Photo by Chris Lawton on Unsplash

Member-only story

Time Series DIY: Seasonal Decomposition

Learn what happens under the hood of `statsmodel's seasonal_decompose`

Eryk Lewinson · Follow
Published in Towards Data Science
7 min read · Mar 31, 2022

Listen Share More

If you have worked with time series, you have probably already used `seasonal_decompose` from `statsmodel` (or R's equivalent). Long story short, it splits a time series into three components: trend, seasonality, and the residuals. After running the command, you see something like the plot below.

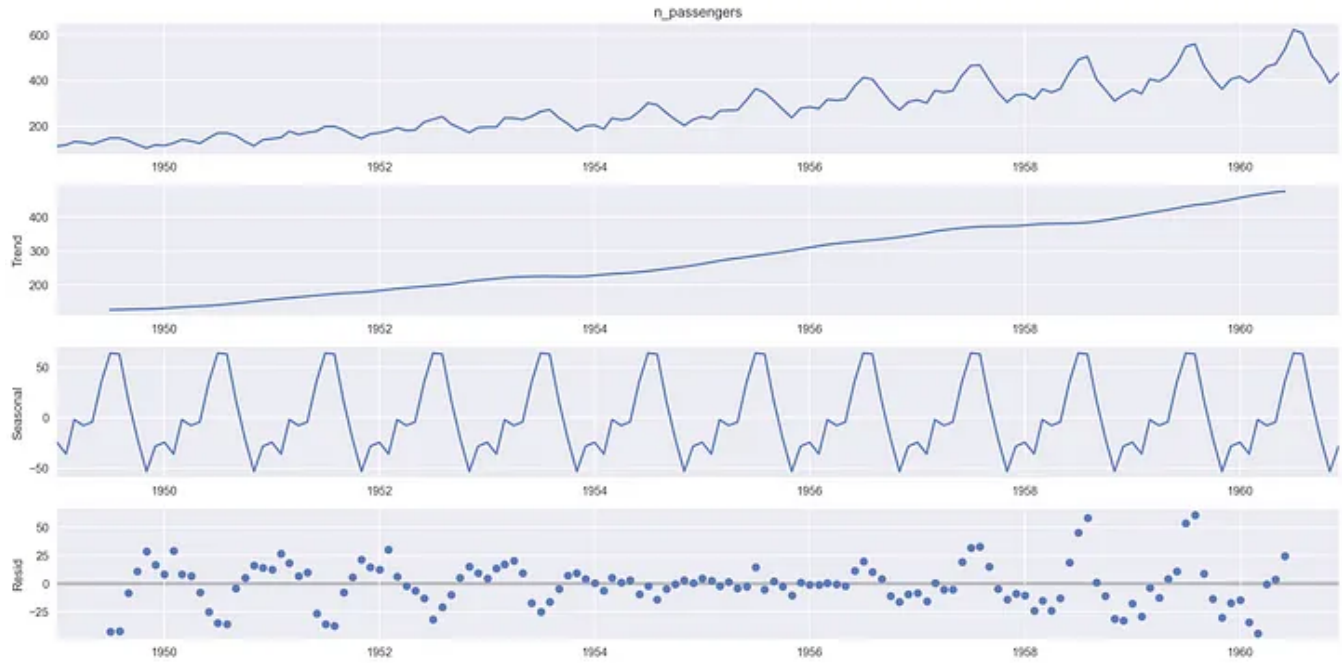


Image by author

The interpretation of the components is also very intuitive:

- trend — the general direction of the series over a long period of time
- seasonality — a distinct, repeating pattern observed in regular intervals due to various seasonal factors. Could be monthly, weekly, etc.
- residual — the irregular component consisting of the fluctuations in the time series after removing the previous components

This should be enough of a recap. I will also not go into much detail on the goal of seasonal decomposition (understanding the data, forecasting, outlier detection). Instead, I wanted to explore a less popular angle — what actually happens when you call the `seasonal_decompose` function. In this hand-on article, we will answer the question: *How are those components estimated?* If you are curious, read on!

Theory

Let's assume we are dealing with the additive model, that is, consisting of a linear trend and seasonal cycle with the same frequency (width) and amplitude (height).

For the multiplicative model, you just need to replace the additions with multiplications and subtractions with divisions.

Trend component

Trend is calculated using a centered moving average of the time series. The moving average is calculated using a window length corresponding to the frequency of the time series. For example, we would use a window of length 12 for monthly data.

Smoothing the series using such a moving average comes together with some disadvantages. First, we are "losing" the first and last few observations of the series. Second, the MA tends to over-smooth the series, which makes it less reactive to sudden changes in the trend or jumps.

