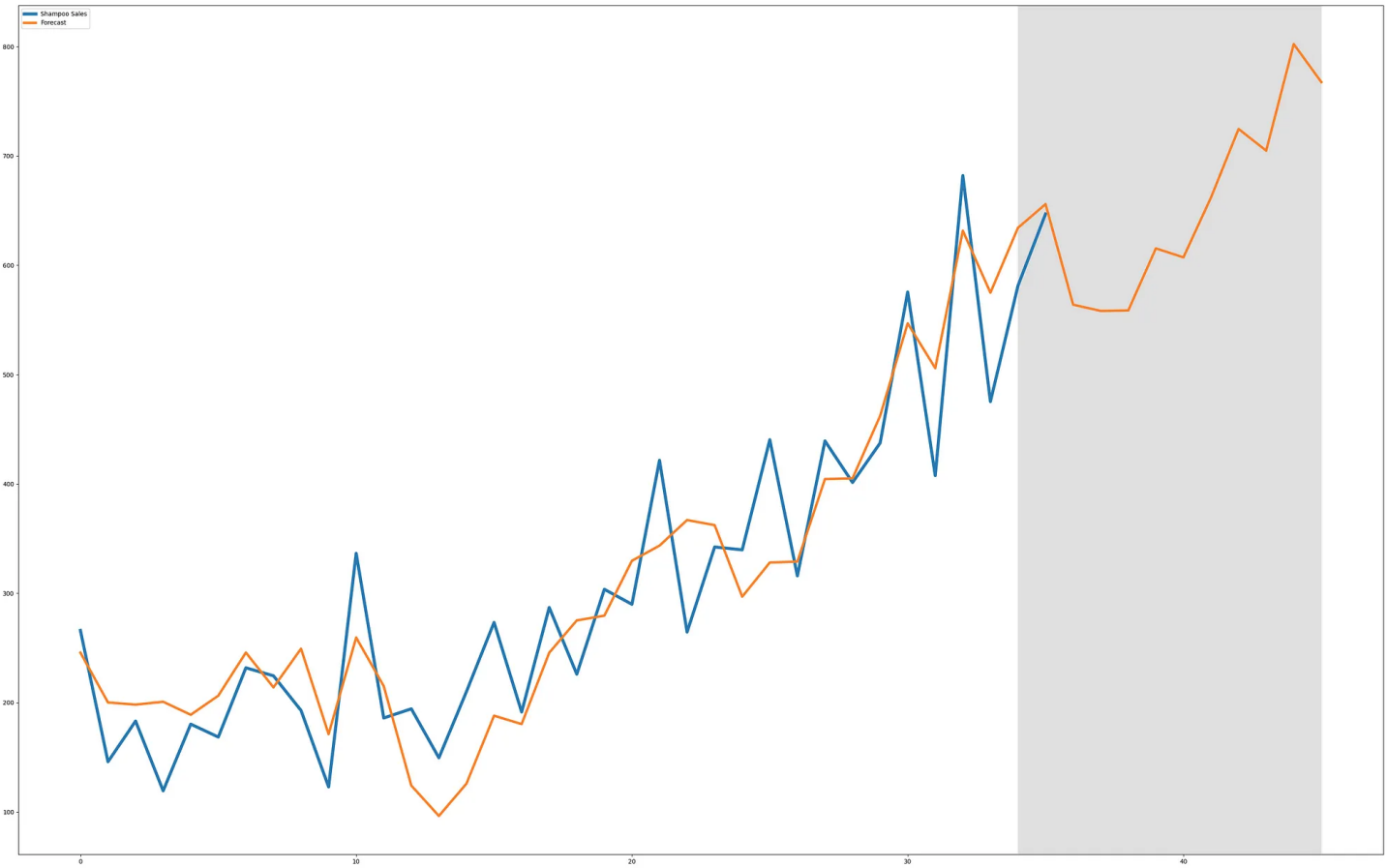
A Thorough Introduction to Holt-Winters Forecasting

Lleyton Ariton · Follow
Published in Analytics Vidhya
14 min read · Feb 22, 2021



The Holt-Winters method — also known as triple exponential smoothing — is an incredibly popular and relatively simple method for time series forecasting. This

article will be a somewhat thorough introduction into the math and theory of the Holt-Winters method, complete with a Python implementation from scratch.

HOLT-WINTERS METHOD

The Holt-Winters method is a very common time series forecasting procedure capable of including both trend and seasonality. The Holt-Winters method itself is a combination of 3 other much simpler components, all of which are smoothing methods:

- **Simple Exponential Smoothing (SES):** Simple exponential smoothing assumes that the time series has no change in level. Thus, it can not be used with series that contain trend, seasonality, or both.
- **Holt's Exponential Smoothing (HES):** Holt's exponential smoothing is one step above simple exponential smoothing, as it allows the time series data to have a trend component. Holt's exponential smoothing is still incapable of cope with seasonal data.
- **Winter's Exponential Smoothing (WES):** Winter's exponential smoothing is an extension to Holt's exponential smoothing that finally allows for the inclusion of seasonality. Winter's exponential smoothing is what is referred to as the Holt-Winters method.

The Holt-Winters method therefore is often referred to as triple exponential smoothing, as it is literally the combination of 3 smoothing methods built on top of each-other.

SIMPLE EXPONENTIAL SMOOTHING

The simple exponential smoothing method does not take into account any trend or seasonality. Rather, it assumes that the time series data only has a level, *L*.

UPDATE EQUATION

We can define the simple exponential smoothing method as:

$$L_t = \alpha y_t + (1 - \alpha) L_{t-1}$$

Where:

y_t is the value at current time step t, L_t is the level estimate for t, L_{t-1} is the previous level estimate, and α is a smoothing constant.

This equation is known as the *level update equation*, as it *updates* the level of the current time step based on the previous level estimate. The equation is therefore recursive, since every level estimate must be computed using every estimate before it.

The alpha, α, is known as the smoothing constant. The smoothing constant has domain $0 \leq \alpha \leq 1$, and it dictates how much weight is given to past values when being included in the current level estimate. This is due to the recursive nature of the level update equation, as if we were to begin to unroll the equation, we would see that the weights on each previous value exponentially decreases:

$$L_t = \alpha y_t + (1-\alpha)[\alpha y_{t-1} + (1-\alpha)L_{t-2}]$$

$$L_t = \alpha y_t + (1-\alpha)[\alpha y_{t-1} + (1-\alpha)L_{t-2}]$$

$$L_t = \alpha y_t + \alpha(1-\alpha)y_{t-1} + (1-\alpha)^2 L_{t-2}$$

...

...

...

The weights are exponentially decreasing with every previous unrolled estimate

The unrolling will continue all the way until the very first time step. Since there is obviously no way we can calculate the level for the very first time step, (since we have no previous level to look back on) we will begin the entire sequence by just setting the very first level as the value of the very first time step.

The *SES* method is essentially just a weighted average across all time steps, with the weights *exponentially* decaying. Hence the name: simple *exponential* smoothing.

IMPLEMENTATION

We will implement simple exponential smoothing in Python, and see some examples.

Recalling the equation, simple exponential smoothing can be implemented as:

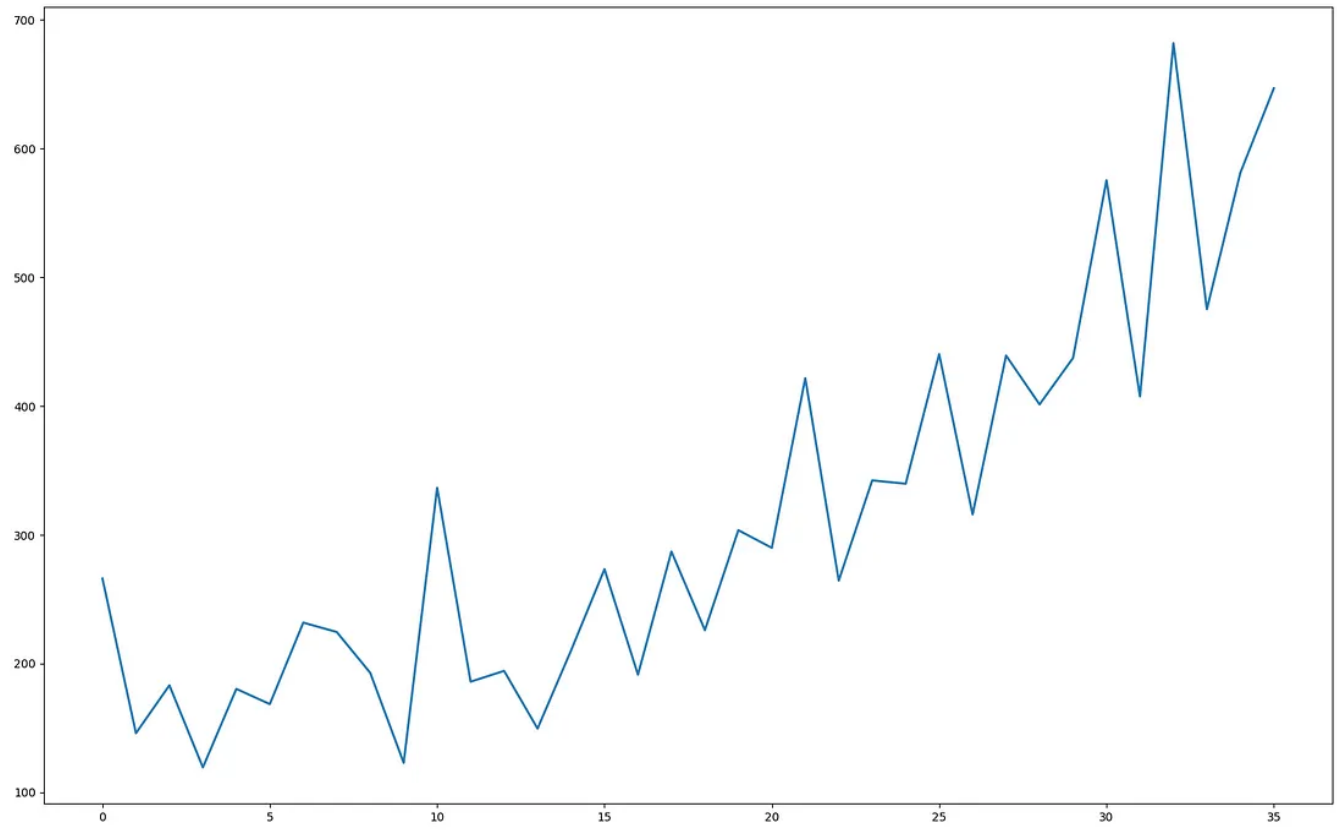
```
from typing import List

def simple_es(series: List, alpha: float) -> float:
    if len(series) < 2:
        return series[0]

    return (alpha * series[-1]) + ((1 - alpha) * ses(series[:-1],
alpha))
```

Like the level update equation, we can see that the Python implementation is subsequently recursive as well. The very first level estimate is set to the value of the very first data point — this is our base case.

FORECASTING

When forecasting with the *SES* method, the future time step is just the level of the current time step. Therefore, the forecast *is* the level *L* at time step *t*.

As an example, we will apply our Python *SES* implementation to some real world data:



Above is a set of shampoo sales over a 3 year period, with every time step being the sales for one month. The data can be downloaded here.

Let's keep the final month hidden when using our *SES*, since we want to be able to contrast our forecast with the ground truth.

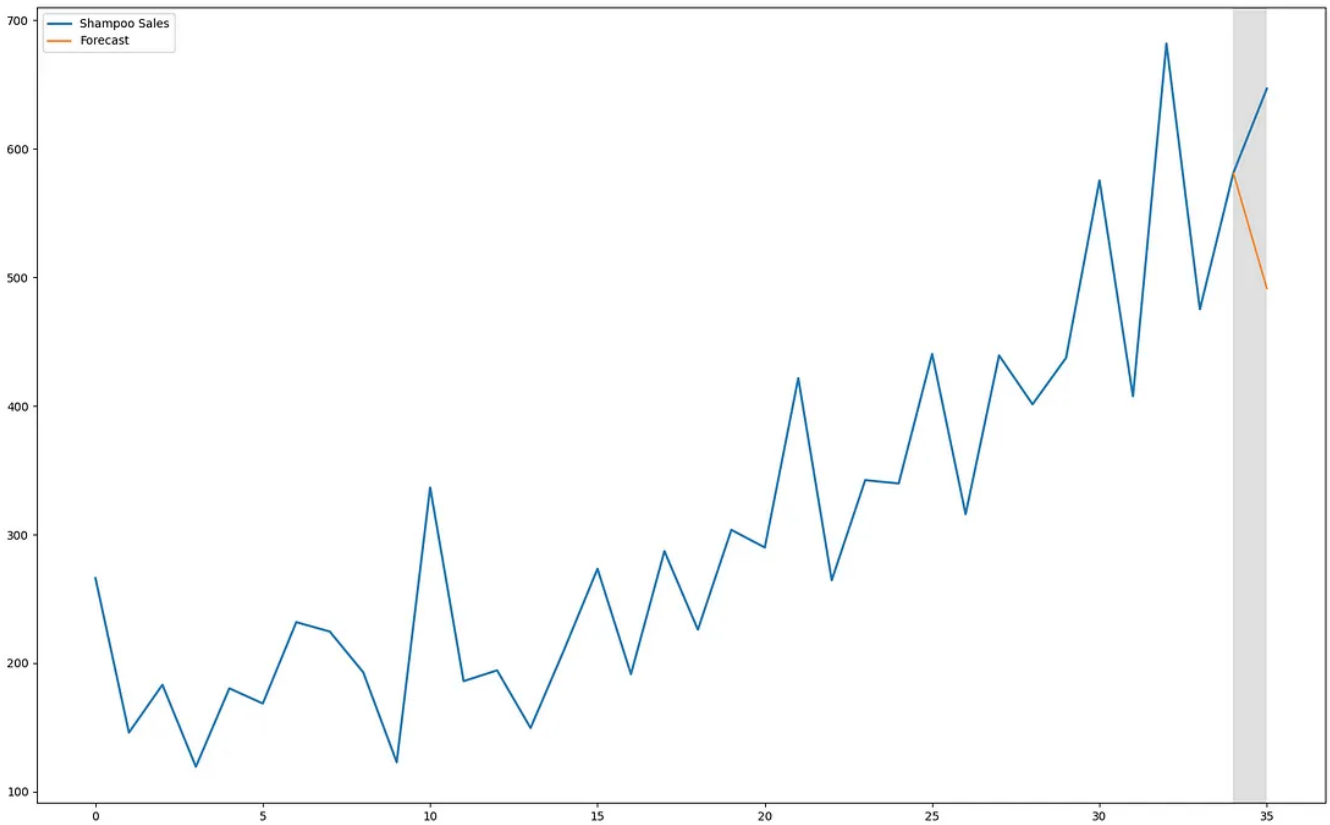
Since a *SES* forecast is just simply the estimated level at time step t, forecasting with our *SES* implementation is as simple as:

```
forecast = simple_es(data[:-1], alpha=0.2)
```

In this particular case the alpha parameter was not optimized, rather it was quickly chosen manually as an alpha around 0.1–0.4 is generally a good rule of thumb.

We will now plot the forecast against the ground truth to get a better visualization:

