Probabilistic Forecasting with Randomized/Quasi-Randomized networks

T. Moudiki

2024-07-03

Plan

- 1 - Context
- 2 - Quasi-randomized *neural* networks (QRNs)
- 3 - QRN forecasting with `nnetsauce`

Plan

- **1 - Context**
- 2 - Quasi-randomized *neural* networks (QRNs)
- 3 - QRN forecasting with `nnetsauce`

1 - Context

- **Time series**:

A set of data collected sequentially usually at fixed intervals of time (Merriam-Webster dictionary onlike)

- **Objective:**
  - Forecasting one time series (**univariate forecasting**). **Example**: **Measurements of the annual flow of the river Nile** (in $10^8 m^3$) at Aswan from 1871 to 1970 $=>$ What happens in 1971, 1972, etc.? How ***"certain"*** can we be about the forecast?
  - Forecasting many related time series (**multivariate forecasting**). **Example**: Growth rates of personal **consumption** and personal **income** in the USA from 1970 to 2010 $=>$ What happens in 2011, 2012, etc.? How ***"certain"*** can we be?
  - Forecasting tool? **Quasi-randomized *neural* networks** applied to time series lags
  - Implemented in Python package `nnetsauce` version `0.22.4`

1 - Context

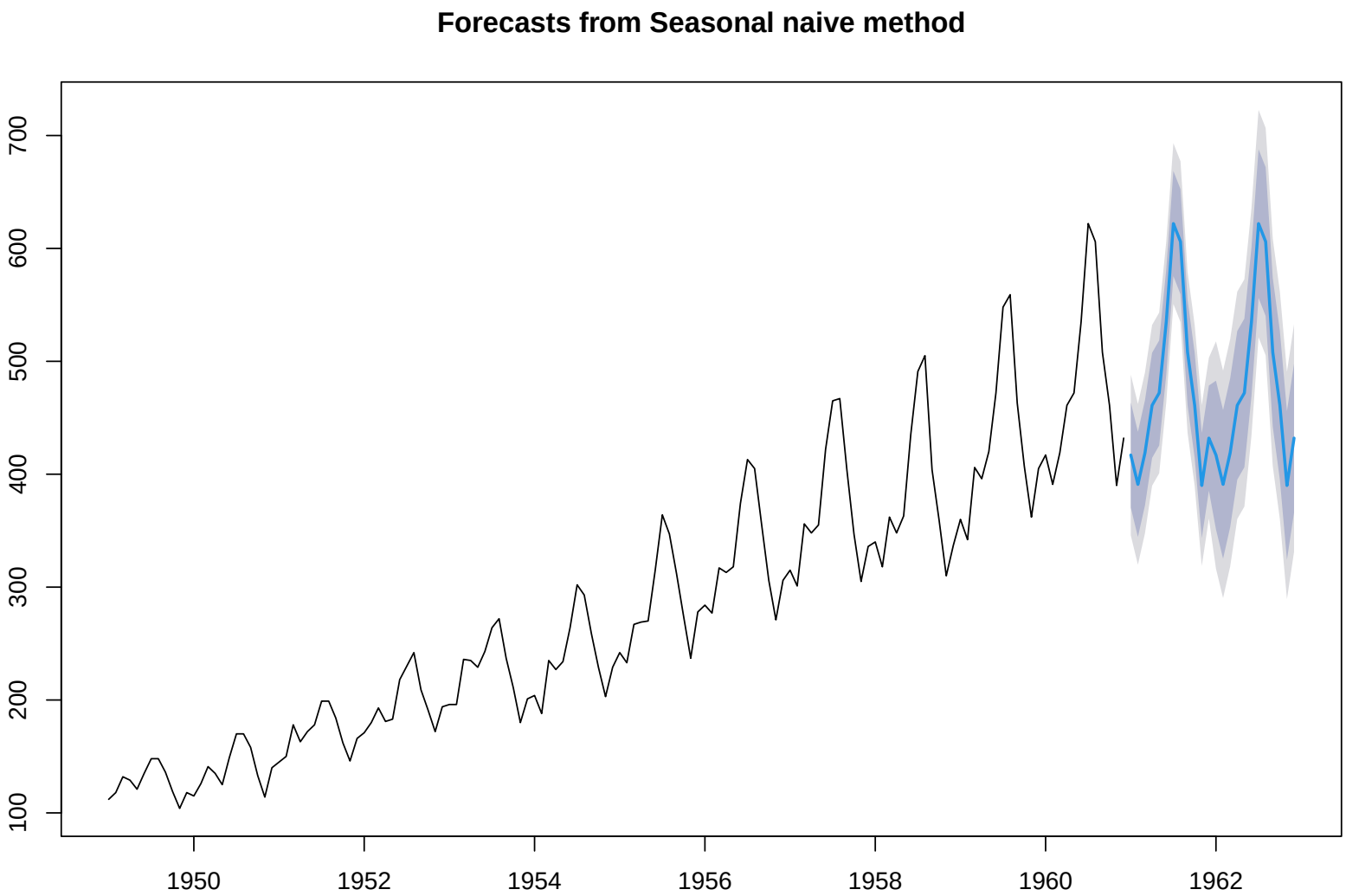
Point forecasts/Uncertainty quantification

- **Point forecasts** for one time series and for multivariate time series: **answers** what happens in 1971, 1972, etc.? What happens in 2011, 2012, etc.?
- **Point forecasts**: cool, but not very informative. How wrong can we be, based on the assumptions that we made: **answers** how ***"certain"*** can we be about the forecast?
- **Uncertainty quantification** needed: **prediction intervals** and/or **predictive simulations**.
- **prediction intervals**: point forecast +/- a term (with a level of confidence)
- **predictive simulations**: future scenarios for the variables of interest

1 - Context

Point forecasts/Uncertainty quantification

```
plot(forecast::snaive(AirPassengers))
```



1 - Context

Uncertainty quantification for time series

- Based on:
  - **Bayesian priors**
  - **In-sample residuals** = model fit - true observation on the *whole training set*
  - **Calibrated residuals** = model fit - true observation on a *held-out calibration set*
- **Calibrated residuals** used in `nnetsauce` for methods based on **sequential split conformal prediction** (more on this later)

1 - Context

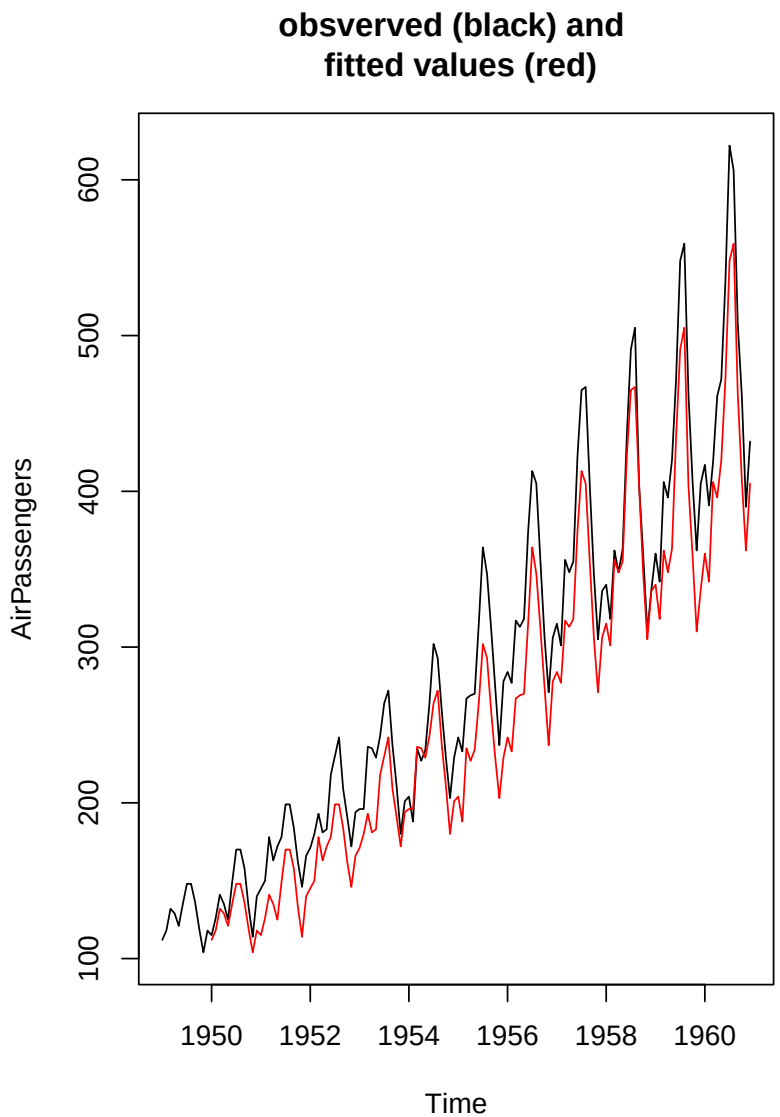
Uncertainty quantification for time series