Seasonal component

To calculate the seasonal component, we first need to detrend the time series. We do it by subtracting the trend component from the original time series (remember, we divide for the multiplicative variant).

Having done that, we calculate the average values of the detrended series for each seasonal period. In the case of months, we would calculate the average detrended value for each month.

The seasonal component is simply built from the seasonal averages repeated for the length of the entire series Again, this is one of the arguments against using the simple seasonal decomposition — the seasonal component is not allowed to change over time, which can be a very strict and often unrealistic assumption for longer time series.

On a side note, in the additive decomposition the detrended series is centered at zero, as adding zero makes no change to the trend. The same logic is applied in the multiplicative approach, with the difference that it is centered around one. That is because multiplying the trend by one also has no effect on it.

Residuals

The last component is simply what is left after removing (by subtracting or dividing) the trend and seasonal components from the original time series.

That would be all for the theory, let's code!

Step-by-step tutorial

Setup

As always, we start by importing the libraries.

```
import pandas as pd
import numpy as np
from statsmodels.tsa.seasonal import seasonal_decompose

import matplotlib.pyplot as plt
```

ts_decomposition_1.py hosted with ❤ by GitHub view raw

Data

We will be using probably the most popular dataset for time series analysis — the Australian airline passengers time series. We load it from a CSV file (available here), but you can also get it from other sources, for example, from the `seaborn` library (here).

```
df = pd.read_csv("../data/air_passengers.csv", index_col=0)
df.index = pd.to_datetime(df.index)
y = df["#Passengers"]
y.name = "n_passengers"

y.plot(title="Airline passengers");
```

ts_decomposition_2.py hosted with ❤ by GitHub view raw

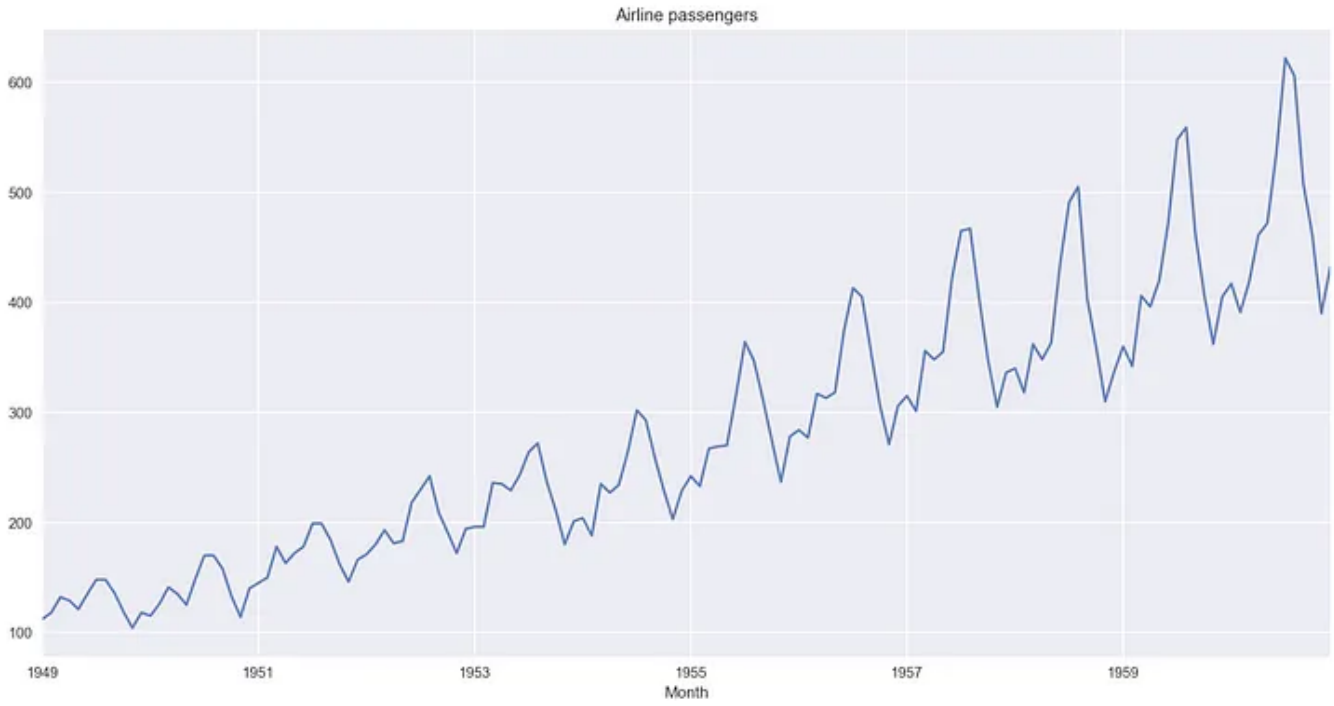


Image by author

Just by eye-balling the plot, it seems like the multiplicative decomposition might be a better choice (especially when looking at the increasing strength of the seasonal component over time). But we will stay in line with what we assumed in the intro and carry out the additive decomposition. We leave the multiplicative one as an optional exercise.

