Lecture 2: Introduction to Forecasting

Contents

Lecture Outline

- Describe and implement the following simple forecasting methods: naive, average, seasonally-adjusted naive, seasonal naive, and drift
- At a high level, explain how exponential smoothing models work
- Describe the difference between simple exponential smoothing, Holt's method and Holt-Winter's method
- Build an exponential smoothing model using the `statsmodels` functions `SimpleExpSmoothing()`, `Holt()`, and `ExponentialSmoothing()`
- Explain the difference between the `statsmodels` functions `ExponentialSmoothing()` and `ETSModel()`
- Describe common approaches for in-sample and out-of-sample time series model evaluation

Lecture Learning Objectives

Imports

```
import numpy as np
import pandas as pd
import plotly.express as px
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.api import SimpleExpSmoothing, Holt, ExponentialSmoothing, ET
from sklearn.metrics import mean_absolute_error, mean_squared_error, mean_absolute
# Custom modules
from scripts.plotting import *
from scripts.utils import mean_absolute_scaled_error
# Plot defaults
px.defaults.height = 600; px.defaults.width = 800
plt.style.use("bmh"); plt.rcParams.update({"font.size": 16, "figure.figsize": (7,5
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 10
      8 # Custom modules
      9 from scripts.plotting import *
---> 10 from scripts.utils import mean_absolute_scaled_error
     11 # Plot defaults
     12 px.defaults.height = 600; px.defaults.width = 800

File ~/Desktop/MDS/DSCI_574_spat-temp-mod_students/lectures/scripts/utils.py:4
      2 import numpy as np
      3 import pandas as pd
----> 4 from shapely.geometry import Polygon
      5 from sklearn.metrics import mean_absolute_error
      8 def mean_absolute_scaled_error(y_true, y_pred, y_train):

ModuleNotFoundError: No module named 'shapely'
```

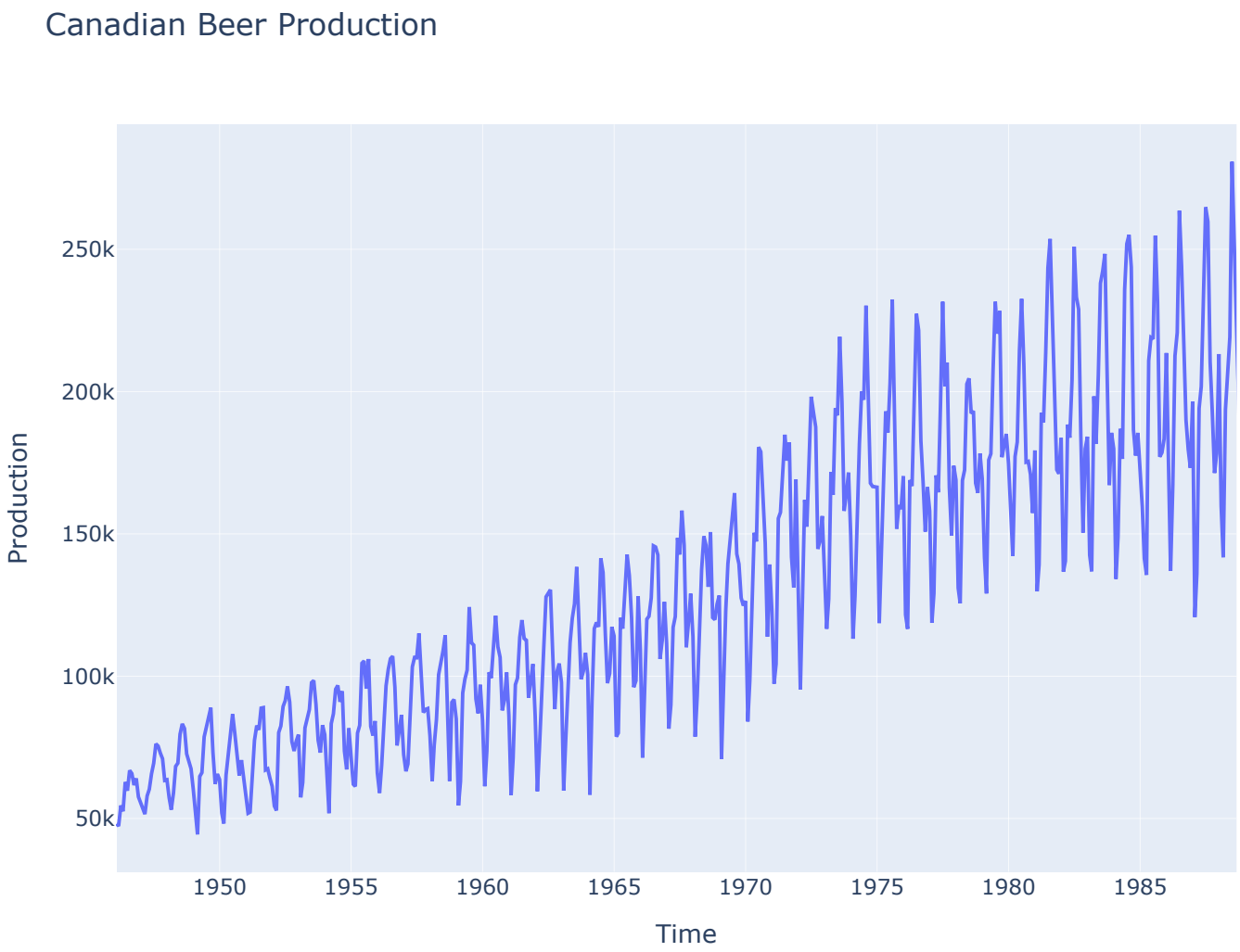
1. Forecasting

- In this lecture we'll play with the Canadian beer production dataset (obtained from Statistics Canada, see Appendix A)

```
def get_beer(split="train", label="Observed"):
    df = pd.read_csv(f"data/beer_{split}.csv", index_col=0, parse_dates=["Time"]).
    df.index.freq = "ME"
    return df
```

- Let's start with the beer production dataset:

```
df = get_beer()
px.line(df, y="Production", title="Canadian Beer Production")
```



Stop & think: What do you notice about the time-series

- Trend?
- Seasonality?
- Multiplicative or additive model?

2. Baseline forecasting methods

- Here we will explore 5 simple forecasting methods:
    1. **Average**
    2. **Naive**
    3. **Seasonally-adjusted naive**
    4. **Seasonal naive**
    5. **Drift**
- While these simple methods typically serve as benchmarks to compare more advanced methods to, they are often very difficult to beat!
- For the remainder of these notes we'll use:
    - y_t to denote the value of a time series at time t
    - h to represent a forecast horizon, i.e., $h = 1$ means we want to predict one time step ahead
    - T to be the length of a time series
    - $\hat{y}$ to be a forecasted value, y to be an observed value
    - $\hat{y}_{t|t-1}$ to mean the value of $\hat{y}_t$ given y_{t-1}

2.1. Average forecast

- Use the average of the series for all future forecasts:

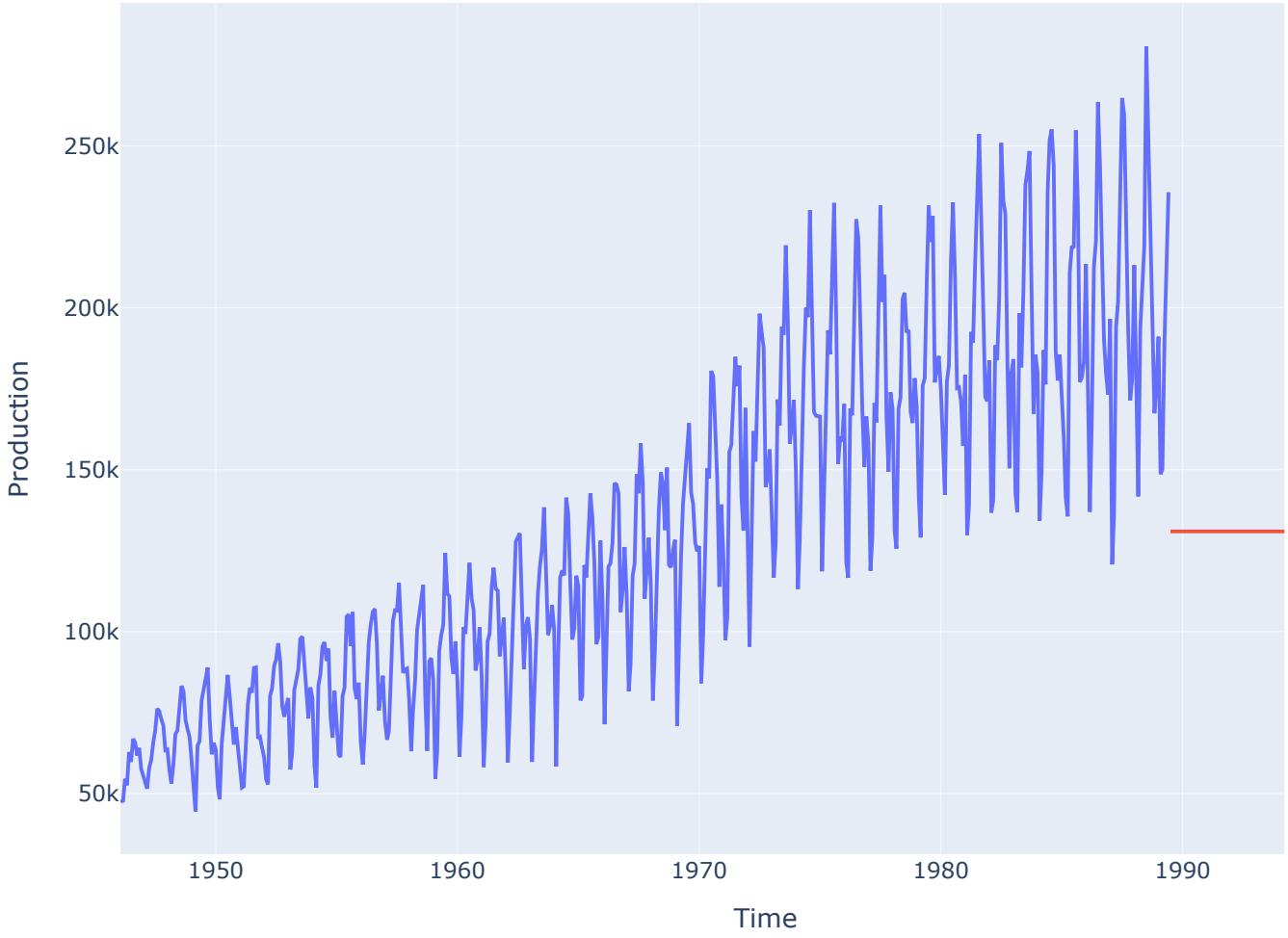
$$\hat{y}_{T+h} = \bar{y}$$

```
# Forecast variables
def create_forecast_index(start, horizon, freq="ME"):
    return pd.date_range(start + pd.tseries.frequencies.to_offset(freq), periods=h

forecast_index = create_forecast_index(df.index[-1], 120)
# forecast_index
```

```
# Make the forecast and store in a dataframe
average = pd.DataFrame({"Production": df["Production"].mean(),
                        "Label": "Average forecast"},
                       index=forecast_index)
average.head()
```

Time	Production	Label
1989-06-30	130945.180422	Average forecast
1989-07-31	130945.180422	Average forecast
1989-08-31	130945.180422	Average forecast
1989-09-30	130945.180422	Average forecast
1989-10-31	130945.180422	Average forecast

```
px.line(pd.concat((df, average)), y="Production", width=950, color="Label")
```



2.2. Naive

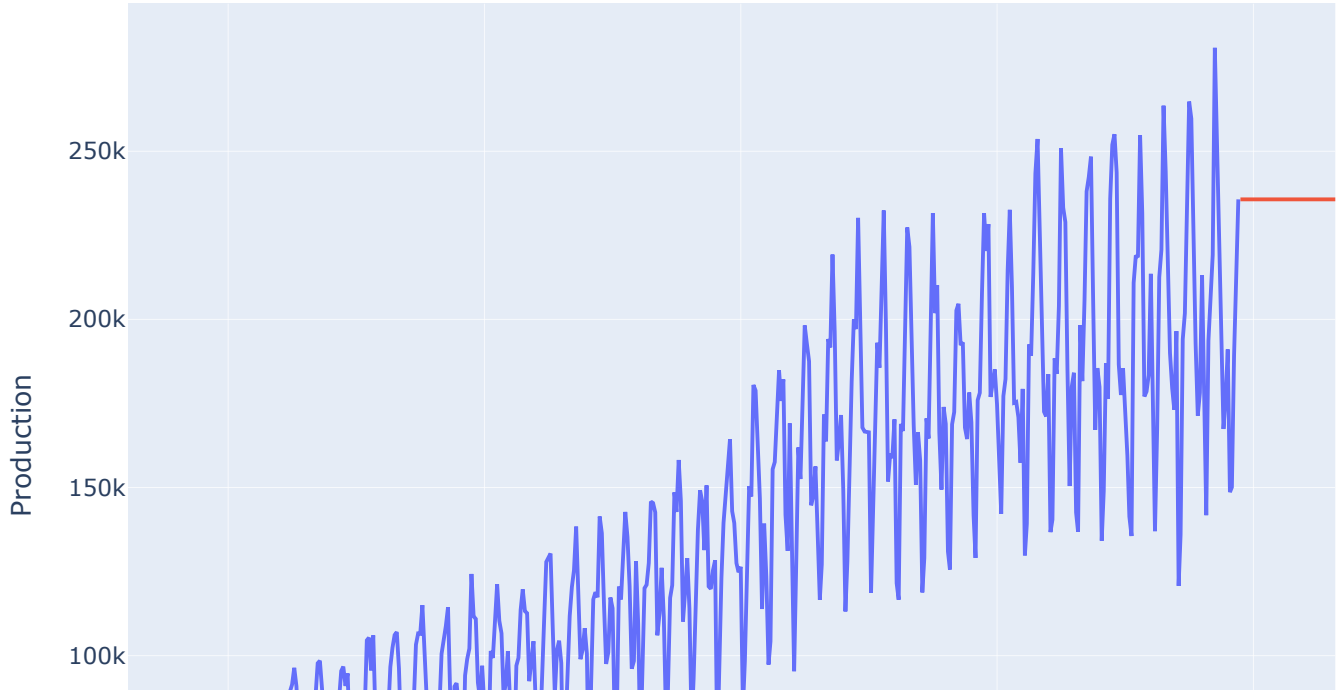
Use the last observation for all forecasts:

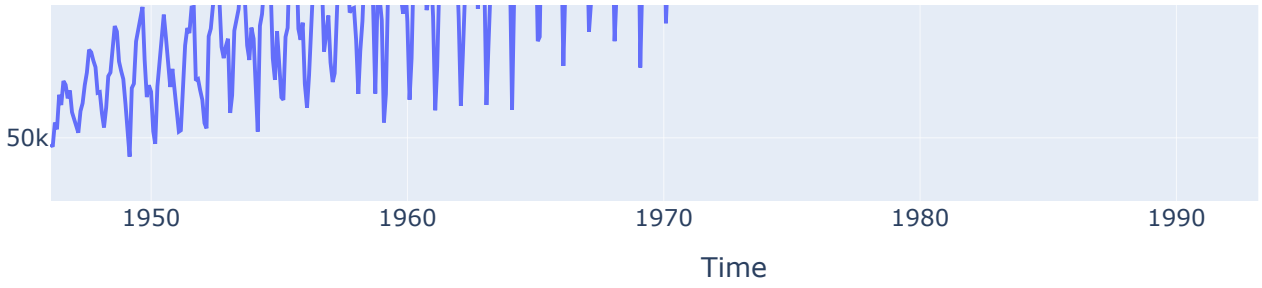
$$\hat{y}_{T+h|T} = y_T$$

```
naive = pd.DataFrame({"Production": df["Production"].iloc[-1],
                      "Label": "Naive forecast"},
                     index=forecast_index)
naive.head()
```