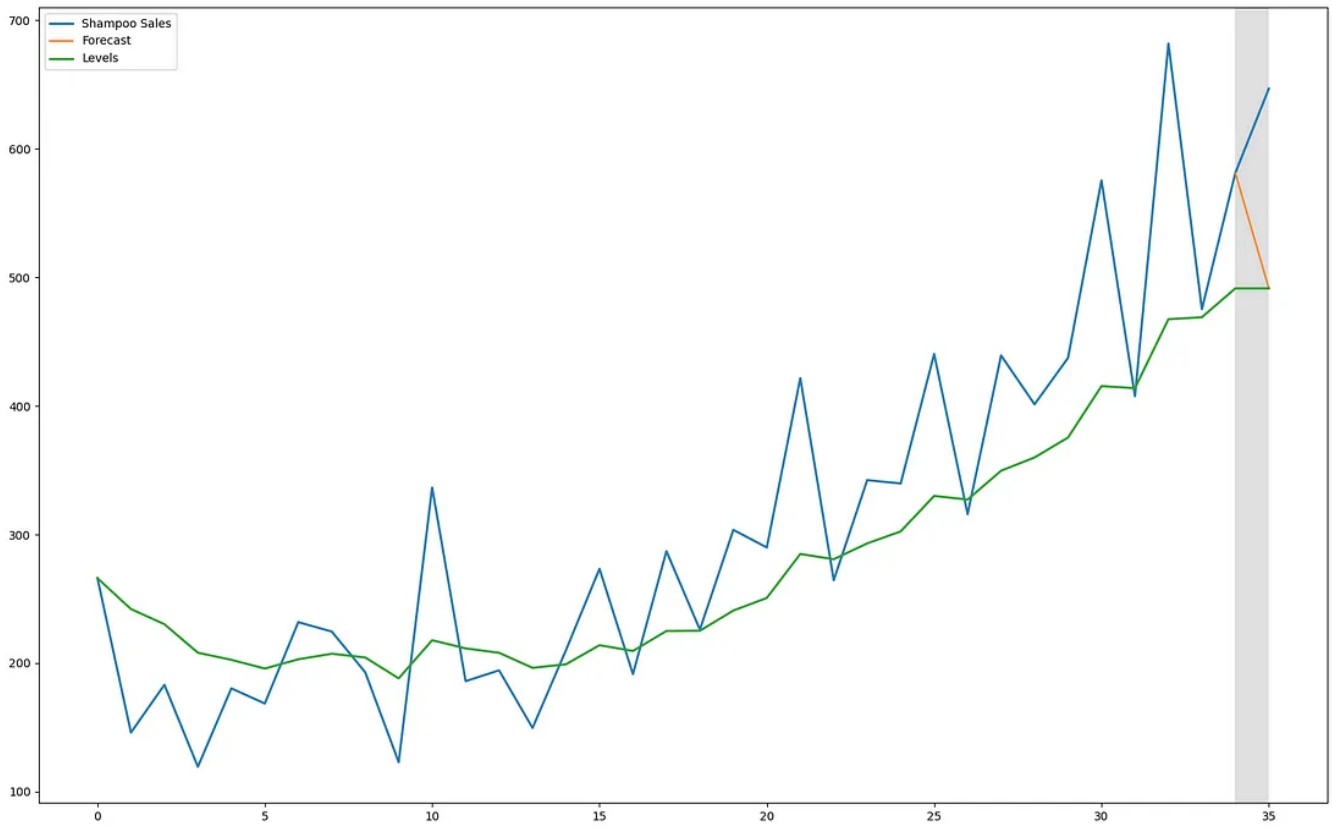
```
plt.figure(
    figsize=(16, 10)
)

data = pd.read_csv('../data/shampoo.csv')["Sales"]
data = data.values.tolist()

final_months = (34, 35)
forecast = ses(data[:-1], alpha=0.2)

plt.plot(data, linewidth=2, label='Shampoo Sales')
plt.plot(final_months,
         [data[-2], forecast],
         label='Forecast')

plt.axvspan(*final_months, facecolor='grey', alpha=0.25)

plt.legend()
plt.show()
```



The grey bar represents the forecasted month. No surprise, the forecast is terrible — quite literally in the opposite direction. This is to be expected as simple exponential

smoothing is just that: simple. It does not consider any kind of trend or seasonality and was therefore doomed from the start with this particular data set.

SES is still a smoothing method after all, and to get a better visual of how it operates we can plot every single estimated level:



The smoothing action can clearly be seen.

In order to improve the forecast, a more sophisticated method that is capable of incorporating trend must be used.

HOLT'S EXPONENTIAL SMOOTHING

A step up from simple exponential smoothing, Holt's exponential smoothing method is capable of taking into account a trend component. Holt's method is often referred to as double exponential smoothing.

Holt's method extends simple exponential smoothing by assuming that the time series has both a level and a trend. A forecast with Holt's method can therefore be defined as:

$$F_{t+k} = L_t + kT_t$$

Where:

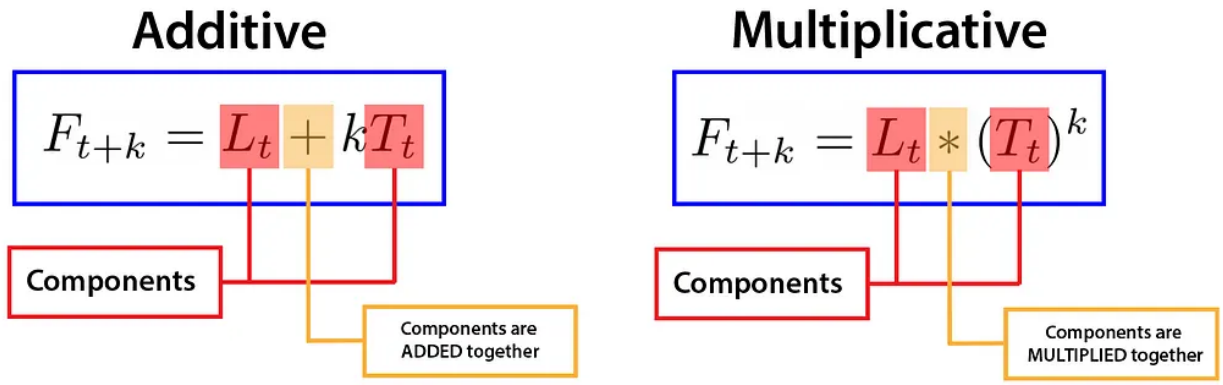
L_t is the level estimate for time t, k is the number of forecasts into the future, and T_t is the trend at time t.

As we can see, it is literally just a simple extenuation of original *SES* method, just with the inclusion of the trend, *T*, component.

However, it is important to note that there are two types of time series, each with their own slightly different forecasting equation:

- **Additive:** In an additive time series, the time series is the sum of its components.
- **Multiplicative:** In a multiplicative time series, the time series is the product of its components.

The forecasting equation must correspond to the particular type of time series:



We must therefore firstly identify the type of time series, and then forecast accordingly.

UPDATE EQUATION

The trend estimate for a given time *t* can be computed as*:*

$$T_t = \beta(L_t - L_{t-1}) + (1 - \beta)T_{t-1}$$

This equation is known as the *trend update equation*, as it *updates* the trend estimate of the current time step based on the difference between previous level estimates. One may notice it is very akin to the original *level update equation* in simple exponential smoothing.

Similarly to the *level update equation*, the *trend update equation* also takes its own parameter, β. The beta is the trend equation's own smoothing constant, and is therefore more or less analogous to the alpha smoothing constant seen earlier in *SES*.

Beta also has the domain $0 \leq \beta \leq 1$, and it dictates how how fast a given trend estimate should be adjusted.

IMPLEMENTATION

As was earlier done with simple exponential smoothing, we will implement Holt's exponential smoothing in Python, and see some examples.

Recalling the *trend update equation,* Holt's exponential smoothing can be implemented as:

```
def holt_es(series: List, alpha: float,
            beta: float, initial_trend: float) -> float:

    if len(series) < 2:
        return initial_trend

    l_t = ses(series, alpha)
```

```
    l_t_prev = ses(series[:-1], alpha)

    trend_t_prev = hes(series[:-1], alpha, beta, initial_trend)

    return (beta * (l_t - l_t_prev)) + ((1 - beta) * trend_t_prev)
```

The Python implementation for Holt's exponential smoothing builds on top of our earlier *SES* function, as that gives us the level estimate. The *HES* function is recursive just like the *SES* function, however note the difference in the base case and the addition of an *"initial trend"* parameter. The initial trend is required because unlike simple exponential smoothing, the start of the trend sequence is not just simply the value of the oldest time step. Rather, the beginning is a trend estimate itself, calculated as the difference between the first two time steps. This implementation assumes an *additive* time series with no seasonal component.

FORECASTING

When forecasting with the *HES* method, we must remember the different types of forecasts — **Additive** and **Multiplicative** — and choose the forecasting equation accordingly. In the case of our Shampoo data, we will use the *additive* forecasting equation. In code, this translates to:

```
initial_trend = data[1] - data[0]

level_estimate = simple_es(data[:-1], 0.2)

forecast = level_estimate + holt_es(data[:-1],
                                    0.2, 0.3, initial_trend)
```