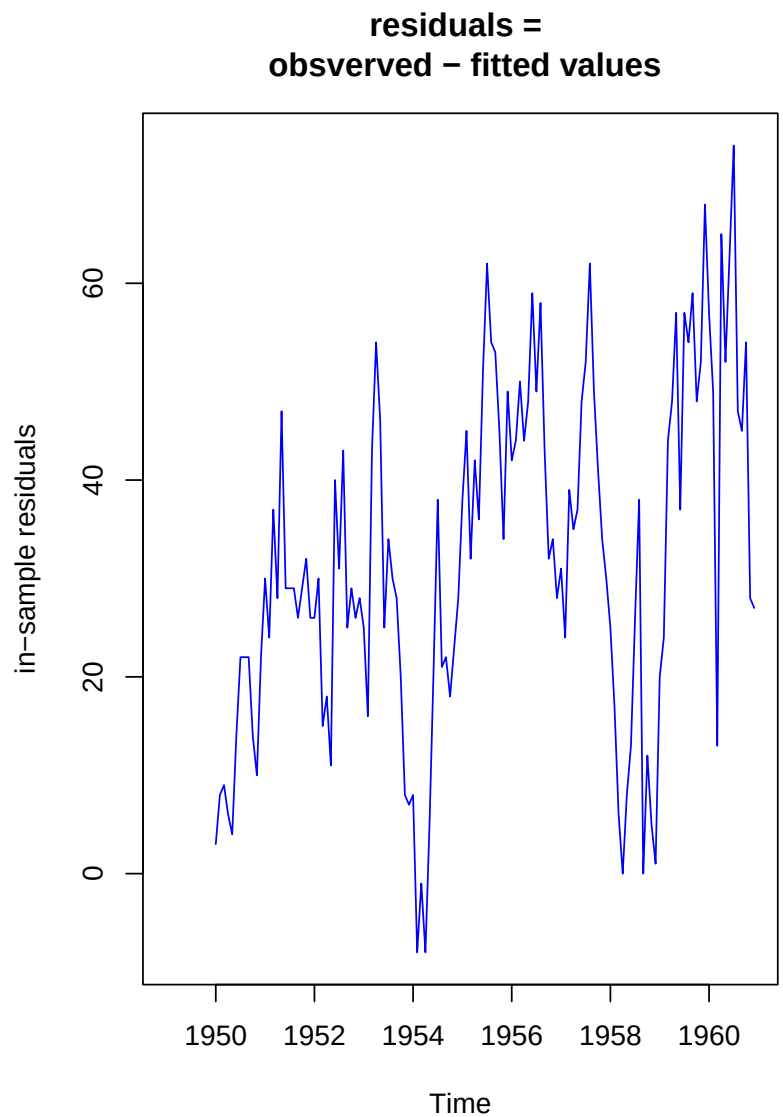
- Time series **residuals** matter:
  - can contain information => need to augment model capacity for example, so that they become smaller (and hopefully centered, with constant variance).
  - used for uncertainty quantification => need to infer their distribution, either parametric or by simulation.
  - assumption for residuals-based uncertainty quantification: **"pattern" observed in residuals is continuing** in the future.
- Important property of residuals: **stationarity**
  - **Stationarity**: no significant correlation of time series lags
  - **Stationarity** => time series mean and variance do not change over time.