	Production	Label
Time		
1989-06-30	235698.0	Naive forecast
1989-07-31	235698.0	Naive forecast
1989-08-31	235698.0	Naive forecast
1989-09-30	235698.0	Naive forecast
1989-10-31	235698.0	Naive forecast

```
px.line(pd.concat((df, naive)), y="Production", width=950, color="Label")
```

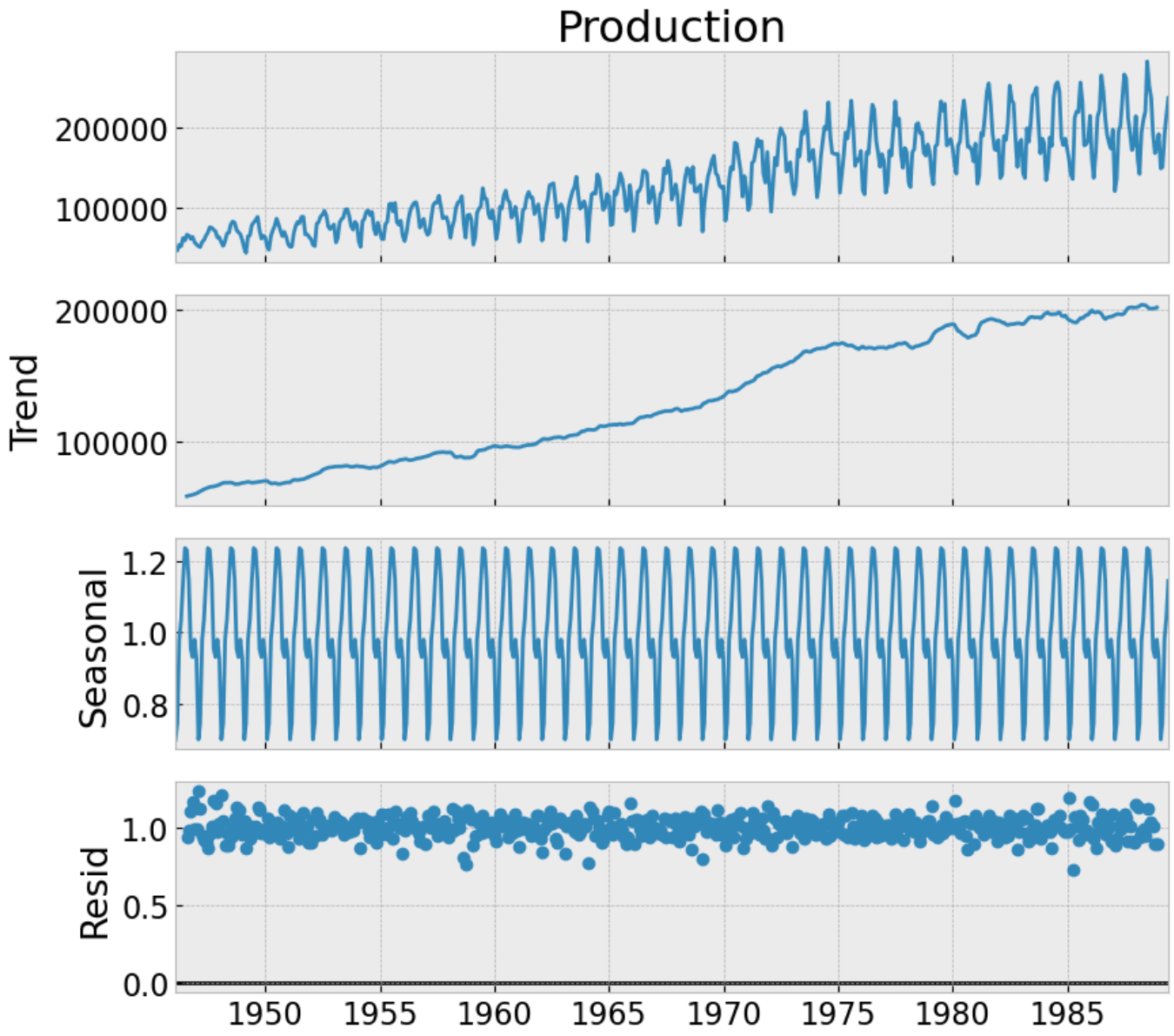




2.3. Seasonally-adjusted naive

- Exactly the same as the naive method but the data are seasonally adjusted by applying a classical decomposition (in this case, a multiplicative model):

```
# decompose using multiplicative model
model = seasonal_decompose(df["Production"], model="mult", period=12)
plt.rcParams.update({"font.size": 16, "figure.figsize": (9,8), "lines.linewidth":
model.plot();
```

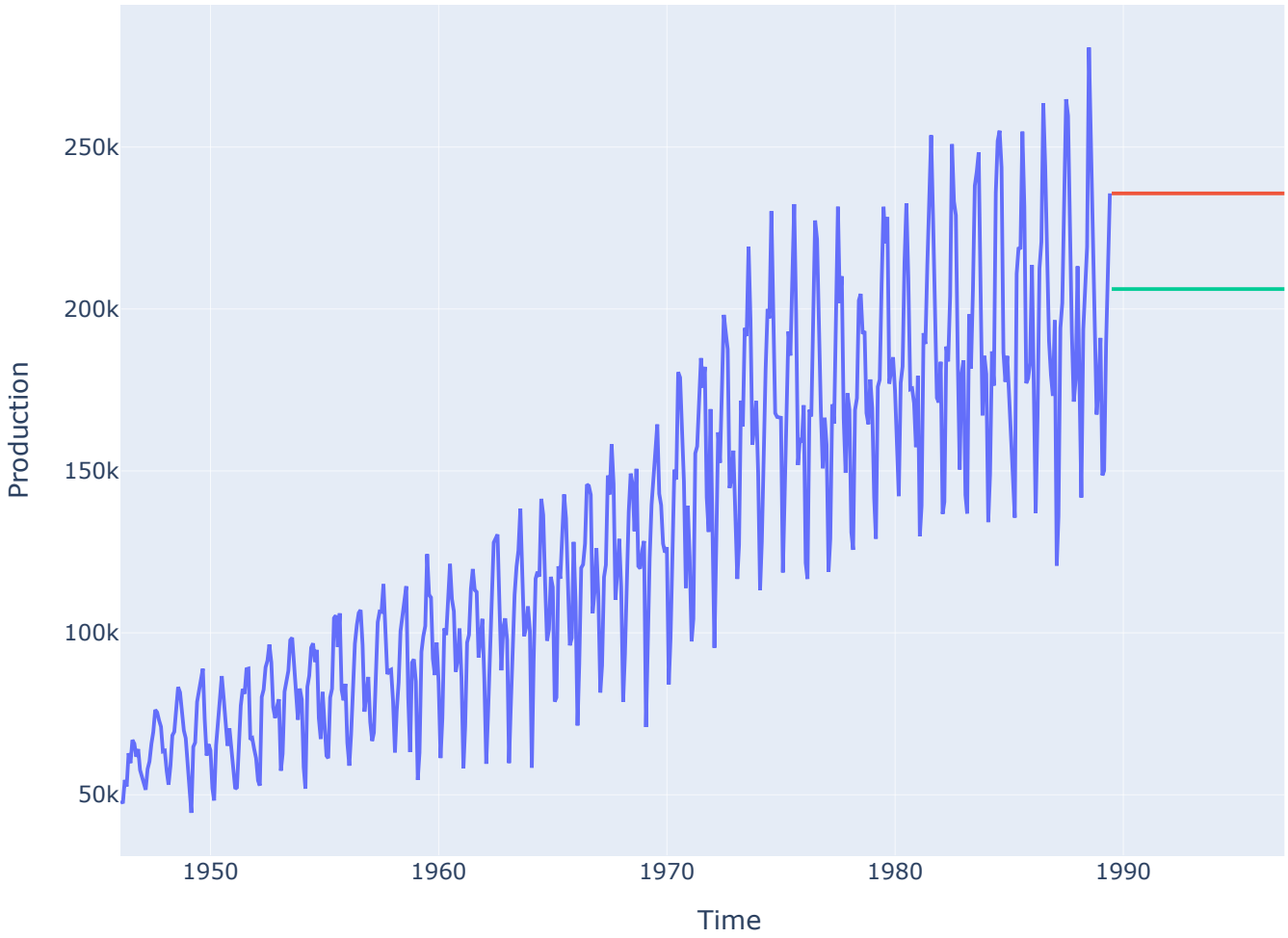


Stop & think

Do you know why the residuals of a multiplicative decomposition center around 1 instead of 0 (in additive composition)?

```
# divided by seasonal component
naive_adj = pd.DataFrame({"Production": (df["Production"] / model.seasonal).iloc[-
                          "Label": "Adjusted naive forecast"},
                         index=forecast_index)
```

```
px.line(pd.concat((df, naive, naive_adj)), y="Production", color="Label", width=95
```



2.4. Seasonal naive

- Set each forecast as the last observed value from the same season of the year (e.g., the same month of the previous year)
- E.g., for our monthly data, the forecasts for all future Januarys is the last observed January value:

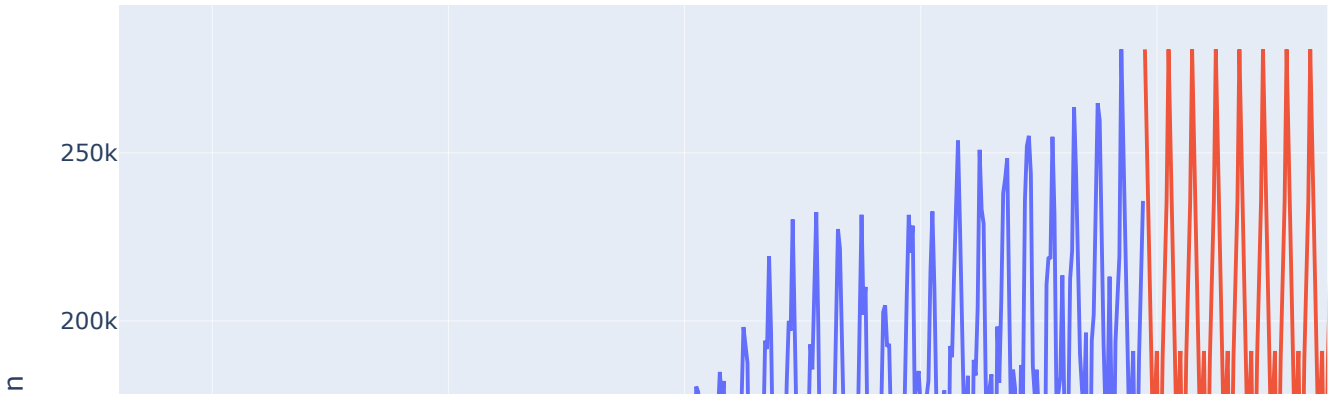
```
df["month"] = df.index.month
last_season = (df.drop_duplicates("month", keep="last")
                 .sort_values(by="month")
                 .set_index("month")["Production"]
               )
df = df.drop(columns="month")
last_season
```

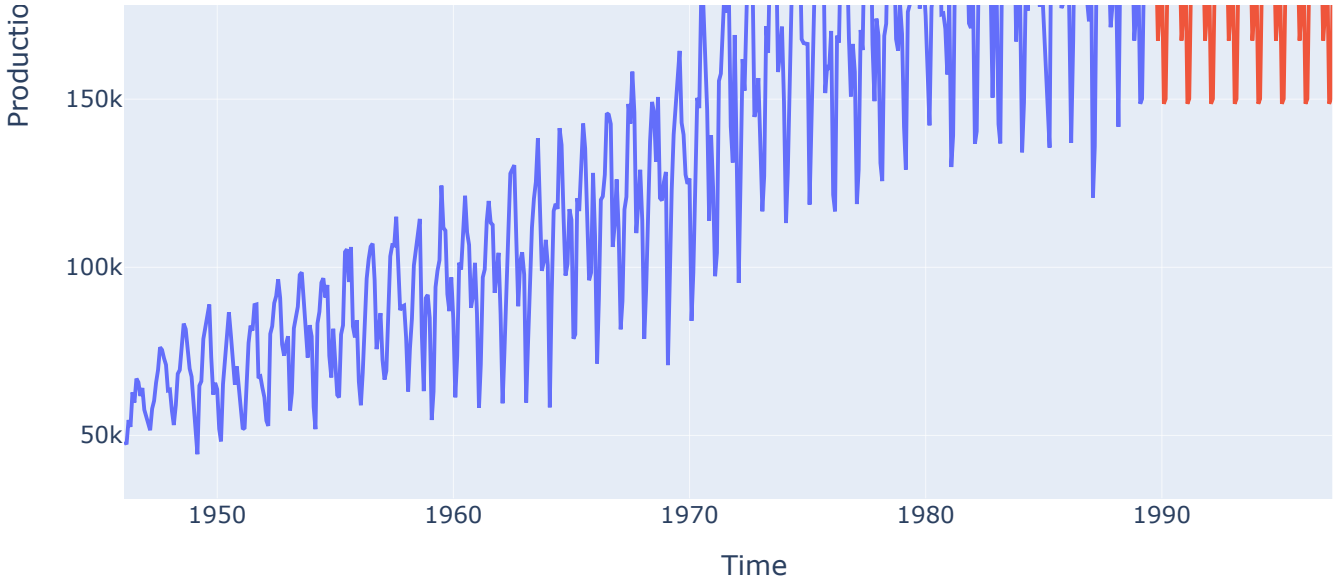
```
month
1     148629.0
2     150100.0
3     189178.0
4     211568.0
5     235698.0
6     280766.0
7     251298.0
8     236171.0
9     192124.0
10    167422.0
11    176063.0
12    191076.0
Name: Production, dtype: float64
```

```
seasonal = pd.DataFrame(index=forecast_index)
seasonal = seasonal.assign(**{"Production": seasonal.index.month.map(lambda x: las
                              "Label": "Seasonal naive forecast"}
                           )
seasonal.head()
```

	Production	Label
Time		
1989-06-30	280766.0	Seasonal naive forecast
1989-07-31	251298.0	Seasonal naive forecast
1989-08-31	236171.0	Seasonal naive forecast
1989-09-30	192124.0	Seasonal naive forecast
1989-10-31	167422.0	Seasonal naive forecast

```
px.line(pd.concat((df, seasonal)), y="Production", color="Label", width=950)
```





2.5. Drift

- Forecasts equal to last value in the series plus the average change of the series:

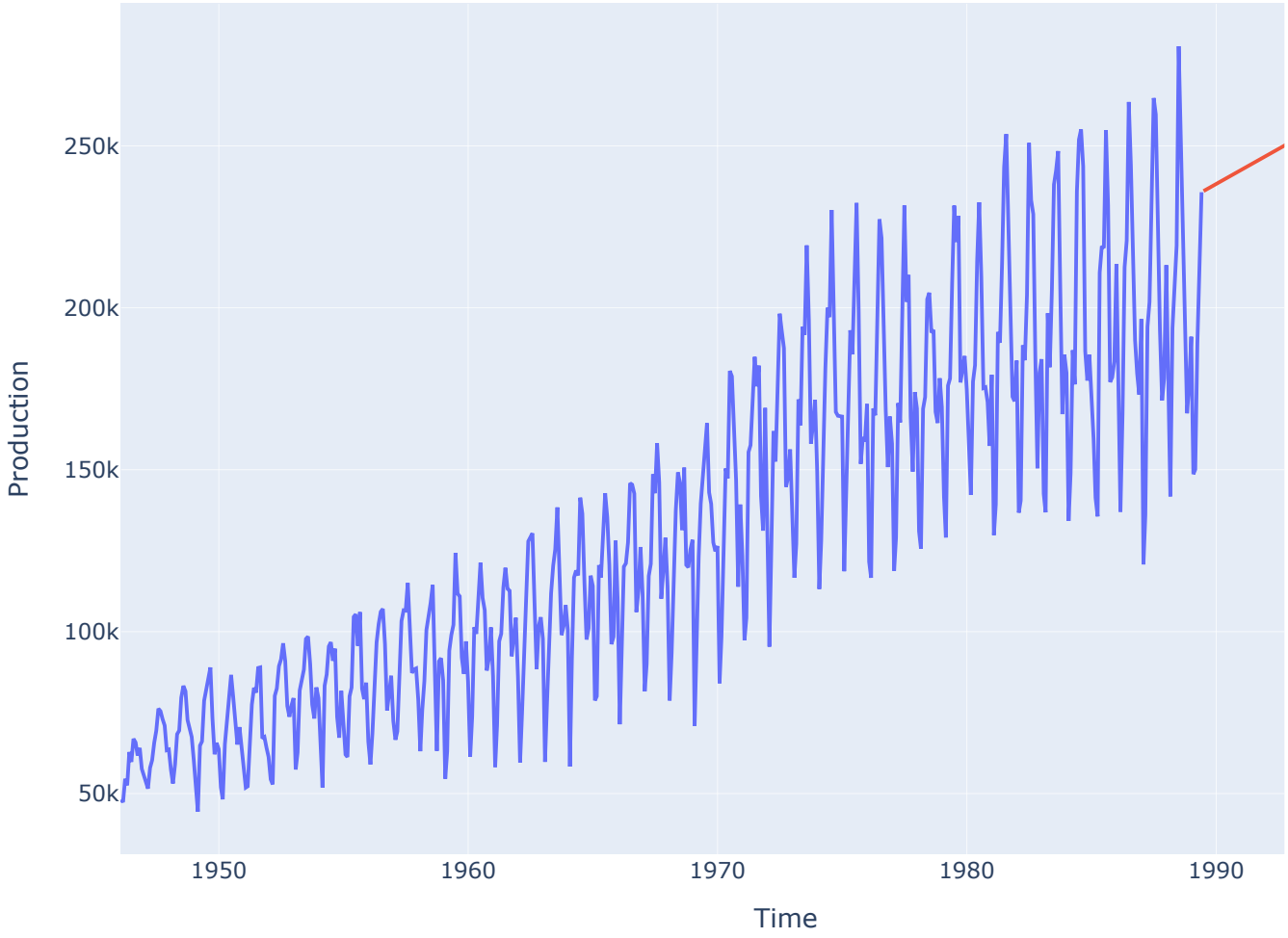
$$\hat{y}_{T+h|T} = y_T + h\left(\frac{y_T - y_1}{T-1}\right)$$

This is equivalent to drawing a line between the first and last observations, and extrapolating it into the future.

```
drift = pd.DataFrame({"Production": df["Production"].iloc[-1],
                      "Label": "Drift forecast"},
                     index=forecast_index)
slope = (df["Production"].iloc[-1] - df["Production"].iloc[0]) / (len(df) - 1)
drift["Production"] += np.full_like(drift["Production"], slope).cumsum()
drift.head()
```

Time	Production	Label
1989-06-30	236058.655769	Drift forecast
1989-07-31	236419.311538	Drift forecast
1989-08-31	236779.967308	Drift forecast
1989-09-30	237140.623077	Drift forecast
1989-10-31	237501.278846	Drift forecast

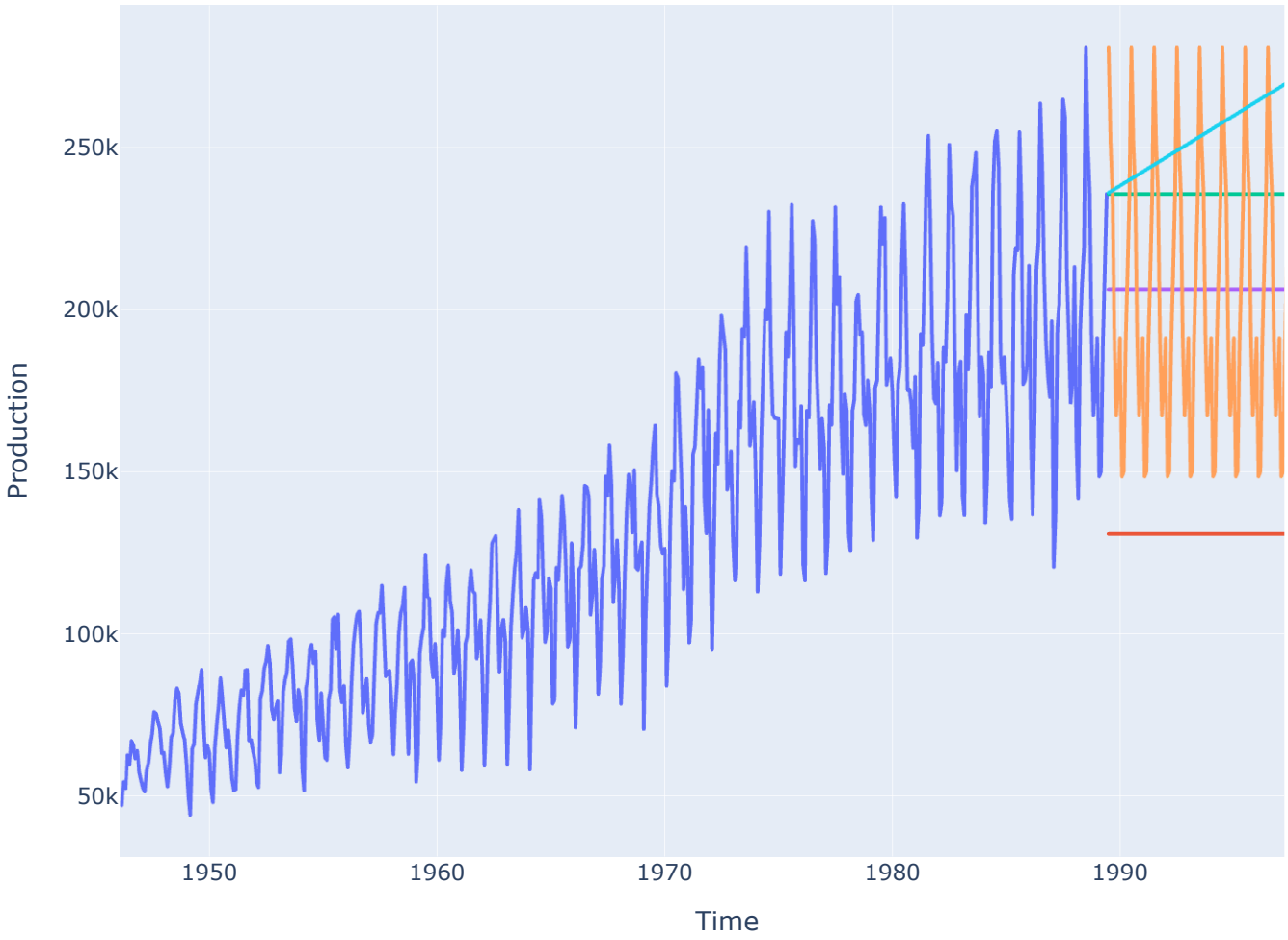
```
px.line(pd.concat((df, drift)), y="Production", color="Label", width=950)
```



2.6. All together now

- Which method would you choose? (we'll look at objective ways to make this decision later on)

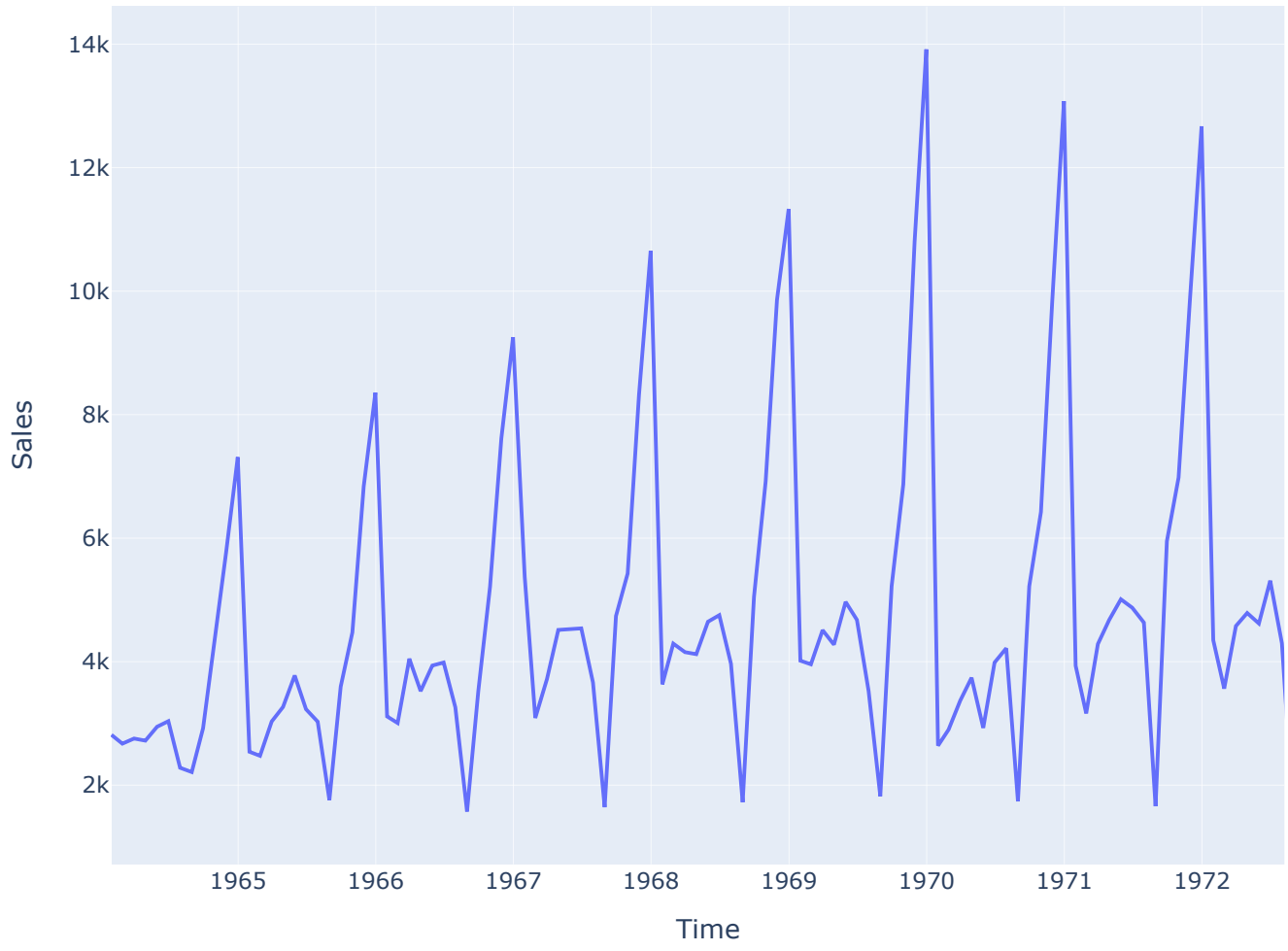
```
px.line(pd.concat((df, average, naive, naive_adj, seasonal, drift)), y="Production
```



iClicker Exercise

Identify the baseline forecasting method that was used for monthly champagne sales for the Perrin Freres label from 1964 to 1972 (as always, check out Appendix A to see where this data came from):

```
df = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)

df['Label'] = 'Observed'
px.line(df, y="Sales")
```



Question 1

A. Average

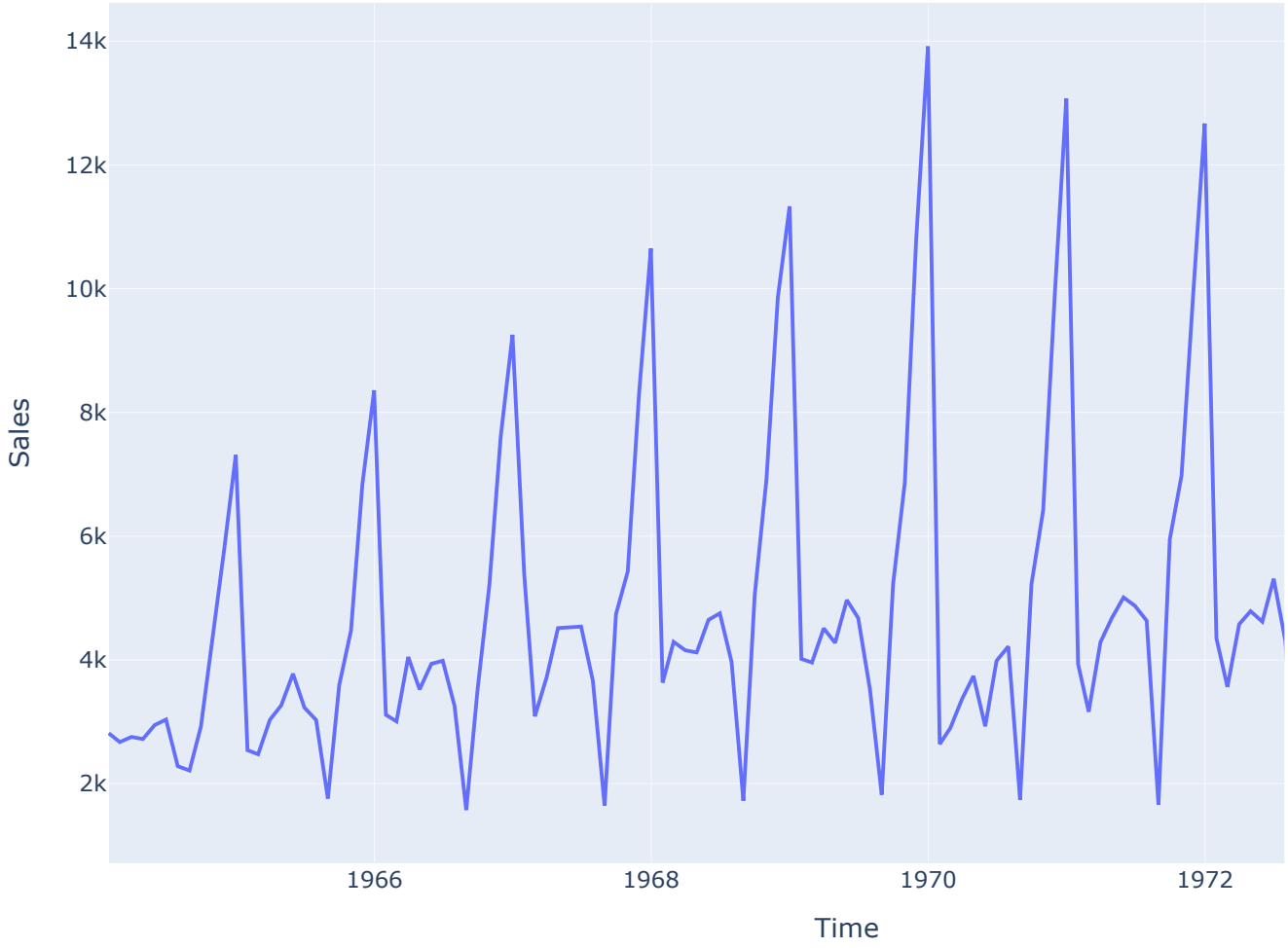
B. Naive

C. Adjusted naive

D. Seasonal naive

E. Drift

```
forecast_index = create_forecast_index(df.index[-1], 24)

average = pd.DataFrame({"Sales": df["Sales"].mean(),
                        "Label": "Average forecast"},
                       index=forecast_index)

fig = px.line(pd.concat((df, average)), y="Sales", width=950, color="Label")

fig.update_layout(showlegend=False)
```