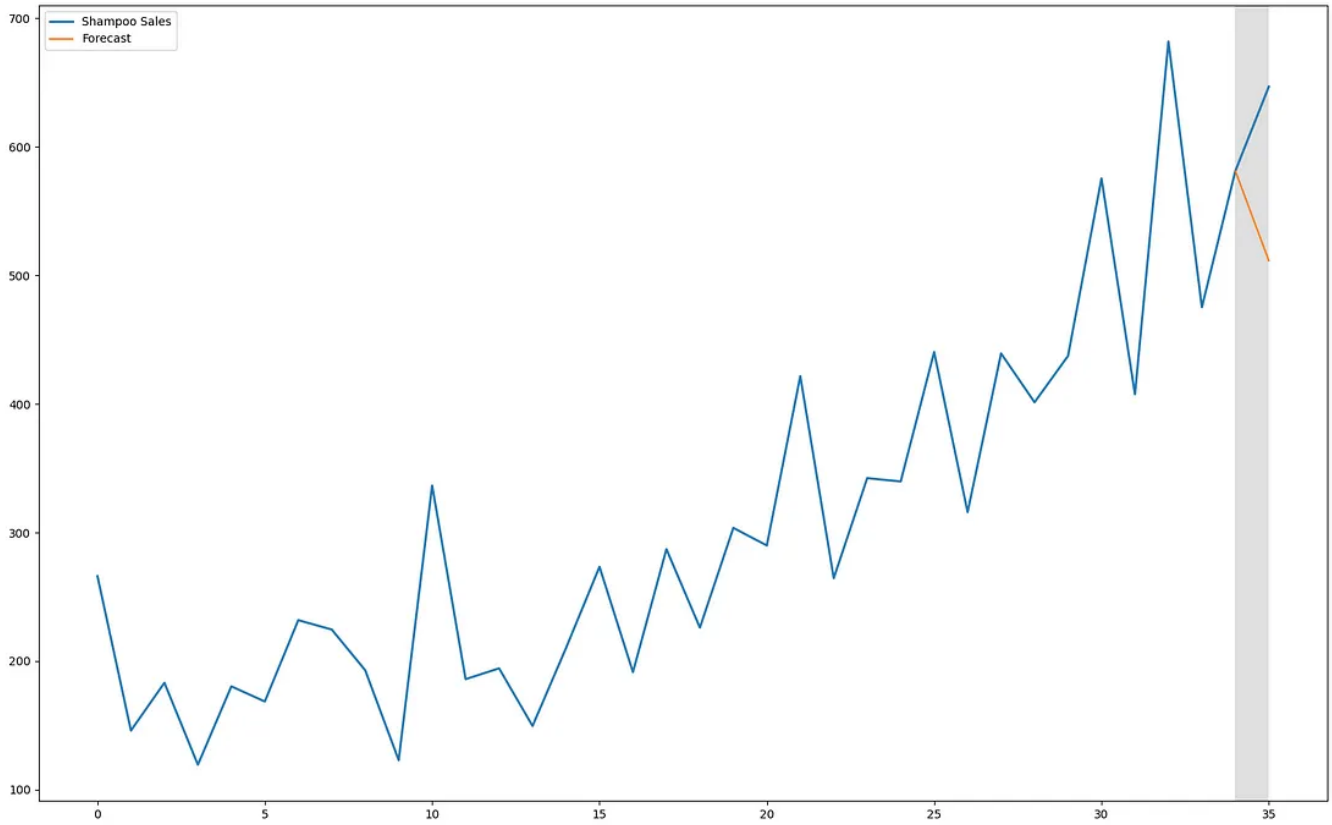
Just like the example with *SES*, we are intentionally forecasting the final already known value so that we can contrast the forecast with the ground truth.

The alpha and beta parameters for this example were not optimized, rather they were picked manually.

It is important to note that this specific forecast example is only for a single step in the future. Since *k=1* we basically take the *HES* result as is. If we were to predict multiple time steps into the future, we would need to multiply the trend component

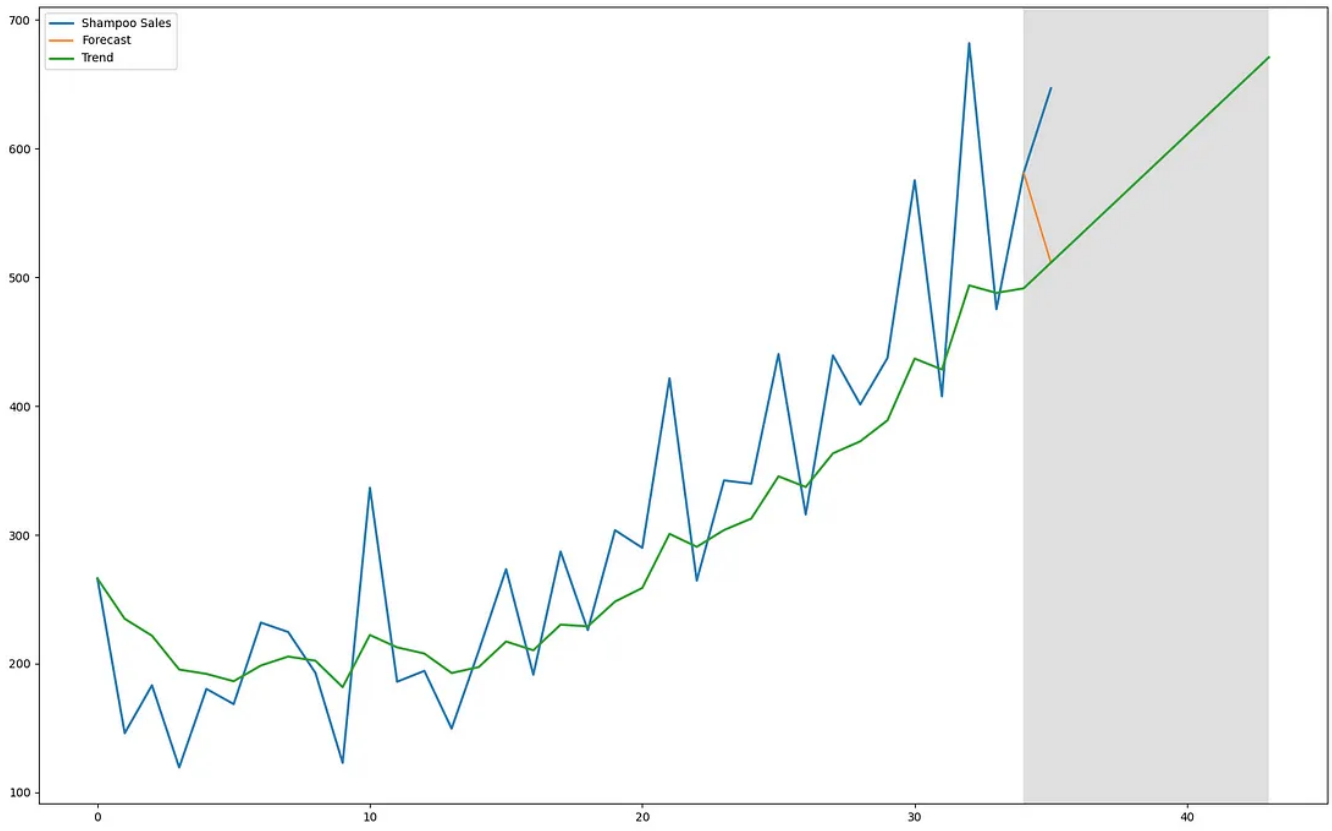
with the desired *k* number of steps.

We will now plot the forecast against the ground truth to get a better visualization:



Once again, the grey bar is the forecasted window, the orange is the forecast, and the blue is the actual series.

Unfortunately, Holt's forecast is still not very different from the *SES* forecast. However, by plotting every forecast plus a few steps ahead, we can see that Holt's method does indeed clearly display the capturing of a trend component:



The final improvement that must be made is the capturing of a seasonal component.

WINTERS' EXPONENTIAL SMOOTHING

Winters' exponential smoothing method is an extension to Holt's method that finally allows for the capturing of a seasonal component. Since Winter's exponential smoothing is built on top of both single and double exponential smoothing, Winter's method is thus also known as triple exponential smoothing.

Winter's method assumes that the time series has a level, trend and seasonal component. A forecast with Winter's exponential smoothing can be expressed as:

$$F_{t+k} = L_t + kT_t + S_{t+k-M}$$

Where:

L_t is the level estimate for time t, k is the number of forecasts into the future, T_t is the trend estimate at time t, S_t is the seasonal estimate at time t, and M is the number of seasons.

The forecast equation is the extenuation of both the *SES* and *HES* methods, finally augmented with the inclusion of the Seasonal, *S*, component.