1 - Context

Uncertainty quantification for time series

- **In-sample residuals**

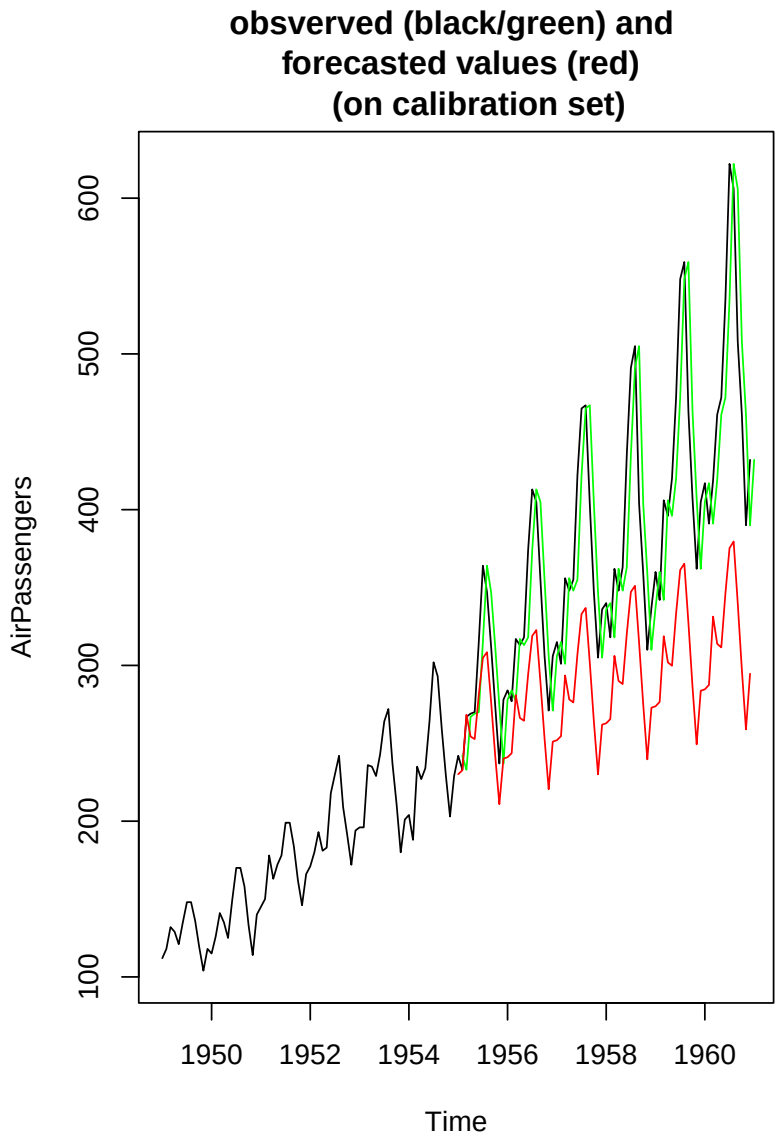


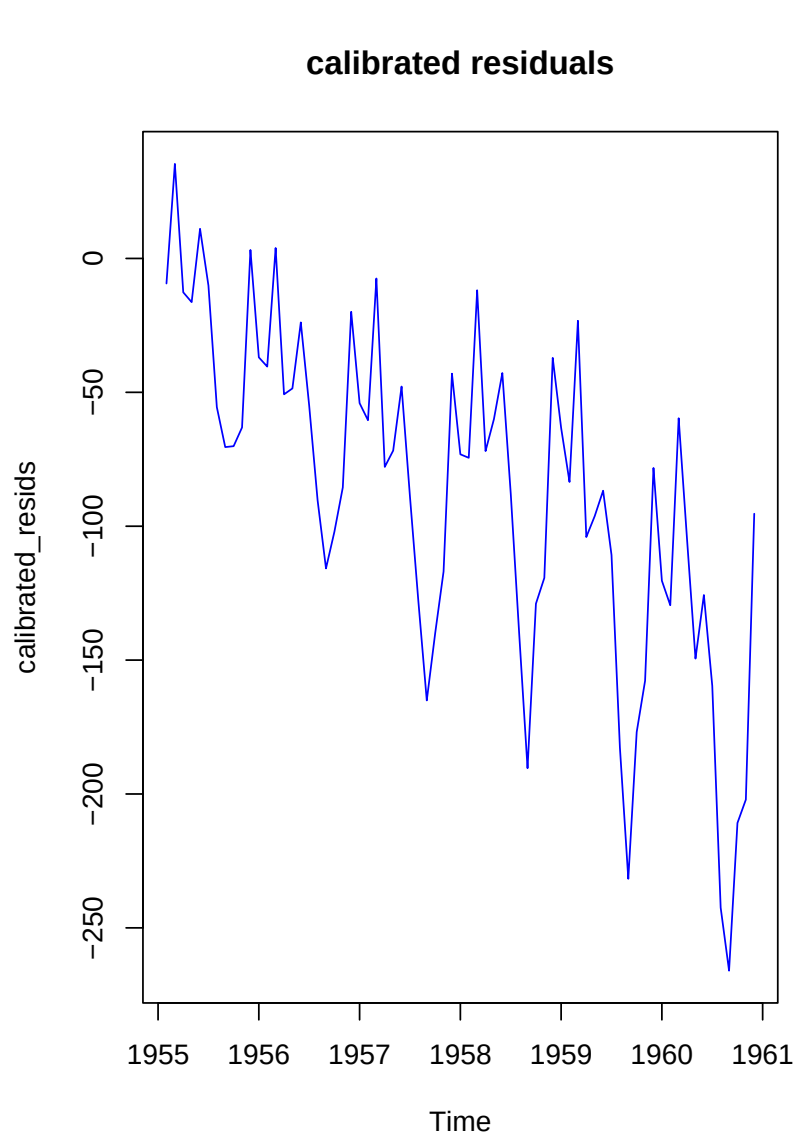


1 - Context

Uncertainty quantification for time series

- **Calibrated residuals?/Sequential split conformal prediction**





1 - Context

Uncertainty quantification for time series

- In `nnetsauce` version `0.22.4` (10 methods):
  - Can be achieved by a **Bayesian base learner**
  - Can be achieved by a **conformalized based learner**
  - Can rely on **in-sample** residuals for methods based on:
    - parametric residuals distribution inference (`gaussian`)
    - density estimation and simulation of residuals (`kde`)
    - bootstrap resampling (`bootstrap` and `block-bootstrap`)
  - Can rely on **calibrated** residuals for methods based on:
    - **sequential split conformal prediction** (`scp*-kde`, `scp*-bootstrap`, `scp*-block-bootstrap`)

Plan

- 1 - Context
- **2 - Quasi-randomized *neural* networks (QRNs)**
- 3 - QRN forecasting with `nnetsauce`

2 - Quasi-randomized *neural* networks (QRNs)

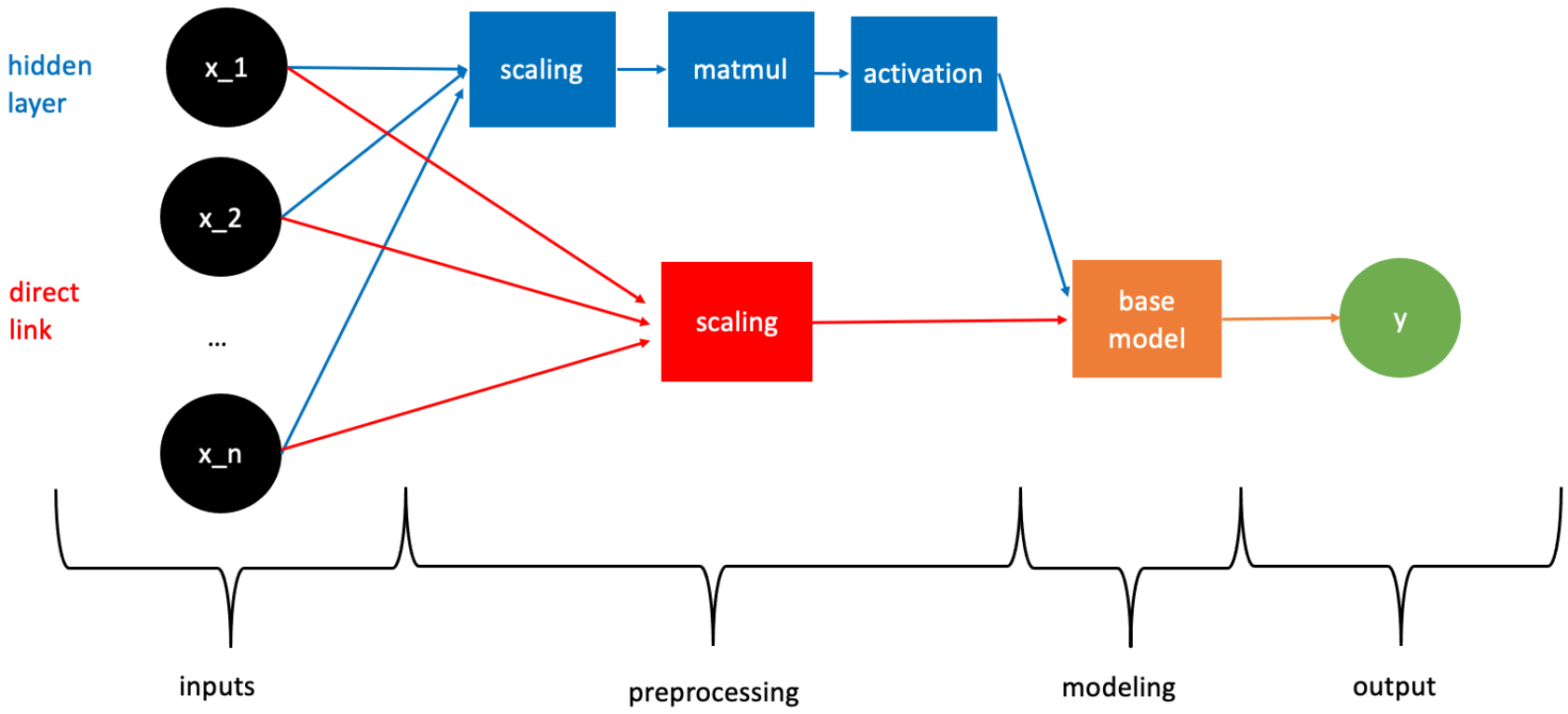


Figure 1: QRN principle

- A simple case (next slide): base model = **base learner** = Linear Regression

2 - Quasi-randomized *neural* networks (QRNs)

Simple case: **base learner** = Linear Regression; $y \in \mathbb{R}^n$, to be explained by $X^{(j)}, j \in \{1, \ldots, p\}$

$$y = \beta_0 + \sum_{j=1}^{p} \beta_j X^{(j)} + \sum_{l=1}^{L} \gamma_l g\left(\sum_{j=1}^{p} W^{(j,l)} X^{(j)}\right) + \epsilon$$

With:

- g : **activation function** $\rightarrow$ nonlinearity
- L : number of nodes in the **hidden layer**
- $W^{(j,l)}$, hidden layer: **pseudo/quasi-random**
- **Quasi-random**: designed to cover the space *parsimoniously*
- β_j and γ_l : linear model coefficients
- ϵ : residuals

2 - Quasi-randomized *neural* networks (QRNs)

QRNs applied to time series

- **Response** y = most recent time series observations
- **Covariates** X = time series lags
- **base learner**: can be any Machine Learning model
- **Multivariate forecasting case**: the base learner **shared** by all the time series

2 - Quasi-randomized *neural* networks (QRNs)

QRNs applied to time series (illustration)

- $p = 2$ time series $X^{(1)}$ and $X^{(2)}$; $n = 5$ dates $t_1 < \ldots < t_5$
- $X^{(2)} := \left(X_{t_1}^{(1)}, \ldots, X_{t_5}^{(1)}\right)$ and $X^{(2)} := \left(X_{t_1}^{(2)}, \ldots, X_{t_5}^{(2)}\right)$
- $k = 2$ lags; $L = 3$ nodes in hidden layer.

Response variables:

$$\mathbf{Y} = \begin{pmatrix} X_{t_5}^{(1)} & X_{t_5}^{(2)} \\ X_{t_4}^{(1)} & X_{t_4}^{(2)} \\ X_{t_3}^{(1)} & X_{t_3}^{(2)} \end{pmatrix}$$

Covariates:

$$\mathbf{X} = \begin{pmatrix} X_{t_4}^{(1)} & X_{t_3}^{(1)} & X_{t_4}^{(2)} & X_{t_3}^{(2)} \\ X_{t_3}^{(1)} & X_{t_2}^{(1)} & X_{t_3}^{(2)} & X_{t_2}^{(2)} \\ X_{t_2}^{(1)} & X_{t_1}^{(1)} & X_{t_2}^{(2)} & X_{t_1}^{(2)} \end{pmatrix}$$

2 - Quasi-randomized *neural* networks (QRNs)

QRNs applied to time series (illustration)

- $p = 2$ time series $X^{(1)}$ and $X^{(2)}$
- $n = 5$ dates $t_1 < \ldots < t_5$,
- $X^{(2)} := \left(X_{t_1}^{(1)}, \ldots, X_{t_5}^{(1)}\right)$ and $X^{(2)} := \left(X_{t_1}^{(2)}, \ldots, X_{t_5}^{(2)}\right)$
- $k = 2$ lags
- $L = 3$ nodes in hidden layer.

