Introduction to `darts`

In this notebook, we will go over the main functionalities of the library: how to build and manipulate time series, train forecasting models, make predictions, evaluate metrics, backtest models and ensemble several models.

As a toy example, we will use the well known monthly airline passengers dataset (https://github.com/jbrownlee/Datasets/blob/master/monthly-airline-passengers.csv).

In [4]:
```
# fix python path if working locally
from utils import fix_pythonpath_if_working_locally
fix_pythonpath_if_working_locally()
```

In [5]:

```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import sys
import time
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from functools import reduce

from darts import TimeSeries
from darts.models import (
    NaiveSeasonal,
    NaiveDrift,
    Prophet,
    ExponentialSmoothing,
    ARIMA,
    AutoARIMA,
    RegressionEnsembleModel,
    RegressionModel,
    Theta,
    FFT
)

from darts.metrics import mape, mase
from darts.utils.statistics import check_seasonality, plot_acf, plot_residuals_analysis, plot_hi
from darts.datasets import AirPassengersDataset

import warnings
warnings.filterwarnings("ignore")
import logging
logging.disable(logging.CRITICAL)
```

Read data and build a `TimeSeries`

A `TimeSeries` represents a univariate or multivariate time series, with a proper time index. The time index can either be of type `pandas.DatetimeIndex` (containing datetimes), or of type `pandas.Int64Index` (containing integers; useful for representing sequential data without specific timestamps). In some cases, `TimeSeries` can even represent *probabilistic* series, in order for

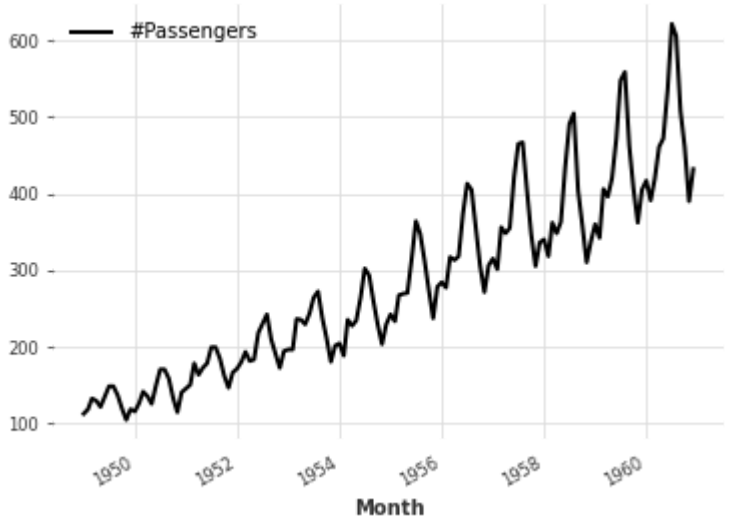
instance to obtain confidence intervals.

`TimeSeries` can be built easily using a few factory methods:

- From an entire Pandas `DataFrame`, using `TimeSeries.from_dataframe()`.
- From a time index and an array of corresponding values, using `TimeSeries.from_times_and_values()`.
- From a 1-D or 2-D Numpy array, using `TimeSeries.from_values()`.
- From a Pandas `Series`, using `TimeSeries.from_series()`.
- From an `xarray.DataArray`, using `TimeSeries.from_xarray()`.
- From a csv file, using `TimeSeries.from_csv()`.

Here, we directly load the air passanger series from a Darts dataset:

In [6]:
```
series = AirPassengersDataset().load()
series.plot()
```



In [7]:
```
type(series)
```

Out[7]:
```
darts.timeseries.TimeSeries
```

In [8]:
```
1 print(series)
```

```
<TimeSeries (DataArray) (Month: 144, component: 1, sample: 1)>
array([[[112.]],

       [[118.]],

       [[132.]],

       [[129.]],

       [[121.]],

       [[135.]],

       [[148.]],

       [[148.]],

       [[136.]],

       [[119.]],

...

       [[419.]],

       [[461.]],

       [[472.]],

       [[535.]],

       [[622.]],

       [[606.]],

       [[508.]],

       [[461.]],

       [[390.]],
```