Just like with Holt's method, the forecasting equation has multiple variations for each of the types of time series:

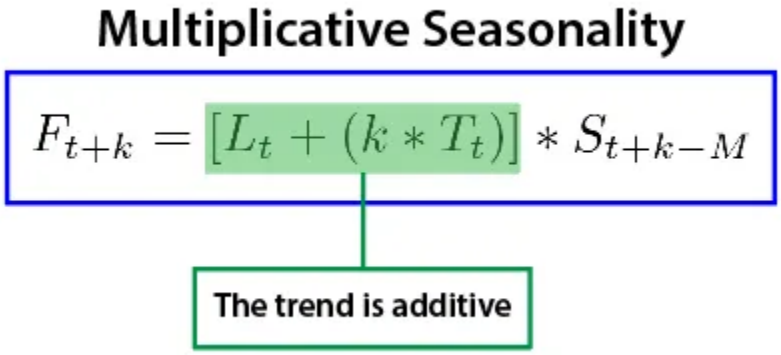
Additive Seasonality

$$F_{t+k} = L_t + (k * T_t) + S_{t+k-M}$$

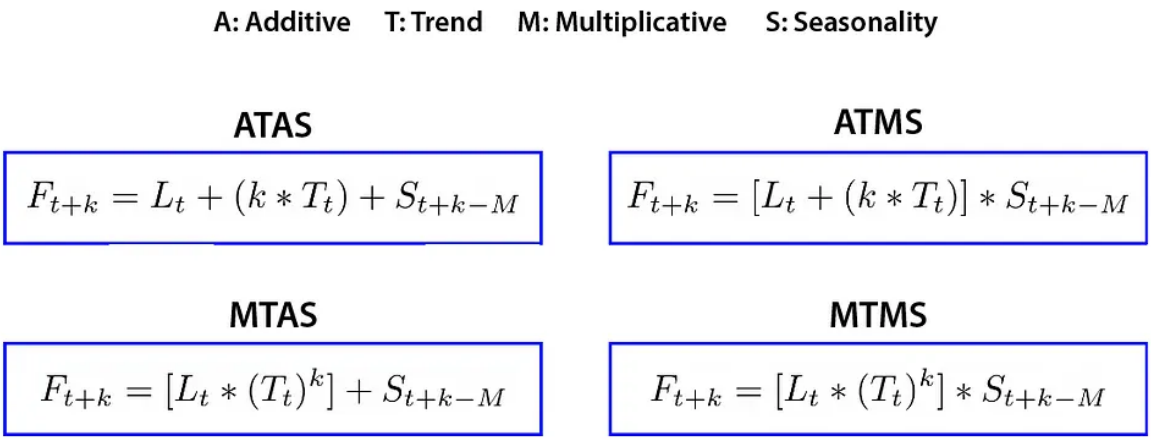
Multiplicative Seasonality

$$F_{t+k} = [L_t + (k * T_t)] * S_{t+k-M}$$

It is important to notice that each of the time series components do not need to exhibit the same behaviour. For instance, the multiplicative seasonality formula does — obviously — have multiplicative seasonality, however the trend portion is additive:



This independence allows the mixing and matching of different behaviours to model many different types of time series:



UPDATE EQUATION

The inclusion of the seasonality component now adds the third update equation:

$$S_t = \gamma \frac{y_t}{L_t} + (1 - \gamma) S_{t-M}$$

Once again, there is the introduction of another smoothing constant: γ. The gamma smoothing constant also has the domain 0≤γ≤1. This specific update equation is for a time series with multiplicative level. For an additive level, use the seasonal update equation below:

$$S_t = \gamma(y_t - L_t) + (1 - \gamma) S_{t-M}$$

In both equations, we see that the seasonality estimate calculation requires us to

remove the level component from the time step, which is accomplished by doing the inverse operation for the behavioural type of the level. In other words, the level component must be subtracted if it is *additive*, and divided out if it is *multiplicative*.

Winter's method not only introduces the *seasonality update equation*, but it actually also introduces a modified version of the original *level update equation*.

The original *level update equation* does not work properly on seasonal data. Therefore, since the time series is assumed to now have a seasonal component, the level equation must firstly "deseasonalize" the data in order to achieve a proper level estimate.

The new *level update equation* can be expressed as:

$$L_t = \alpha(y_t - S_{t-M}) + (1-\alpha)(L_{t-1} + T_{t-1})$$

This specific update equation is for *additive* seasonality and *additive* trend. The update equations for each type of seasonality can be seen below:

Muliplicative

$$L_t = \alpha\frac{y_t}{S_{t-M}} + (1-\alpha)(L_{t-1} + T_{t-1})$$

Additive

$$L_t = \alpha(y_t - S_{t-M}) + (1-\alpha)(L_{t-1} + T_{t-1})$$

We can see the specific deseasonalization effect here:

Muliplicative

$$L_t = \alpha\frac{y_t}{S_{t-M}} + (1-\alpha)(L_{t-1} + T_{t-1})$$

Additive

$$L_t = \alpha(y_t - S_{t-M}) + (1-\alpha)(L_{t-1} + T_{t-1})$$

In order to remove the seasonal component, we must know what type of time series seasonality we are attempting to model, and do the inverse. This translates into either dividing or subtracting the seasonal component for multiplicative and additive time series seasonality, respectively.

IMPLEMENTATION

We can finally implement Winter's method in Python, and see some examples.

Recalling all of our update equations, Winter's exponential smoothing can be implemented as:

```
def winters_es(series: List,
               uppercase_m: int,
               alpha: float=0.2,
               beta: float=0.2,
               gamma: float=0.15,
               future_steps: int=1) -> List:

    i_l = [series[0]]
    i_t = [initial_trend(series, uppercase_m)]
    i_s = initial_seasonality(series, uppercase_m)

    forecasts = []
    for t in range(len(series) + future_steps):

        if t >= len(series):
            k = t - len(series) + 1
            forecasts.append(
                (i_l[-1] + k * i_t[-1]) + i_s[t % uppercase_m]
            )

        else:
            l_t = alpha * (series[t] - i_s[t % uppercase_m]) + (1 -
alpha) * (i_l[-1] + i_t[-1])

            i_t[-1] = beta * (l_t - i_l[-1]) + (1 - beta) * i_t[-1]
            i_l[-1] = l_t

            i_s[t % uppercase_m] = gamma * (series[t] - l_t) + (1 -
gamma) * i_s[t % uppercase_m]

            forecasts.append(
                (i_l[-1] + i_t[-1]) + i_s[t % uppercase_m]
            )
```

```
    return forecasts
```

As we can see, Winters' method is the conglomeration of every single smoothing method and equation thus far. This specific implementation is for all *additive* components, and was inspired by this. Due to the increased complexity of the implementation, it was simpler to make it non-recursive and to add the forecasting capability internally.

It is important to notice the addition of the *"initial_trend"* and *"initial_seasonality"* functions. This is due to the introduction of the seasonality component which now changes the initial conditions.

The initial trend for a time series with an *additive* seasonality can be computed as:

$$T_0 = \frac{1}{M}\left[\frac{y_{M+1} - y_1}{M} + \frac{y_{M+2} - y_2}{M} + \cdots + \frac{y_{M+M} - y_M}{M}\right]$$

Where:
T_0 is the initial trend, M is the seasonal length and y is the value of the time series at a given time step

In Python code, this translates as:

```
def initial_trend(series: List, uppercase_m: int) -> float:
    return sum([
        float(series[i+uppercase_m] - series[i]) / uppercase_m
        for i in range(uppercase_m)
    ]) / uppercase_m
```

Essentially, this is just the average of trend averages across seasons.

As for the initial seasonality, it can be computed as:

```
def initial_seasonality(series: List, uppercase_m: int) -> List:
    initial_season = []
    n_seasons = int(len(series)/uppercase_m)

    season_averages = [sum(
        series[uppercase_m * i:uppercase_m * i + uppercase_m]
    ) / uppercase_m for i in range(n_seasons)]

    initial_season.extend([
        sum([series[uppercase_m*j+i]-season_averages[j]
             for j in range(n_seasons)]) / n_seasons
        for i in range(uppercase_m)
    ])

    return initial_season
```

Notice that unlike the other initial states, the initial seasonality is a vector. This is because we need to start at minimum with at least one entire season.

We firstly calculate the average level for every season, then divide every single time step by the level average of the time step's corresponding season. Finally, the results are averaged across each season, and we are left with the initial seasonality vector. The full mathematical explanation gets rather complex, and will thus be omitted from this article.

