Lecture 4: Forecasting Time Series with Machine Learning

Contents

UBC Master of Data Science

DSCI 574
Spatial & Temporal Models

Lecture Outline

- Lecture Learning Objectives
- Imports
- 1. Machine learning with time series
- 2. Forecasting

Lecture Learning Objectives

- Describe how to properly split a time series into train, validation, and test sets.
- Forecast a univariate time series using a machine learning algorithm.
- Forecast a multivariate time series using a machine learning algorithm.
- Describe the difference between recursive and direct forecasts for multi-step forecast horizons.
- Explain common time series pre-processing steps like: coercing to stationary, smoothing, and removing outliers.
- Explain common time series feature engineering techniques like: lagging features, adding time stamps, using rolling statistics.

Imports

```python
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
import plotly.express as px
import plotly.graph_objects as go
import matplotlib.pyplot as plt
from plotly.subplots import make_subplots
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import FunctionTransformer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split, TimeSeriesSplit
from sklearn.pipeline import make_pipeline
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
# Custom modules
from scripts.utils import *
from scripts.plotting import *
# Plot defaults
px.defaults.height = 400; px.defaults.width = 550
```

Machine learning workflow recap

Import

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_percentage_error
```

Data preparation/Feature engineering

```
X, y = make_regression(n_samples=1000, n_features=10, noise=0.1, random_state=
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
```

Model specification

```
rf = RandomForestRegressor()
```

Fit

```
rf.fit(X_train, y_train)
```

```
▾ RandomForestRegressor
RandomForestRegressor()
```

Predict

```
y_pred = rf.predict(X_test)
```

Evaluation

```
mape = mean_absolute_percentage_error(y_test, y_pred)
print('MAPE:', mape)
```

```
MAPE: 0.9399489915338621
```

Cross-validation/Hyperparameter optimization

Transformer

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestRegressor())
])
```

Grid search cross validation

```
param_grid = {
    'n_estimators': [10, 50, 100],
    'max_depth': [None, 5, 10]
}
```

```
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, n_jobs=-
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print('Best parameters:', best_params)
```

```
Best parameters: {'max_depth': None, 'n_estimators': 50}
```

Re-fit with the optimized parameters

```python
rf = RandomForestRegressor(**best_params)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
mape = mean_absolute_percentage_error(y_test, y_pred)
print('MAPE:', mape)
```

```
MAPE: 1.123971909275135
```

Key Differences with Time Series Data

Train/test split

- We don't shuffle the data to preserve the temporal order

Cross-validation split

- We use sliding window or expanding window approach

Feature engineering

- We use the lagged variables $(y_{t-1}, y_{t-2}, \ldots)$ of the target y_t as predictors
- We transform the data (log, Box-Cox, detrended, deseasonalized)

Prediction

- We need to specify the forecast horizon
- We fit the model using a recursive, direct, or hybrid approach

The `sktime` package

The `sktime` package provides an easy-to-use, flexible and modular open-source framework for a wide range of time series machine learning tasks. It offers `scikit-learn` compatible interfaces and model composition tools, with the goal to make the ecosystem more usable and interoperable as a whole.

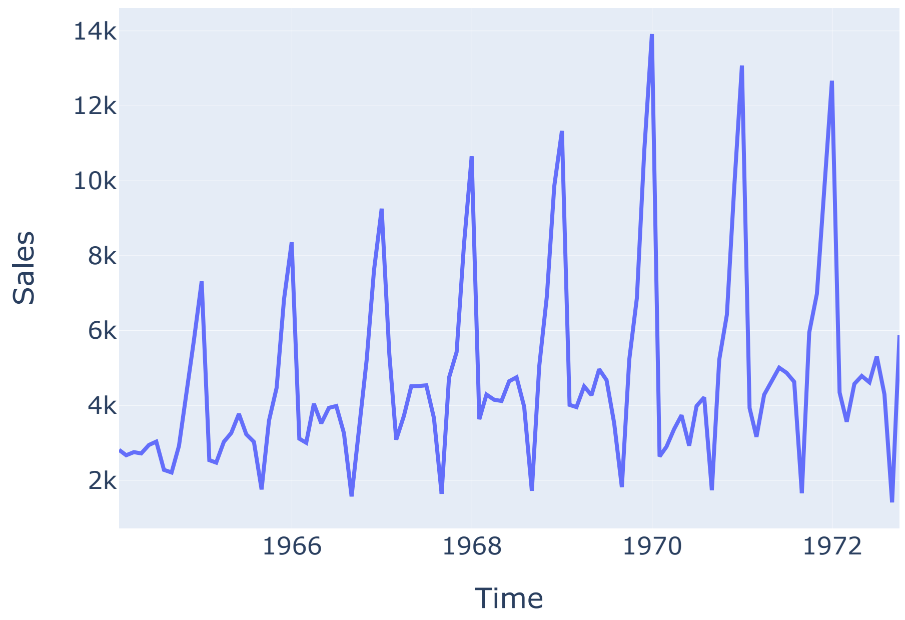
- It offers a wide range of time-series specific data transformation techniques (generate lagged values, box-cox, fourier, etc.)

- It offeres a wide range of classical time-series model with automated optimization (STLforecaster, autoETS, autoarima, etc.) and deep learning (CNNregressor)
- It resembles the scikit-learn workflow and you can easily prepare the data and fit using classic ML models (random forest, SVM, KNN, etc..) with a few lines of code.
- The `sktime` package is relatively new and still under active development but it has gained substantial traction over the past few years.

1. Machine learning with time series

- I'm going to demonstrate concepts in this lecture using a relatively simple time series of 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)
px.line(df, y="Sales")
```



1.1. Features engineering

- As we wish to use past information to help predict the future, we'll be approaching our ML implementation similarly to how `AR(p)` models work, by having features that are lagged versions of the response
- Recall that we can do this easily in Pandas using the dataframe method `.shift()`:

```
df["Sales-1"] = df.shift(1)
df.head()
```

	Sales	Sales-1
Time		
1964-01-31	2815.0	NaN
1964-02-29	2672.0	2815.0
1964-03-31	2755.0	2672.0
1964-04-30	2721.0	2755.0
1964-05-31	2946.0	2721.0

You can do the same for multiple lags using the `Lag` function from `sktime` package:

```
from sktime.transformations.series.lag import Lag
t = Lag(lags=[1,2,3], index_out="original") # I want to create lag 1, lag 2, a
# t = Lag(lags=list(range(1,12)), index_out="original")
t.get_params()
```

```
{'flatten_transform_index': True,
 'freq': None,
 'index_out': 'original',
 'keep_column_names': False,
 'lags': [1, 2, 3],
 'remember_data': True}
```

```
df = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
df.index.freq = "M"
pd.concat([df,t.fit_transform(df)],axis=1).head()
```

	Sales	lag_1__Sales	lag_2__Sales	lag_3__Sales
Time				
1964-01-31	2815.0	NaN	NaN	NaN
1964-02-29	2672.0	2815.0	NaN	NaN
1964-03-31	2755.0	2672.0	2815.0	NaN
1964-04-30	2721.0	2755.0	2672.0	2815.0
1964-05-31	2946.0	2721.0	2755.0	2672.0

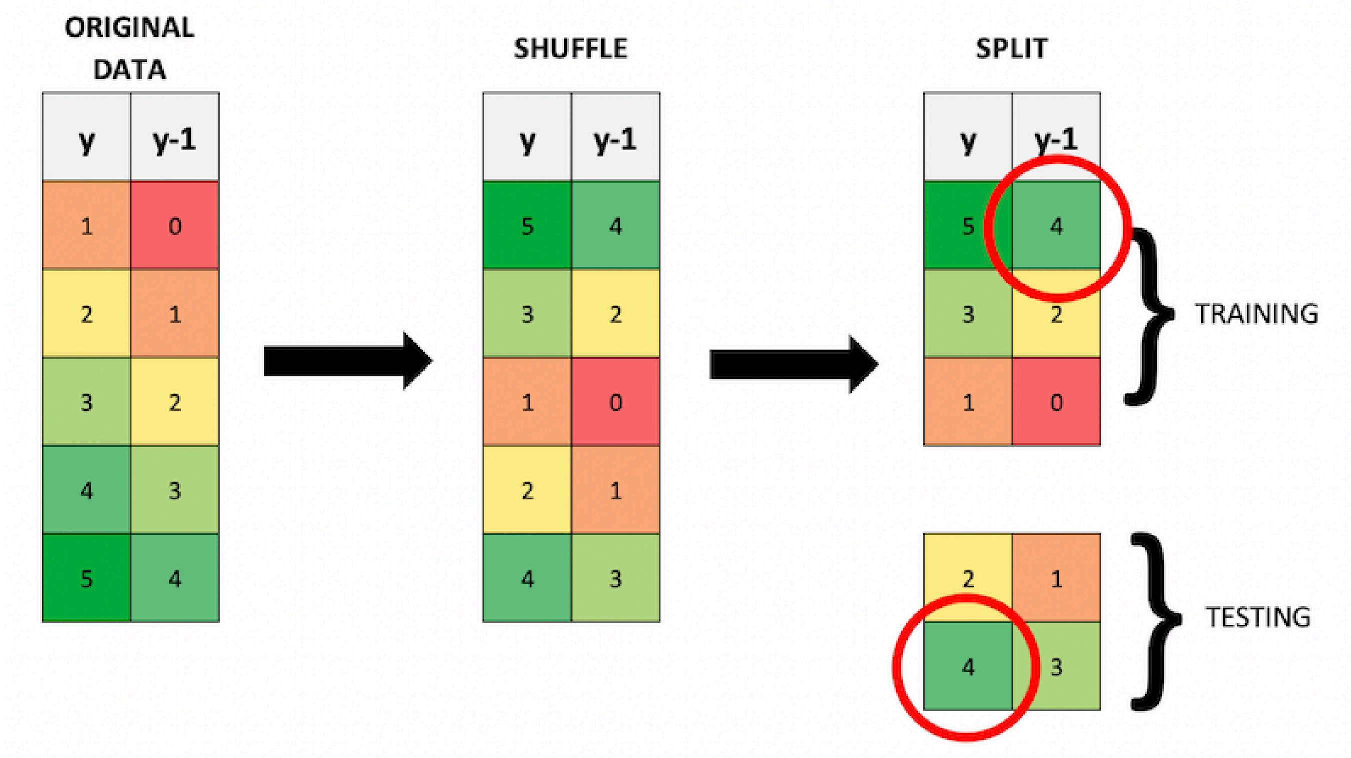
Now you have a tabular dataset with X and y.

- X will be lag_1__Sales, lag_2__Sales, lag_2__Sales
- y will be Sales

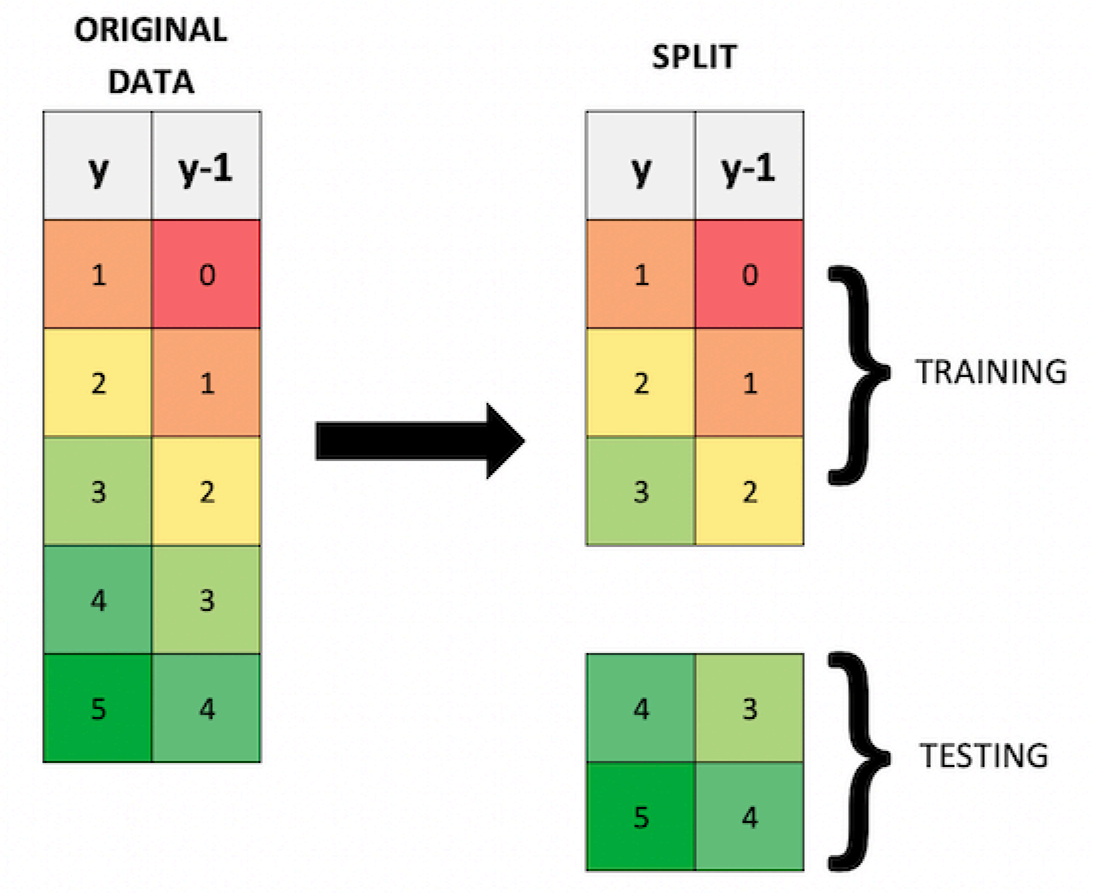
You can start fitting any sklearn model here, but there's a better way to do this with sktime, which we will see later on

1.2. Splitting into train, validation, and test sets

- We *never* split a time series with random shuffling involved, which is different to how we learned to do things with non-time series data
- Why? Because this would disrupt the temporal order of the data and can leak information from the test/validation sets into the training set
- In the example below, after randomly shuffling and splitting our data, we get the same elements appearing in the train and test sets (i.e., we're training our model on something we later want to test it on!)



- Instead we need to split data in a way that preserves the temporal order of the data
- The simplest method is to just take a chunk off the end:

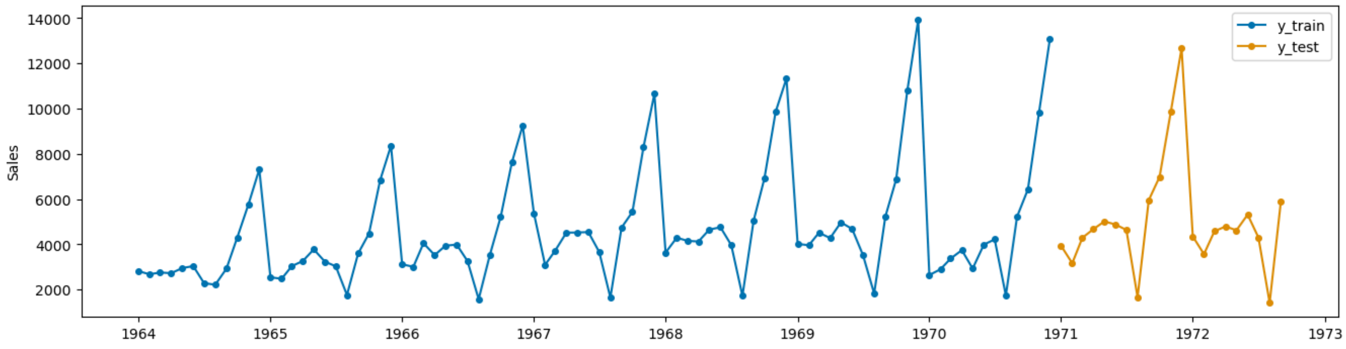


- We can accomplish this kind of splitting with sktime's `temporal_train_test_split()`:

```python
from sktime.split import (
    CutoffSplitter,
    ExpandingWindowSplitter,
    SingleWindowSplitter,
    SlidingWindowSplitter,
    temporal_train_test_split,
)

from sktime.utils.plotting import plot_series
from sktime.forecasting.base import ForecastingHorizon
```

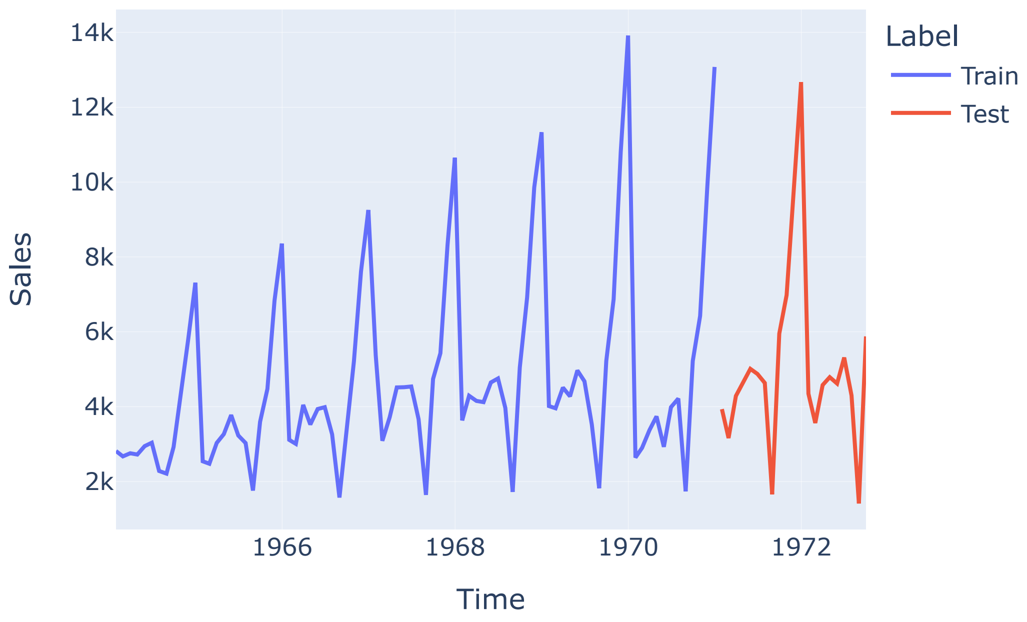
```python
def get_windows(y, cv):
    """Generate windows"""
    train_windows = []
    test_windows = []
    for i, (train, test) in enumerate(cv.split(y)):
        train_windows.append(train)
        test_windows.append(test)
    return train_windows, test_windows
```

```python
y = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
y.index.freq = "M"
y.index = y.index.to_period("M")
y_train, y_test = temporal_train_test_split(y, test_size=0.2)
fig, ax = plot_series(y_train, y_test, labels=["y_train", "y_test"])
```

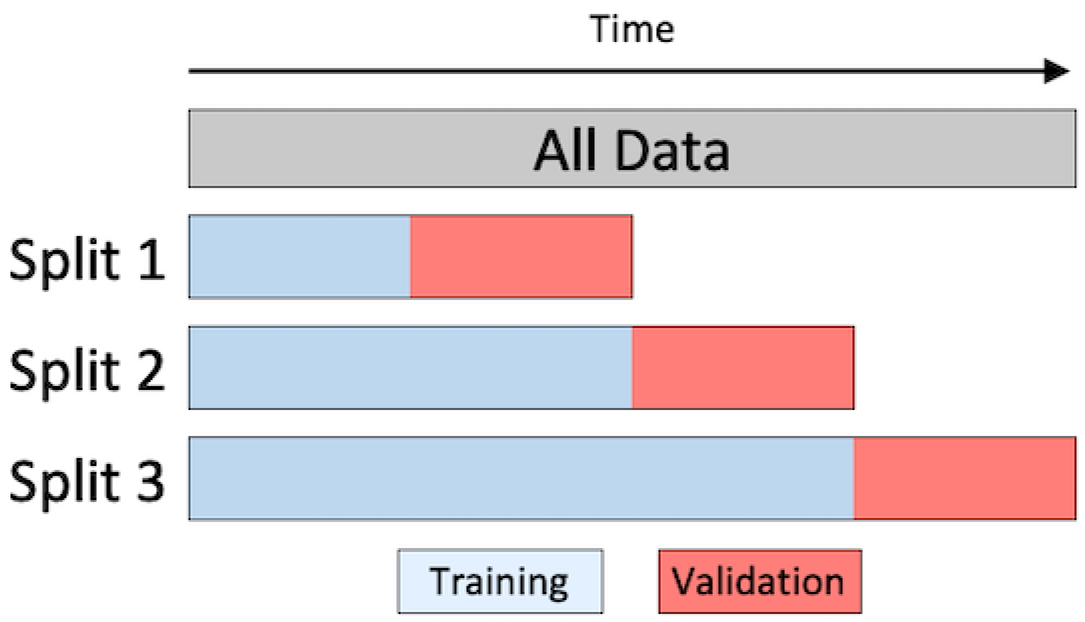


- You could also split off your data with sklearn's `train_test_split(..., shuffle=False)`:

```python
df_train, df_test = train_test_split(df.dropna(), test_size=0.2, shuffle=False
df_train = df_train.assign(Label="Train")
df_test = df_test.assign(Label="Test")

px.line(pd.concat((df_train, df_test)), y="Sales", color="Label")
```



- We now need a validation set to help optimize model hyperparameters
- If we had a lot of data, we could just repeat the above process and split the "Train" set into a "Train" and "Validation" set and be done with it
- But often, we'll want to make more effective use of our data via cross-validation (CV)!
- Time series CV is similar to regular CV, except that we retain the temporal order of our data
- Expanding window approach:



Expanding window split

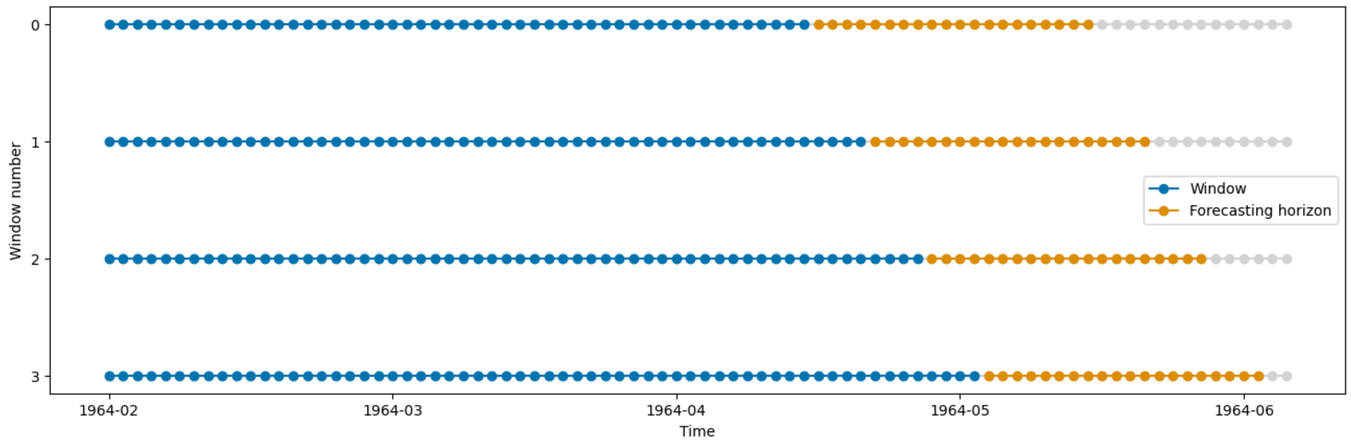
```
# Specify forecast horizon
fh = ForecastingHorizon(np.arange(1,len(y_test)))
```

```
cv = ExpandingWindowSplitter(initial_window=50, fh=fh
                              , step_length=4
                             )

n_splits = cv.get_n_splits(y_train)
print(f"Number of Folds = {n_splits}")
```

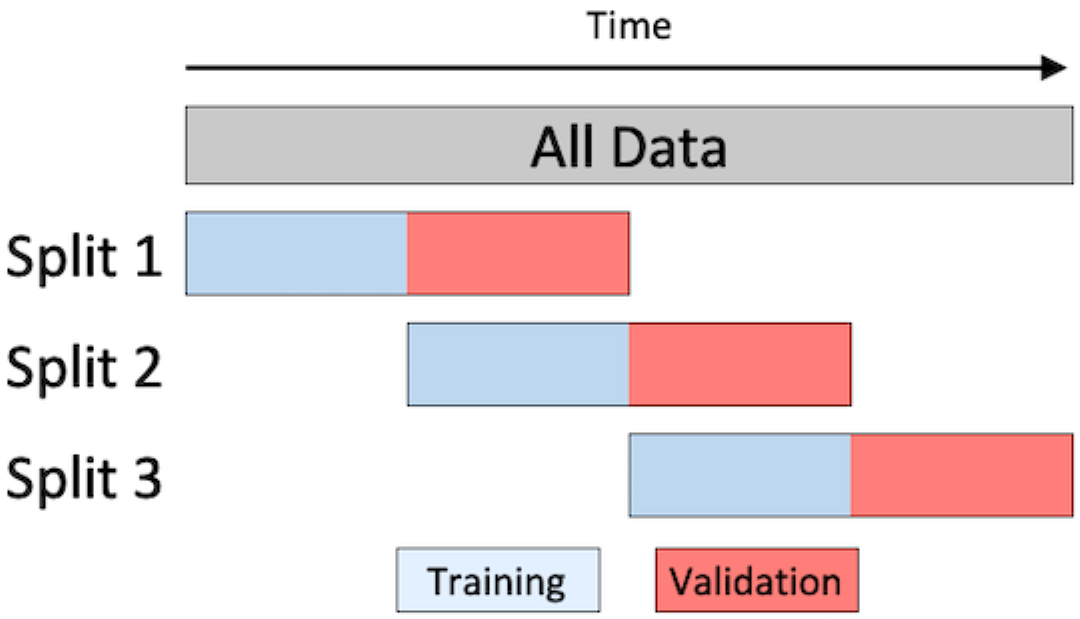
```
Number of Folds = 4
```

```python
from warnings import simplefilter

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from matplotlib.ticker import MaxNLocator

def plot_windows(y, train_windows, test_windows, title=""):
    """Visualize training and test windows"""

    # simplefilter("ignore", category=UserWarning)

    def get_y(length, split):
        # Create a constant vector based on the split for y-axis."""
        return np.ones(length) * split

    n_splits = len(train_windows)
    n_timepoints = len(y)
    len_test = len(test_windows[0])

    train_color, test_color = sns.color_palette("colorblind")[:2]

    fig, ax = plt.subplots(figsize=plt.figaspect(0.3))

    for i in range(n_splits):
        train = train_windows[i]
        test = test_windows[i]

        ax.plot(
            np.arange(n_timepoints), get_y(n_timepoints, i), marker="o", c="li
        )
        ax.plot(
            train,
            get_y(len(train), i),
            marker="o",
            c=train_color,
            label="Window",
        )
        ax.plot(
            test,
            get_y(len_test, i),
            marker="o",
            c=test_color,
            label="Forecasting horizon",
        )
    ax.invert_yaxis()
    ax.yaxis.set_major_locator(MaxNLocator(integer=True))
    ax.set(
        title=title,
        ylabel="Window number",
        xlabel="Time",
        xticklabels=y.index,
    )
    # remove duplicate labels/handles
    handles, labels = [(leg[:2]) for leg in ax.get_legend_handles_labels()]
    ax.legend(handles, labels);
```