Coefficients in hidden layer:

$$\mathbf{W} = \begin{pmatrix} W^{(1,1)} & W^{(1,2)} & W^{(1,3)} \\ W^{(2,1)} & W^{(2,2)} & W^{(2,3)} \\ W^{(3,1)} & W^{(3,2)} & W^{(3,3)} \\ W^{(4,1)} & W^{(4,2)} & W^{(4,3)} \end{pmatrix}$$

Plan

- 1 - Context
- 2 - Quasi-randomized *neural* networks (QRNs)
- **3 - QRN forecasting with `nnetsauce`**

Plan

- **3 - QRN forecasting with `nnetsauce`**
- 3 - 1 `nnetsauce`'s description (Python version)
- 3 - 2 Install+import Python packages (including `nnetsauce`)
- 3 - 3 Import data for the demo
- 3 - 4 Using the `fit` + `predict` interface
- 3 - 5 Using GPUs
- 3 - 6 AutoML with `LazyMTS`
- 3 - 7 Time series cross-validation

3 - QRN forecasting with `nnetsauce`

3 - 1 `nnetsauce`'s description (Python version)

- General-purpose tool for Machine Learning using **Randomized** and **Quasi-Randomized *neural* networks**
  - GitHub: https://github.com/Techtonique/nnetsauce
  - PyPI: https://pypi.org/project/nnetsauce/
  - Conda: https://anaconda.org/conda-forge/nnetsauce
- **Tasks:**
  - Classification
  - Regression
  - **Univariate/Multivariate time series forecasting**

3 - 1 `nnetsauce`'s description (Python version) (cont'd)

- **Simple interface** for each model:
  - `fit`: fitting model to training data
  - `predict`: model inference on unseen data
- **GPU** version optimizes matrices multiplication using **JAX** (not *magical*)
- Classes `MTS` and `DeepMTS` for **time series forecasting**
- Here: **focus on `MTS`**
- Also: automated Machine Learning (**AutoML**) with class `LazyMTS`

3 - 2 Install+import Python packages (including nnetsauce)

```
pip install nnetsauce
pip install git+https://github.com/Techtonique/mlsauce.git
pip install xgboost
```

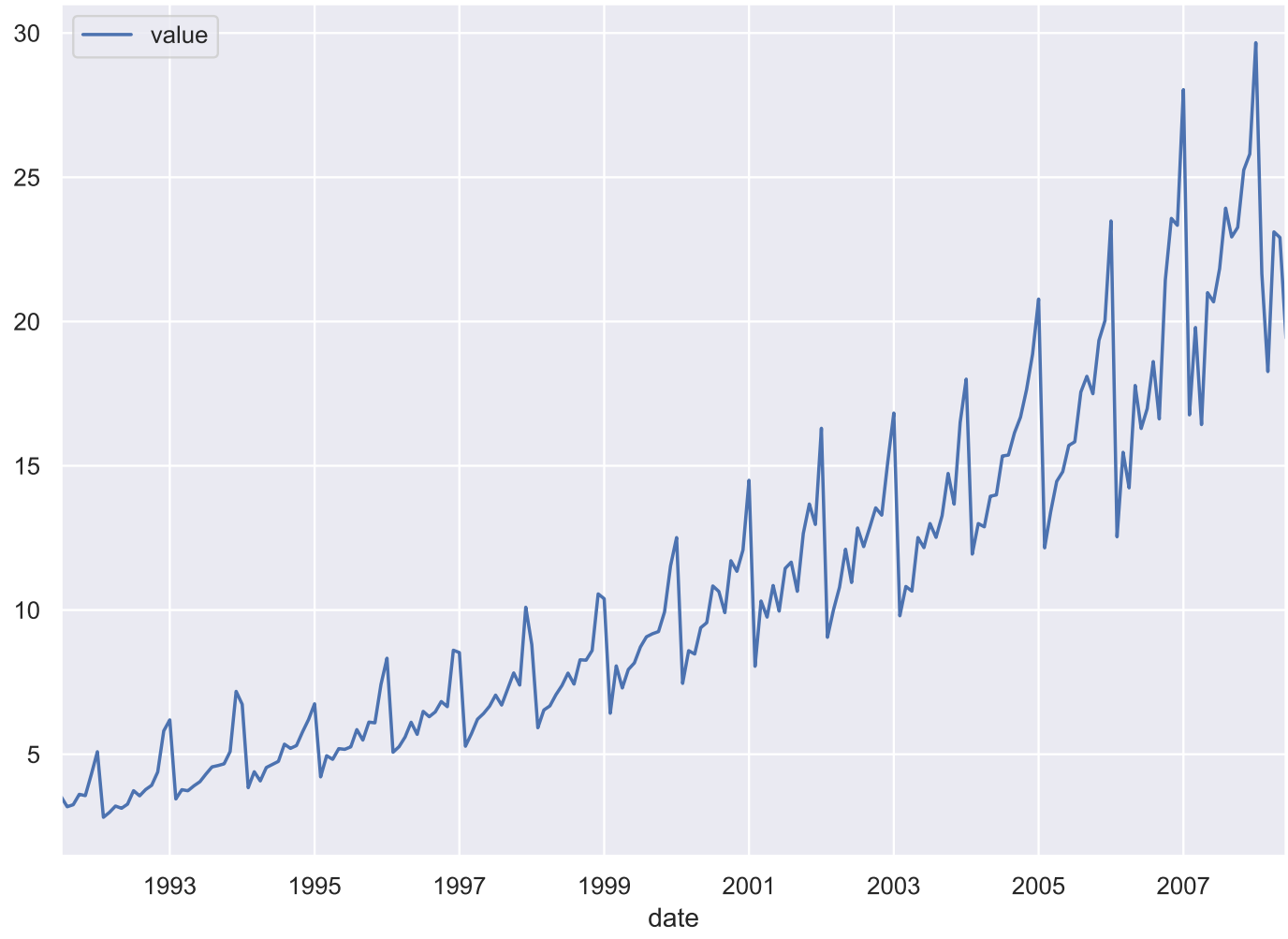
```
import nnetsauce as ns
import mlsauce as ms
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.linear_model import Ridge
from statsmodels.tsa.seasonal import STL

sns.set_theme(style="darkgrid")
```

3 - 3 Import data for the demo

Univariate: the famous `AirPassengers` data set: **Monthly Airline Passenger Numbers 1949-1960**

```
url = "https://raw.githubusercontent.com/Techtonique/"
url += "datasets/main/time_series/univariate/"
url += "a10.csv"
df_a10 = pd.read_csv(url)
df_a10.index = pd.DatetimeIndex(df_a10.date)
df_a10.drop(columns=['date'], inplace=True)
```

```
df_a10.plot()
```