The entire implementation for the Holt-Winters method:

```
from typing import *

def initial_trend(series: List, uppercase_m: int) -> float:
    return sum([
        float(series[i+uppercase_m] - series[i]) / uppercase_m
        for i in range(uppercase_m)
    ]) / uppercase_m


def initial_seasonality(series: List, uppercase_m: int) -> List:
    initial_season = []
    n_seasons = int(len(series)/uppercase_m)
```

```
    season_averages = [sum(
        series[uppercase_m * i:uppercase_m * i + uppercase_m]
    ) / uppercase_m for i in range(n_seasons)]

    initial_season.extend([
        sum([series[uppercase_m*j+i]-season_averages[j]
             for j in range(n_seasons)]) / n_seasons
        for i in range(uppercase_m)
    ])

    return initial_season
```

```
def winters_es(series: List,
               uppercase_m: int,
               alpha: float=0.2,
               beta: float=0.2,
               gamma: float=0.15,
               future_steps: int=1) -> List:

    i_l = [series[0]]
    i_t = [initial_trend(series, uppercase_m)]
    i_s = initial_seasonality(series, uppercase_m)

    forecasts = []
    for t in range(len(series) + future_steps):

        if t >= len(series):
            m = t - len(series) + 1
            forecasts.append(
                (i_l[-1] + m * i_t[-1]) + i_s[t % uppercase_m]
            )

        else:
            l_t = alpha * (series[t] - i_s[t % uppercase_m]) + (1 -
alpha) * (i_l[-1] + i_t[-1])

            i_t[-1] = beta * (l_t - i_l[-1]) + (1 - beta) * i_t[-1]
            i_l[-1] = l_t

            i_s[t % uppercase_m] = gamma * (series[t] - l_t) + (1 -
gamma) * i_s[t % uppercase_m]

            forecasts.append(
                (i_l[-1] + i_t[-1]) + i_s[t % uppercase_m]
            )

    return forecasts
```

FORECASTING

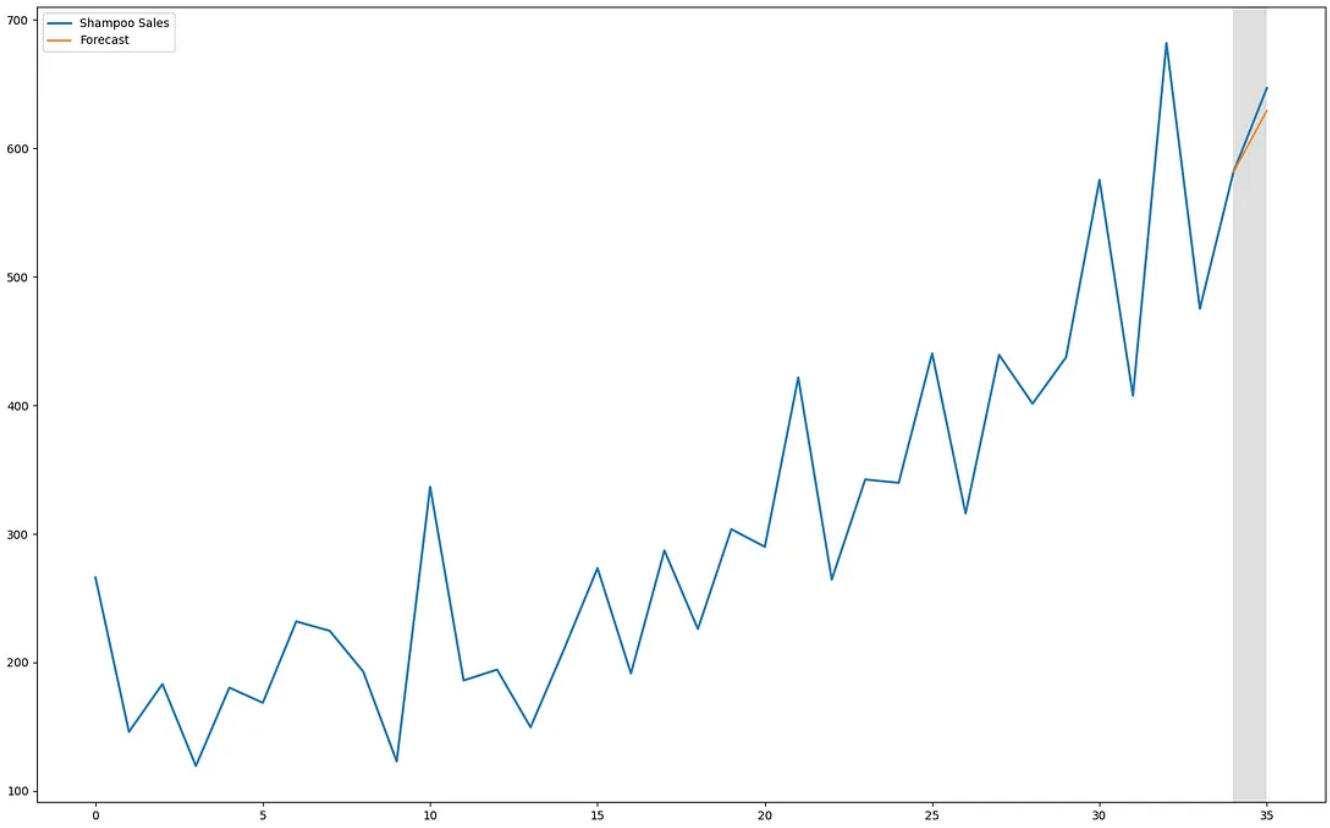
We can now use the completed Holt-Winters implementation to do some forecasting. As usual, we will use the *additive* Shampoo dataset and forecast the last known value so that we may contrast the estimation with the ground truth.

The alpha, beta and gamma parameters for this example were not optimized, rather they were picked manually.

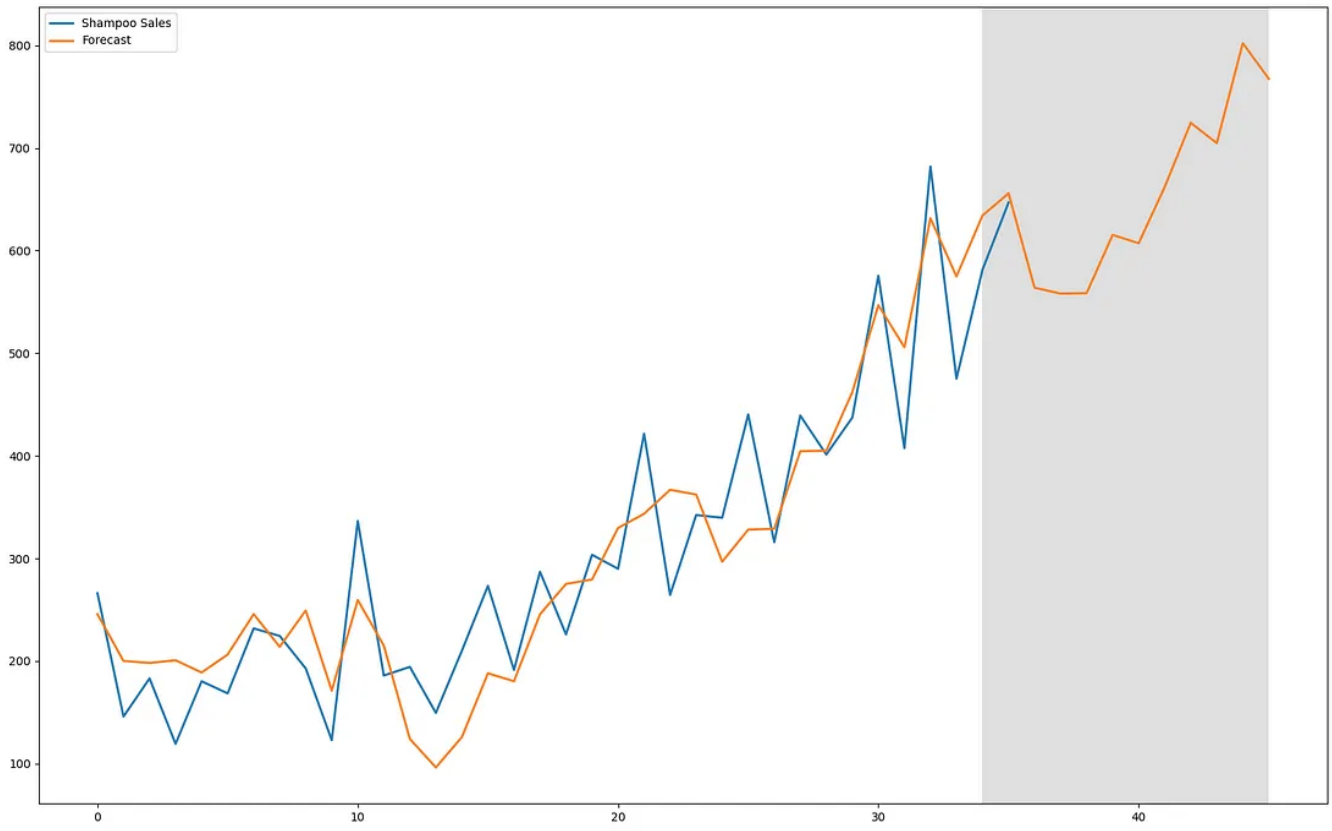
Since our Holt-Winters function already returns the result for an *additive* forecast internally, we can use the output directly:

```
forecast = winters_es(data, 12)
```

We will now plot the forecast against the ground truth to get a better visualization:



Finally, a proper forecast! For further exploration, let's plot every forecast plus a few steps ahead:



We can see that there is now the capturing of a seasonal component.