Question 2

A. Average

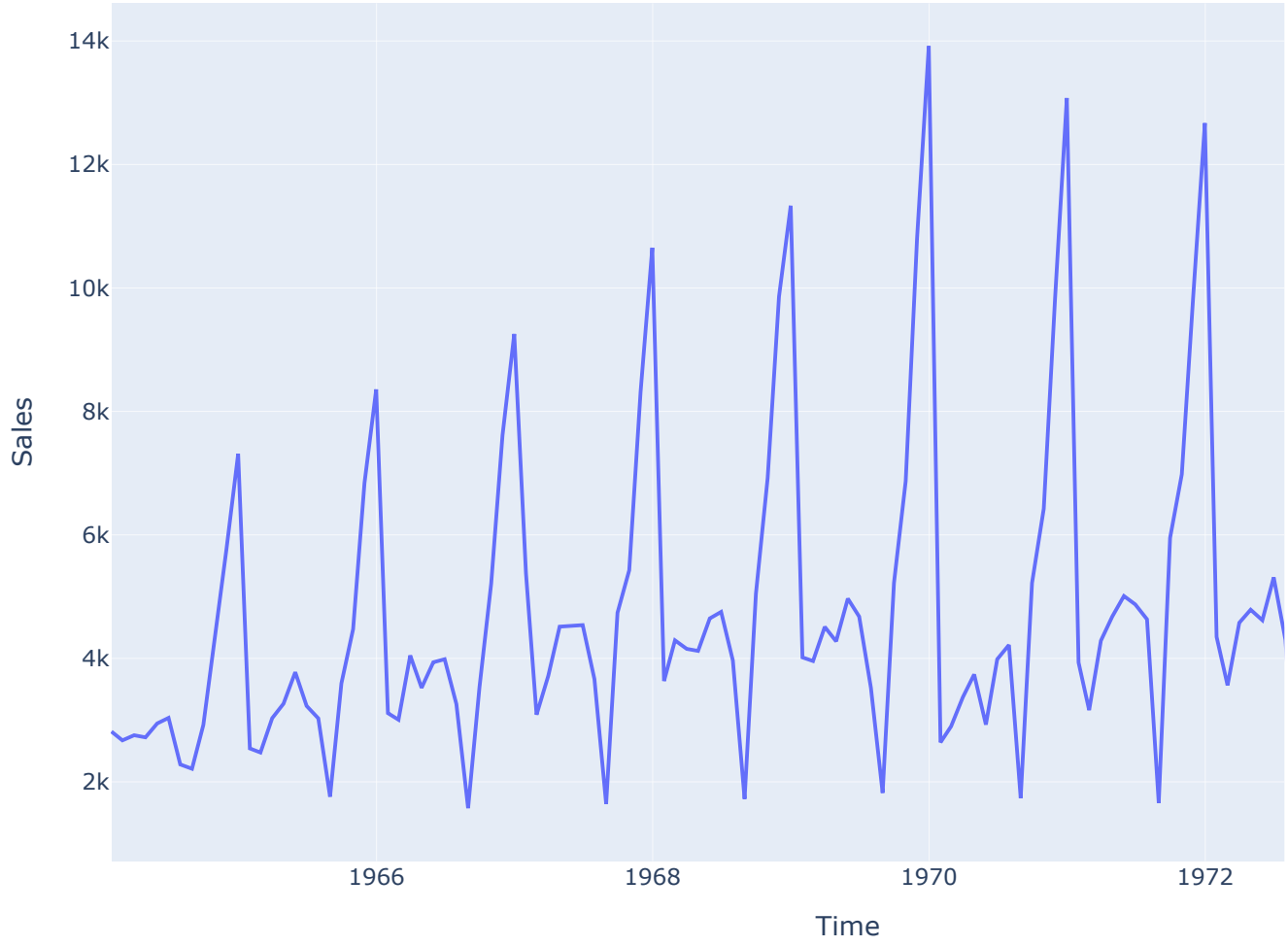
B. Naive

C. Adjusted naive

D. Seasonal naive

E. Drift

```
drift = pd.DataFrame({"Sales": df["Sales"].iloc[-1],
                      "Label": "Drift forecast"},
                     index=forecast_index)
slope = (df["Sales"].iloc[-1] - df["Sales"].iloc[0]) / (len(df) - 1)
drift["Sales"] += np.full_like(drift["Sales"], slope).cumsum()

fig = px.line(pd.concat((df, drift)), y="Sales", width=950, color="Label")

fig.update_layout(showlegend=False)
```



Question 3

A. Average

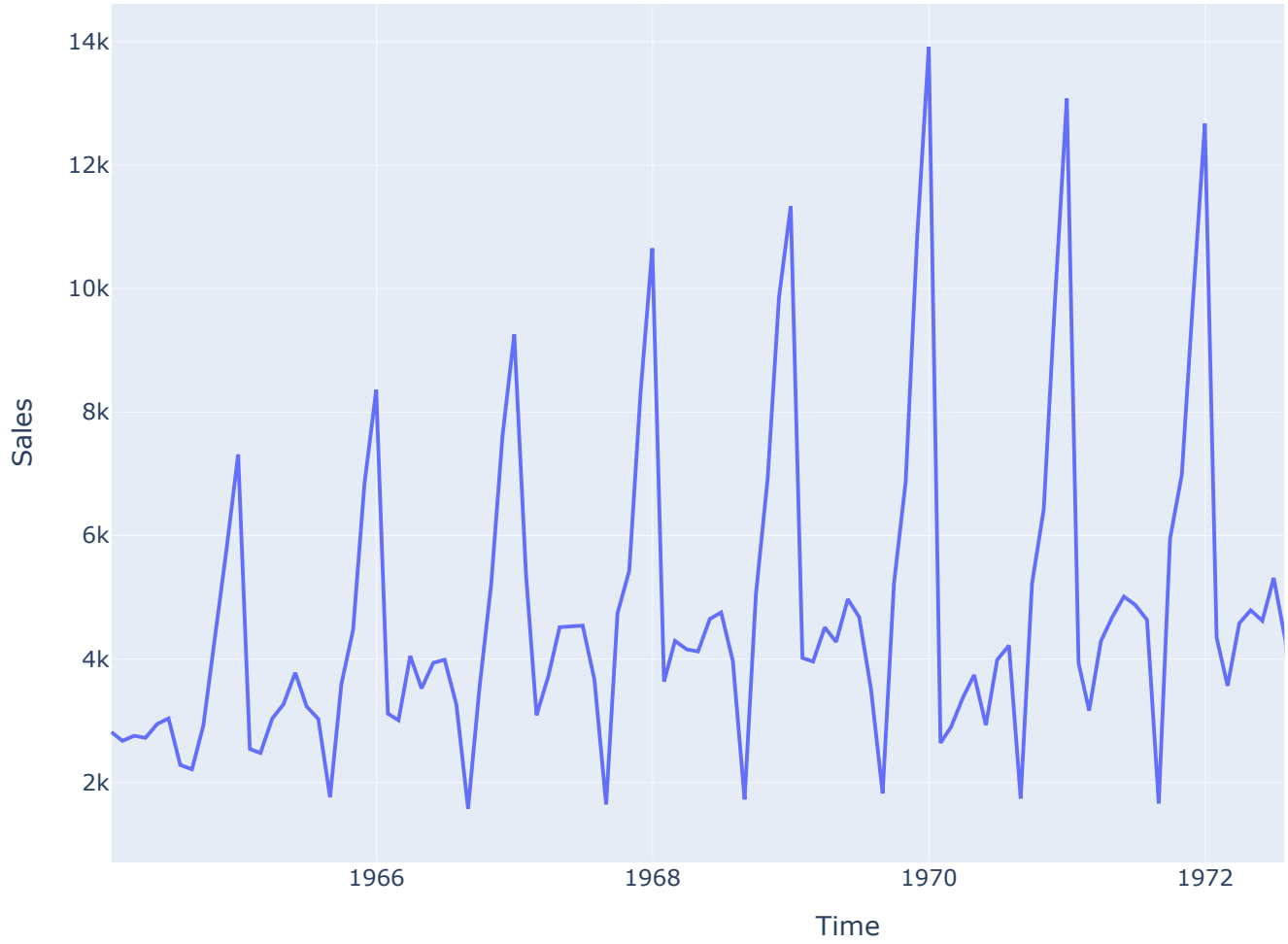
B. Naive

C. Adjusted naive

D. Seasonal naive

E. Drift

```
naive = pd.DataFrame({"Sales": df["Sales"].iloc[-1],
                      "Label": "Naive forecast"},
                     index=forecast_index)

fig = px.line(pd.concat((df, naive)), y="Sales", width=950, color="Label")

fig.update_layout(showlegend=False)
```



Question 4

A. Average

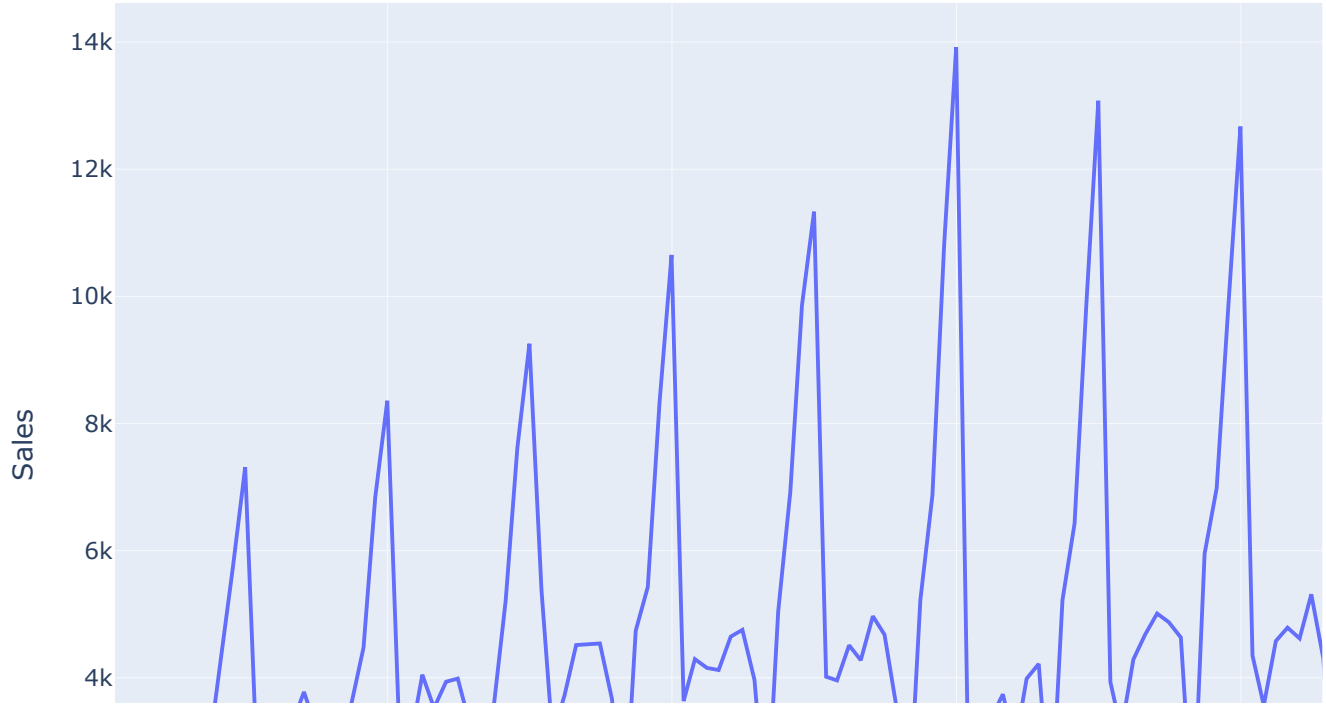
B. Naive

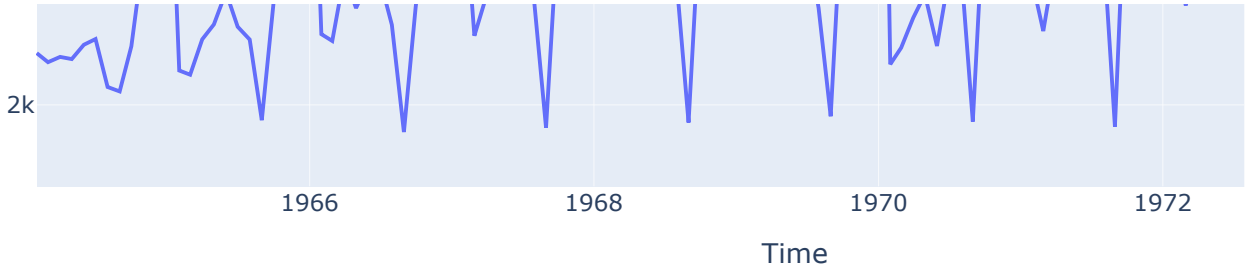
C. Adjusted naive

D. Seasonal naive

E. Drift

```
df["month"] = df.index.month
last_season = (df.drop_duplicates("month", keep="last")
                 .sort_values(by="month")
                 .set_index("month")["Sales"]
               )
df = df.drop(columns="month")
last_season

seasonal = pd.DataFrame(index=forecast_index)
seasonal = seasonal.assign(**{"Sales": seasonal.index.month.map(lambda x: last_sea
                              "Label": "Seasonal naive forecast"}
                           )

fig = px.line(pd.concat((df, seasonal)), y="Sales", width=950, color="Label")

fig.update_layout(showlegend=False)
```





3. Exponential Models

- The above simple methods take one of two broad approaches:
  1. Forecasts based on last observed value (Naive/Seasonally-adjusted Naive/Seasonal Naive)
  2. Forecasts based on all observed values (Average/Drift)
- Is there something in between these two?
- Can we produce a forecast using multiple past data points but that weights recent data more than historical data?
- The most common way to do this is with exponential smoothing models

3.1. Simple Exponential Smoothing

In SES, our forecast is an exponentially weighted average of past values:

$$\hat{y}_{t+1} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \dots$$

- $0 \leq \alpha \leq 1$ is the smoothing parameter
- $\hat{y}$ refers to a forecasted value.