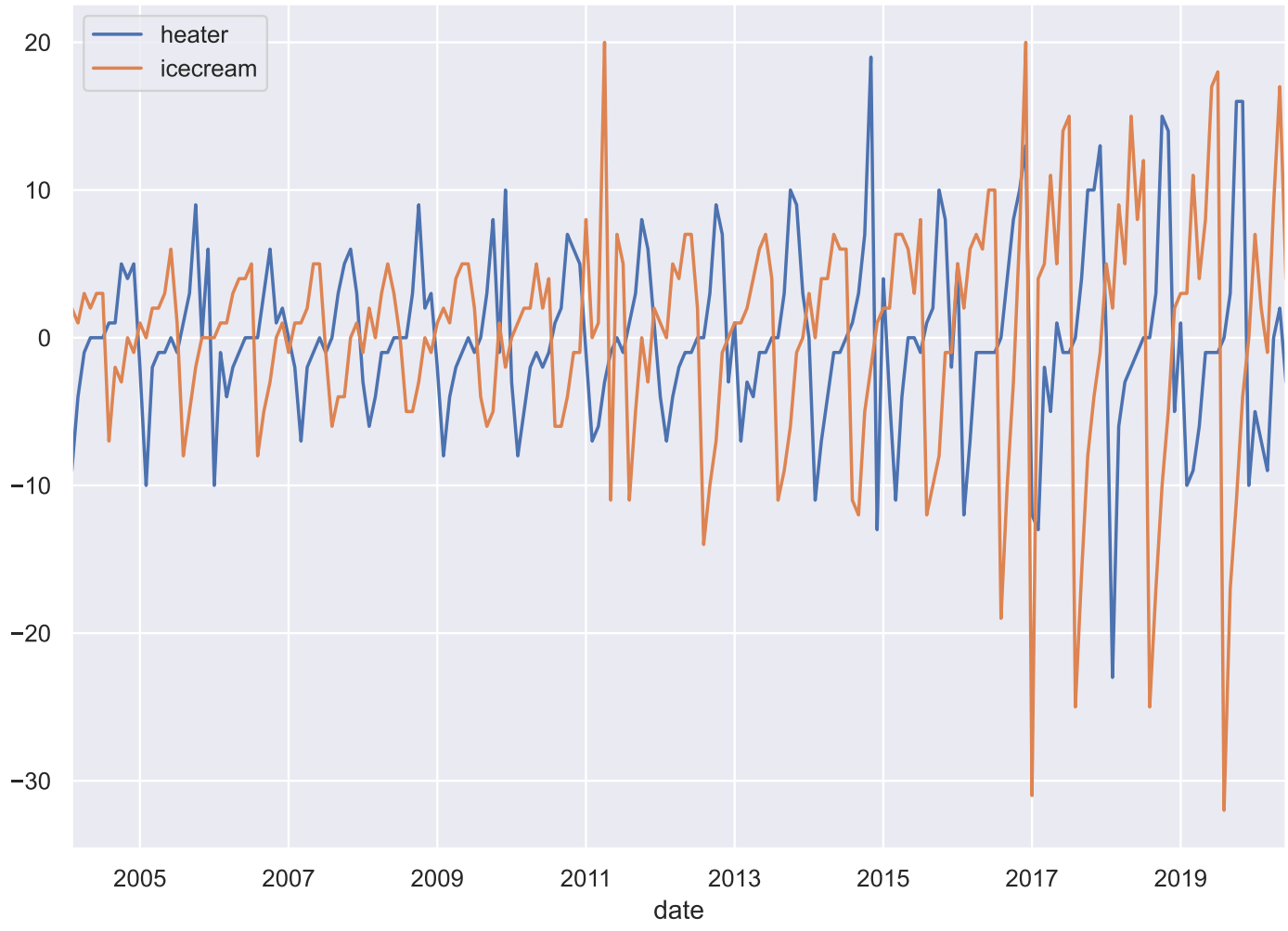

value
30
25
20
15
10
5
1993
1995
1997
1999
2001
2003
2005
2007
date

3 - 3 Import data for the demo (cont'd)

Multivariate: **Heater vs Ice cream sales** data set

```
url = "https://raw.githubusercontent.com/Techtonique/"
url += "datasets/main/time_series/multivariate/"
url += "ice_cream_vs_heater.csv"
df_temp = pd.read_csv(url)
df_temp.index = pd.DatetimeIndex(df_temp.date)
df_icecream = df_temp.drop(columns=['date']).diff()\
.dropna()
```

```
df_icecream.plot()
```



3 - 4 Using the `fit` + `predict` interface

- Gaussian
- Bayesian (Gaussian prior)
- Kernel Density Estimation (KDE)
- Conformalized base learner: `TweedieRegressor` + sequential split conformal

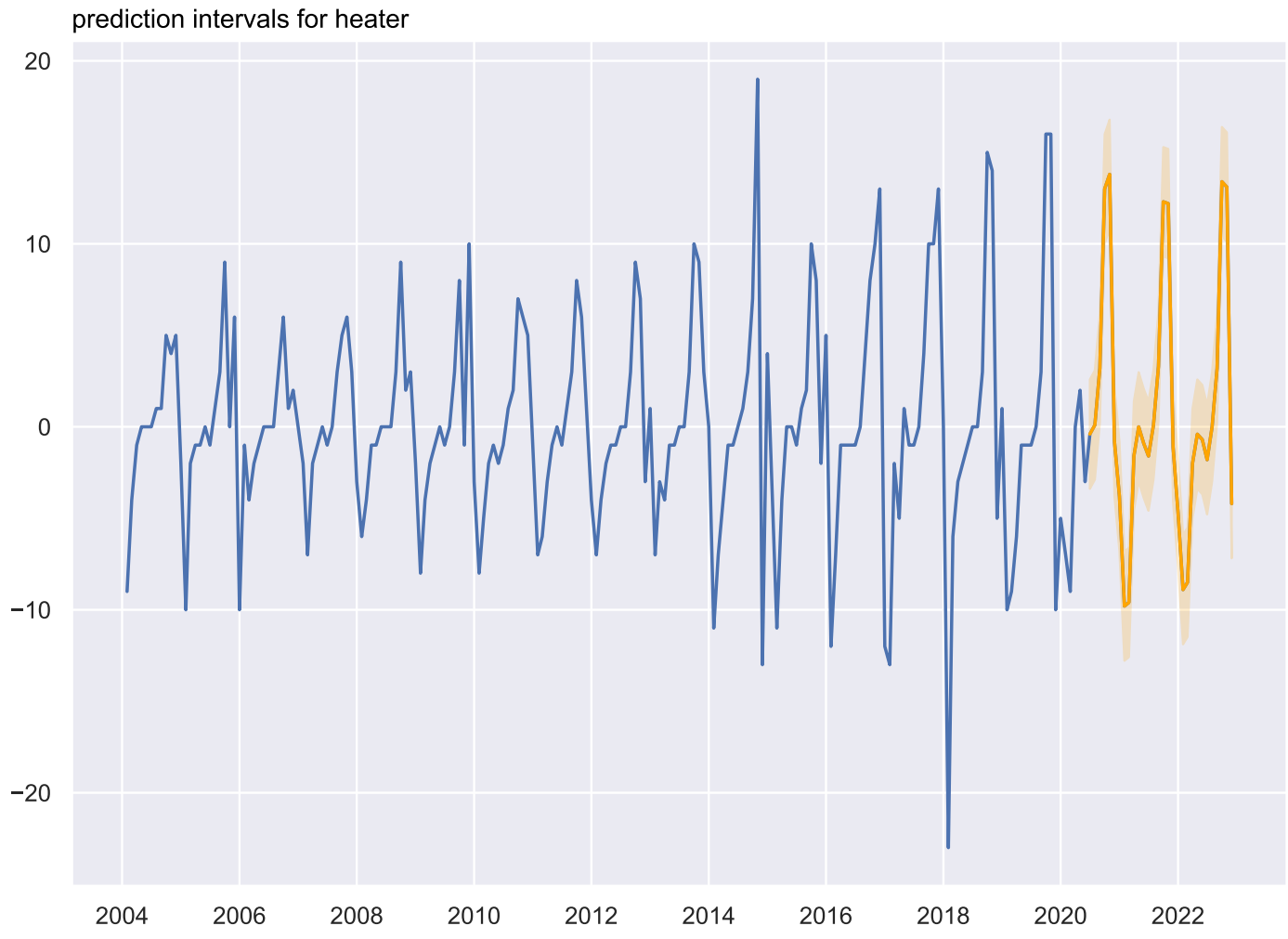
3 - 4 Using the `fit` + `predict` interface

- **Gaussian**

```
from sklearn.ensemble import BaggingRegressor

regr = ns.MTS(obj=BaggingRegressor(),
              type_pi="gaussian",
              lags=20,
              show_progress=False)
regr.fit(df_icecream);
regr.predict(h=30);
```

3 - 4 Using the `fit` + `predict` interface

```
regr.plot("heater", type_plot="pi")
```