The complete code for this generation can be seen below:

```
import matplotlib.pyplot as plt

from typing import *

def initial_trend(series: List, uppercase_m: int) -> float:
    return sum([
        float(series[i+uppercase_m] - series[i]) / uppercase_m
        for i in range(uppercase_m)
    ]) / uppercase_m


def initial_seasonality(series: List, uppercase_m: int) -> List:
    initial_season = []
    n_seasons = int(len(series)/uppercase_m)

    season_averages = [sum(
        series[uppercase_m * i:uppercase_m * i + uppercase_m]
    ) / uppercase_m for i in range(n_seasons)]
```

```
    initial_season.extend([
        sum([series[uppercase_m*j+i]-season_averages[j]
             for j in range(n_seasons)]) / n_seasons
        for i in range(uppercase_m)
    ])

    return initial_season

def winters_es(series: List,
               uppercase_m: int,
               alpha: float=0.2,
               beta: float=0.2,
               gamma: float=0.15,
               future_steps: int=1) -> List:

    i_l = [series[0]]
    i_t = [initial_trend(series, uppercase_m)]
    i_s = initial_seasonality(series, uppercase_m)

    forecasts = []
    for t in range(len(series) + future_steps):

        if t >= len(series):
            m = t - len(series) + 1
            forecasts.append(
                (i_l[-1] + m * i_t[-1]) + i_s[t % uppercase_m]
            )

        else:
            l_t = alpha * (series[t] - i_s[t % uppercase_m]) + (1 -
alpha) * (i_l[-1] + i_t[-1])

            i_t[-1] = beta * (l_t - i_l[-1]) + (1 - beta) * i_t[-1]
            i_l[-1] = l_t

            i_s[t % uppercase_m] = gamma * (series[t] - l_t) + (1 -
gamma) * i_s[t % uppercase_m]

            forecasts.append(
                (i_l[-1] + i_t[-1]) + i_s[t % uppercase_m]
            )

    return forecasts

if __name__ == '__main__':

    plt.figure(
        figsize=(32, 20)
    )
```

```
data = pd.read_csv('../data/shampoo.csv')["Sales"]
data = data.values.tolist()

k = 10

last_months = list(range(34, 34 + k))

forecast = winters_es(data, 12, future_steps=10)

plt.plot(data, linewidth=5, label='Shampoo Sales')
plt.plot(forecast, linewidth=4, label='Forecast')

plt.axvspan(*(last_months[0], last_months[-1]),
            facecolor='grey',
            alpha=0.25)

plt.legend()
plt.show()
```

Data Science Python Statistics Forecasting Mathematics

CONCLUSION

We have covered the math, theory and implementation of the Holt-Winters method in Python, complete with a forecasting example. Being triple exponential smoothing, it just hierarchically builds on top of both double exponential smoothing (Holt's method) and simple exponential smoothing. Therefore the method is capable of capturing level, trend and seasonality components, and promptly utilize them in a forecast.

Winters method is an incredibly intuitive and relatively simple forecasting capable of modelling a plethora of time series.

FURTHER READING/REFERENCES

Article sources and extra information for your Holt-Winters needs

- Python implementation
- Initial season explanation
- Amazing video series
- More initial season explanation
- More Holt-Winters explanation

Lleyton Ariton in Analytics Vidhya

No, LSTMs Can't Predict Stock Prices

Despite being an incredibly popular approach, LSTMs are an inherently terrible way of estimating stock prices

8 min read · Feb 4, 2021

762 26

Kia Eisinga in Analytics Vidhya

How to create a Python library

Ever wanted to create a Python library, albeit for your team at work or for some open source project online? In this blog you will learn...

7 min read · Jan 26, 2020

1.91K 23

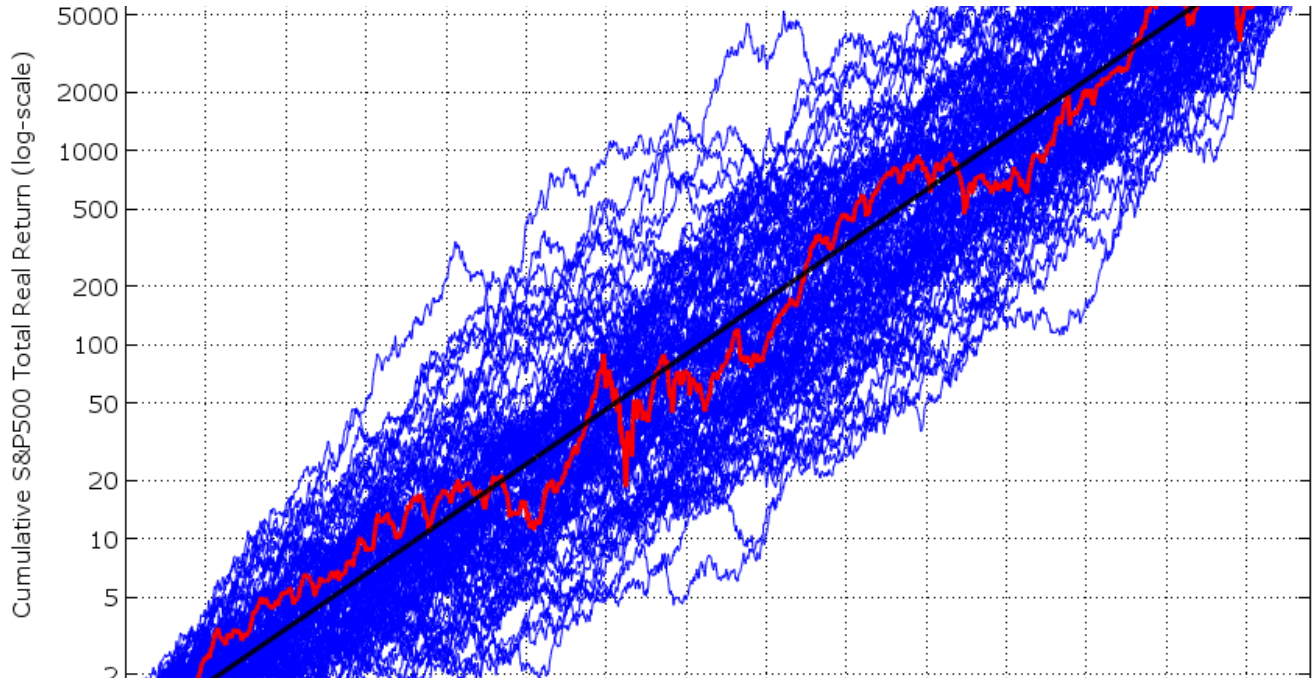
Harikrishnan N B in Analytics Vidhya

Confusion Matrix, Accuracy, Precision, Recall, F1 Score

Binary Classification Metric

6 min read · Dec 10, 2019

536 6



Lleyton Ariton in Analytics Vidhya

Can LSTMs Predict Stock Prices? — A Complete Analysis (Part 1)

A thorough analysis of the Random Walk Theory and the subsequent efficacy of using LSTMs for stock price prediction.