We can re-write that in the recursive form:

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}$$

The table below shows the weights attached to observations for four different values of α when forecasting using simple exponential smoothing

	α =0.2	α =0.4	α =0.6	α =0.8
y_t	0.2	0.4	0.6	0.8
y_{t-1}	0.16	0.24	0.24	0.16
y_{t-2}	0.128	0.144	0.096	0.032
y_{t-3}	0.1024	0.0864	0.0384	0.0064
y_{t-4}	0.0819	0.0518	0.0154	0.0013
y_{t-5}	0.0655	0.0311	0.0061	0.0003

α controls the flexibility of the level

- As α approaches 1, more weight is given to recent observations (model learns fast)
- As α approaches 0, more weight is given to observations from the more distant past

Notice that for the very start of the series at $t = 1$ the last term in the equation is $\hat{y}_{1|0}$. There's no such thing as y_0 but we need to start our model somewhere so we usually call $\hat{y}_{1|0} = \ell_0$ the "initial level" and it's a model parameter. How do we choose it? We have three options:

1. Fixing it to a value: e.g., $\ell_0 = y_1$
2. Using a heuristic: the "heuristic" method in `statsmodels` computes a linear trend on the first ten observations of your series and sets ℓ_0 as the intercept of the trend as per Hyndman et al., 2008)
3. Learn it: by optimizing to minimise the sum of squares

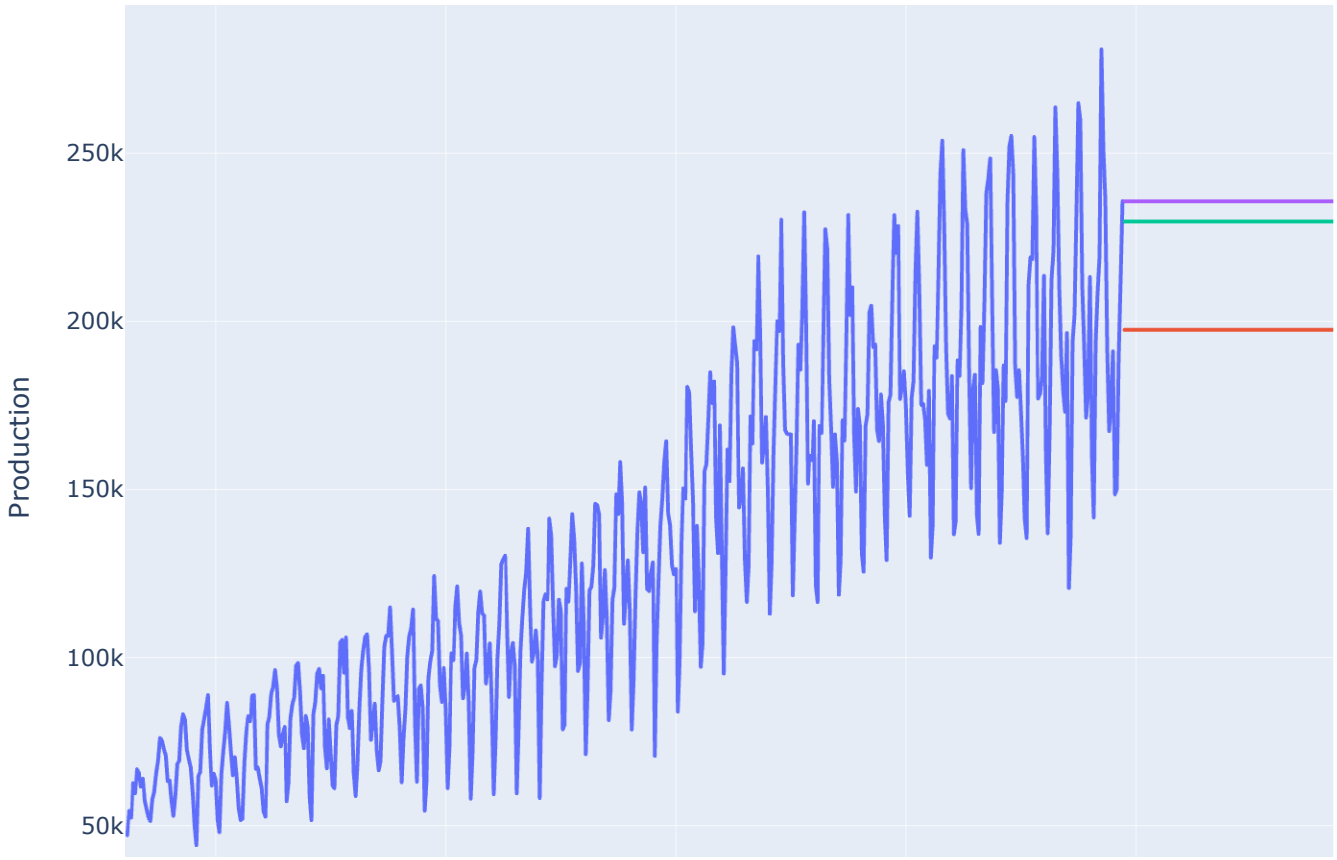
The initial level is actually not that important unless your time series is very short or α is close to 0.

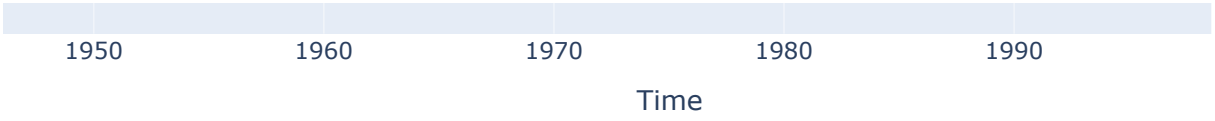
- Okay, let's implement an SES model using the `statsmodels` function `SimpleExpSmoothing()`
- SES is suitable for modelling data without a trend or seasonality, let's try it out on the beer production dataset. We'll use three values: $\alpha = 0.1$, $\alpha = 0.8$, and an optimized α:

```
# Load data
df = get_beer()
forecast_index = create_forecast_index(df.index[-1], 175)
```

```
import warnings
warnings.filterwarnings("ignore")

# Instaniate model
SES = SimpleExpSmoothing(df["Production"], initialization_method="heuristic")

# Fit model
model = SES.fit(smoothing_level=0.1, optimized=False)
ses02 = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                      "Label": "SES: alpha = 0.1"},
                     index=forecast_index)

model = SES.fit(smoothing_level=0.8, optimized=False)
ses08 = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                      "Label": "SES: alpha = 0.8"},
                     index=forecast_index)

# Optimized alpha
SES = SimpleExpSmoothing(df["Production"], initialization_method="estimated")
model = SES.fit(optimized=True)
ses = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                    "Label": "SES: optimized"},
                   index=forecast_index)
```

```
px.line(pd.concat((df, ses02, ses08, ses)), y="Production", color="Label", width=9
```





Stop & think

- Why with a higher α, we get something closer to a Naive forecast?
- Why are the forecasts flat?
- What are these forecasts missing?

For a time series of length T the first forecast at $T+1$ is:

$$\hat{y}_{T+1|T} = \alpha y_T + (1-\alpha)\hat{y}_{T|T-1}$$

Our next forecast at $T+2$ would then be:

$$\begin{aligned}\hat{y}_{T+2|T+1} &= \alpha y_{T+1} + (1-\alpha)\hat{y}_{T+1|T} \\ &= \alpha \hat{y}_{T+1|T} + (1-\alpha)\hat{y}_{T+1|T} \\ &= \hat{y}_{T+1|T}\end{aligned}$$

And so on and so forth. Hence, the forecasts are flat.

What if we want to capture information about the trend and/or seasonality?

3.2. Holt's method

- Holt (1957) extends SES to allow forecasting of data with trends
- It is similar to SES but now includes a way to smooth the "level" (the magnitude of the series) and the "trend":

$$\hat{y}_{t+h|t} = \ell_t + hb_t$$

$$\ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1}$$

- We now have two key parameters α (to control smoothness of the level) and β (control smoothness of trend).

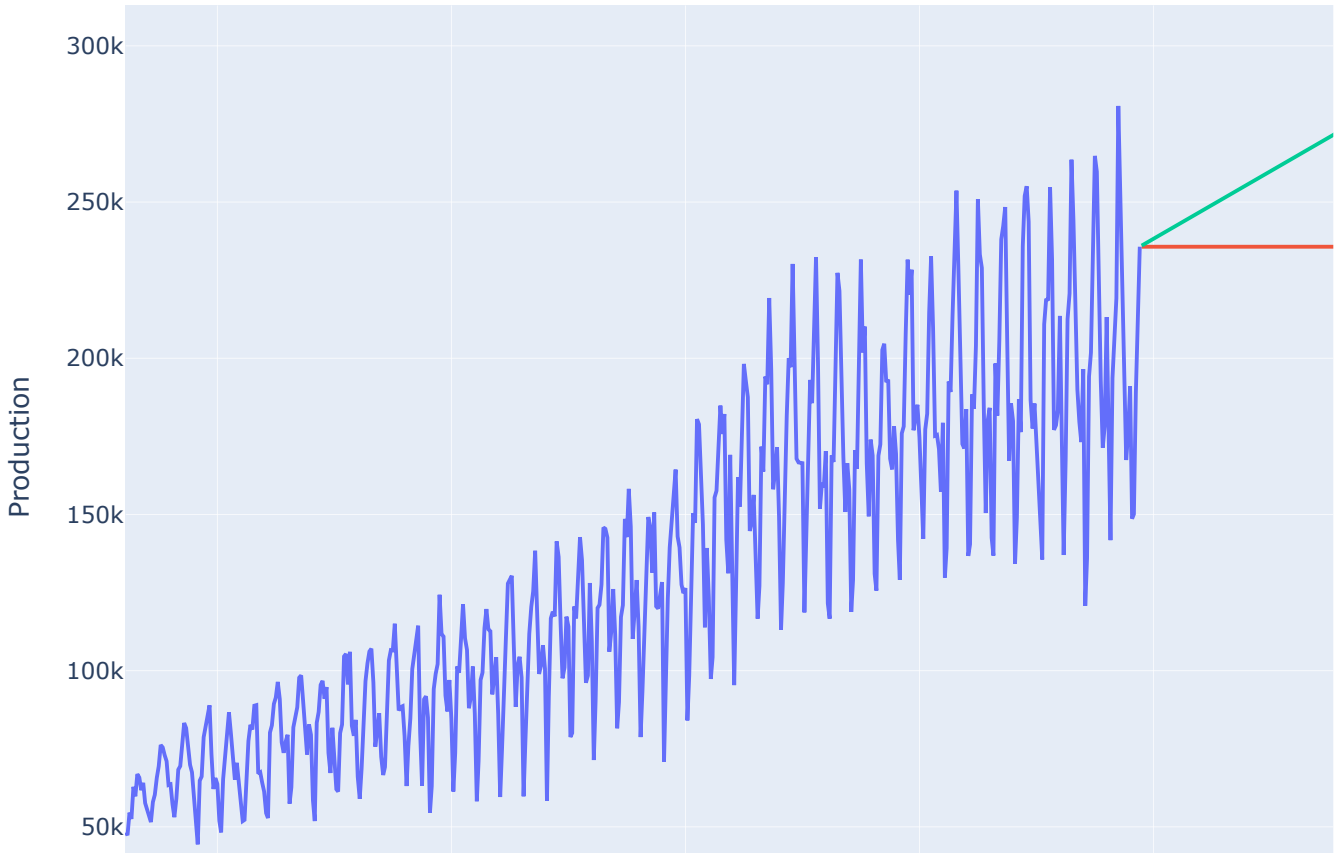
β controls the flexibility of the trend

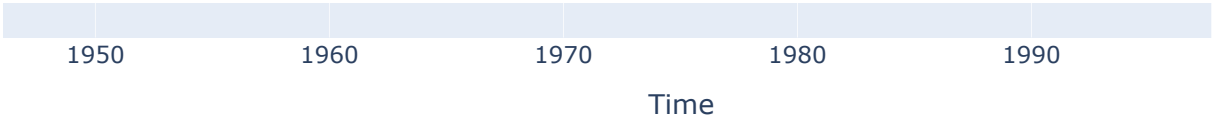
- As β approaches 0, the trend becomes more linear
- As β approaches 1, the trend changes with every observation

- All we're doing here is forecasting the next values as an exponentially weighted average of past values and past trend
- How do we choose α and β? Again, we minimize SSE
- Let's fit a model using the `statsmodels` function `Holt()`:

```
model = Holt(df["Production"], initialization_method="estimated").fit(method='leas

holt = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                     "Label": "Holt"},
                    index=forecast_index)
```

```
px.line(pd.concat((df, ses, holt)), y="Production", color="Label", width=950)
```





- Our forecasts now have a trend as expected.

3.3. Holt-Winter's method

- Holt (1957) and Winters (1960) extended Holt's method to capture seasonality (in addition to trend)
- The method is similar to before but now includes a way to smooth the "level", "trend" and "seasonality" of the series
- There's a version for additive seasonality:

$$\hat{y}_{t+h|t} = \ell_t + hb_t + s_{t+h-m(k+1)}$$

$$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1}$$

$$s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$$

- Or multiplicative seasonality:

$$\hat{y}_{t+h|t} = (\ell_t + hb_t)s_{t+h-m(k+1)}$$

$$\ell_t = \alpha\frac{y_t}{s_{t-m}} + (1-\alpha)(\ell_{t-1} + b_{t-1})$$

$$b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1}$$

$$s_t = \gamma\frac{y_t}{\ell_{t-1} + b_{t-1}} + (1-\gamma)s_{t-m}$$

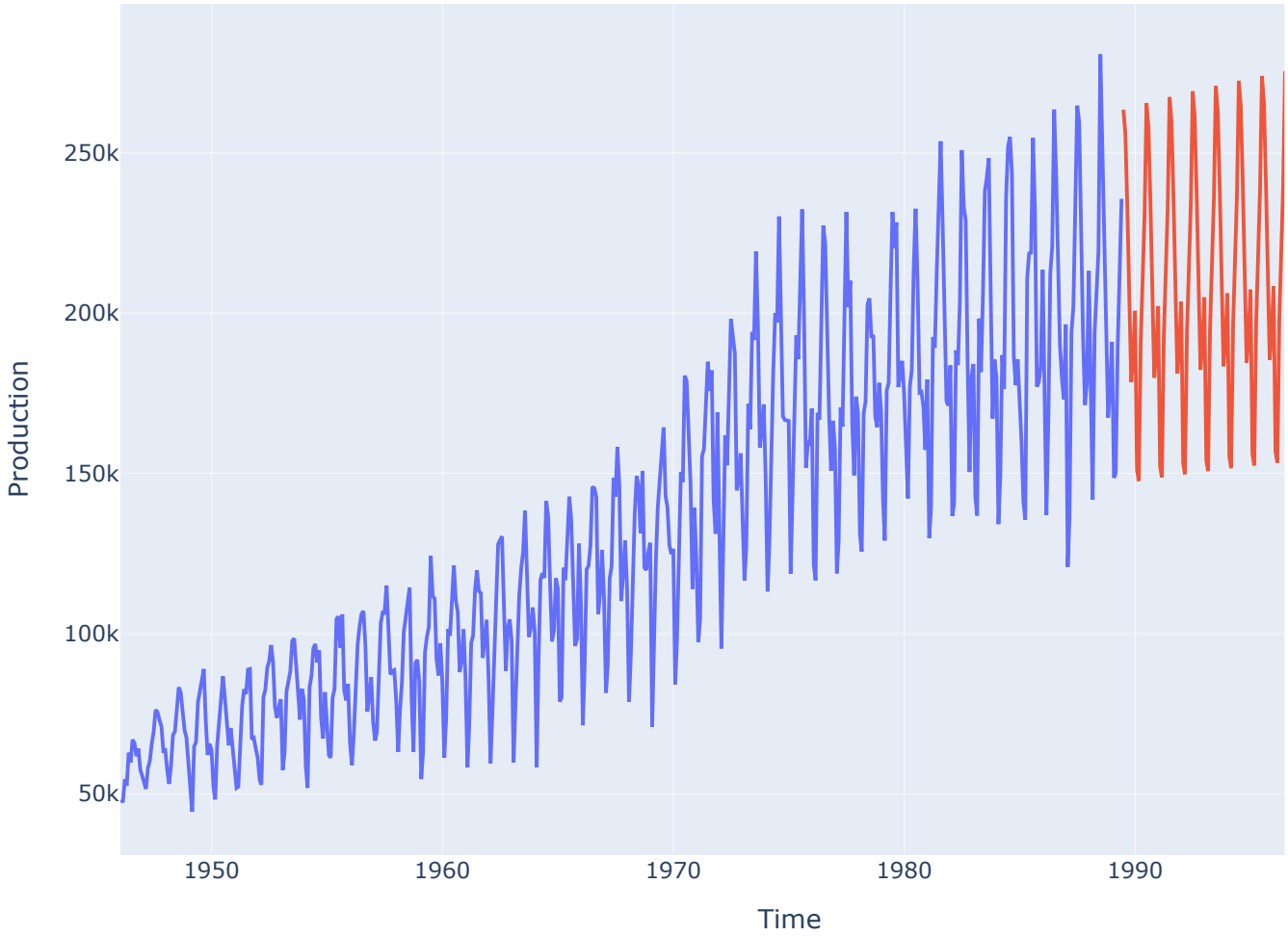
- Once again, the equations are fairly intuitive, even though it looks messy, it's just forecasting based on historically weighted level, trend, and seasonality. You can read more about it here.

γ controls the flexibility of the seasonality

- As γ approaches 0, the seasonality is fixed (e.g., seasonal mean)
- As γ approaches 1, the seasonality updates completely (e.g., seasonal naive)
- Let's fit a model using the `statsmodels` function `ExponentialSmoothing()` (I'll talk more about the argument I'm using in the model after the code)

```
model = ExponentialSmoothing(df["Production"], trend="add", seasonal="mul", season
                             damped_trend = True, # damped trend option
                             initialization_method="estimated")
model = model.fit(method="least_squares")
holtwin = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                        "Label": "Holt-Winters"},
                       index=forecast_index)
```

```
px.line(pd.concat((df, holtwin)), y="Production", color="Label", width=950)
```



Damped?