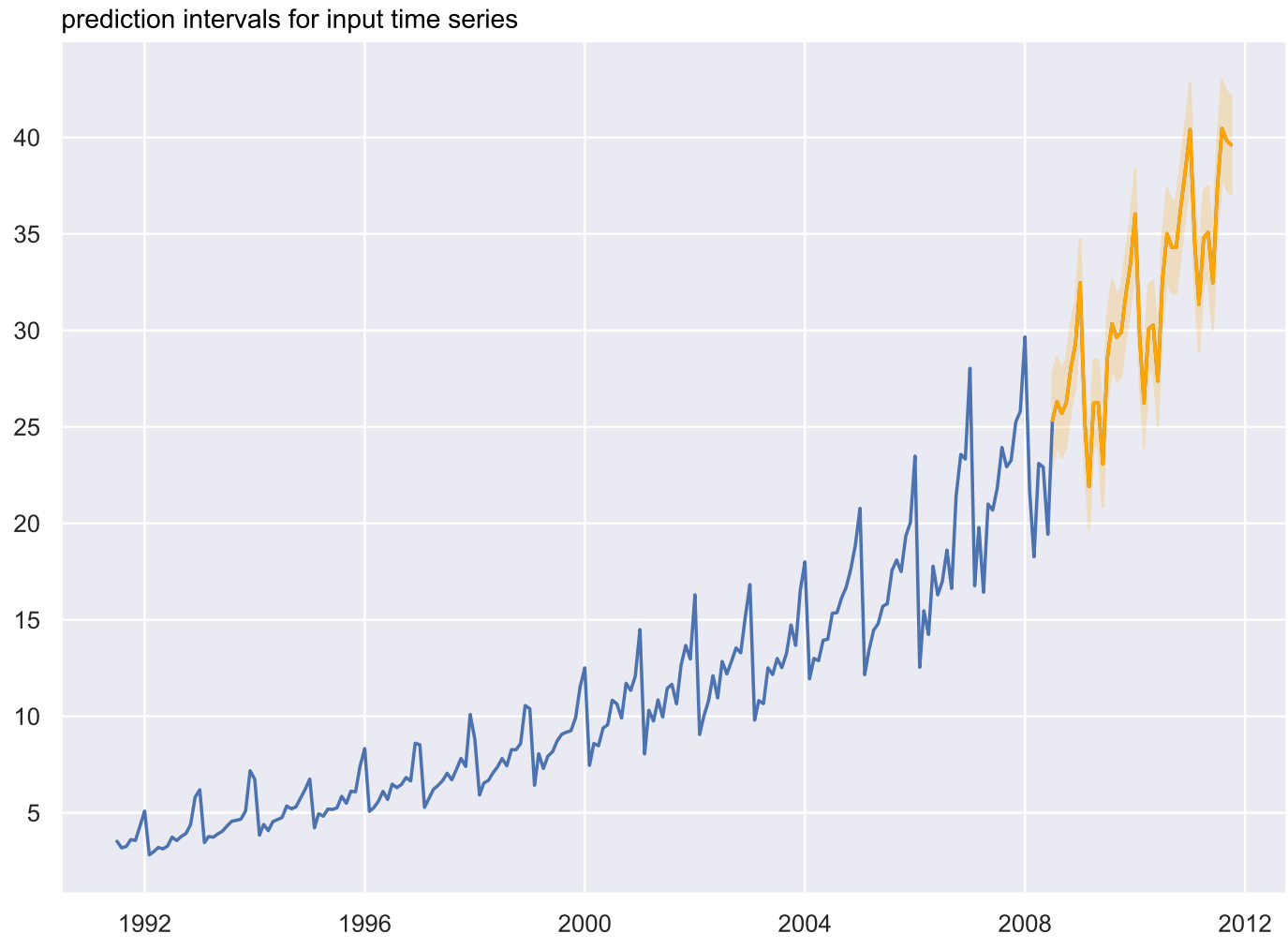
3 - 4 Using the `fit` + `predict` interface

- **Bayesian (Gaussian prior)**

```
from sklearn.linear_model import BayesianRidge

regr = ns.MTS(obj=BayesianRidge(),
              lags=15,
              show_progress=False)
regr.fit(df_a10);
regr.predict(h=40, return_std=True);
```

3 - 4 Using the `fit` + `predict` interface

```
regr.plot(type_plot="pi")
```



3 - 4 Using the `fit` + `predict` interface

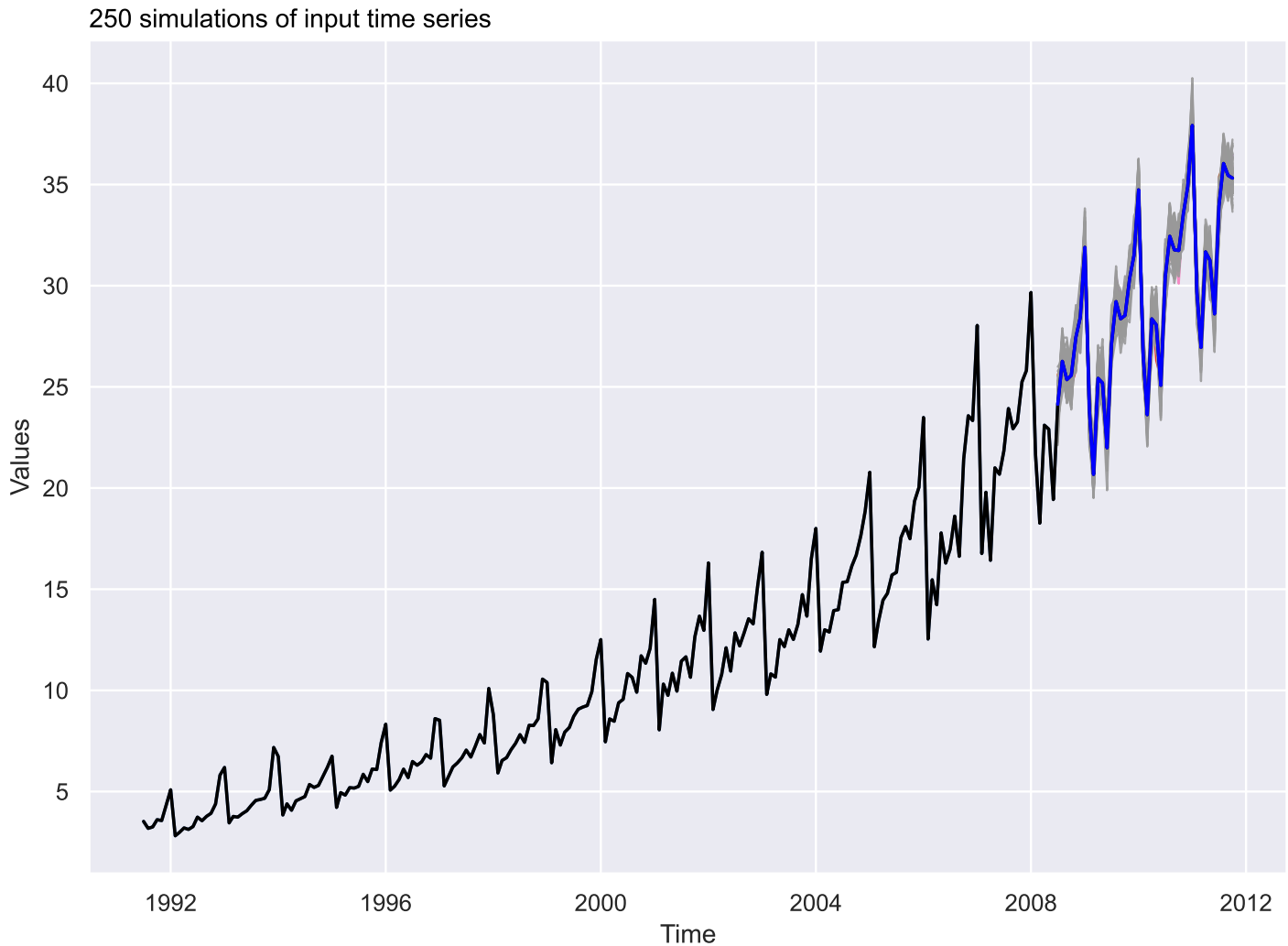
- **SCP-KDE**

```
from sklearn.linear_model import LassoLarsIC

regr = ns.MTS(obj=LassoLarsIC(),
              type_pi="scp-kde",
              replications=250,
              kernel='gaussian',
              lags=15,
              show_progress=False)
regr.fit(df_a10);
regr.predict(h=40);
```

```
##   0%|          | 0/250 [00:00<?, ?it/s]100%|##########|
##   0%|          | 0/250 [00:00<?, ?it/s]100%|##########|
```

3 - 4 Using the `fit` + `predict` interface

```
regr.plot(type_plot="spaghetti")
```