Benchmark from statsmodels

Before decomposing the time series ourselves, we can get the benchmark using `statsmodels`.

```
1  seasonal_decomp = seasonal_decompose(y, model="additive")
2  seasonal_decomp.plot();
```

ts_decomposition_3.py hosted with ❤ by GitHub view raw

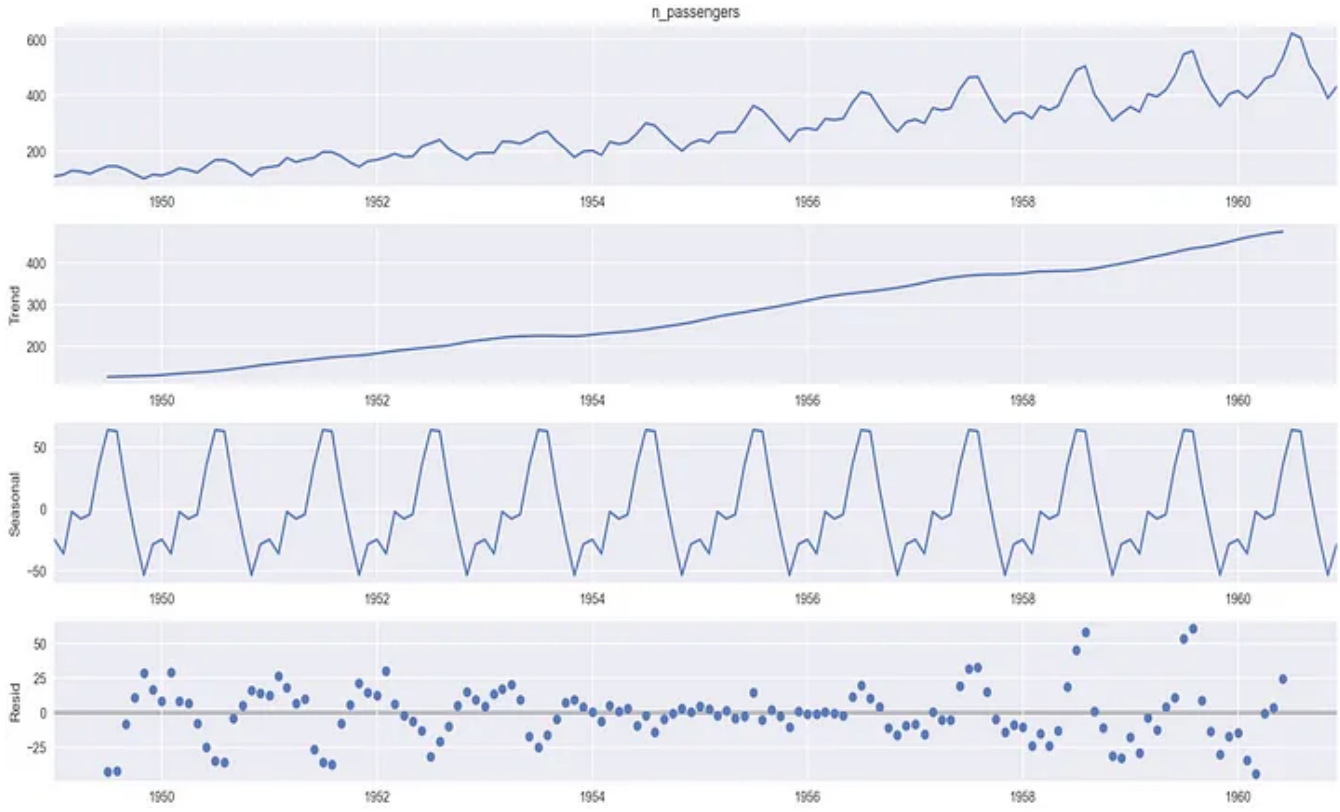


Image by author

In the plot we can see another hint that the additive model is not the right choice here —there are clear patterns in the residuals over time. In case of a good fit, we would expect the residuals to behave randomly without any pattern.

For reference, we can easily extract the components from the `DecomposeResult` object. They are stored under `trend`, `seasonal`, and `resid`.

Manual decomposition

For the manual decomposition, we will use a `pandas` DataFrame to store the original series, the extracted components, and all the intermediate results.