```
train_windows, test_windows = get_windows(y_train, cv)
plot_windows(y_train, train_windows, test_windows)
```



- Fixed/sliding window approach:



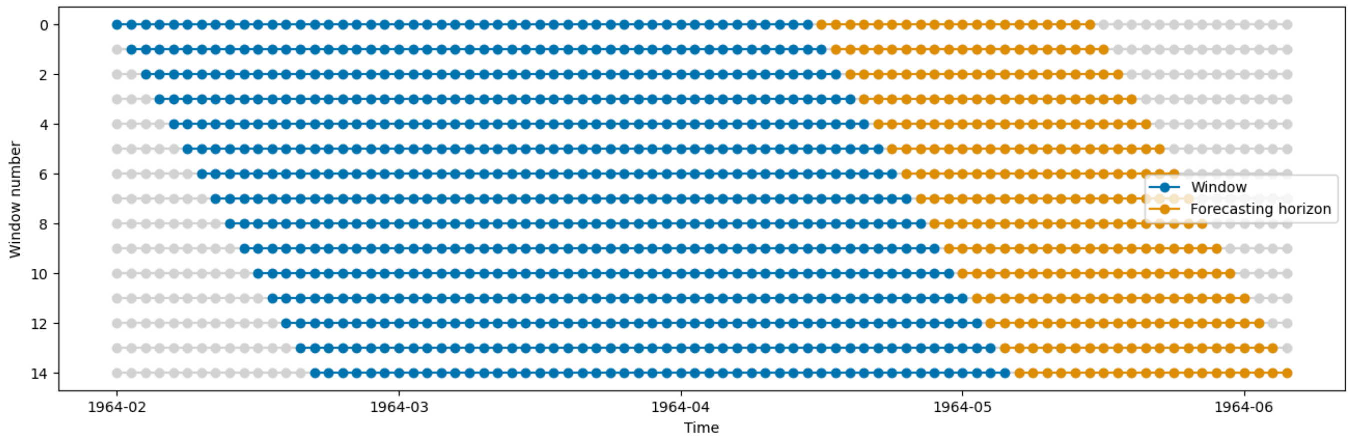
- There is a trade-off to think about: fixed (smaller) windows have less computation but may result in poorer parameter estimates. Expanding windows use more data to estimate parameters but data way in the past may not be as representative as current (and future data)
- I usually go for an expanding window approach and don't think about it too much, but it's good to be aware of

Sliding window split

```
cv = SlidingWindowSplitter(window_length=50, fh=fh
#                               , step_length=5
                           )

n_splits = cv.get_n_splits(y_train)
print(f"Number of Folds = {n_splits}")
```

```
Number of Folds = 15
```

```
train_windows, test_windows = get_windows(y_train, cv)
plot_windows(y_train, train_windows, test_windows)
```



1.3. Fit a model

- Okay, that was fairly straight-forward, let’s fit a model!
- You might be wondering how we implement our time series cross-validation with a time-series model or a sklearn model

Time-series models in sktime

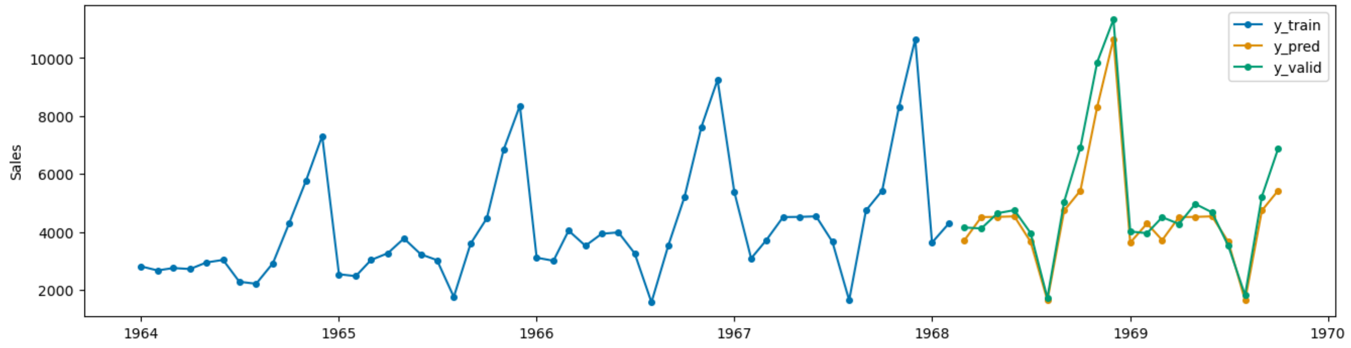
- You can print out a list of built-in forecaster such as ARIMA, ETS, Naive in sktime
- Let’s fit an ARIMA model:

```
# from sktime.registry import all_estimators
# all_estimators("forecaster", as_dataframe=True)
```

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

from sktime.forecasting.model_evaluation import evaluate
from sktime.forecasting.naive import NaiveForecaster
from sktime.forecasting.arima import AutoARIMA

forecaster = NaiveForecaster(strategy="last", sp=12) # seasonal naive
# forecaster = AutoARIMA(sp=12)

results = evaluate(forecaster=forecaster, y=y_train, cv=cv, strategy="refit",
results.head()
```

	test_MeanAbsolutePercentageError	fit_time	pred_time	len_train_window	c
0	0.093002	0.001086	0.012347	50	
1	0.094441	0.001047	0.009062	50	
2	0.100433	0.001020	0.008687	50	
3	0.116621	0.001021	0.008631	50	
4	0.137733	0.001031	0.009394	50	

```python
# Plot the 1st fold
fig, ax = plot_series(
#     y_train,
    results["y_train"].iloc[0],
    results["y_pred"].iloc[0],
    results["y_test"].iloc[0],
    labels=["y_train","y_pred","y_valid"])
```



Fit sk-learn models in sktime

Recall that a typical sklearn ML models requires X and y. With a time-series, we only have y and our predictors X will be the lagged values of y itself. In order to prepare our data for sklearn models, we need to tabluraize the time-series data, meaning that create X and y.

We could do it manually

```
from sktime.transformations.series.lag import Lag
t = Lag(lags=[1,2,3], index_out="original") # I want to create lag 1, lag 2, a
t.get_params()
pd.concat([df,t.fit_transform(df)],axis=1).head()
```

	Sales	lag_1__Sales	lag_2__Sales	lag_3__Sales
Time				
1964-01-31	2815.0	NaN	NaN	NaN
1964-02-29	2672.0	2815.0	NaN	NaN
1964-03-31	2755.0	2672.0	2815.0	NaN
1964-04-30	2721.0	2755.0	2672.0	2815.0
1964-05-31	2946.0	2721.0	2755.0	2672.0

And then you can call a sklearn model and fit.

In sktime, we can use the `make_reduction` function to do this automatically, what it does is:

- Take a regressor (e.g., any sklearn regressor)
- Tabulate the data into X and y using sliding window. For example, if we want to use the last lag1, lag2, lag3 as X, we specify window_length = 3

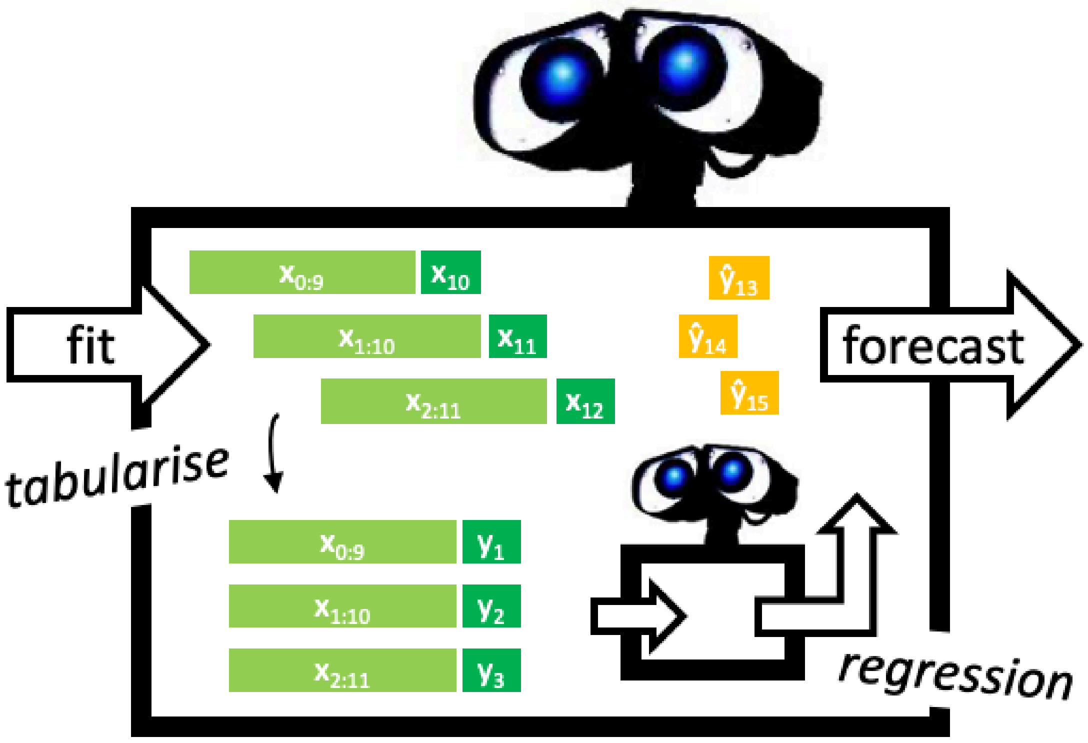
- Fit the regressor to the tabulated data

Note: The `window_length` in `make_reduction` is DIFFERENT from the `window_length` in cross validation `SlidingWindowSplitter`!

- `window_length` in `make_reduction` refers to the **number of lagged values** we use as predictors
- `window_length` in cross validation `SlidingWindowSplitter` is the **length of the training set** in cross-validation

From sktime: https://www.sktime.net/en/stable/examples/01_forecasting.html#3.-Advanced-composition-patterns—pipelines,-reduction,-autoML,-and-more

> *we will define a tabulation reduction strategy to convert a k-nearest neighbors regressor (sklearn KNeighborsRegressor) into a forecaster. The composite algorithm is an object compliant with the sktime forecaster interface (picture: big robot), and contains the regressor as a parameter accessible component (picture: little robot). In fit, the composite algorithm uses a sliding window strategy to tabulate the data, and fit the regressor to the tabulated data (picture: left half). In predict, the composite algorithm presents the regressor with the last observed window to obtain predictions (picture: right half).*



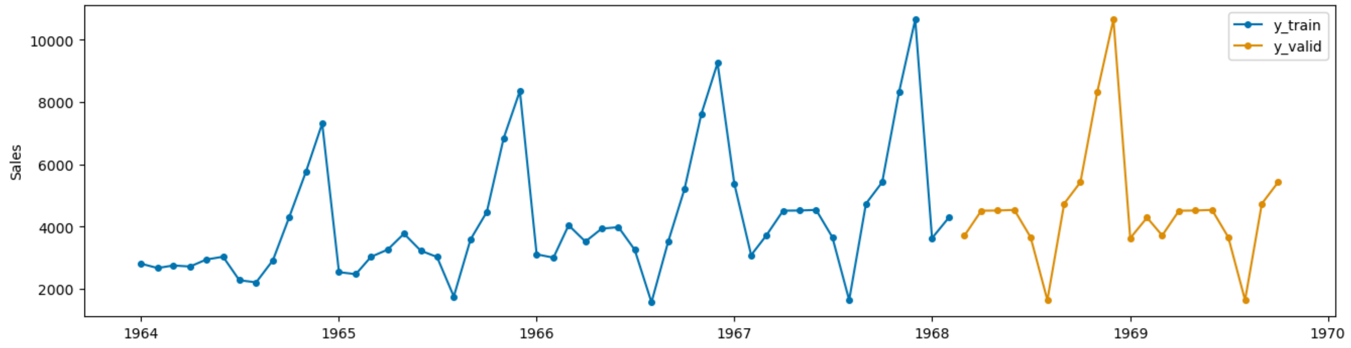
source: https://www.sktime.net/en/stable/examples/01_forecasting.html

```
# ?make_reduction
```

```
from sktime.forecasting.model_evaluation import evaluate
from sklearn.neighbors import KNeighborsRegressor
from sktime.forecasting.compose import make_reduction


regressor = KNeighborsRegressor(n_neighbors=1) # specify a sklearn model
forecaster = make_reduction(regressor,
                            strategy="recursive", # specify the fit strategy
                            window_length=12  # specify the window length to c
                                              # (e.g. windown_length = 12, the
                                              # think of it as X_train = lag 1
                           )
results = evaluate(forecaster=forecaster, y=y_train, cv=cv, strategy="refit",
results.head()
```

	test_MeanAbsolutePercentageError	fit_time	pred_time	len_train_window
0	0.093002	0.003065	0.004614	50
1	0.094441	0.001824	0.004194	50
2	0.100433	0.001823	0.004158	50
3	0.116621	0.001821	0.004143	50
4	0.137733	0.001820	0.004101	50