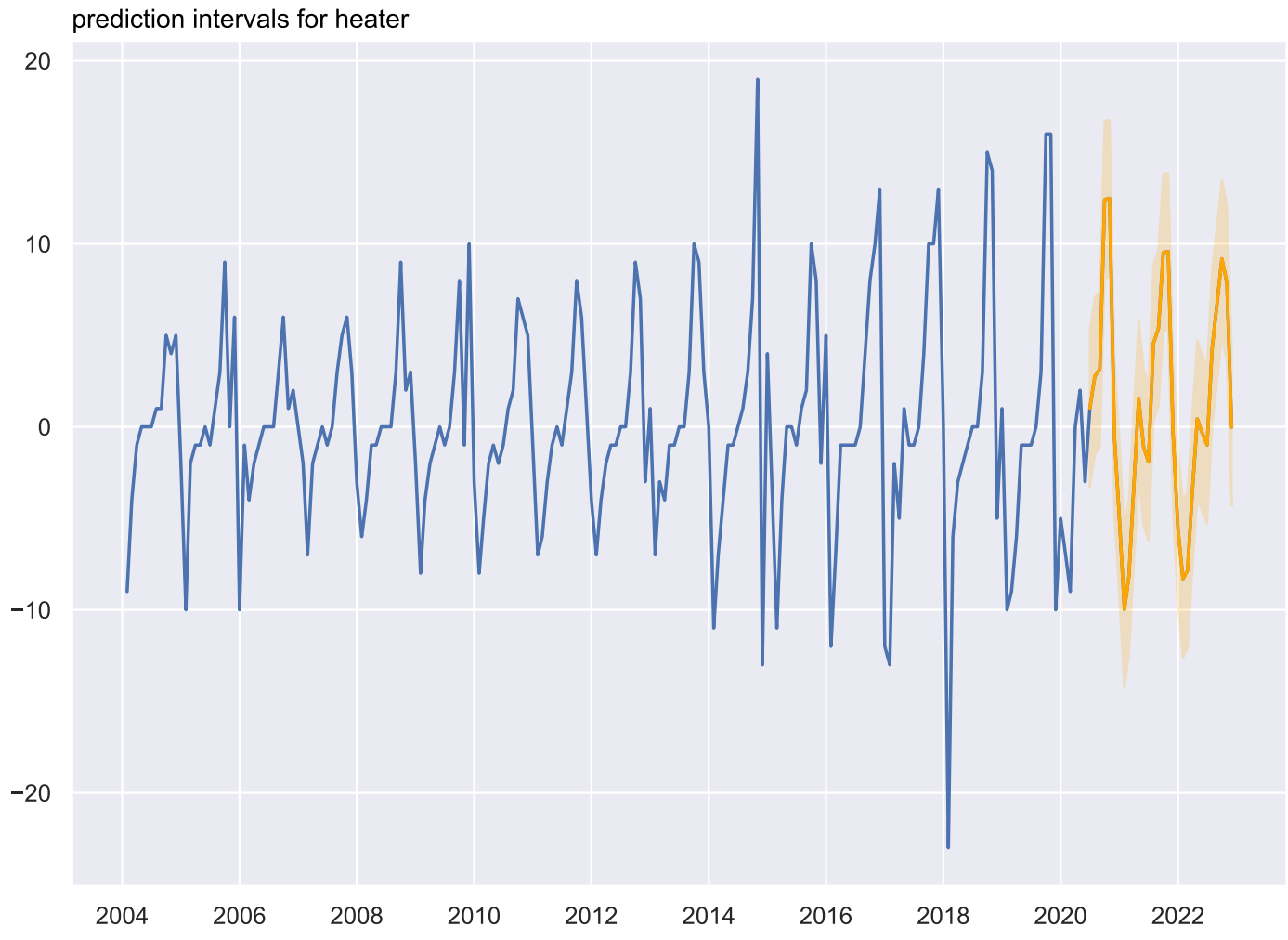
3 - 4 Using the `fit` + `predict` interface

- **Conformalized base learner**

```
from sklearn.linear_model import TweedieRegressor

obj0 = ns.PredictionInterval(obj=TweedieRegressor(),
                             method="splitconformal",
                             type_split="sequential",
                             level=95)
regr = ns.MTS(obj=obj0,
              lags=20,
              show_progress=False)
regr.fit(df_icecream);
regr.predict(h=30, return_pi=True);
```

3 - 4 Using the `fit` + `predict` interface

```
regr.plot("heater", type_plot="pi")
```



3 - 5 Using GPUs

- Public notebook on GitHub (https://bit.ly/45RchgD)
- Simulated multivariate time series: 100 series, 10000 observations
- Ran on **Kaggle notebooks**, with **accelerator GPU P100**

```
import numpy as np
import pandas as pd
import nnetsauce as ns
from sklearn.linear_model import Ridge
from time import time
```

```
df = generate_synthetic_mts(n_steps=10000, n_series=100,
                            amplitude=40, seed=14531)
df_ = df.diff().dropna()
```

3 - 5 Using GPUs (cont'd)

- **Example 1** on CPU, using Ridge regression

```
regr = Ridge()
obj_MTS = ns.MTS(regr,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="cpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 64.46652388572693
```

3 - 5 Using GPUs (cont'd)

- **Example 2** on GPU (uses **JAX** behind the scenes), using Ridge regression

```
regr = Ridge()
obj_MTS = ns.MTS(regr,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="gpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 40.530698537826544
```

- **37% time gain**

3 - 5 Using GPUs (cont'd)

- **Example 3** on GPU (uses **JAX** behind the scenes), using Ridge regression on GPU (`mlsauce` implementation) **too**

```
regr = ms.RidgeRegressor(reg_lambda=1.0, backend="gpu")
obj_MTS = ns.MTS(regr,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="gpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 23.551459312438965
```

- **63% time gain**

3 - 5 Using GPUs (cont'd)

- **Example 4** Using XGBoost

```
import xgboost as xgb

xgb1 = xgb.XGBRegressor()
xgb2 = xgb.XGBRegressor(tree_method = "gpu_hist")
```

```
obj_MTS = ns.MTS(xgb1,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="cpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 3285.469851732254
```

3 - 5 Using GPUs (cont'd)

- **Example 5** Using XGBoost on GPU

```
obj_MTS = ns.MTS(xgb1,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="gpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 3266.556359767914
```

- No substantial gain (19 seconds), **but...**

3 - 5 Using GPUs (cont'd)

- **Example 2** Using XGBoost's GPU implementation

```
obj_MTS = ns.MTS(xgb2,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="cpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 249.40361714363098
```

- **92% time gain** over `backend="cpu"`

3 - 5 Using GPUs (cont'd)

- **Example 2** Using XGBoost's GPU implementation

```
obj_MTS = ns.MTS(xgb2,
                 lags = 15,
                 n_hidden_features=5,
                 nodes_sim="uniform",
                 backend="gpu", # specify backend
                 verbose = 1)
start = time()
obj_MTS.fit(df_)
print(f"Elapsed: {time()-start}")
```

```
Elapsed: 249.10436272621155
```

- a little faster than `backend="cpu"`
- **92% time gain** over `backend="gpu"` with XGBoost CPU

Conclusion: **No magic trick**, experiment to see what works *best*!