We have already written out the battle plan in the theory section above, so let's execute all the steps in one code snippet (you can follow the cell-by-cell flow in the Notebook on GitHub).

```
# create the DF
seasonal_df = y.to_frame()

# calculate the trend component
seasonal_df["trend"] = seasonal_df["n_passengers"].rolling(window=13, center=True).mean()

# detrend the series
seasonal_df["detrended"] = seasonal_df["n_passengers"] - seasonal_df["trend"]

# calculate the seasonal component
seasonal_df.index = pd.to_datetime(seasonal_df.index)
seasonal_df["month"] = seasonal_df.index.month
seasonal_df["seasonality"] = seasonal_df.groupby("month")["detrended"].transform("mean")

# get the residuals
seasonal_df["resid"] = seasonal_df["detrended"] - seasonal_df["seasonality"]

# display the DF
seasonal_df.head(15)
```

ts_decomposition_4.py hosted with ❤ by GitHub view raw

Running the snippet generates the following table:

Month	n_passengers	trend	detrended	month	seasonality	resid
1949-01-01	112	NaN	NaN	1	-30.825175	NaN
1949-02-01	118	NaN	NaN	2	-42.027972	NaN
1949-03-01	132	NaN	NaN	3	-4.139860	NaN
1949-04-01	129	NaN	NaN	4	-6.944056	NaN
1949-05-01	121	NaN	NaN	5	-0.699301	NaN
1949-06-01	135	NaN	NaN	6	37.146853	NaN
1949-07-01	148	125.769231	22.230769	7	64.923077	-42.692308
1949-08-01	148	126.846154	21.153846	8	64.580420	-43.426573
1949-09-01	136	128.615385	7.384615	9	15.636364	-8.251748
1949-10-01	119	128.846154	-9.846154	10	-20.951049	11.104895
1949-11-01	104	128.538462	-24.538462	11	-54.090909	29.552448
1949-12-01	118	130.692308	-12.692308	12	-31.972028	19.279720
1950-01-01	115	133.384615	-18.384615	1	-30.825175	12.440559
1950-02-01	126	135.076923	-9.076923	2	-42.027972	32.951049
1950-03-01	141	135.846154	5.153846	3	-4.139860	9.293706

Image by author

A few things worth mentioning about the calculations:

- we have used a rolling window of length 13 (12 months + 1 to make it an odd number for the centered average).
- we used a very handy method called `transform` to calculate the average values per group. We used it to avoid the need of creating a separate DataFrame with the aggregated values and then joining it back to the original DF. You can read more about it (and some other useful `pandas` functionalities) here.
- we displayed the first 15 rows so we can see that the aggregated seasonal component is calculated correctly for all the months (including the ones that have missing values for the detrended series).

Lastly, we plot the decomposition.

```
(
    seasonal_df
    .loc[:, ["n_passengers", "trend", "seasonality", "resid"]]
    .plot(subplots=True, title="Seasonal decomposition - additive")
);
```