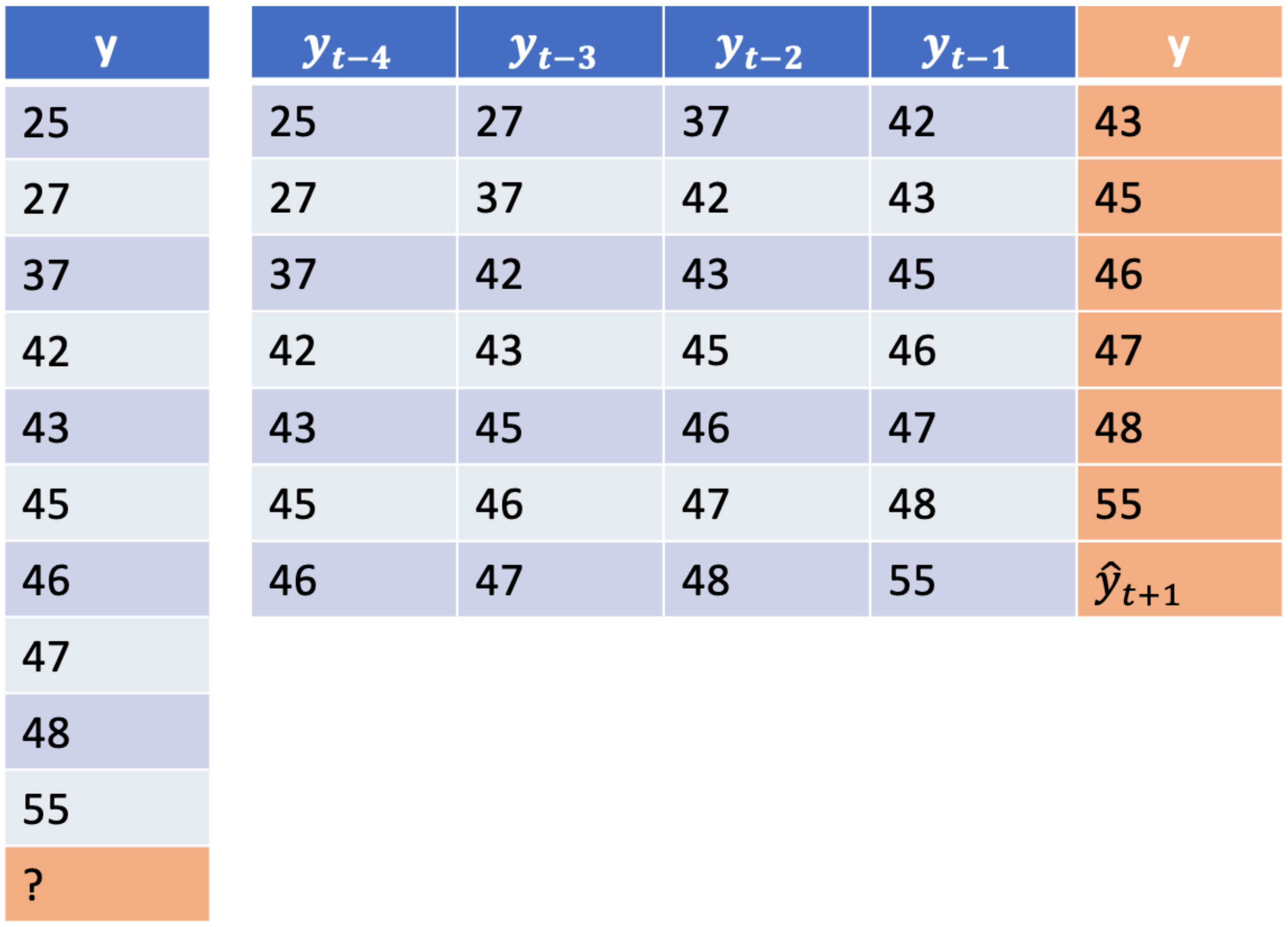
```
# Plot the 1st fold
fig, ax = plot_series(
#     y_train,
    results["y_train"].iloc[0],
    results["y_pred"].iloc[0],
    labels=["y_train","y_valid"])
```



2. Forecasting

2.1. One-step forecasts

- Okay nice! So we've been able to build a model that looks reasonable
- Think about what this model is doing, it takes in lagged values of the response and predicts the next value



y
25
27
37
42
43
45
46
47
48
55
?

y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}	y
25	27	37	42	43
27	37	42	43	45
37	42	43	45	46
42	43	45	46	47
43	45	46	47	48
45	46	47	48	55
46	47	48	55	$\hat{y}_{t+1}$

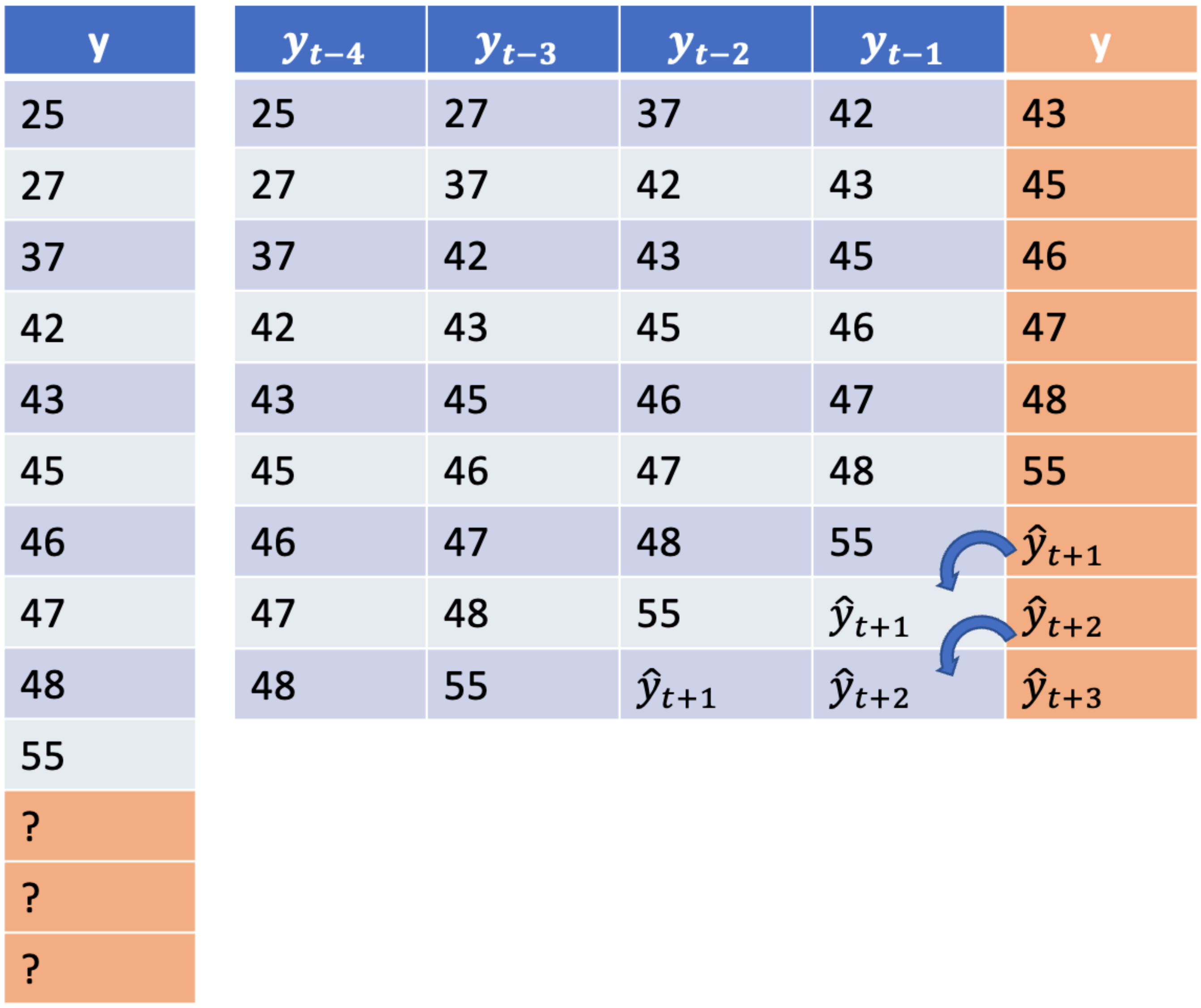
- But what if we did want to forecast more than just one step into the future? We don’t have future observations to use as input features, so how do we do it?

2.2. Multi-step forecasts

- Say we have monthly data and we want to forecast our time series 12 months into the future
- There are 4 main ways we can approach this problem:
  1. Recursive forecasts
  2. Direct forecast
  3. Hybrid
  4. Multi-output

2.2.1. Recursive

- In the recursive approach, as we forecast into the future, the prediction at time `t` will become an input to our model for predicting at time `t+1` and so on and so forth
- So, we need a **for** loop!

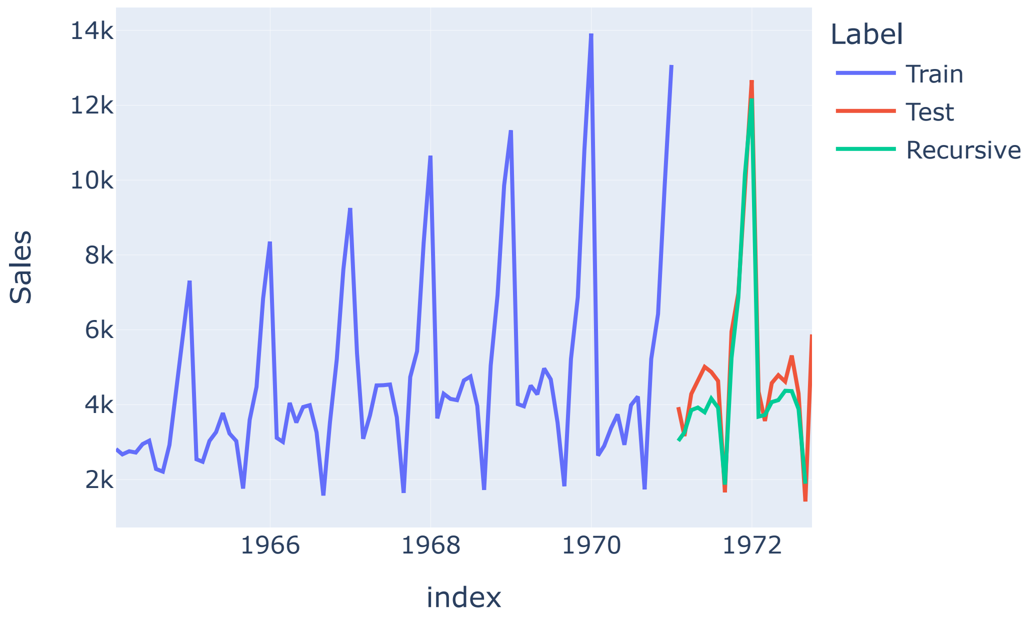


y
25
27
37
42
43
45
46
47
48
55
?
?
?

y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}	y
25	27	37	42	43
27	37	42	43	45
37	42	43	45	46
42	43	45	46	47
43	45	46	47	48
45	46	47	48	55
46	47	48	55	$\hat{y}_{t+1}$
47	48	55	$\hat{y}_{t+1}$	$\hat{y}_{t+2}$
48	55	$\hat{y}_{t+1}$	$\hat{y}_{t+2}$	$\hat{y}_{t+3}$

When can specify the fit strategy (e.g., recursive) in the `make_reduction` function