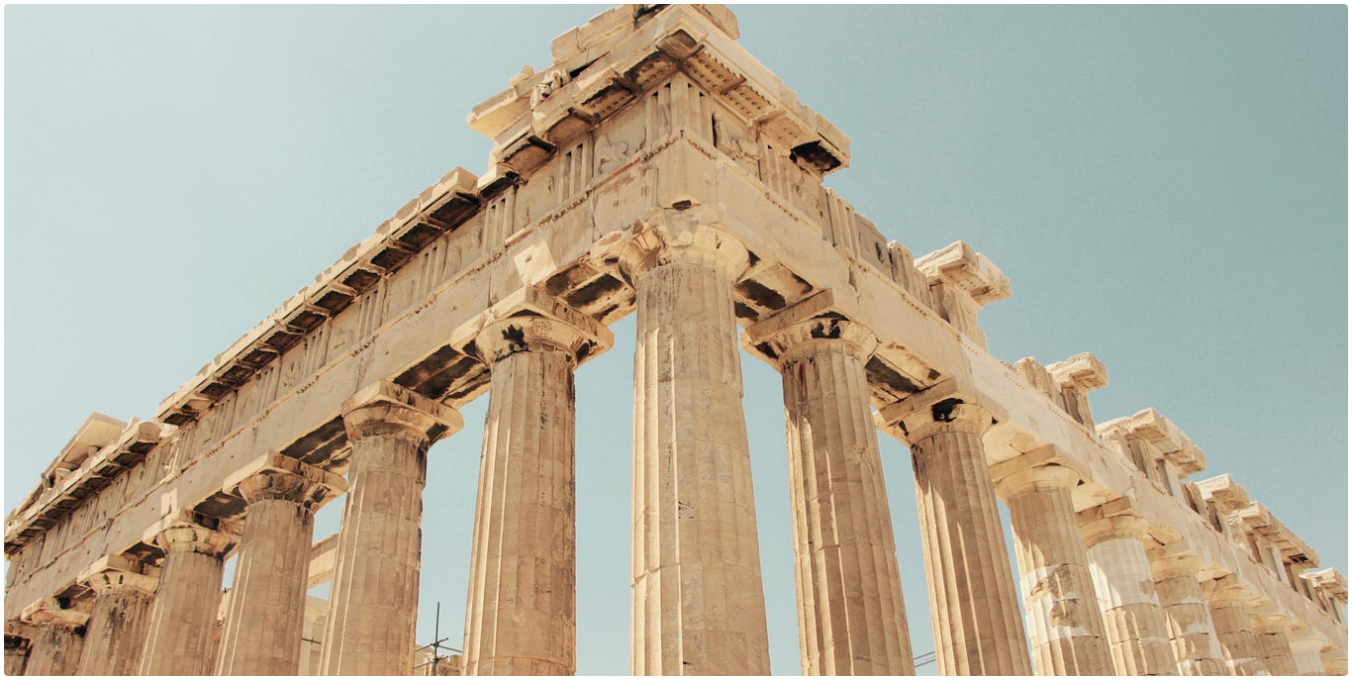
18 min read · Mar 16, 2021

612 12

See all from Lleyton Ariton

See all from Analytics Vidhya

Recommended from Medium

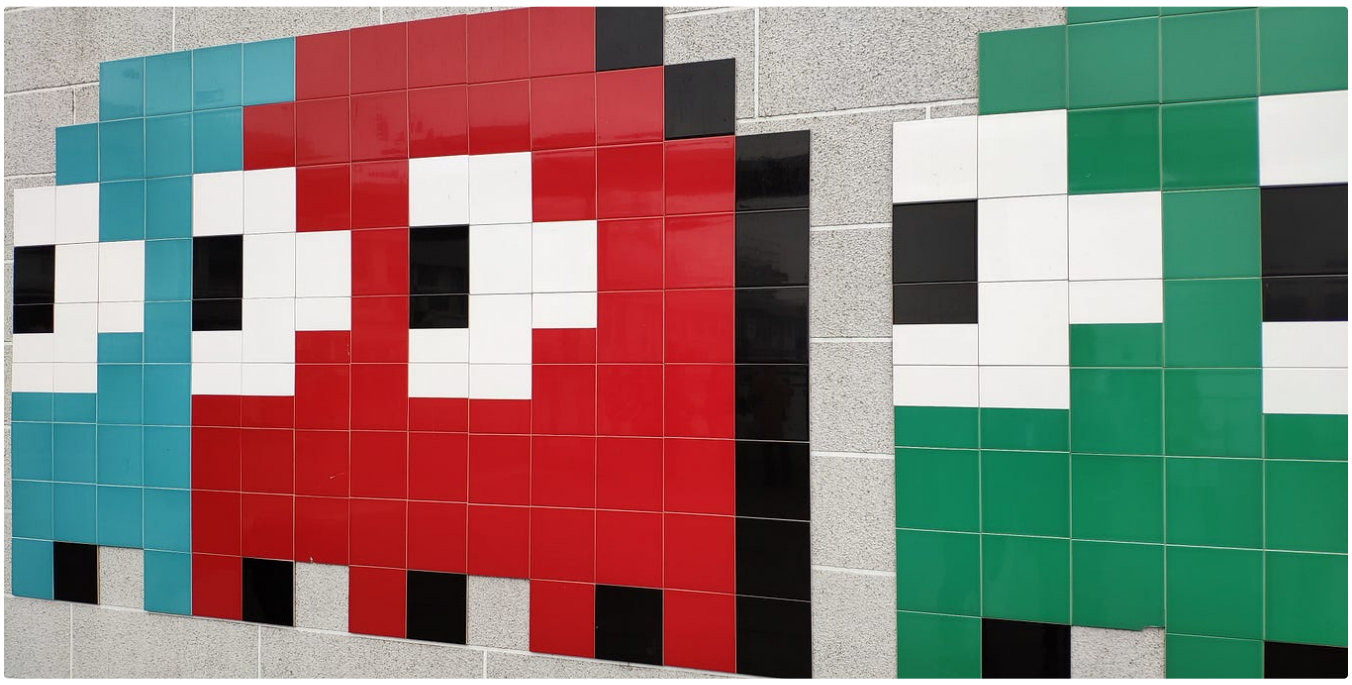


Marco Peixeiro in Towards Data Science

Theta Model for Time Series Forecasting

A hands-on tutorial on how to apply the Theta model for time series forecasting in Python

· 9 min read · Nov 2, 2022

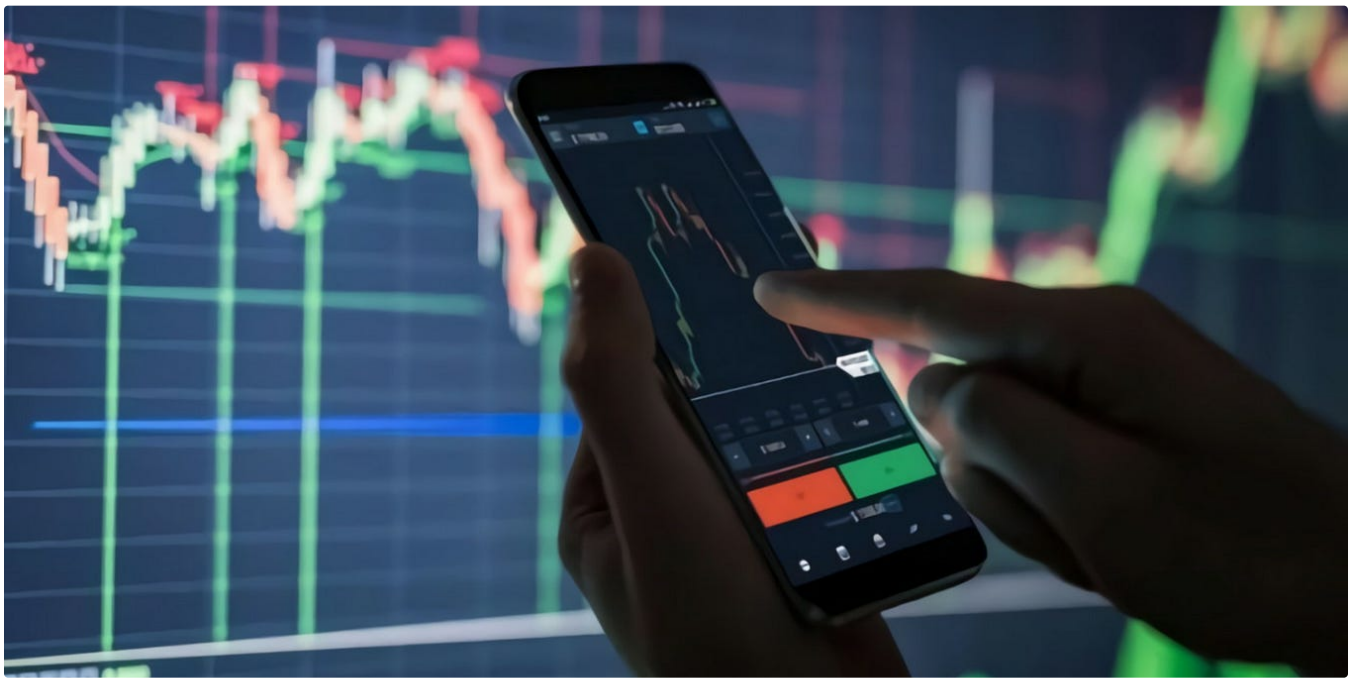


Marco Cerliani in Towards Data Science

Forecast Time Series with Missing Values: Beyond Linear Interpolation

Comparing Alternatives to Handle Missing Values in Time Series

· 5 min read · Oct 13, 2022



Egor Howell in Towards AI

How To Forecast With SARIMA

A deep dive into the SARIMA model and its applications

✨ · 6 min read · Feb 28

172

Egor Howell in Towards Data Science

How To Forecast With ARIMA

An introduction to the ARIMA forecasting model and how to use it

✨ · 7 min read · Jan 31

219 1

Cloud & Data Science

Introduction to Time Series Analysis in Python

Time series analysis is the process of using statistical methods to analyze time-based data in order to understand trends and patterns. It...

· 3 min read · Feb 11

51

Vitor Cerqueira in Towards Data Science

3 Types of Seasonality and How to Detect Them

Understanding time series seasonality

✦ · 6 min read · Jun 30

 308 

See more recommendations