ts_decomposition_5.py hosted with ❤ by GitHub view raw

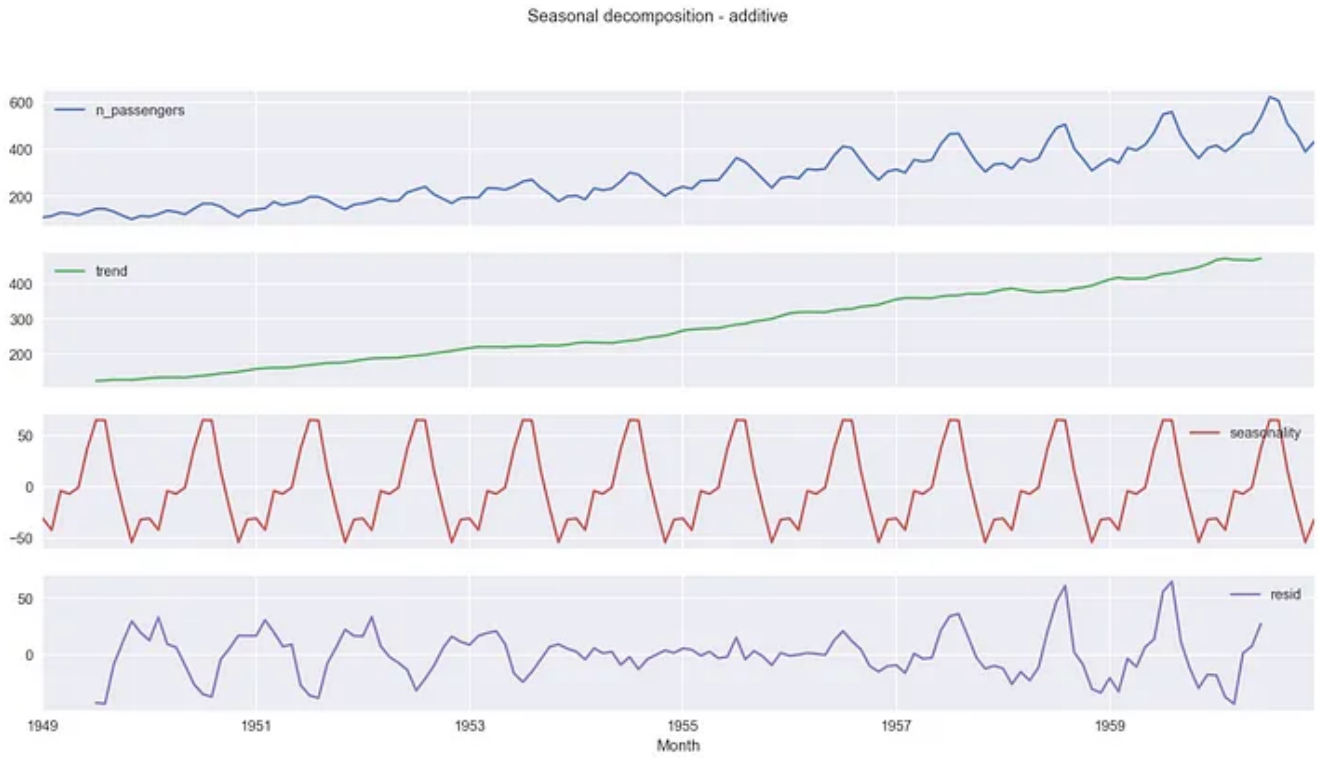


Image by author

The results look very similar to what we have obtained using `statsmodels`. We did not plot the residuals as points (instead of the default line), however, we should still be able to see the overlapping patterns quite easily.

Results comparison

To compare if we have an exact match, we could look at the components extracted with `seasonal_decompose` and compare them to the ones we calculated manually. Spoiler alert: they are very similar, yet not the same.

We opted for another approach, that is, comparing the two decompositions visually. First, we store the results of the manual decomposition in the `DecomposeResult` object. Then, we borrowed a great helper function to add the results of a decomposition to an existing decomposition plot (source of the function).

```python
from statsmodels.tsa.seasonal import DecomposeResult

manual_decomposition = DecomposeResult(
    seasonal=seasonal_df["seasonality"],
    trend=seasonal_df["trend"],
    resid=seasonal_df["resid"],
    observed=seasonal_df["n_passengers"],
)

def add_second_decomp_plot(fig, res, legend):
    axs = fig.get_axes()
    comps = ["trend", "seasonal", "resid"]
    for ax, comp in zip(axs[1:], comps):
        series = getattr(res, comp)
        if comp == "resid":
            ax.plot(series, marker="o", linestyle="none")
        else:
            ax.plot(series)
            if comp == "trend":
                ax.legend(legend, frameon=False)

fig = seasonal_decomp.plot()
add_second_decomp_plot(fig, manual_decomposition, ["statsmodels", "manual"]);
```


Image by author

In the figure above, we see that the results are very close. The differences can be attributed to the way that the components are calculated — as always, the devil is in the details. In `statsmodels`, the moving average used for extracting the trend component is calculated using a 1-D convolution filter (calculated using the `convolution_filter` function). As you can see, the outcome is very similar, but still a bit different. This then propagates to the seasonal component.

Takeaways

- the basic approach to seasonal decomposition splits the time series into three components: trend, seasonal and residuals,
- the trend component is calculated as a centered moving average of the original series,
- the seasonal component is calculated as the per period average of the detrended series,
- the residual component is obtained after removing the trend and seasonal components from the time series.

You can find the code used for this article on my GitHub. Also, any constructive feedback is welcome. You can reach out to me on Twitter or in the comments.

Liked the article? Become a Medium member to continue learning by reading without limits. If you use this link to become a member, you will support me at no extra cost to you. Thanks in advance and see you around!

You might also be interested in one of the following:

Pandas Is Not Enough? A Comprehensive Guide To Alternative Data Wrangling Solutions

Including Dask, Modin, polars, Vaex, Terality and 6 others

towardsdatascience.com

A Step-by-Step Guide to Calculating Autocorrelation and Partial Autocorrelation

How to calculate the ACF and PACF values from scratch in Python

towardsdatascience.com

LinkedIn's response to Prophet — Silverkite and Greykite

An overview of a new algorithm for time series forecasting

towardsdatascience.com

References

- Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) *Time Series Analysis, Forecasting and Control.* Third Edition. Holden-Day. Series G.