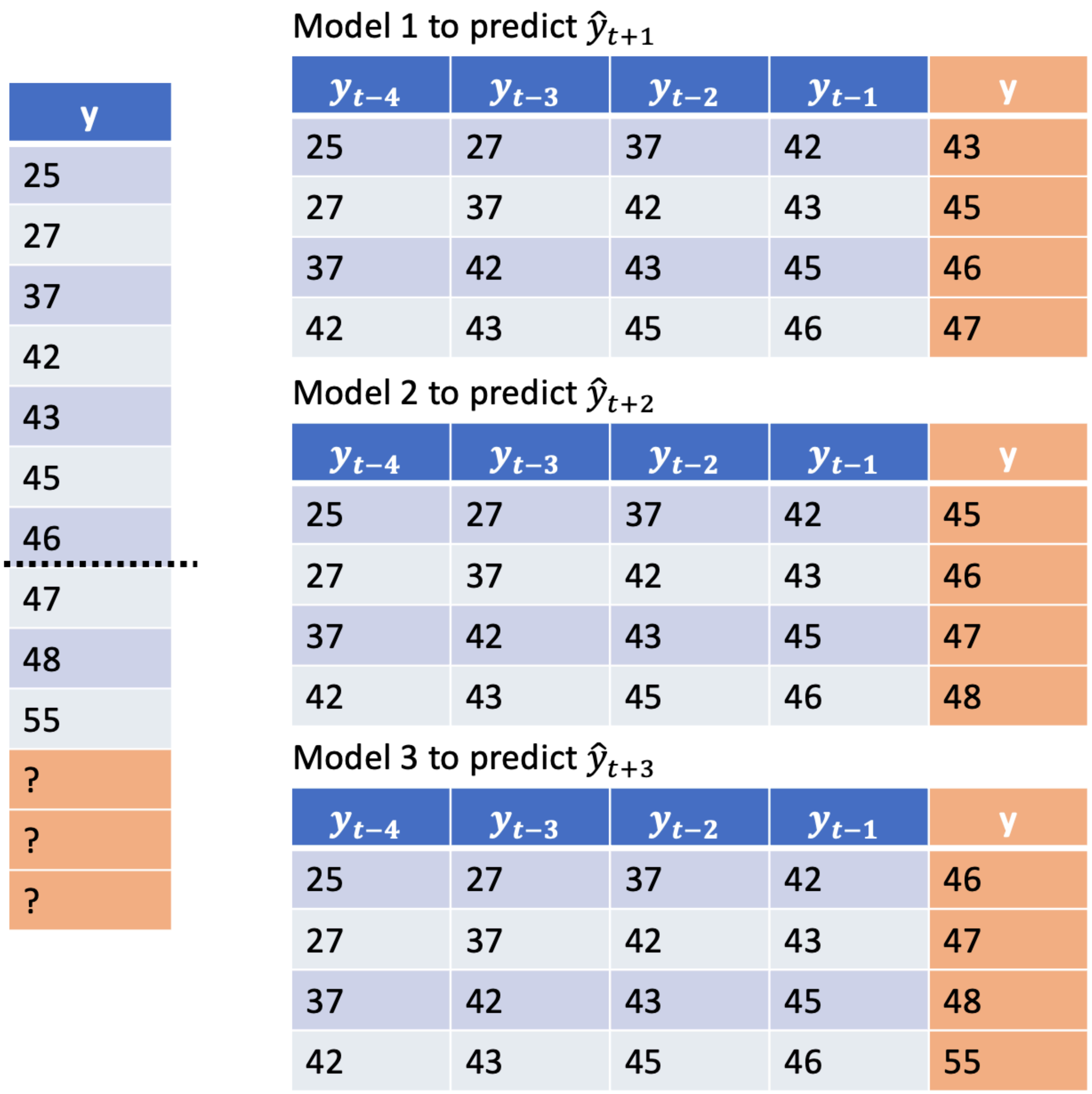
```
regressor = RandomForestRegressor() # specify a sklearn model
forecaster = make_reduction(regressor,
                            strategy="recursive", # specify the fit strategy
                            window_length=12  # specify the window length to c
                                              # (e.g. windown_length = 12, the
                                              # think of it as X_train = lag 1
                           )
y = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
y.index.freq = "M"
y_train, y_test = temporal_train_test_split(y, test_size=0.2)

forecaster.fit(y_train, fh=fh)
y_pred_recursive = forecaster.predict(fh)

y_pred_recursive['Label'] = 'Recursive'
y_train['Label'] = 'Train'
y_test['Label'] = 'Test'


px.line(pd.concat([y_train, y_test, y_pred_recursive]), y="Sales", color="Labe
```



- Although errors accummulate in this approach as the forecast horizon expands, it is the most popular approach I see because of its ease of use

2.2.2. Direct

- In the direct approach we make a model for every forecast horizon we wish to predict
- For example, if we want to predict the next 12 years worth of Sales, we need 12 models (and 12 datasets)
  1. A model to predict `t+1`
  2. A model to predict `t+2`
  3. A model to predict `t+3`
  4. etc…
- Same features but different target variable for each forecast step



y
25
27
37
42
43
45
46
47
48
55
?
?
?

Model 1 to predict $\hat{y}_{t+1}$

y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}	y
25	27	37	42	43
27	37	42	43	45
37	42	43	45	46
42	43	45	46	47

Model 2 to predict $\hat{y}_{t+2}$

y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}	y
25	27	37	42	45
27	37	42	43	46
37	42	43	45	47
42	43	45	46	48

Model 3 to predict $\hat{y}_{t+3}$

y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}	y
25	27	37	42	46
27	37	42	43	47
37	42	43	45	48
42	43	45	46	55

By training a model to predict the value at time 't' using the values at t-2, t-3, t-4 etc., you are effectively training the model to forecast a value 2 time points into the future. The same logic would apply for the other models (training a model to predict the value at time 't' using the values at t-3, t-4 etc. is effectively training the model to predict 3 time steps into the future). Since we don't have new inputs, we need to train models using further lags to replicate that process. For example, when predicting 2-step ahead (y_{t+2}), we don't have access to t+1. That's equivalent of predicting y_t from y_{t-2} (we purposely omit y_{t-1}).

You can specify the direct strategy in `make_reduction`

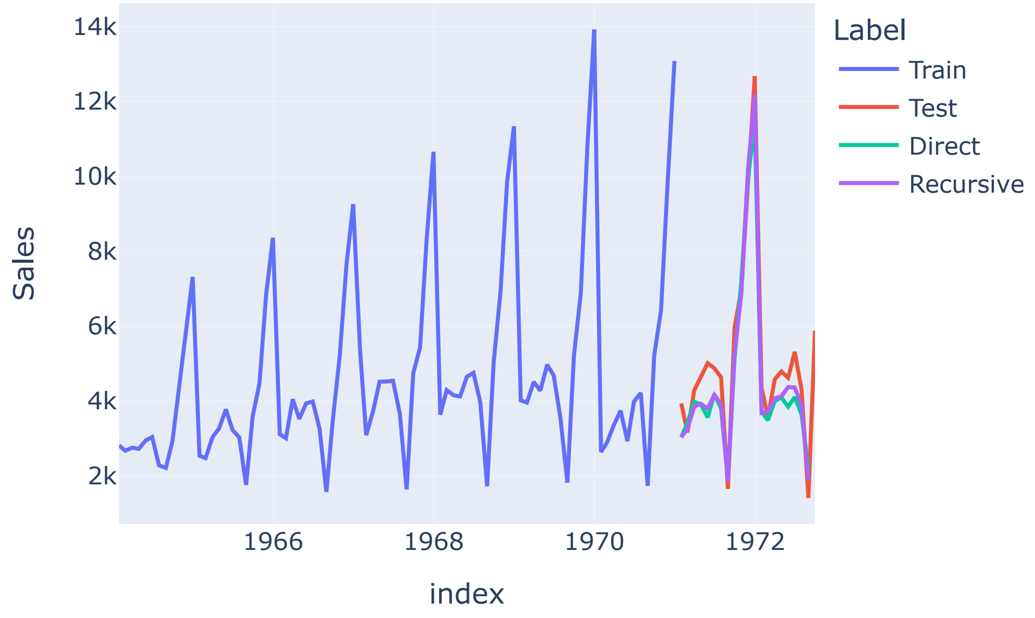
```
regressor = RandomForestRegressor() # specify a sklearn model
forecaster = make_reduction(regressor,
                            strategy="direct", # specify the fit strategy
                            window_length=12  # specify the window length to c
                                              # (e.g. windown_length = 12, the
                                              # think of it as X_train = lag 1
                           )

y = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
y.index.freq = "M"
y_train, y_test = temporal_train_test_split(y, test_size=0.2)

forecaster.fit(y_train, fh=fh)
y_pred_direct = forecaster.predict(fh)
```

```
y_pred_direct['Label'] = 'Direct'
y_pred_recursive['Label'] = 'Recursive'
y_train['Label'] = 'Train'
y_test['Label'] = 'Test'

pd.concat([y_train, y_test, y_pred_direct, y_pred_recursive])
```

	Sales	Label
1964-01-31	2815.00	Train
1964-02-29	2672.00	Train
1964-03-31	2755.00	Train
1964-04-30	2721.00	Train
1964-05-31	2946.00	Train
...	...	...
1972-04-30	4122.22	Recursive
1972-05-31	4370.86	Recursive
1972-06-30	4362.10	Recursive
1972-07-31	3881.96	Recursive
1972-08-31	1888.44	Recursive

145 rows × 2 columns

```
px.line(pd.concat([y_train, y_test, y_pred_direct, y_pred_recursive]), y="Sale
```



- We get similar answers to before which is nice
- The direct method is another popular approach. It has a much higher computational and maintenance cost (needing one model for every forecast horizon), but it doesn’t propagate uncertainty like the recursive method.

2.2.3. Recursive-Direct Hybrid

- This approach is a combination of the recursive and direct strategies
- A separate model is developed for each forecast horizon, but each model can incorporate the predictions of the previous timestep:

$$\hat{y}_{t+1} = model_1(y_t, y_{t-1}, y_{t-2}, \dots)$$

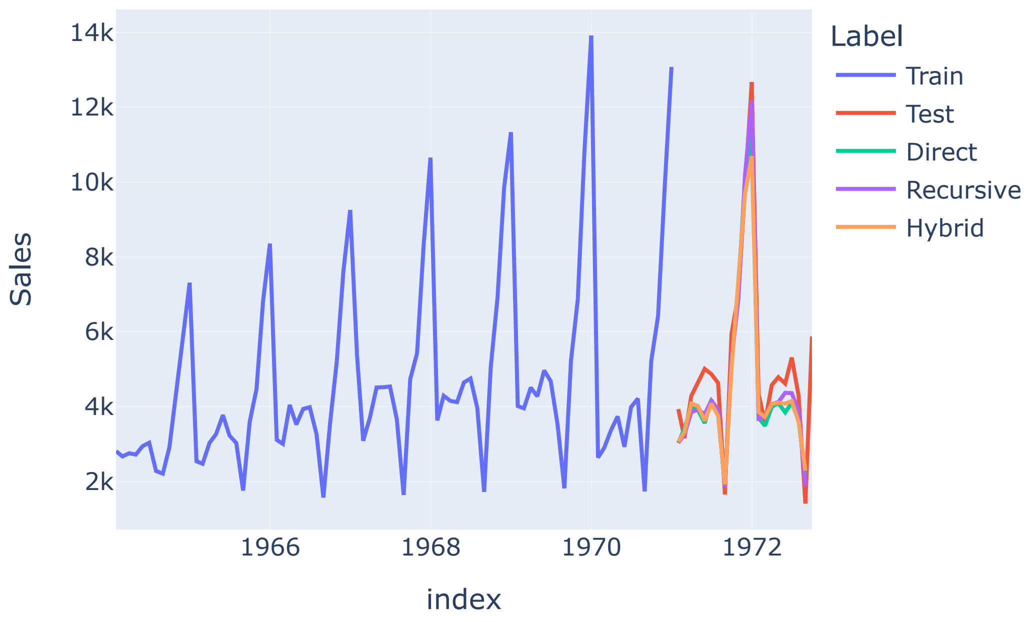
$$\hat{y}_{t+2} = model_2(\hat{y}_{t+1}, y_t, y_{t-1}, \dots)$$

$$\hat{y}_{t+3} = model_3(\hat{y}_{t+2}, \hat{y}_{t+1}, y_t, \dots)$$

- This approach obviously takes more care and thought to set up, and I find it doesn’t often improve on either of the individual approaches

```
y = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
y.index.freq = "M"
y_train, y_test = temporal_train_test_split(y, test_size=0.2)
regressor = RandomForestRegressor() # specify a sklearn model
forecaster = make_reduction(regressor,
                            strategy="dirrec", # specify the fit strategy
                            window_length=12  # specify the window length to c
                                              # (e.g. windown_length = 12, the
                                              # think of it as X_train = lag 1
                           )
forecaster.fit(y_train, fh=fh)
y_pred_hybrid = forecaster.predict(fh)
```

```
y_pred_direct['Label'] = 'Direct'
y_pred_recursive['Label'] = 'Recursive'
y_pred_hybrid['Label'] = 'Hybrid'
y_train['Label'] = 'Train'
y_test['Label'] = 'Test'
px.line(pd.concat([y_train, y_test, y_pred_direct, y_pred_recursive, y_pred_hy
```

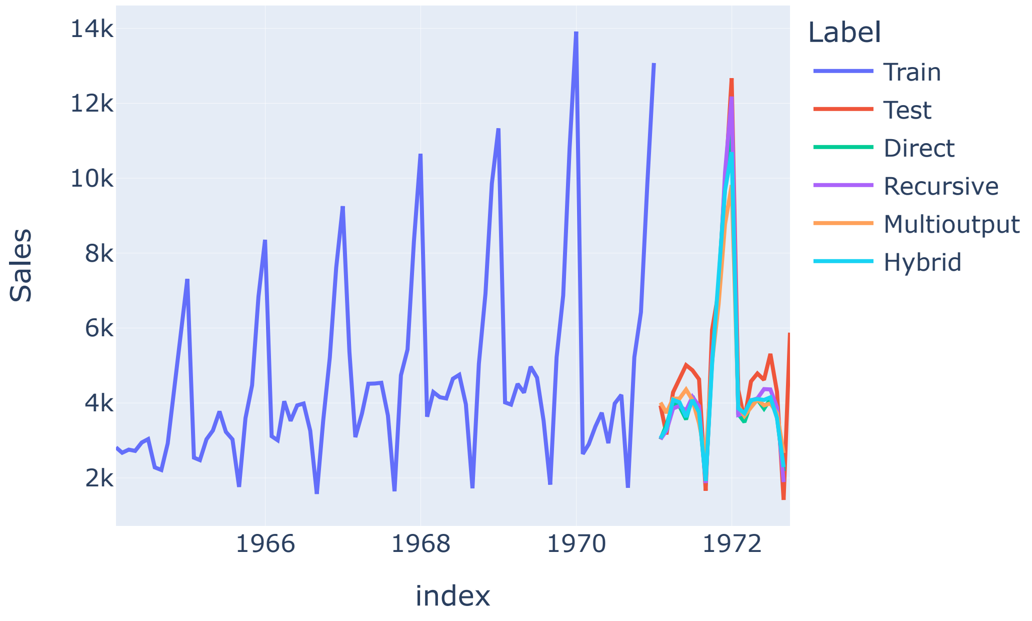


2.2.4. Multi-output

- In this approach, we develop a model that makes multiple predictions at once
- For example, a model that spits out predictions for `t+1` and `t+2`
- Multi-output strategies are most useful when dealing with multivariate time series data

```
y = pd.read_csv("data/champagne.csv", index_col=0, parse_dates=True)
y.index.freq = "M"
y_train, y_test = temporal_train_test_split(y, test_size=0.2)
regressor = RandomForestRegressor() # specify a sklearn model
forecaster = make_reduction(regressor,
                            strategy="multioutput", # specify the fit strategy
                            window_length=12  # specify the window length to c
                                              # (e.g. windown_length = 12, the
                                              # think of it as X_train = lag 1
                           )
forecaster.fit(y_train, fh=fh)
y_pred_multioutput = forecaster.predict(fh)
```

```
y_pred_direct['Label'] = 'Direct'
y_pred_recursive['Label'] = 'Recursive'
y_pred_hybrid['Label'] = 'Hybrid'
y_pred_multioutput['Label'] = 'Multioutput'
y_train['Label'] = 'Train'
y_test['Label'] = 'Test'
px.line(pd.concat([y_train, y_test, y_pred_direct, y_pred_recursive, y_pred_mu
```