- Exponential smoothing methods are not restricted to those we have presented so far, there are actually nine combinations of the trend and seasonal components!
- In practice, estimating the trend in exponential models sometimes leads to over-forecasts. So there's also a version where we damp the trend over time:

Trend Component	Seasonal Component
None `(N)`	None `(N)`
Additive `(A)`	Additive `(A)`
Additive damped `(Ad)`	Multiplicative `(M)`

- We denote these models with a tuple `(trend, seasonal)`
- We've seen some specific combinations already:

Notation	Method
`(N,N)`	Simple exponential smoothing
`(A,N)`	Holt's method
`(A,A)`	Additive Holt-Winter's method

- The `ExponentialSmoothing()` is really a generic class that gives us access to all these types, as we saw in the example above, we can modify these components as we please.

> Note: You can also consider multiplicative/multiplicative damped trend, but we don't consider them in this course as they don't typically produce good forecasts

3.4. ETS models

- The methods presented above are algorithms which generate point forecasts. But often we'll want to also generate prediction intervals
- The generalisation of exponential smoothing algorithms to statistical models that model distributions, include an error term, and can generate prediction intervals are known as **ETS**

models (**E**rror, **T**rend, **S**easonal), where:

  - E = {additive, multiplicative}
  - T = {none, additive, additive damped}
  - S = {none, additive, multiplicative}

- You can read more about ETS models and their derivation here and in Appendix B but their derivation is not really important to know and beyond the scope of this course. What you need to know is that an ETS model will usually give the same/simlar results to the algorithms above, but with the added bonus of being able to generate prediction intervals
- In Rob Hyndman's own words "the results from ETS are usually more reliable (than the algorithmic exponential models)"
- Rather than optimizing based on minimzing the SSE, ETS models optimize by maximizing the likelihood (we assume errors are normally distributed). See Appendix B for a more detailed derviation of ETS Models

```
model = ETSModel(df["Production"], error="add", trend="add").fit()
ets_aan = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                        "Label": "ETS (AAN)"},
                       index=forecast_index)

model = ETSModel(df["Production"], error="add", trend="add", seasonal="mul", seasc
ets_aam = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                        "Label": "ETS (AAM)"},
                       index=forecast_index)
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            4     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.16106D+01    |proj g|=  9.99000D-02

At iterate    1    f=  1.15953D+01    |proj g|=  7.31937D-02

At iterate    2    f=  1.15927D+01    |proj g|=  4.01890D-02

At iterate    3    f=  1.15927D+01    |proj g|=  2.06777D-02

At iterate    4    f=  1.15927D+01    |proj g|=  7.91545D-04

At iterate    5    f=  1.15927D+01    |proj g|=  4.02167D-04

At iterate    6    f=  1.15927D+01    |proj g|=  3.48876D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    4      6      9      8     0     0   3.489D-04   1.159D+01
  F =   11.592691029283568

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =           16     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.11646D+01    |proj g|=  9.88789D-01

At iterate    1    f=  1.06913D+01    |proj g|=  2.60184D-01

At iterate    2    f=  1.06869D+01    |proj g|=  1.95657D-01

At iterate    3    f=  1.06757D+01    |proj g|=  1.95400D-01
```

```
At iterate    4    f=  1.06613D+01    |proj g|=  1.84737D-01

At iterate    5    f=  1.06428D+01    |proj g|=  1.68173D-01

At iterate    6    f=  1.06228D+01    |proj g|=  1.22400D-01

At iterate    7    f=  1.06183D+01    |proj g|=  1.21381D-01

At iterate    8    f=  1.06155D+01    |proj g|=  1.08645D-01

At iterate    9    f=  1.06130D+01    |proj g|=  8.24247D-02

At iterate   10    f=  1.06119D+01    |proj g|=  2.21643D-02

At iterate   11    f=  1.06117D+01    |proj g|=  8.00089D-03

At iterate   12    f=  1.06117D+01    |proj g|=  7.27045D-03

At iterate   13    f=  1.06116D+01    |proj g|=  8.10552D-03

At iterate   14    f=  1.06115D+01    |proj g|=  1.34476D-02

At iterate   15    f=  1.06114D+01    |proj g|=  2.10907D-02

At iterate   16    f=  1.06113D+01    |proj g|=  2.02388D-02

At iterate   17    f=  1.06111D+01    |proj g|=  1.23643D-02

At iterate   18    f=  1.06111D+01    |proj g|=  2.09894D-03

At iterate   19    f=  1.06111D+01    |proj g|=  4.30358D-03

At iterate   20    f=  1.06111D+01    |proj g|=  1.23848D-03

At iterate   21    f=  1.06111D+01    |proj g|=  7.25997D-04

At iterate   22    f=  1.06111D+01    |proj g|=  4.79083D-04

At iterate   23    f=  1.06111D+01    |proj g|=  5.57421D-04

At iterate   24    f=  1.06111D+01    |proj g|=  4.69136D-04

At iterate   25    f=  1.06111D+01    |proj g|=  1.66622D-04

At iterate   26    f=  1.06111D+01    |proj g|=  1.82965D-05

At iterate   27    f=  1.06111D+01    |proj g|=  1.79412D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value
```

```
           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
   16     27     34     33     0     0   1.794D-05   1.061D+01
  F =   10.611092054644224

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

```
model = ETSModel(df["Production"], error="add", trend="add"
#                 ,bounds={"smoothing_trend": (0.0412, 1)}
                ).fit()
ets_aan = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                        "Label": "ETS (AAN)"},
                       index=forecast_index)
model.summary()
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            4     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.16106D+01    |proj g|=  9.99000D-02

At iterate    1    f=  1.15953D+01    |proj g|=  7.31937D-02

At iterate    2    f=  1.15927D+01    |proj g|=  4.01890D-02

At iterate    3    f=  1.15927D+01    |proj g|=  2.06777D-02

At iterate    4    f=  1.15927D+01    |proj g|=  7.91545D-04

At iterate    5    f=  1.15927D+01    |proj g|=  4.02167D-04

At iterate    6    f=  1.15927D+01    |proj g|=  3.48876D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    4      6      9      8     0     0   3.489D-04   1.159D+01
  F =   11.592691029283568

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

ETS Results

Dep. Variable:	Production	**No. Observations:**	521
Model:	ETS(AAN)	**Log Likelihood**	-6039.792
Date:	Wed, 08 Jan 2025	**AIC**	12089.584
Time:	09:18:01	**BIC**	12110.863
Sample:	01-31-1946	**HQIC**	12097.919
	- 05-31-1989	**Scale**	686766557.205
Covariance Type:	approx		

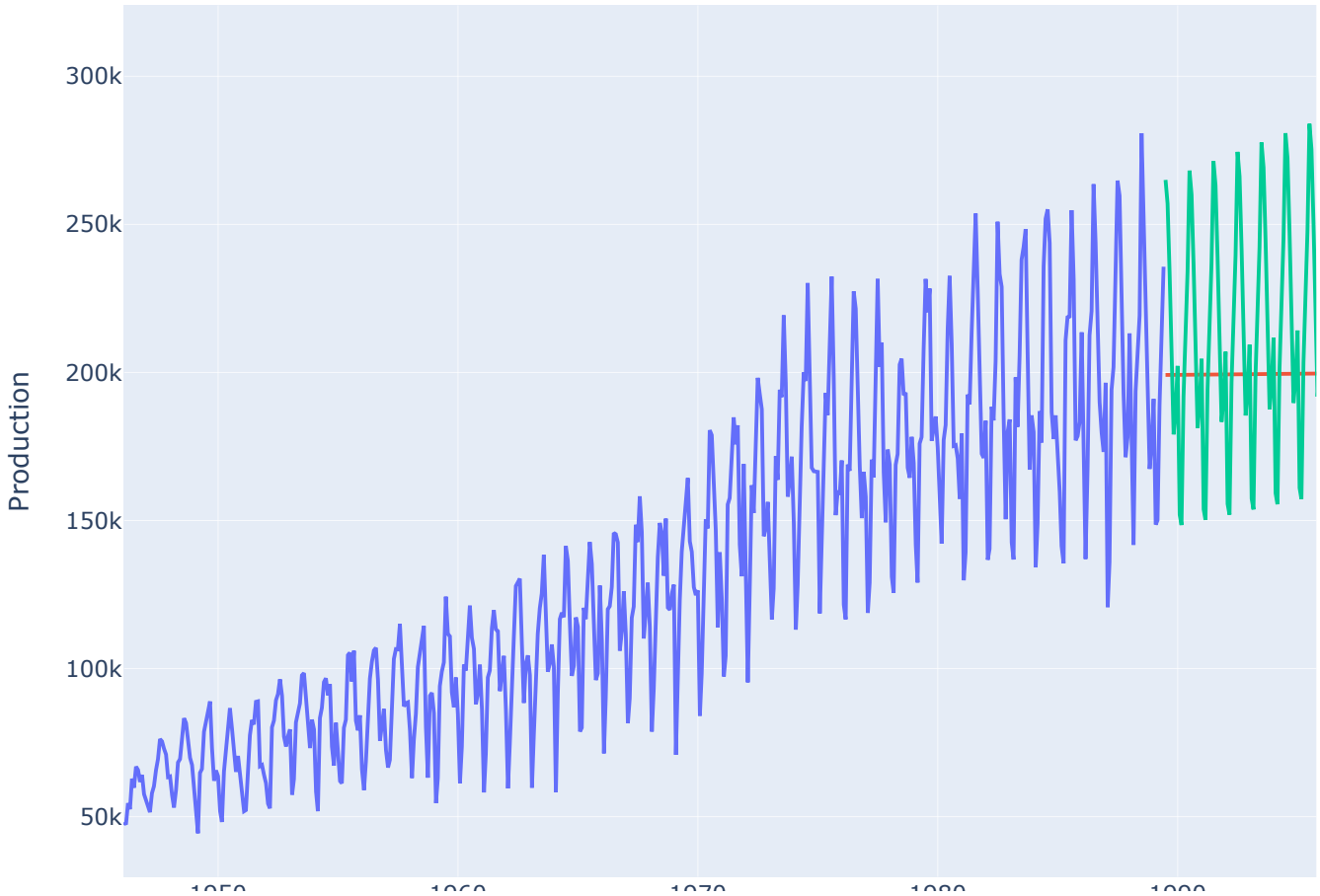
	coef	std err	z	P>|z|	[0.025	0.975]
smoothing_level	0.0589	0.018	3.337	0.001	0.024	0.094
smoothing_trend	0.0039	nan	nan	nan	nan	nan
initial_level	4.703e+04	9828.031	4.785	0.000	2.78e+04	6.63e+04
initial_trend	2062.2545	nan	nan	nan	nan	nan

Ljung-Box (Q):	234.78	**Jarque-Bera (JB):**	1.50
Prob(Q):	0.00	**Prob(JB):**	0.47
Heteroskedasticity (H):	5.20	**Skew:**	0.07
Prob(H) (two-sided):	0.00	**Kurtosis:**	3.22

Warnings:

[1] Covariance matrix calculated using numerical (complex-step) differentiation.

```
px.line(pd.concat((df, ets_aan, ets_aam)), y="Production", color="Label", width=95
```

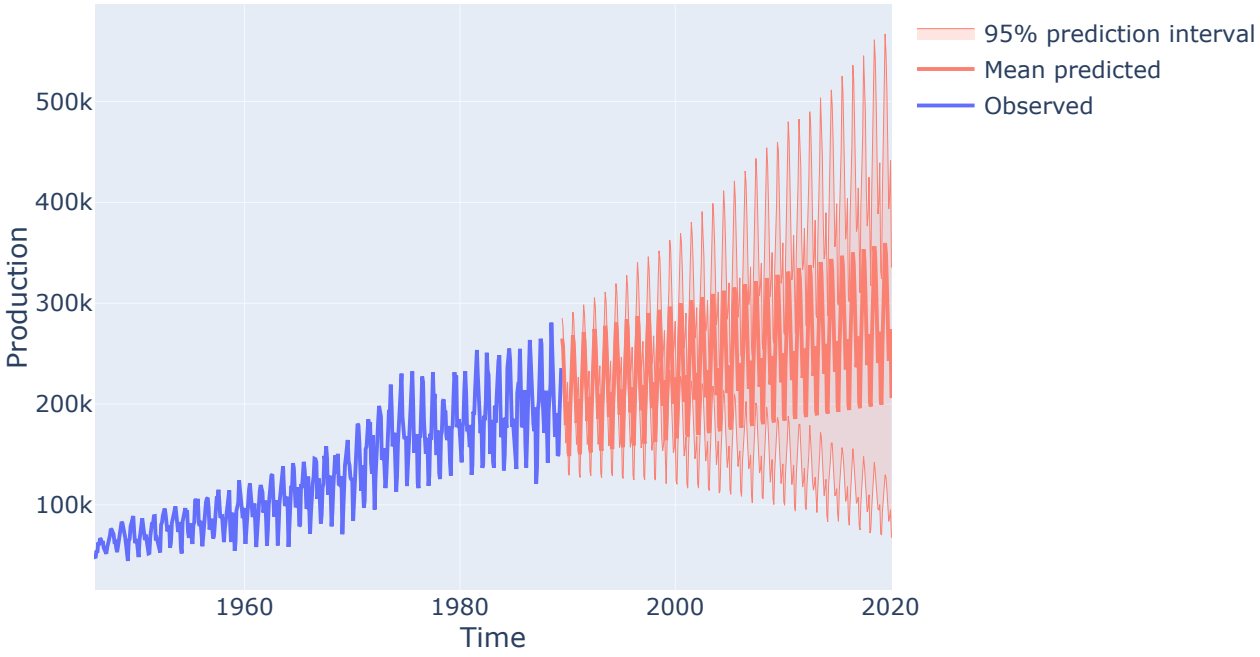


- As mentioned above, we can generate prediction intervals with ETS models
- There are two main ways to do this:
  1. `model.get_prediction()` (analytical)
  2. `model.simulate()`
- We'll talk more about these and how to evaluate intervals in a later lecture, but as a teaser:

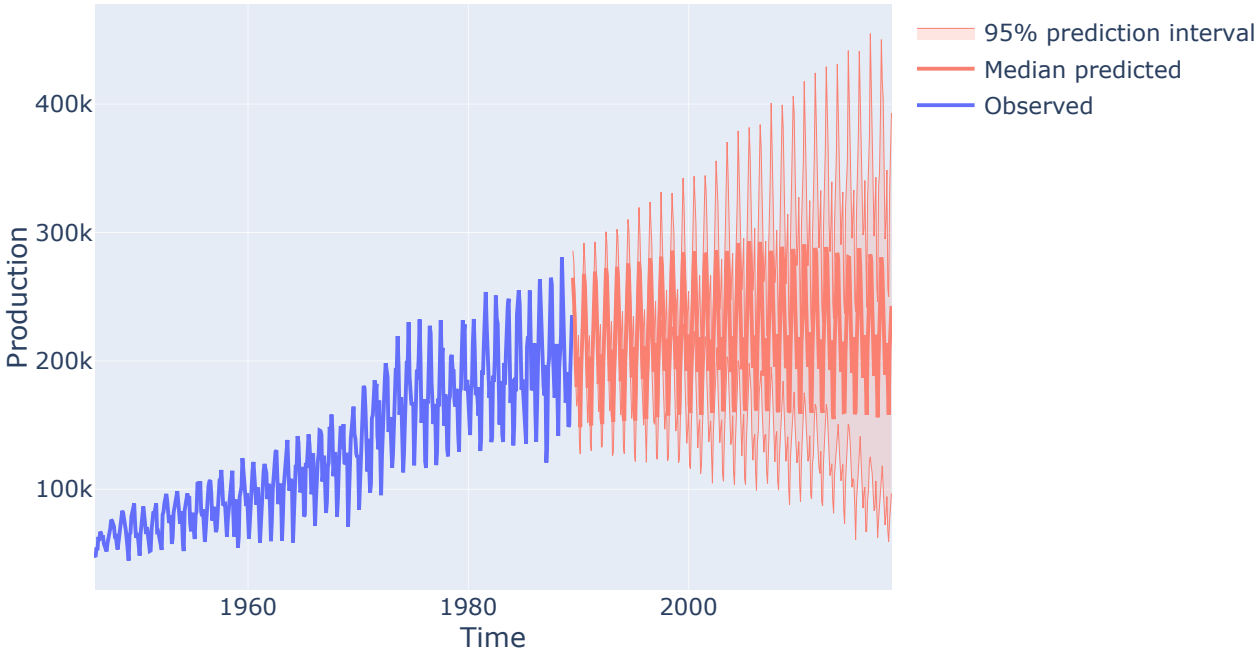
```
model = ETSModel(df["Production"], error="add", trend="add", seasonal="mul", seasc
predictions = model.get_prediction(start=df.index[-1] + pd.DateOffset(months=1), e
plot_prediction_intervals(df["Production"], predictions)
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =           16     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.11646D+01    |proj g|=  9.88789D-01

At iterate    1    f=  1.06913D+01    |proj g|=  2.60184D-01

At iterate    2    f=  1.06869D+01    |proj g|=  1.95657D-01

At iterate    3    f=  1.06757D+01    |proj g|=  1.95400D-01

At iterate    4    f=  1.06613D+01    |proj g|=  1.84737D-01

At iterate    5    f=  1.06428D+01    |proj g|=  1.68173D-01

At iterate    6    f=  1.06228D+01    |proj g|=  1.22400D-01

At iterate    7    f=  1.06183D+01    |proj g|=  1.21381D-01

At iterate    8    f=  1.06155D+01    |proj g|=  1.08645D-01

At iterate    9    f=  1.06130D+01    |proj g|=  8.24247D-02

At iterate   10    f=  1.06119D+01    |proj g|=  2.21643D-02

At iterate   11    f=  1.06117D+01    |proj g|=  8.00089D-03

At iterate   12    f=  1.06117D+01    |proj g|=  7.27045D-03

At iterate   13    f=  1.06116D+01    |proj g|=  8.10552D-03

At iterate   14    f=  1.06115D+01    |proj g|=  1.34476D-02

At iterate   15    f=  1.06114D+01    |proj g|=  2.10907D-02

At iterate   16    f=  1.06113D+01    |proj g|=  2.02388D-02

At iterate   17    f=  1.06111D+01    |proj g|=  1.23643D-02

At iterate   18    f=  1.06111D+01    |proj g|=  2.09894D-03

At iterate   19    f=  1.06111D+01    |proj g|=  4.30358D-03

At iterate   20    f=  1.06111D+01    |proj g|=  1.23848D-03

At iterate   21    f=  1.06111D+01    |proj g|=  7.25997D-04

At iterate   22    f=  1.06111D+01    |proj g|=  4.79083D-04

At iterate   23    f=  1.06111D+01    |proj g|=  5.57421D-04
```

```
At iterate   24    f=  1.06111D+01    |proj g|=  4.69136D-04

At iterate   25    f=  1.06111D+01    |proj g|=  1.66622D-04

At iterate   26    f=  1.06111D+01    |proj g|=  1.82965D-05

At iterate   27    f=  1.06111D+01    |proj g|=  1.79412D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
   16     27     34     33     0     0   1.794D-05   1.061D+01
  F =   10.611092054644224

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```



```
q = 0.975  # 95% prediction intervals
simulations = model.simulate(anchor="end", nsimulations=348, repetitions=100, rand
simulations = pd.DataFrame({"median": simulations.median(axis=1),
                            "pi_lower": simulations.quantile((1 - q), axis=1),
                            "pi_upper": simulations.quantile(q, axis=1)},
                           index=simulations.index)
plot_prediction_intervals(df["Production"], simulations, "median")
```



4. Selecting a model

4.1. In-sample methods

4.1.1. Metrics

- We can use all the same metrics we've seen before to determine how well a given model fits our data. The most common metrics are:
  - AIC
  - BIC
  - SSE/MSE/RMSE
- These are provided for most models in `statsmodels`:

```
model_1 = ETSModel(df["Production"], error="add", trend="add").fit()
model_1.summary()
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            4     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.16106D+01    |proj g|=  9.99000D-02

At iterate    1    f=  1.15953D+01    |proj g|=  7.31937D-02

At iterate    2    f=  1.15927D+01    |proj g|=  4.01890D-02

At iterate    3    f=  1.15927D+01    |proj g|=  2.06777D-02

At iterate    4    f=  1.15927D+01    |proj g|=  7.91545D-04

At iterate    5    f=  1.15927D+01    |proj g|=  4.02167D-04

At iterate    6    f=  1.15927D+01    |proj g|=  3.48876D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    4      6      9      8     0     0   3.489D-04   1.159D+01
  F =   11.592691029283568

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

ETS Results

Dep. Variable:	Production	**No. Observations:**	521
Model:	ETS(AAN)	**Log Likelihood**	-6039.792
Date:	Wed, 08 Jan 2025	**AIC**	12089.584
Time:	09:18:02	**BIC**	12110.863
Sample:	01-31-1946	**HQIC**	12097.919
	- 05-31-1989	**Scale**	686766557.205
Covariance Type:	approx		

	coef	std err	z	P>|z|	[0.025	0.975]
smoothing_level	0.0589	0.018	3.337	0.001	0.024	0.094
smoothing_trend	0.0039	nan	nan	nan	nan	nan
initial_level	4.703e+04	9828.031	4.785	0.000	2.78e+04	6.63e+04
initial_trend	2062.2545	nan	nan	nan	nan	nan

Ljung-Box (Q):	234.78	**Jarque-Bera (JB):**	1.50
Prob(Q):	0.00	**Prob(JB):**	0.47
Heteroskedasticity (H):	5.20	**Skew:**	0.07
Prob(H) (two-sided):	0.00	**Kurtosis:**	3.22

Warnings:

[1] Covariance matrix calculated using numerical (complex-step) differentiation.

- And we can extract relevant metrics ourselves to compare different models:

```
model_1 = ETSModel(df["Production"], error="add", trend="add", seasonal=None).fit(
model_2 = ETSModel(df["Production"], error="add", trend="add", seasonal="add", sea
model_3 = ETSModel(df["Production"], error="add", trend="add", damped_trend=True,

pd.DataFrame({"AIC": [model_1.aic, model_2.aic, model_3.aic],
              "BIC": [model_1.bic, model_2.bic, model_3.bic],
              "MSE": [model_1.mse, model_2.mse, model_3.mse]},
             index=["(AAN)", "(AAA)", "(AAdM)"]).round(2).style.apply(highlight_mi
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            4     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.16106D+01    |proj g|=  9.99000D-02

At iterate    1    f=  1.15953D+01    |proj g|=  7.31937D-02

At iterate    2    f=  1.15927D+01    |proj g|=  4.01890D-02

At iterate    3    f=  1.15927D+01    |proj g|=  2.06777D-02

At iterate    4    f=  1.15927D+01    |proj g|=  7.91545D-04

At iterate    5    f=  1.15927D+01    |proj g|=  4.02167D-04

At iterate    6    f=  1.15927D+01    |proj g|=  3.48876D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    4      6      9      8     0     0   3.489D-04   1.159D+01
  F =   11.592691029283568

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =           16     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.12940D+01    |proj g|=  9.88789D-01

At iterate    1    f=  1.08219D+01    |proj g|=  9.99800D-01

At iterate    2    f=  1.07448D+01    |proj g|=  2.70587D-01

At iterate    3    f=  1.07425D+01    |proj g|=  2.47881D-01
```

```
At iterate    4    f=  1.07328D+01    |proj g|=  2.25321D-01

At iterate    5    f=  1.07119D+01    |proj g|=  1.83298D-01

At iterate    6    f=  1.06860D+01    |proj g|=  9.99800D-01

At iterate    7    f=  1.06757D+01    |proj g|=  2.13323D-01

At iterate    8    f=  1.06699D+01    |proj g|=  2.41669D-01

At iterate    9    f=  1.06604D+01    |proj g|=  2.13520D-01

At iterate   10    f=  1.06487D+01    |proj g|=  9.18894D-01

At iterate   11    f=  1.06444D+01    |proj g|=  8.31979D-01

At iterate   12    f=  1.06429D+01    |proj g|=  3.39755D-01

At iterate   13    f=  1.06422D+01    |proj g|=  2.68396D-01

At iterate   14    f=  1.06417D+01    |proj g|=  1.12358D-01

At iterate   15    f=  1.06413D+01    |proj g|=  6.71056D-02

At iterate   16    f=  1.06411D+01    |proj g|=  2.60291D-02

At iterate   17    f=  1.06411D+01    |proj g|=  6.27107D-03

At iterate   18    f=  1.06411D+01    |proj g|=  5.20295D-04

At iterate   19    f=  1.06411D+01    |proj g|=  2.11386D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
   16     19     26     24     0     0   2.114D-05   1.064D+01
  F =   10.641121023447468

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =           17     M =           10

At X0         1 variables are exactly at the bounds
```

```
At iterate    0    f=  1.11626D+01    |proj g|=  9.88789D-01

At iterate    1    f=  1.07031D+01    |proj g|=  7.24556D-01

At iterate    2    f=  1.06888D+01    |proj g|=  1.91353D-01

At iterate    3    f=  1.06846D+01    |proj g|=  1.93565D-01

At iterate    4    f=  1.06604D+01    |proj g|=  1.79238D-01

At iterate    5    f=  1.06439D+01    |proj g|=  1.53581D-01

At iterate    6    f=  1.06341D+01    |proj g|=  1.60132D-01

At iterate    7    f=  1.06335D+01    |proj g|=  1.53684D-01

At iterate    8    f=  1.06318D+01    |proj g|=  1.55455D-01

At iterate    9    f=  1.06238D+01    |proj g|=  1.31462D-01

At iterate   10    f=  1.06190D+01    |proj g|=  2.26798D-01

At iterate   11    f=  1.06150D+01    |proj g|=  1.00266D-01

At iterate   12    f=  1.06148D+01    |proj g|=  9.16076D-02

At iterate   13    f=  1.06137D+01    |proj g|=  1.02190D-02

At iterate   14    f=  1.06136D+01    |proj g|=  1.80366D-02

At iterate   15    f=  1.06136D+01    |proj g|=  1.00808D-02

At iterate   16    f=  1.06134D+01    |proj g|=  1.08063D-02

At iterate   17    f=  1.06133D+01    |proj g|=  2.42551D-02

At iterate   18    f=  1.06132D+01    |proj g|=  2.11847D-02

At iterate   19    f=  1.06130D+01    |proj g|=  2.73900D-02

At iterate   20    f=  1.06129D+01    |proj g|=  2.00314D-02

At iterate   21    f=  1.06126D+01    |proj g|=  1.85414D-02

At iterate   22    f=  1.06125D+01    |proj g|=  1.52340D-02

At iterate   23    f=  1.06124D+01    |proj g|=  1.41316D-02

At iterate   24    f=  1.06123D+01    |proj g|=  1.44398D-02

At iterate   25    f=  1.06122D+01    |proj g|=  4.27267D-03

At iterate   26    f=  1.06122D+01    |proj g|=  1.35731D-03

At iterate   27    f=  1.06122D+01    |proj g|=  1.76446D-03

At iterate   28    f=  1.06122D+01    |proj g|=  1.16422D-03
```

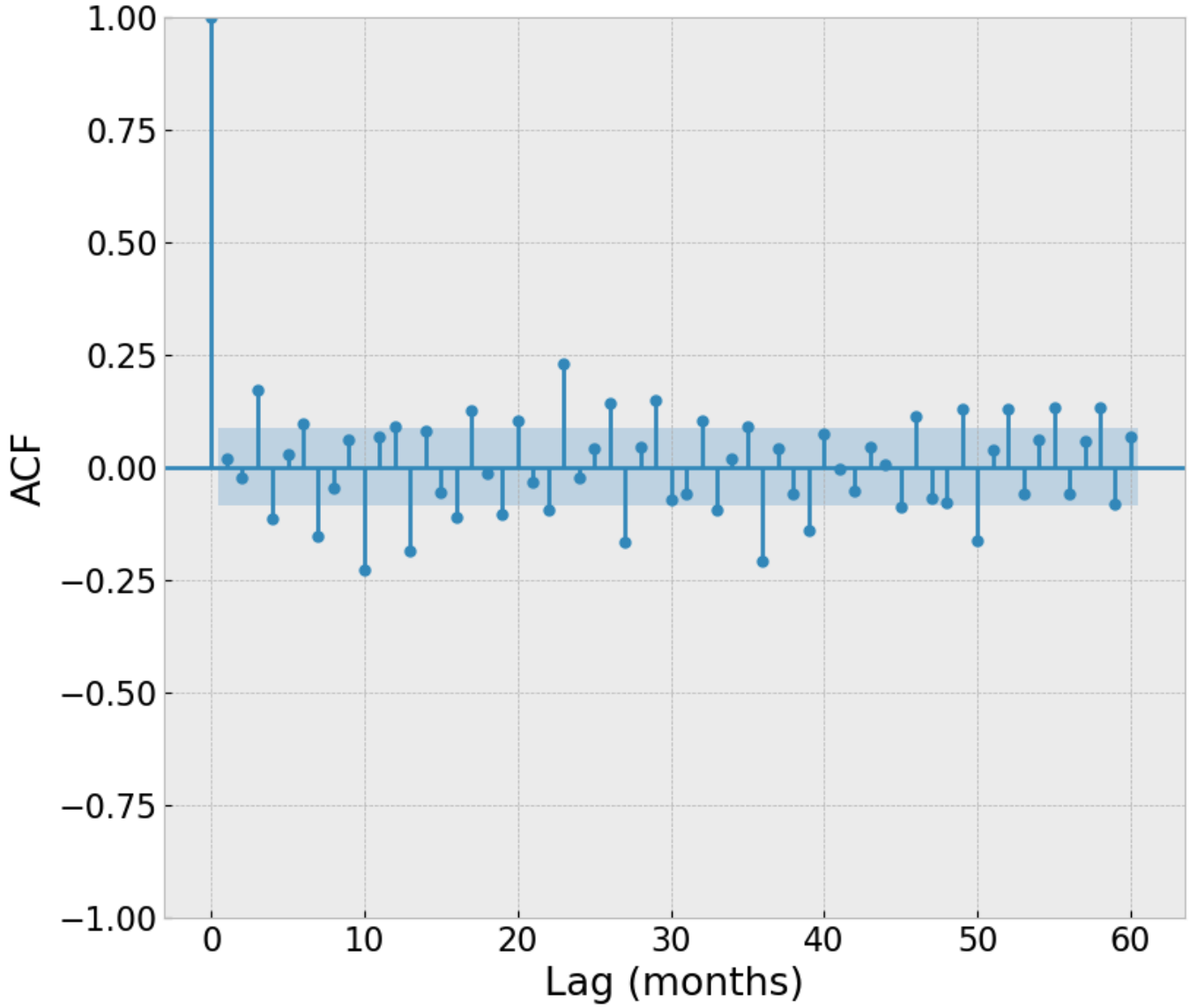
```
At iterate   29    f=  1.06122D+01    |proj g|=  8.11085D-04

At iterate   30    f=  1.06122D+01    |proj g|=  1.98490D-03

At iterate   31    f=  1.06122D+01    |proj g|=  1.69642D-04

At iterate   32    f=  1.06122D+01    |proj g|=  5.91527D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
   17     32     36     36     0     1   5.915D-05   1.061D+01
  F =   10.612233930912643

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

	AIC	BIC	MSE
(AAN)	12089.580000	12110.860000	686766557.210000
(AAA)	11124.050000	11200.650000	102396693.550000
(AAdM)	11095.950000	11176.810000	96648456.690000

4.1.2. Residuals

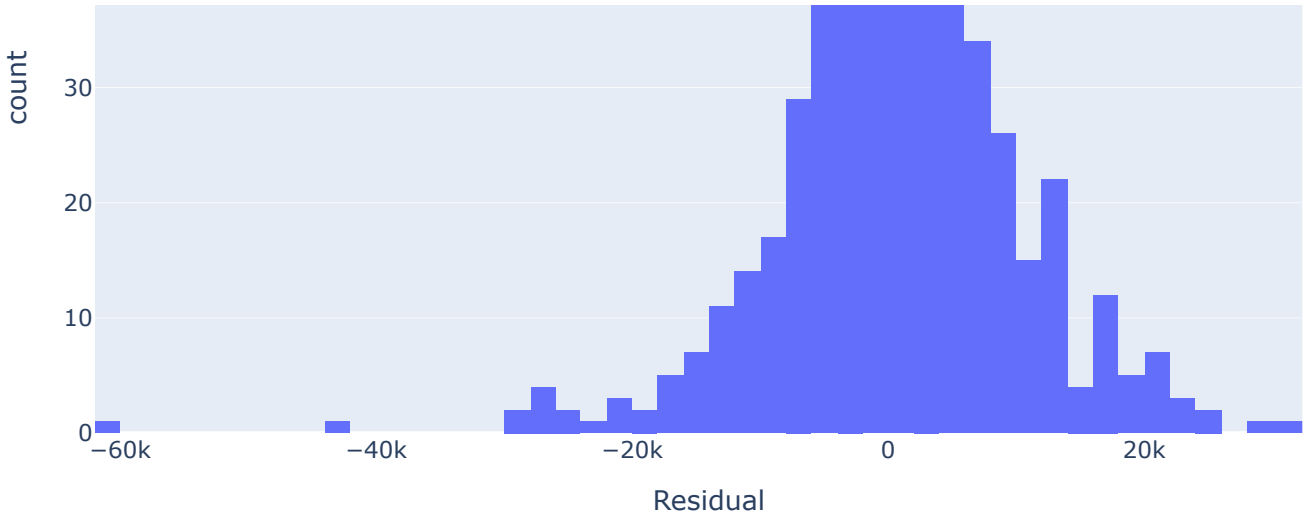
- We can also check the residuals to determine how well our model captures information in our data, by:
    1. Visual inspection (residuals should be uncorrelated, have zero mean, and ideally be normally distributed)
    2. Running diagnostic Portmanteau tests (e.g., Ljung-Box-Pierce test, etc.)
- Let's do a visual inspection first
- I want to see if my residuals are significantly different to white noise:

```
fig = plot_acf(model_3.resid, lags=60, title=None, bartlett_confint=False)
plt.ylabel("ACF")
plt.xlabel("Lag (months)");
```



- It does look like our residuals differ to a white noise series (this is not good because it means there's structure in the data we didn't model well), but it's not too bad
- Ideally our residuals follow a normal distirbution:

```
fig = px.histogram(model_3.resid, height=500)
fig.layout.update(showlegend=False, xaxis=dict(title="Residual"))
fig
```



- There also exists formal tests for determining whether our residuals are correlated or are approximately normal
- The *Ljung–Box test* tests whether a group of autocorrelations is significantly different from white noise (remember, we want the residuals to be white noise, meaning we've extracted all possible information from our series)
- The *Jarque-Bera test* tests whether residuals are significantly different from a normal distribution (based on skewness and kurtosis)
- `statsmodels` provides the test statistics for us:

```
model_3.summary().tables[-1]  # I'm indexing the diagnostic table output by summar
```

Ljung-Box (Q):	162.90	**Jarque-Bera (JB):**	375.16
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	4.64	**Skew:**	-0.66
Prob(H) (two-sided):	0.00	**Kurtosis:**	6.94

- Or we can calculate them explicitly using relevant methods:

```
p = model_3.test_serial_correlation('ljungbox', lags=24)[0, 1, -1]
sig_level = 0.05
print(f"LB test for autocorrelation p-value: {p:.2f}")
if p < sig_level:
    print("Residuals significantly different from white noise. That's bad...")
else:
    print("Residuals not significantly different from white noise. That's good!")
```

```
LB test for autocorrelation p-value: 0.00
Residuals significantly different from white noise. That's bad...
```

```
p = model_3.test_normality('jarquebera')[0, 1]
print(f"JB test for normality p-value: {p:.2f}")
if p < sig_level:
    print("Residuals significantly different from normal distribution. That's bad.
else:
    print("Residuals not significantly different from normal distribution. That's
```

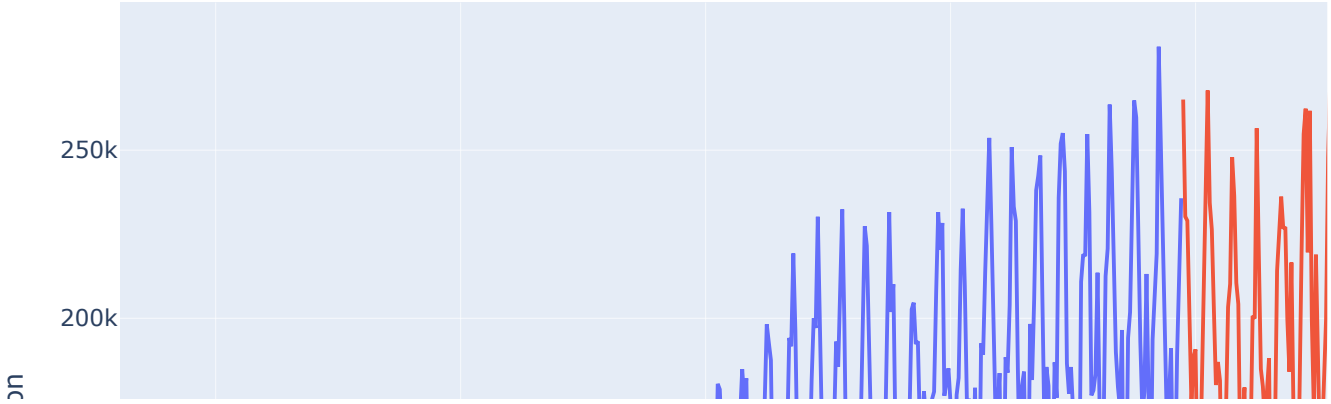
```
JB test for normality p-value: 0.00
Residuals significantly different from normal distribution. That's bad...
```

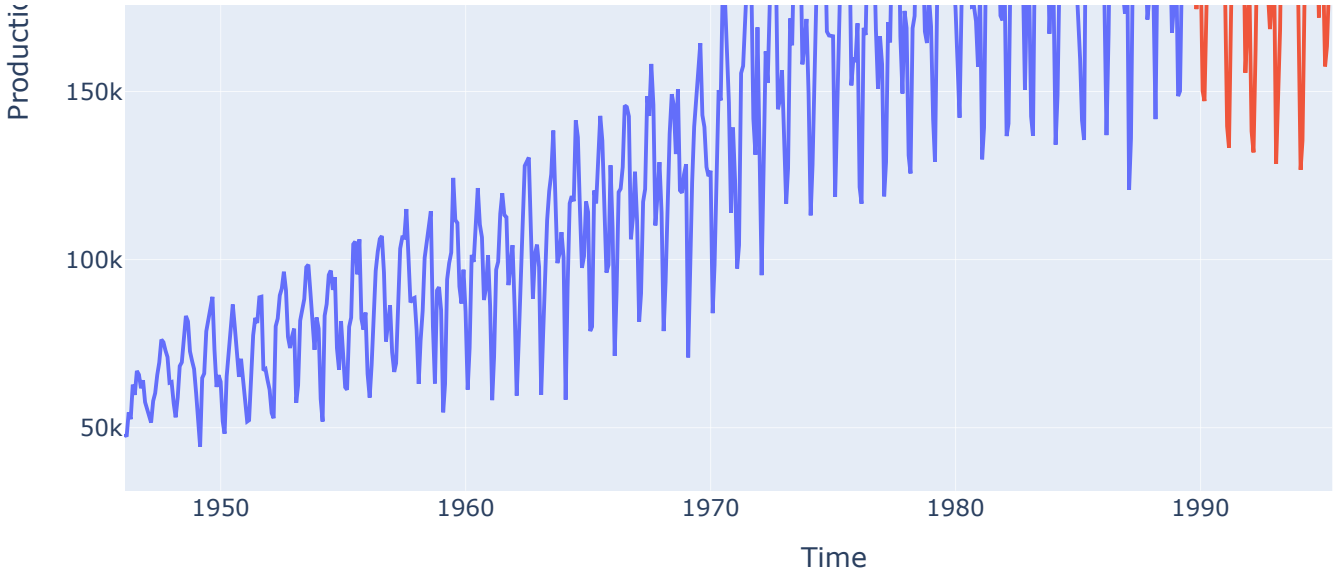
- Unfortunately, these tests tend to be not all the useful in practice, but there's no harm in considering them as they are part of most time series software packages anyway

4.2. Out-of-sample methods

- Most of the time, we'll be interested in using our model to make forecasts, so we care about its performance on unseen data
- For this, we usually use a training set and a validation set of data
- We'll talk more about this next week, because we need to split our data in a specific way when working with time series (we'll discuss cross-validation too), but for now, imagine we have the following train and validation data:

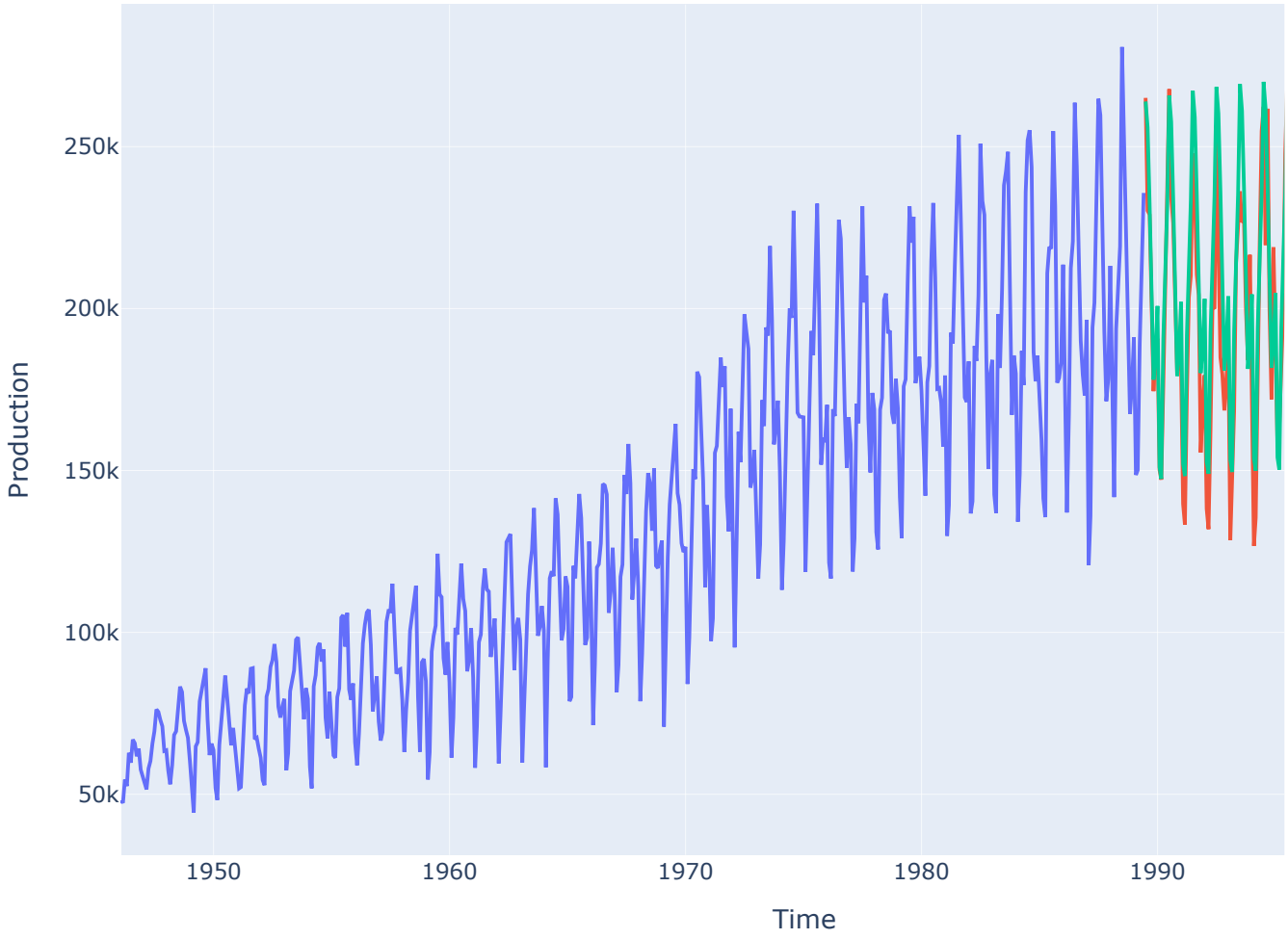
```
df_train = get_beer(label="Train")
df_valid = get_beer(split="valid", label="Validation")
px.line(pd.concat((df_train, df_valid)), y="Production", width=950, color="Label")
```





- So how do we evaluate our model forecasts? The most common regression metrics are:
  1. Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
  2. Root Mean Squared Error (RMSE): $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
  3. Mean Absolute Percentage Error (MAPE): $\frac{1}{n}\sum_{i=1}^{n}|\frac{y_i - \hat{y}_i}{y_i}|$
  4. Mean Absolute Scaled Error (MASE): $\frac{MAE}{\frac{1}{T-1}\sum_{t=2}^{T}|y_t - y_{t-1}|}$
- Some notes:
  - MAE and RMSE are popular in practice because they are easier to interpret. Closer to 0 is better.
- MAPE is scale-free and aims to proportionalize errors, such that the error for $\hat{y} = 12$ and $y = 10$ (MAPE = 20%, MSE = 4) is the same as $\hat{y} = 120$ and $y = 100$ (MAPE = 20%, MSE = 400). Closer to 0 is better.
- MAPE is problematic if 0 values are expected (divide by 0 error) and it is also not symmetrical, i.e., $\hat{y} = 150$ and $y = 100$ gives $MAPE = \frac{|100-150|}{100} = 33.33\%$, but $\hat{y} = 100$ and $y = 150$ gives $MAPE = \frac{|150-100|}{150} = 50\%$. There is a version, `sMAPE` available that is symmetrical, but MASE is often preferred.
- MASE scales the MAE based on the MAE of a naive forecast on the training data. I think of this as the "r-squared" of the forecasting world. It corrects the above errors. Value < 1 indicate forecasts are better than in-sample naive forecasts.

```
forecast = pd.DataFrame({"Production": model_3.forecast(len(forecast_index)),
                         "Label": "Forecast"},
                        index=forecast_index)
px.line(pd.concat((df_train, df_valid, forecast)), y="Production", width=950, colo
```



```
def accuracy(y_pred, y_test, y_train, index="Results"):
    """Calculate forecast accuracy.

    Parameters
    ----------
    y_pred : pandas.Series
        Predictions of y_test.
    y_test : pandas.Series
        Test observations.
    y_train : pandas.Series
        Train observations. Required for MASE.
    index : str, optional
        Name of output dataframe index.
        By default = "Results".

    Returns
    -------
    pandas.DataFrame
        A dataframe of the following metrics: MAE,
        RMSE, MAPE, MASE.
    """
    mae = mean_absolute_error(y_test, y_pred)
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    mape = mean_absolute_percentage_error(y_test, y_pred)
    mase = mean_absolute_scaled_error(y_test, y_pred, y_train)
    return pd.DataFrame([[mae, rmse, mape, mase]], columns=["MAE", "RMSE", "MAPE",
```

```
accuracy(forecast["Production"], df_valid["Production"], df_train["Production"])
```

	MAE	RMSE	MAPE	MASE
Results	13794.339855	19328.769833	0.07353	0.849991

5. General guidance

- It's hard to know what model will be best for your data, as always, there's no free lunch!
- My general advice is to:
    1. Start with a baseline model(s)
    2. Try improving forcasts with an ETS model
    3. Use in-sample, out-of-sample, and visual checks of forecasts and residuals to assess model performance
    4. Ensembling multiple methods almost always helps (more on that in a later lecture)