Time Series Analysis | Data Science | Python | Statistics | Time Series Forecasting

Follow

Written by Eryk Lewinson

11.4K Followers · Writer for Towards Data Science

Data Scientist, quantitative finance, gamer. My latest book - Python for Finance Cookbook 2nd ed: https://t.ly/WHHP

More from Eryk Lewinson and Towards Data Science

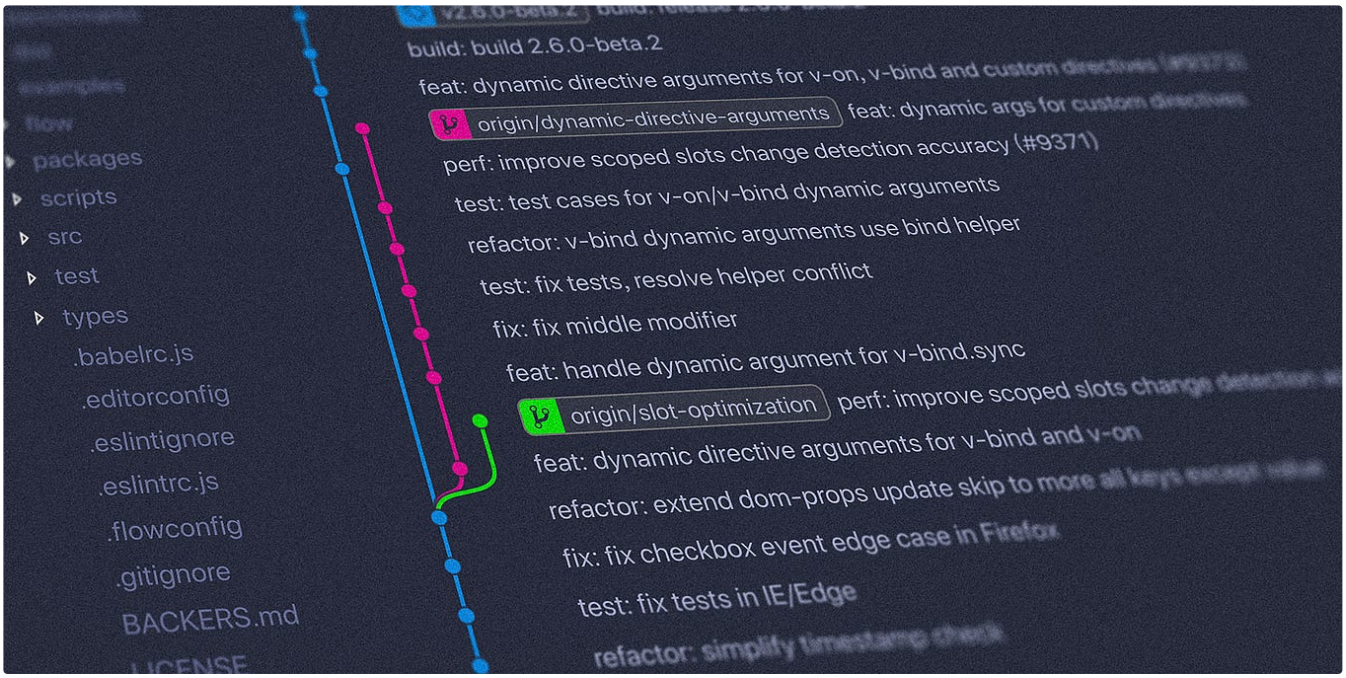
Eryk Lewinson in Geek Culture

Top 10 VS Code Extensions for Data Science

Enhance Your Productivity with These Must-Have Tools!