3. Feature pre-processing and engineering

3.1. Pre-processing

- Common feature pre-processing techniques:
  1. Coerce to stationary (through differencing or transforms)
  2. Smoothing (usually with a moving average)
  3. Impute missing value (e.g., linear interpolation)
  4. Removing outliers (we'll talk more about this next lecture)
- In terms of 1, many ML models (like tree based models) will find it difficult to model non-stationary data. While we can often try to account for seasonal cycles by using lagged

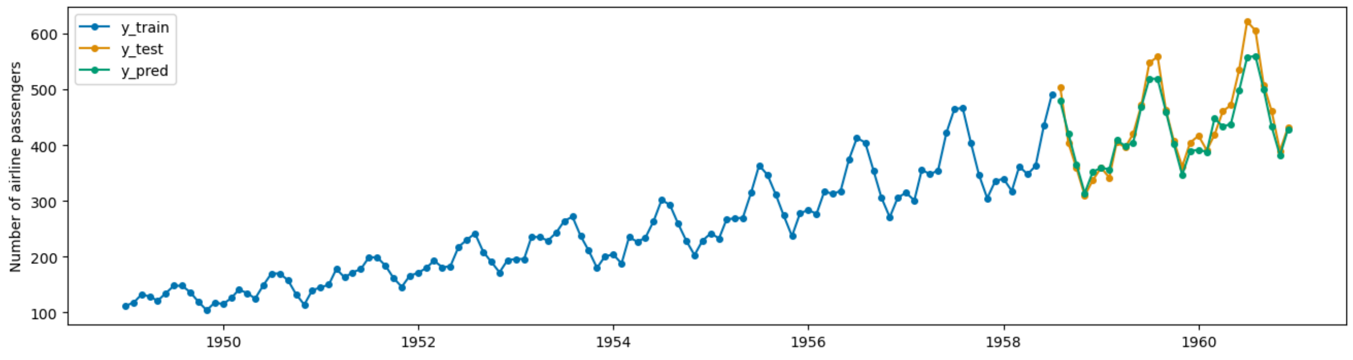
features, or time stamp features (more on that in a bit), trends can be harder to model. Linear models may do better in these situations, but I usually just make my data stationary.

- Below is an example of using the `TransformedTargetForecaster` function in sktime to create a data pre-processing pipeline by applying deseasonalize and detrend the data before fitting a KNN regression

```
# imports necessary for this chapter
from sktime.datasets import load_airline
from sktime.forecasting.base import ForecastingHorizon
from sktime.forecasting.model_selection import temporal_train_test_split
from sktime.performance_metrics.forecasting import mean_absolute_percentage_er
from sktime.utils.plotting import plot_series

# load airline data
y = pd.DataFrame(load_airline())

# split train test
y_train, y_test = temporal_train_test_split(y, test_size=0.2)

# set forecast horizon
fh = ForecastingHorizon(y_test.index, is_relative=False)
```

```
from sktime.forecasting.arima import ARIMA
from sktime.forecasting.compose import TransformedTargetForecaster
from sktime.transformations.series.detrend import Deseasonalizer
from sktime.forecasting.trend import PolynomialTrendForecaster
from sktime.transformations.series.detrend import Detrender

# create a transformer pipeline
forecaster = TransformedTargetForecaster(
    [
        ("deseasonalize", Deseasonalizer(model="multiplicative", sp=12)),
        ("detrend", Detrender(forecaster=PolynomialTrendForecaster(degree=1)))
#         ("forecast", ARIMA()),
        ("forecast", make_reduction(
                KNeighborsRegressor(),
                window_length=15,
                strategy="recursive",
            )
        )
    ]
)

# This can also be done using the *
# forecaster = Deseasonalizer(model="multiplicative", sp=12) * ARIMA()

# Fit
forecaster.fit(y_train)

# Predict
y_pred = forecaster.predict(fh)

# Plot
plot_series(y_train, y_test, y_pred, labels=["y_train", "y_test", "y_pred"])

# Evaluate
mean_absolute_percentage_error(y_test, y_pred, symmetric=False)
```

```
0.03852759736800383
```



3.2. Feature Engineering

- Common feature engineering:
    1. Lagging response/features (we've seen this already)
    2. Adding time stamps (e.g., month of year, day of week, am/pm, etc.)
    3. Rolling statistics (e.g., rolling mean, median, variance, etc)
- Let's do a quick example of adding a time stamp as a feature
- I'll add the month of an observation as a feature and see how it changes predictions:

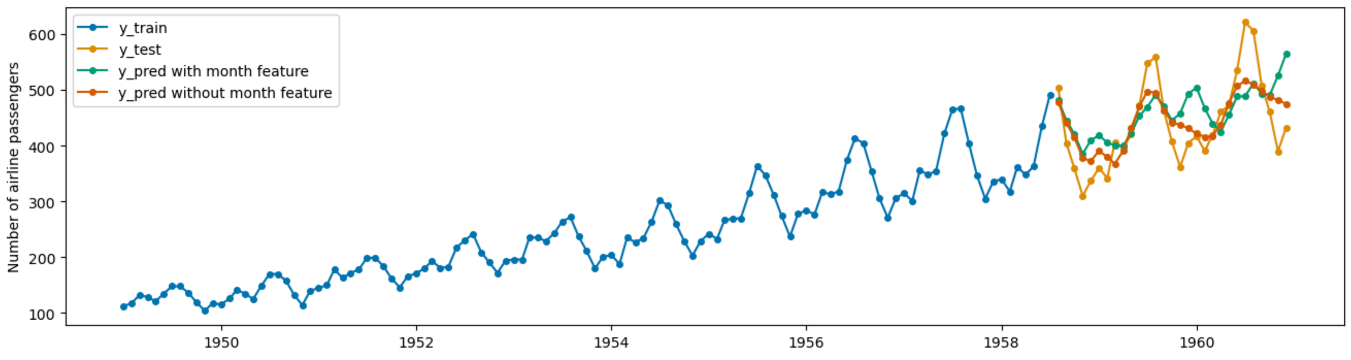
```
X_train = pd.DataFrame(index=y_train.index)
X_train["Month"] = X_train.index.month
X_train
```

	Month
1949-01	1
1949-02	2
1949-03	3
1949-04	4
1949-05	5
...	...
1958-03	3
1958-04	4
1958-05	5
1958-06	6
1958-07	7

115 rows × 1 columns

```
from sktime.forecasting.arima import ARIMA
from sktime.forecasting.compose import TransformedTargetForecaster
from sktime.transformations.series.detrend import Deseasonalizer
from sktime.forecasting.trend import PolynomialTrendForecaster
from sktime.transformations.series.detrend import Detrender
from sklearn.linear_model import LinearRegression
# create a forecaster
forecaster = make_reduction(
                LinearRegression(),
                window_length=10,
                strategy="recursive",
            )
# Fit
forecaster.fit(y_train,X=X_train)

# Predict
y_pred = forecaster.predict(X=X_train, fh=fh)

# Fit
forecaster.fit(y_train)

# Predict
y_pred2 = forecaster.predict(fh=fh)

# Plot
plot_series(y_train, y_test, y_pred, y_pred2, labels=["y_train", "y_test", "y_
```

```
(<Figure size 1600x400 with 1 Axes>,
 <Axes: ylabel='Number of airline passengers'>)
```



- Didn't make much of a difference there

3.3. Hyperparameter optimization

```python
from sklearn.neighbors import KNeighborsRegressor

from sktime.forecasting.compose import make_reduction
from sktime.forecasting.model_selection import (
    ForecastingGridSearchCV,
    SlidingWindowSplitter,
)

# create a forecaster
regressor = KNeighborsRegressor()
forecaster = make_reduction(regressor, strategy="recursive")
```

You can print out the list of tunable parameters

```python
forecaster.get_params()
```

```
{'estimator': KNeighborsRegressor(),
 'pooling': 'local',
 'transformers': None,
 'window_length': 10,
 'estimator__algorithm': 'auto',
 'estimator__leaf_size': 30,
 'estimator__metric': 'minkowski',
 'estimator__metric_params': None,
 'estimator__n_jobs': None,
 'estimator__n_neighbors': 5,
 'estimator__p': 2,
 'estimator__weights': 'uniform'}
```

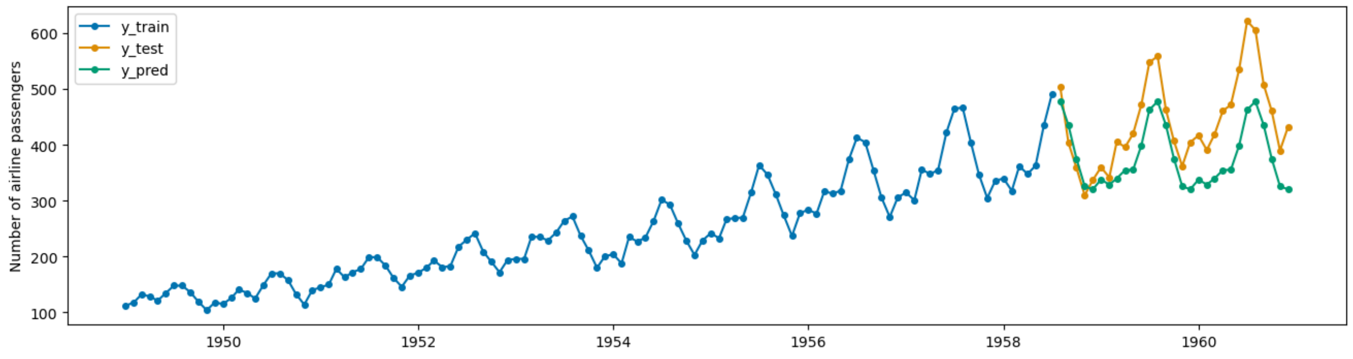
Let’s try to tune the window_length and the number of neighbors

```
# create parameters grid
param_grid = {"window_length": [7, 12, 15], "estimator__n_neighbors": np.arang

# Create cross-validation using sliding window
cv = SlidingWindowSplitter(initial_window=int(len(y_train) * 0.8), window_leng

# grid search with cross valudation
gscv = ForecastingGridSearchCV(forecaster, cv=cv, param_grid=param_grid)

# fit the best model
gscv.fit(y_train)

# predict
y_pred = gscv.predict(fh)

# plot
plot_series(y_train, y_test, y_pred, labels=["y_train", "y_test", "y_pred"])

# MAPE
mean_absolute_percentage_error(y_test, y_pred, symmetric=False)
```

```
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
```

```
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
/Users/katieburak/miniforge3/envs/mds574/lib/python3.9/site-packages/gluonts/j
  warnings.warn(
```

```
0.14410664714651514
```



```
gscv.best_params_
```

```
{'estimator__n_neighbors': 2, 'window_length': 12}
```

You can also print out the results from gridsearch

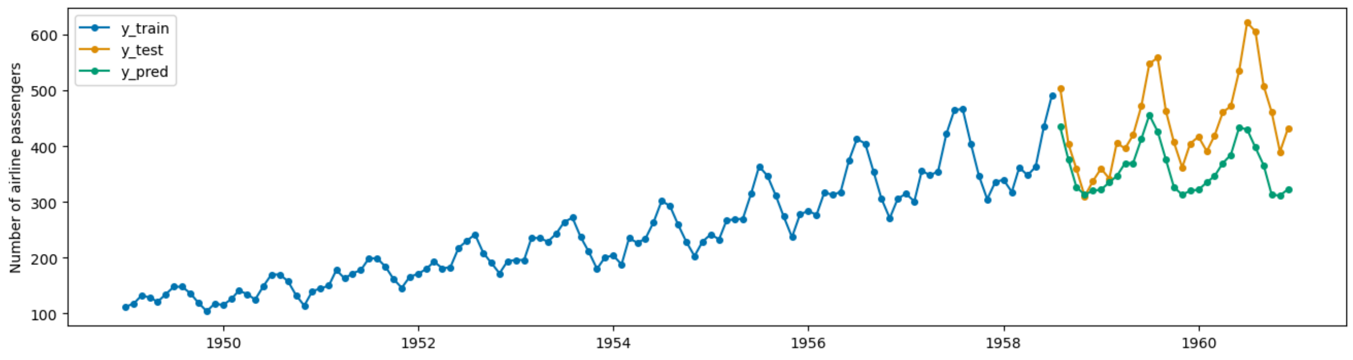
```
# gscv.cv_results_
```

3.4. Model selection

In most cases, we want to try a wide range of models and select the ‘best’ model using cross-validation. In sktime, this can be done with `MultiplexForecaster`, which is constructed with a named list `forecasters`. It has a single hyper-parameter, `selected_forecaster`, which can be set to the name of any forecaster in `forecasters`

```python
from sktime.forecasting.compose import MultiplexForecaster
from sktime.forecasting.exp_smoothing import ExponentialSmoothing
from sktime.forecasting.naive import NaiveForecaster

forecaster = MultiplexForecaster(
    forecasters=[
        ("naive", NaiveForecaster(strategy="last", sp=12)),
        ("knn", make_reduction(KNeighborsRegressor(), strategy="recursive")),

    ]
)

forecaster_param_grid = {"selected_forecaster": ["knn", "naive"]}

cv = SlidingWindowSplitter(initial_window=int(len(y_train) * 0.5), window_leng
gscv = ForecastingGridSearchCV(forecaster, cv=cv, param_grid=forecaster_param_

gscv.fit(y_train)
y_pred = gscv.predict(fh)
plot_series(y_train, y_test, y_pred, labels=["y_train", "y_test", "y_pred"])
mean_absolute_percentage_error(y_test, y_pred, symmetric=False)
```

```
0.16885592056672907
```



```python
gscv.best_params_
```

```
{'selected_forecaster': 'knn'}
```