3 - 6 Automated Machine Learning (AutoML) with LazyMTS

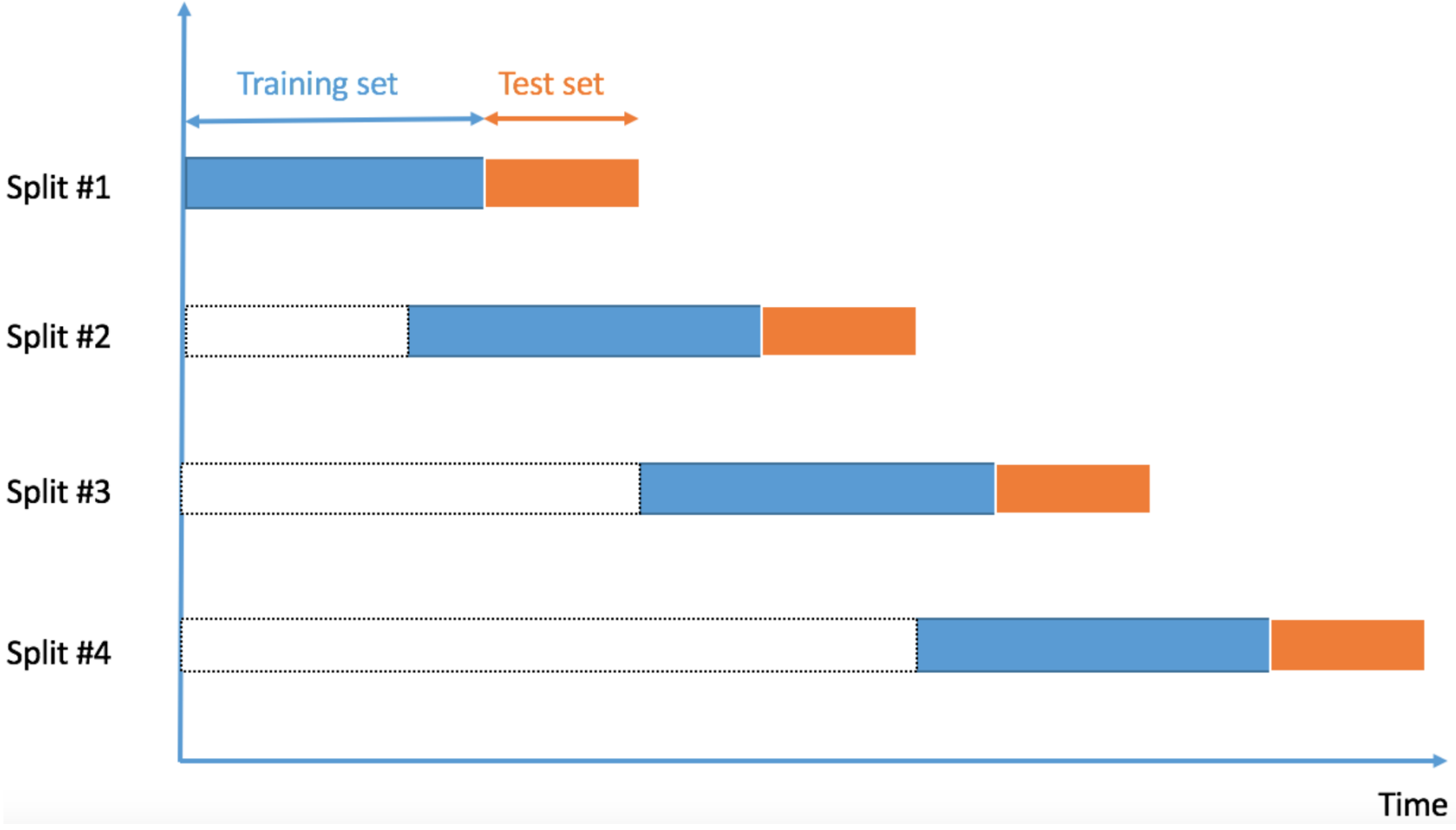
```
# split data into training/testing set
idx_train = int(df_icecream.shape[0]*0.9)
idx_end = df_icecream.shape[0]
df_train = df_icecream.iloc[0:idx_train,:]
df_test = df_icecream.iloc[idx_train:idx_end,:]

# Train + predict on 5 ML models
regr = ns.LazyMTS(verbose=1, ignore_warnings=False,
                  custom_metric=None, show_progress=False,
                  preprocess=True,
                  type_pi="kde",
                  kernel="gaussian",
                  lags = 20, replications=250,
                  estimators=["Ridge",
                  "ElasticNet", "ExtraTreesRegressor",
                  "RandomForestRegressor"]);
models, predictions = regr.fit(df_train, df_test);
```

3 - 6 Automated Machine Learning (AutoML) with LazyMTS (cont'd)

```
print(models[['RMSE', 'WINKLERSCORE']])
```

```
##                                    RMSE  WINKLERSCORE
## Model
## MTS(Ridge)                         8.52        220.35
## MTS(RandomForestRegressor)         8.79        239.63
## MTS(ExtraTreesRegressor)           8.82        266.00
## MTS(ElasticNet)                    9.64        217.62
```

3 - 7 Time series cross-validation (cont'd)

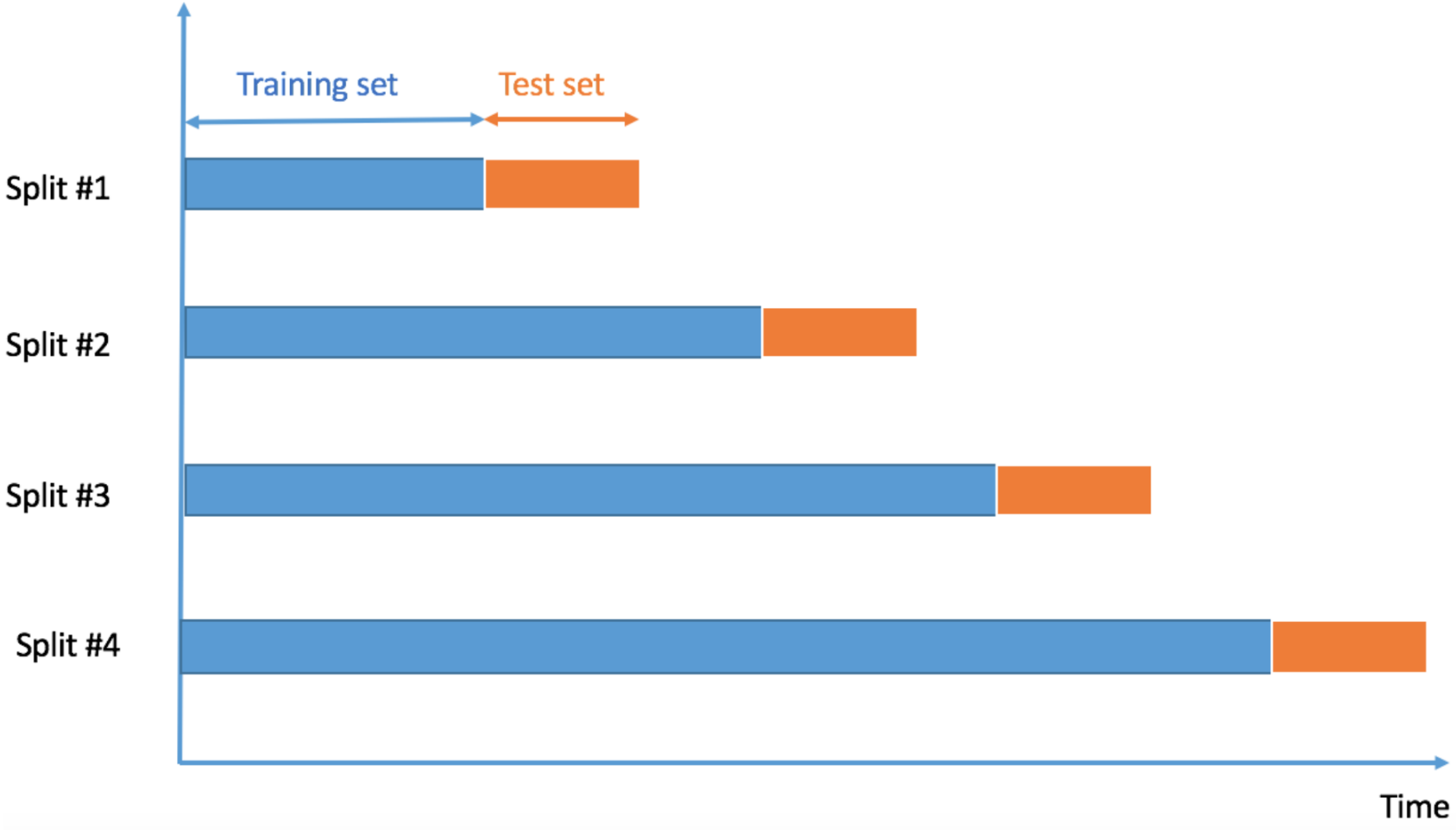


3 - 7 Time series cross-validation (cont'd)

```
tscv = ns.utils.model_selection.TimeSeriesSplit()

X = np.random.rand(10, 2)

# `fixed_window` = type of slicing
tscv_obj = tscv.split(X, initial_window=3, horizon=2,
                      fixed_window=True)

for train_index, test_index in tscv_obj:
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
## TRAIN: [0 1 2] TEST: [3 4]
## TRAIN: [1 2 3] TEST: [4 5]
## TRAIN: [2 3 4] TEST: [5 6]
## TRAIN: [3 4 5] TEST: [6 7]
## TRAIN: [4 5 6] TEST: [7 8]
## TRAIN: [5 6 7] TEST: [8 9]
```

3 - 7 Time series cross-validation (cont'd)



3 - 7 Time series cross-validation (cont'd)

```
tscv = ns.utils.model_selection.TimeSeriesSplit()

X = np.random.rand(10, 2)

# `fixed_window` = type of slicing
tscv_obj = tscv.split(X, initial_window=3, horizon=2,
                      fixed_window=False)

for train_index, test_index in tscv_obj:
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
## TRAIN: [0 1 2] TEST: [3 4]
## TRAIN: [0 1 2 3] TEST: [4 5]
## TRAIN: [0 1 2 3 4] TEST: [5 6]
## TRAIN: [0 1 2 3 4 5] TEST: [6 7]
## TRAIN: [0 1 2 3 4 5 6] TEST: [7 8]
## TRAIN: [0 1 2 3 4 5 6 7] TEST: [8 9]
```