· 7 min read · Apr 1

187 4



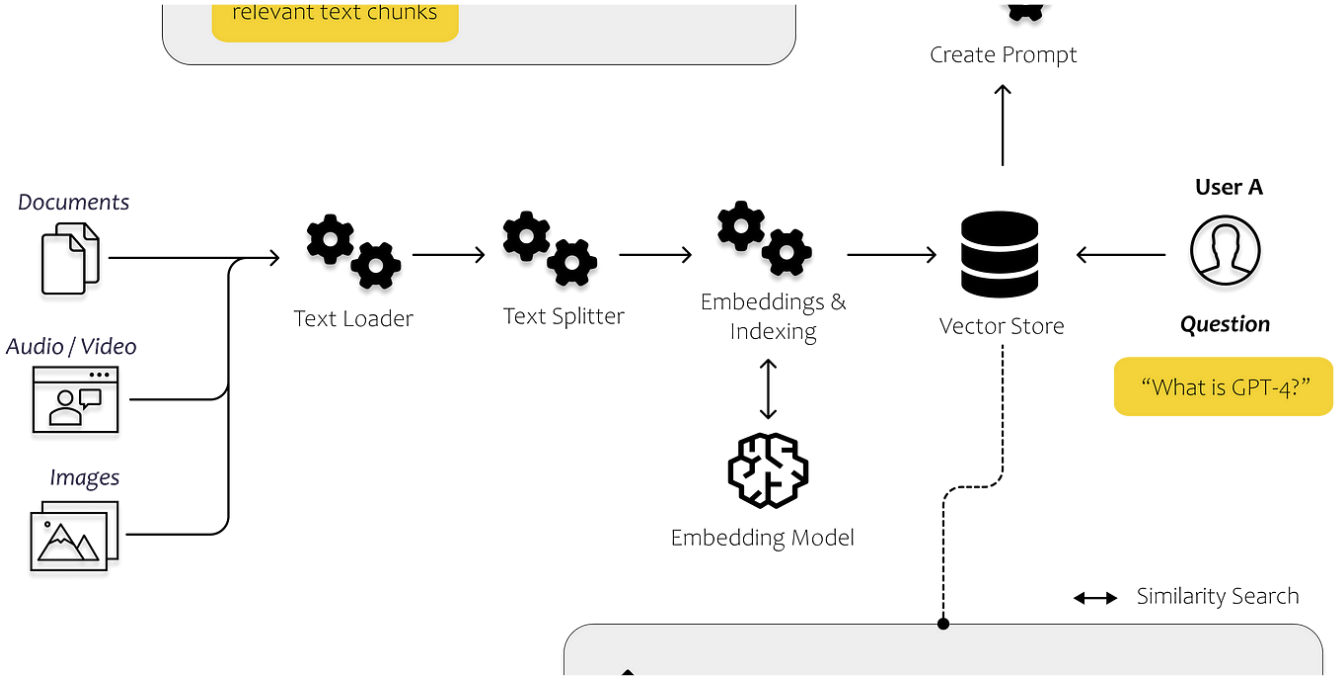
Miriam Santos in Towards Data Science

Pandas 2.0: A Game-Changer for Data Scientists?