There are a lot of decisions that we can optimize:

- Which model to use (ETS, ARIMA, naive, KNN, RF, etc..)
- Which parameter in each model (e.g., k in KNN, p,d,q in ARIMA, additive/multiplicative...)
- Window length (how many lagged values to use)
- Fit strategy (recursive, direct, hybrid)

4. Multivariate time series

- Most of the concepts we just discussed extend to the multivariate case
- You just need to use a model that can produce multiple outputs (e.g., decision tree, random forest, knn, neural network, etc.)
- Let's do a quick example with two time series of pressure and temperature measurements in Beijing from 2010-2014:

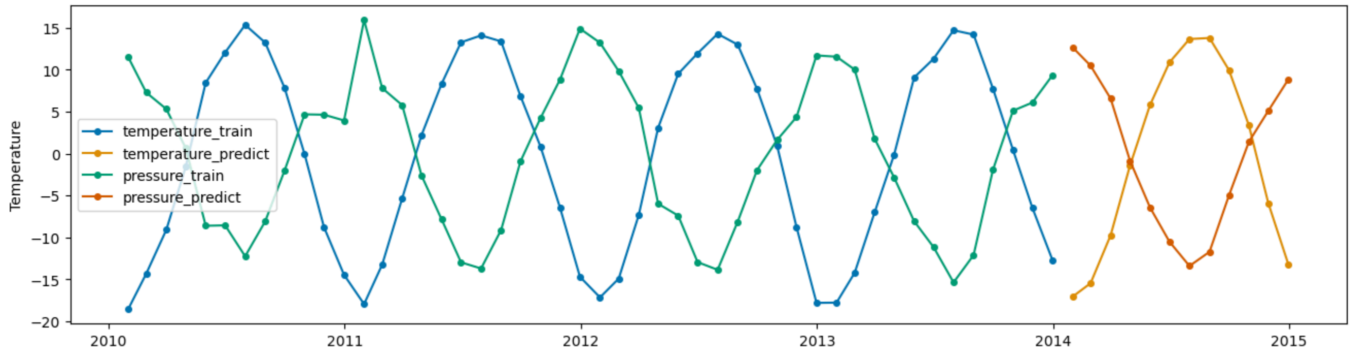
```
y = (pd.read_csv("data/pollution.csv", index_col=0, parse_dates=True, usecols=
        .resample("M")
        .mean()
     )
y -= y.mean()

y_train, y_test = temporal_train_test_split(y, train_size=0.8)
```

```
y_train.head()
```

	Temperature	Pressure
Time		
2010-01-31	-18.543265	11.515210
2010-02-28	-14.303249	7.282587
2010-03-31	-9.087620	5.317629
2010-04-30	-1.573686	0.675246
2010-05-31	8.451359	-8.597693

```
fh = ForecastingHorizon(y_test.index, is_relative=False)
forecaster = make_reduction(KNeighborsRegressor(), strategy="recursive")
forecaster.fit(y_train)
y_pred = forecaster.predict(fh=fh)
plot_series(y_train.Temperature, y_pred.Temperature,y_train.Pressure, y_pred.P
            labels=["temperature_train", "temperature_predict", "pressure_trai
mean_absolute_percentage_error(y_test, y_pred, symmetric=False)
```

```
0.5691676966274577
```



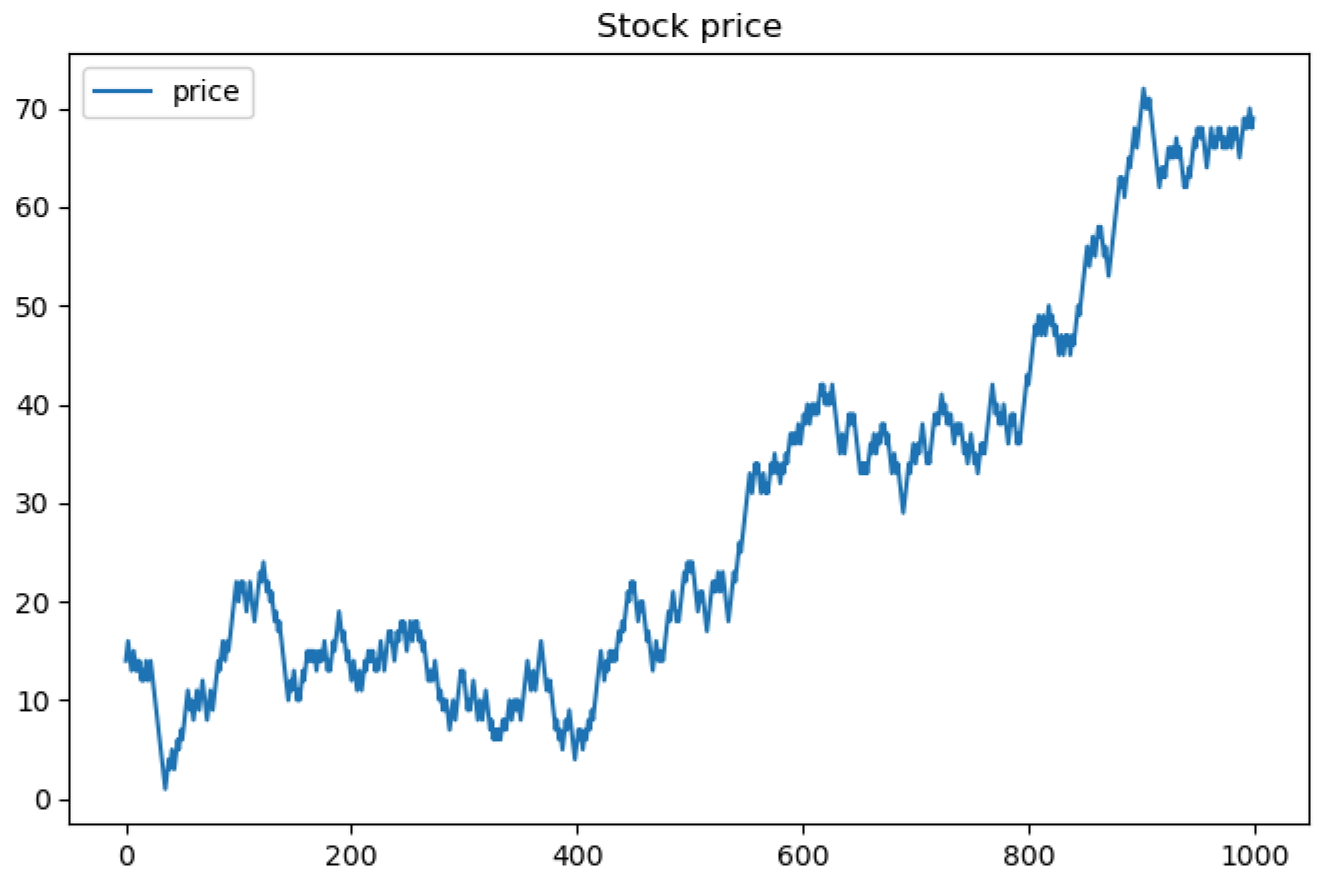
- Not too shabby!

5. Random walk

```
from random import seed
from random import random
from pandas.plotting import autocorrelation_plot
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
    movement = -1 if random() < 0.5 else 1
    value = random_walk[i-1] + movement
    random_walk.append(value)
stock_index = pd.DataFrame({'price': random_walk })
stock_index.price = stock_index.price + 15
```

```
stock_index.plot(figsize=(8,5),title='Stock price')
```

```
<Axes: title={'center': 'Stock price'}>
```

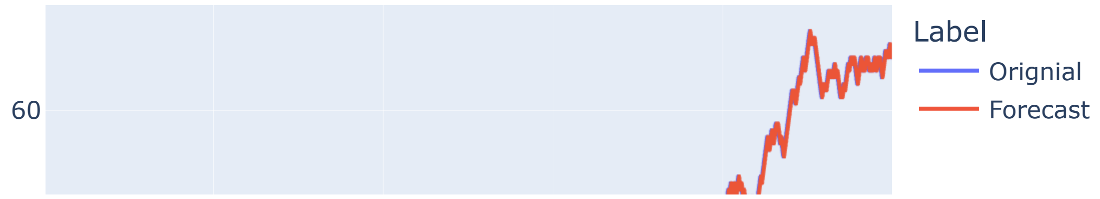


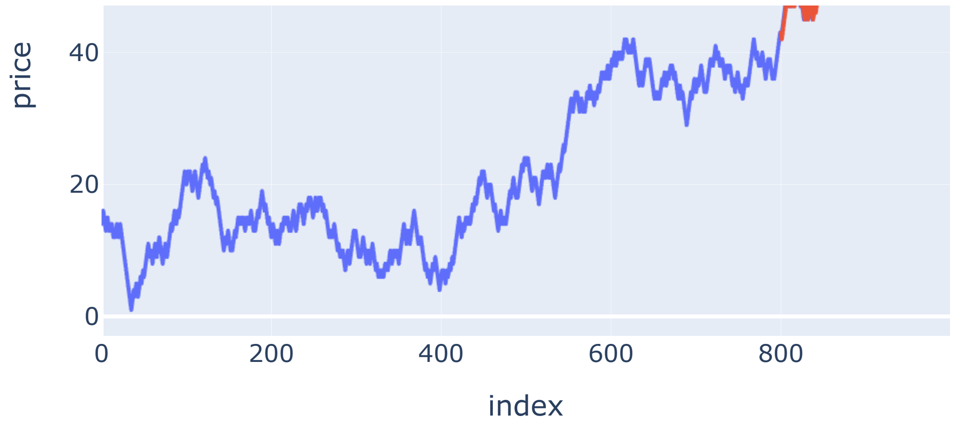
Let’s say I fit a machine learning model

Do you think this is a good model?

```
stock_index['Label'] = 'Orignial'

# Assume a ML model learned to predict future value based on current value (na
stock_index_naive = stock_index.shift(1)
stock_index_naive['Label'] = 'Forecast'
stock_index_naive.price = stock_index_naive.price
```

```
# Plot
px.line(pd.concat((stock_index, stock_index_naive.tail(200))), y="price", colo
```