The Top 5 Features for Efficient Data Manipulation

7 min read · Jun 27

1.5K 18



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 22

2K 19

Eryk Lewinson in Geek Culture

Top 4 Python libraries for technical analysis

With an example of calculating Bollinger Bands

· 5 min read · Nov 5, 2021

Egor Howell in Towards Data Science

How To Forecast With Moving Average Models

Tutorial and theory on how to carry out forecasts with moving average models

· 6 min read · Jan 23

186

Tyler Blume in Towards Data Science

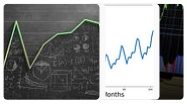
Fixing Prophet's Forecasting Issue

Step 1: Constrain the Insane Trend

✦ · 10 min read · Jan 25

238 2

Lists



Predictive Modeling w/ Python

18 stories · 108 saves

Practical Guides to Machine Learning

10 stories · 120 saves

Coding & Development

11 stories · 46 saves

New_Reading_List

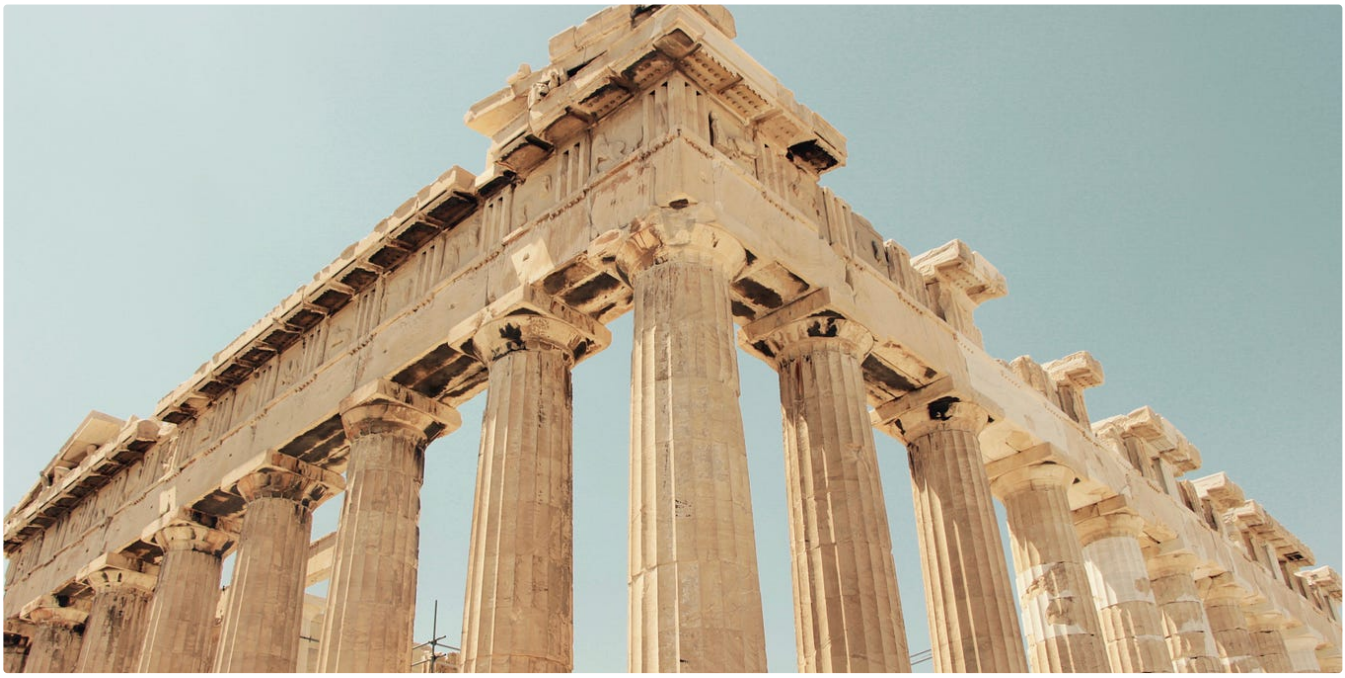
174 stories · 18 saves

Satyam Kumar in Towards Data Science

5 Changepoint Detection algorithms every Data Scientist should know

Essential guide to changepoint detection algorithms for time series analysis

✦ · 3 min read · Mar 7

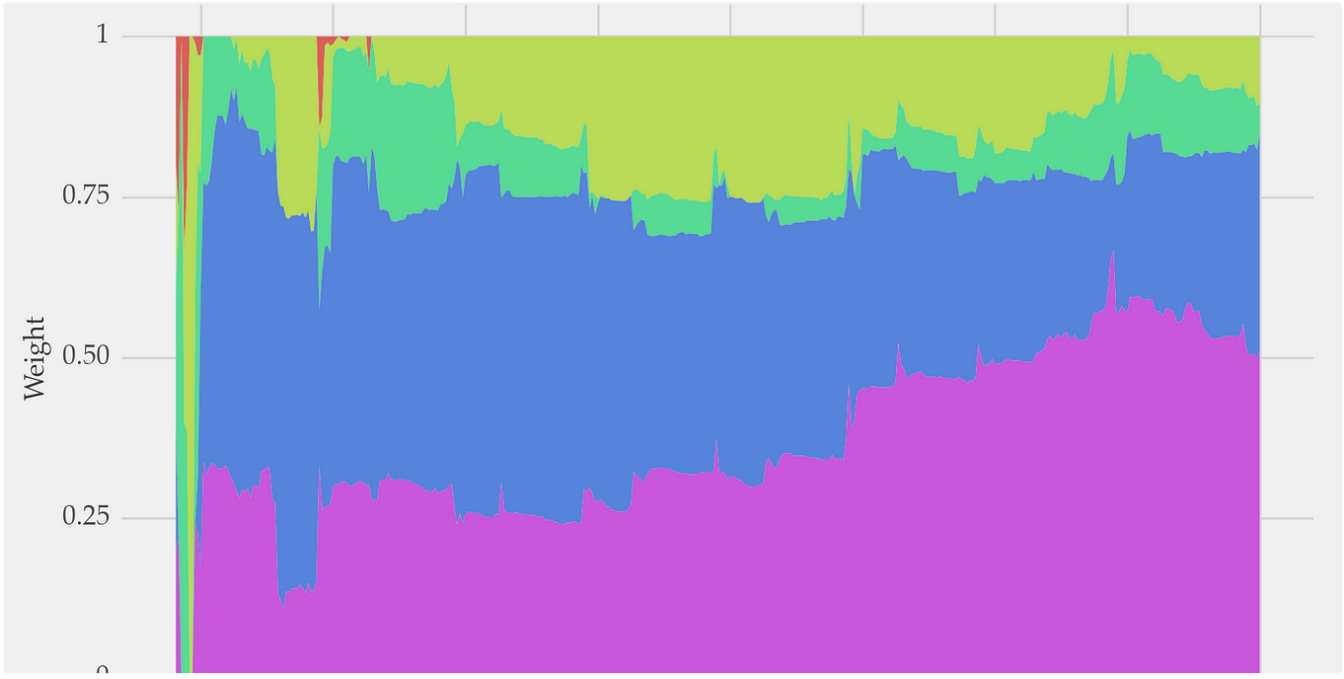


Marco Peixeiro in Towards Data Science

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



Vitor Cerqueira in Towards Data Science

Dynamic Forecast Combination using R from Python

Exploring rpy2 to call R methods from Python

✦ · 6 min read · Jan 25

91 2

Arthur Mello in Geek Culture

Bayesian Time Series Forecasting

Theory and practice using PyBATS

✦ · 6 min read · Jan 20

139 1

See more recommendations