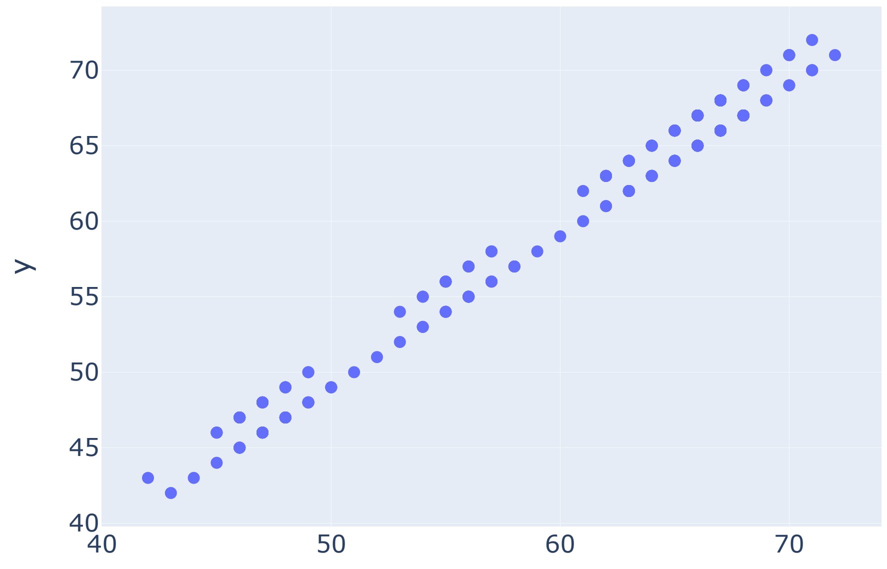




Let’s look at R-Squared

```
from scipy.stats import pearsonr
corr, _ = pearsonr(stock_index.tail(200).price,stock_index_naive.tail(200).pri
print('R-squared: %.3f' % corr)

from sklearn.metrics import mean_squared_error
mse =mean_squared_error(stock_index.tail(200).price,stock_index_naive.tail(200

print('Mean squared error: %.3f' % mse )

px.scatter(x=stock_index.tail(200).price, y = stock_index_naive.tail(200).pric
```

```
R-squared: 0.993
Mean squared error: 1.000
```



x

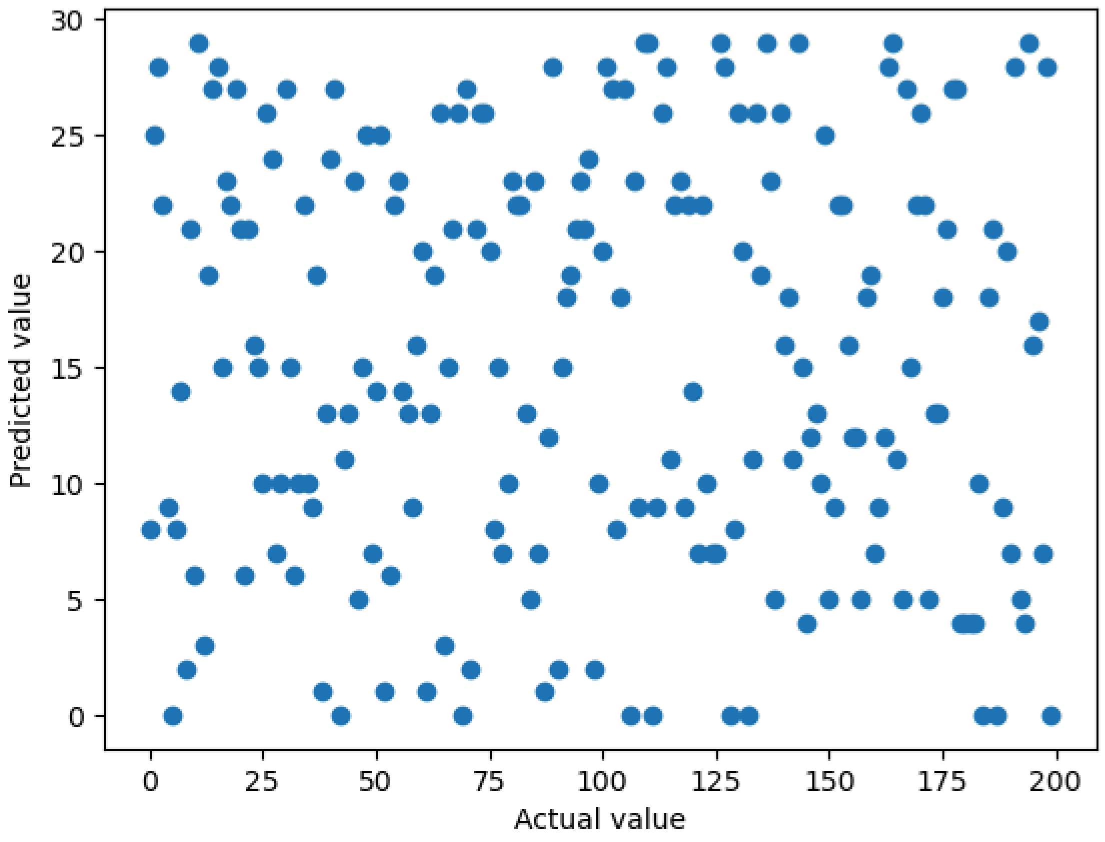
What is our objective in forecasting financial data?

- Our implicit aim is not to predict the price
- We only care about the returns of the stock/securities you're trading (i.e. is it going up or down?)

Re-run the model on 1st order differenced data

```
x = range(200)
y = np.random.randint(0,30,200)
plt.scatter(x, y)
plt.xlabel('Actual value')
plt.ylabel('Predicted value')
```

```
Text(0, 0.5, 'Predicted value')
```



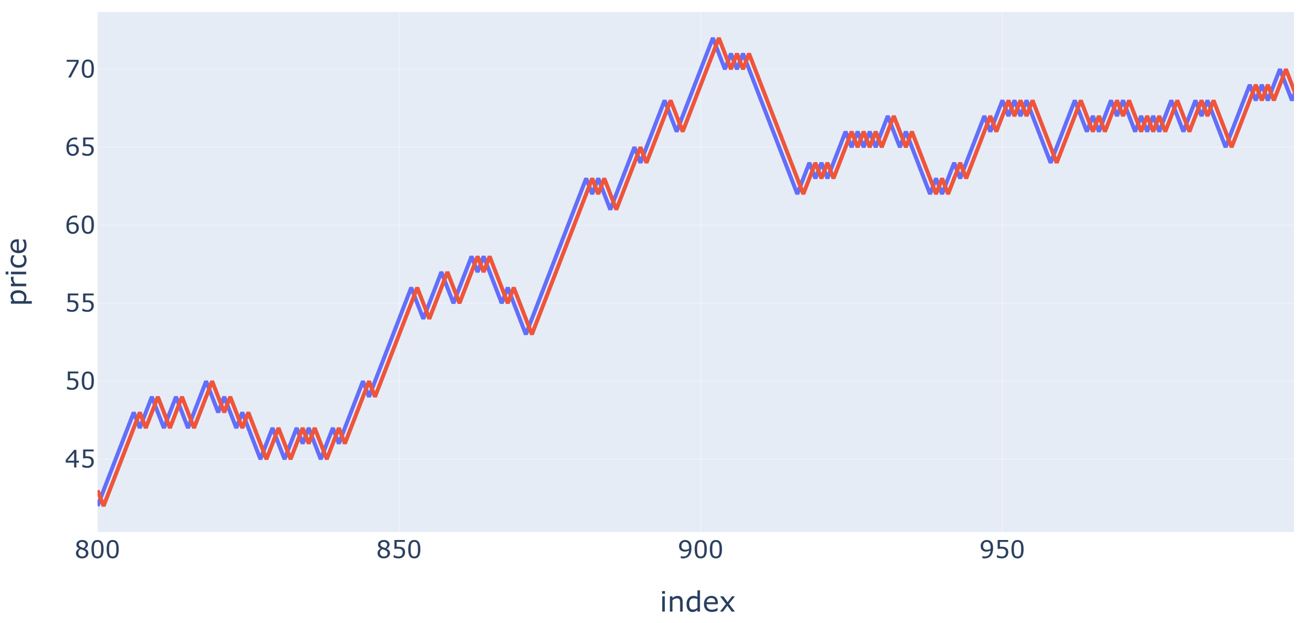
What did the model learn here?

- The algorithm showed you that it's possible to predict the price
- But it's not possible to predict the returns

- the data I called "stock index", was actually modeled using a random walk process.
- As the name indicates, a random walk is a completely stochastic process.
- Due to this, the idea of using historical data as a training set in order to learn the behavior and predict future returns is simply not possible.

- Let's zoom in

```
# Plot
px.line(pd.concat((stock_index.tail(200), stock_index_naive.tail(200))), y="pr
```

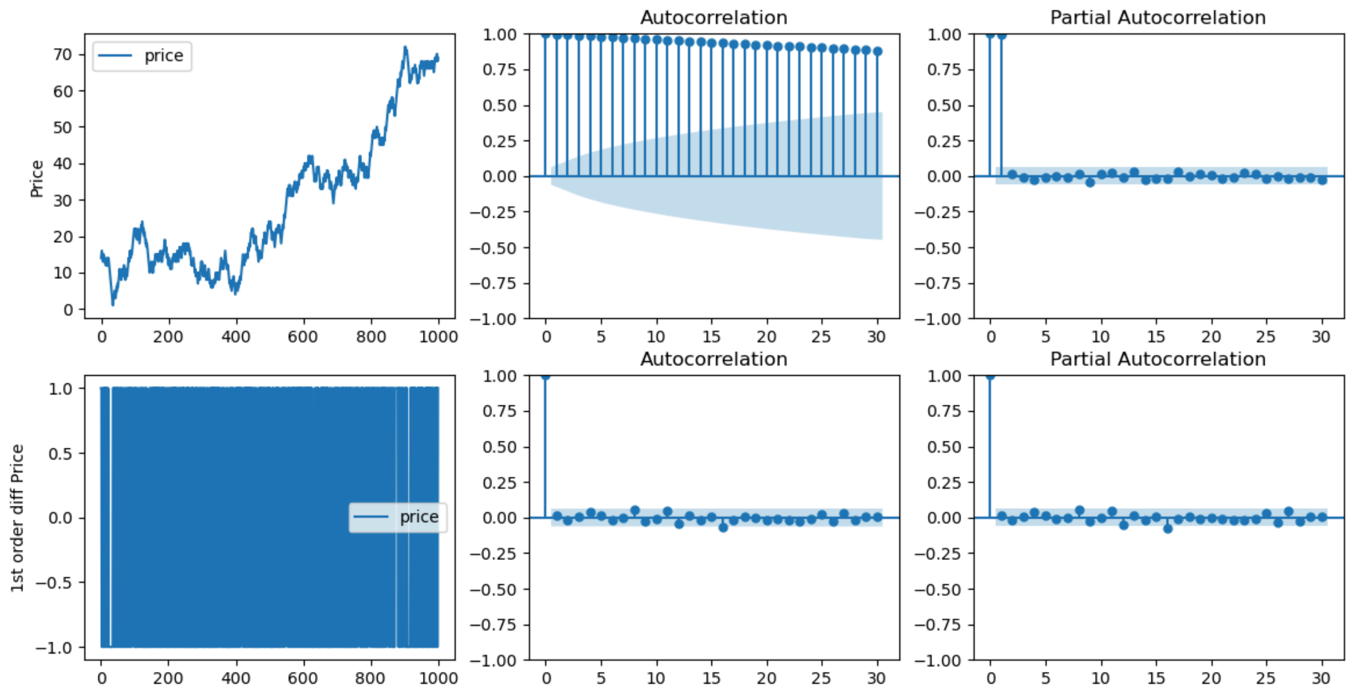


- what the model learned was actually using y_{t-1} to predict y_t (i.e., naive model)

Why?

```
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
    movement = -1 if random() < 0.5 else 1
    value = random_walk[i-1] + movement
    random_walk.append(value)
stock_index = pd.DataFrame({'price': random_walk })
stock_index.price = stock_index.price + 15
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(14, 7))

stock_index.plot(ax=axes[0,0])
plot_acf(stock_index,ax=axes[0,1])
plot_pacf(stock_index,ax=axes[0,2]);
axes[0,0].set(ylabel='Price')

stock_index_diff = stock_index.diff().dropna()
stock_index_diff.plot(ax=axes[1,0])
plot_acf(stock_index_diff,ax=axes[1,1])
plot_pacf(stock_index_diff,ax=axes[1,2]);
axes[1,0].set(ylabel='1st order diff Price')
```

```
[Text(0, 0.5, '1st order diff Price')]
```



- This difference graph also makes it clear that really we have no information to work with here other than a series of random moves.
- There is no structure to learn.

This is why we have baseline models in time-series forecasting

- Naive
- Seasonal naive
- Drift
- Baseline forecasts quickly flesh out whether you can do significantly better.
- If you can't, you're probably working with a random walk.

> A random walk is one in which future steps or directions cannot be predicted on the basis of past history. When the term is applied to the stock market, it means that short-run changes in stock prices are unpredictable.

6. Advanced Models & Summary

Traditionally, classical time-series methods such as exponential smoothing or ARIMA have been the workhorse in forecasting. These methods are well understood, easy to intepret, and highly effective in univariate time-series applications, or time-series with a strong trend/seasonality. However, they suffer from some limitations:

- Focus on linear relationships between x & y
- Inputs must be specified, not learned automatically
- Focus on univariate time-series, lack support for multi-input or multi-output
- Focus on regression, lack support for classification
- Assume complete, non-missing data

Neural networks provide a flexible architecture that can be valuable in forecasting:

- Robust to noise/missing value
- Support non-linear relationships, learn arbitrary mapping functions from inputs to outputs
- Support multivariate inputs/outputs
- Features can be learned, don't need to specify in advance (CNN)
- Temporal dependecies can be learned (RNN, LSTM)

Common applications of deep learning for time-series:

- Sensorial data, data generated with high frequency, data with no clear trend/seasonality
- Multivariate time-series
- Hierachical time-series
- Time-series classification
- Anomaly detection

If you are curious about reading more about how neural networks can be applied to time series data, I encourage you to check out Appendix A

A good workflow when presented with a new time series:

- Understand it. Plot it. Decompose it. What signals are present? What models might be suitable here?
- Split into train, validation, and test sets (or just train and test if you intend on doing cross-validation in the training set).
- Baseline and statistical models are the first stop. ETS and ARIMA are the workhorses of time series modelling, but don't discount the potential of basline methods!
- If you're looking for more performance, delve into machine learning. In particular, this allows us to easily and effectively incorporate explanatory variables and model highly complex functions.
- If you've got the time and the patience, say hello to CNNs, RNNs and LSTMs. They are complex, and take a long time to develop, but they show great potential. To me these methods try to interpret the time series for you, they engineer the features and identify the seasonality. It may be difficult to build neural networks that consistently outperform the classic statistical methods. But in saying that, they offer the ability to get good predictive performance without expert knowledge of the field or data you are forecasting.

> Always remember. We are forecasting the future - nobody knows what's going to happen. In that spirit, I prefer to err on the side of parsimony (simplicity).

- Forecasting is not as advanced as you might think, it's a very difficult field to make significant advances and there are several important, remaining challenges that have not yet been well dealt with, include forecasting:
    - Multivariate data

  - Data with complex/multiple seasonality
  - Irregular data
  - Including external regressors
  - Uncertainty