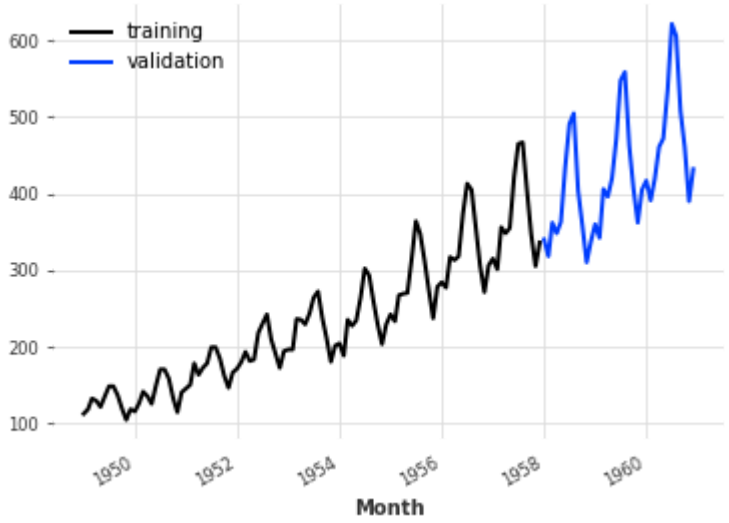
```
       [[432.]]])
Coordinates:
  * Month      (Month) datetime64[ns] 1949-01-01 1949-02-01 ... 1960-12-01
  * component  (component) object '#Passengers'
Dimensions without coordinates: sample
```

Creating a training and validation series

First, let's split our `TimeSeries` into a training and a validation series. Note: in general, it is also a good practice to keep a test series aside and never touch it until the end of the process. Here, we just build a training and a test series for simplicity.

The training series will be a `TimeSeries` containing values until January 1958 (excluded), and the validation series a `TimeSeries` containing the rest:

In [9]:
```
1 train, val = series.split_before(pd.Timestamp('19580101'))
2 train.plot(label='training')
3 val.plot(label='validation')
4 plt.legend();
```



In [11]:

```
1 print(train)
2 print(val)
```

```
<TimeSeries (DataArray) (Month: 108, component: 1, sample: 1)>
array([[[112.]],

       [[118.]],

       [[132.]],

       [[129.]],

       [[121.]],

       [[135.]],

       [[148.]],

       [[148.]],

       [[136.]],

       [[119.]],

...

       [[356.]],

       [[348.]],

       [[355.]],

       [[422.]],

       [[465.]],

       [[467.]],

       [[404.]],

       [[347.]],
```

```
       [[305.]],

       [[336.]]])
Coordinates:
  * Month      (Month) datetime64[ns] 1949-01-01 1949-02-01 ... 1957-12-01
  * component  (component) object '#Passengers'
Dimensions without coordinates: sample
<TimeSeries (DataArray) (Month: 36, component: 1, sample: 1)>
array([[[340.]],

       [[318.]],

       [[362.]],

       [[348.]],

       [[363.]],

       [[435.]],

       [[491.]],

       [[505.]],

       [[404.]],

       [[359.]],

...

       [[419.]],

       [[461.]],

       [[472.]],

       [[535.]],

       [[622.]],

       [[606.]],

       [[508.]],
```

```
       [[461.]],

       [[390.]],

       [[432.]]])
Coordinates:
  * Month      (Month) datetime64[ns] 1958-01-01 1958-02-01 ... 1960-12-01
  * component  (component) object '#Passengers'
Dimensions without coordinates: sample
```

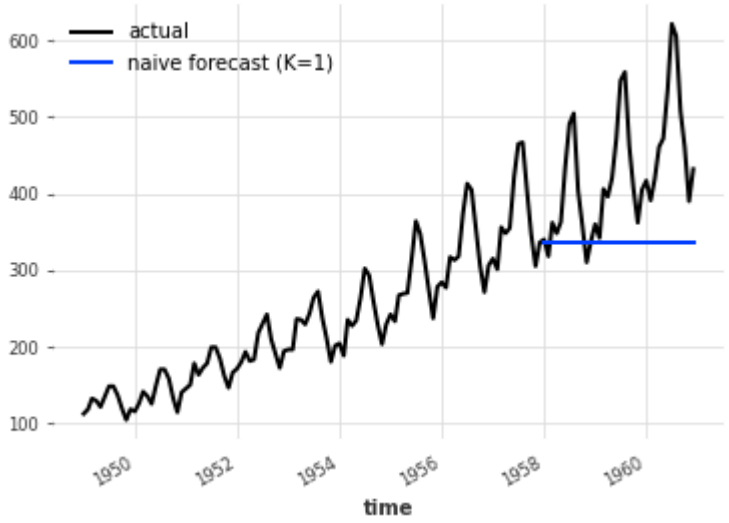
Playing with toy models

There is a collection of "naive" baseline models in `darts`, which can be very useful to get an idea of the bare minimum accuracy that one could expect. For example, the `NaiveSeasonal(K)` model always "repeats" the value that occured `K` time steps ago.

In its most naive form, when `K=1`, this model simply always repeats the last value of the training series:

In [12]:

```
naive_model = NaiveSeasonal(K=1)
naive_model.fit(train)
naive_forecast = naive_model.predict(36)

series.plot(label='actual')
naive_forecast.plot(label='naive forecast (K=1)')
plt.legend();
```

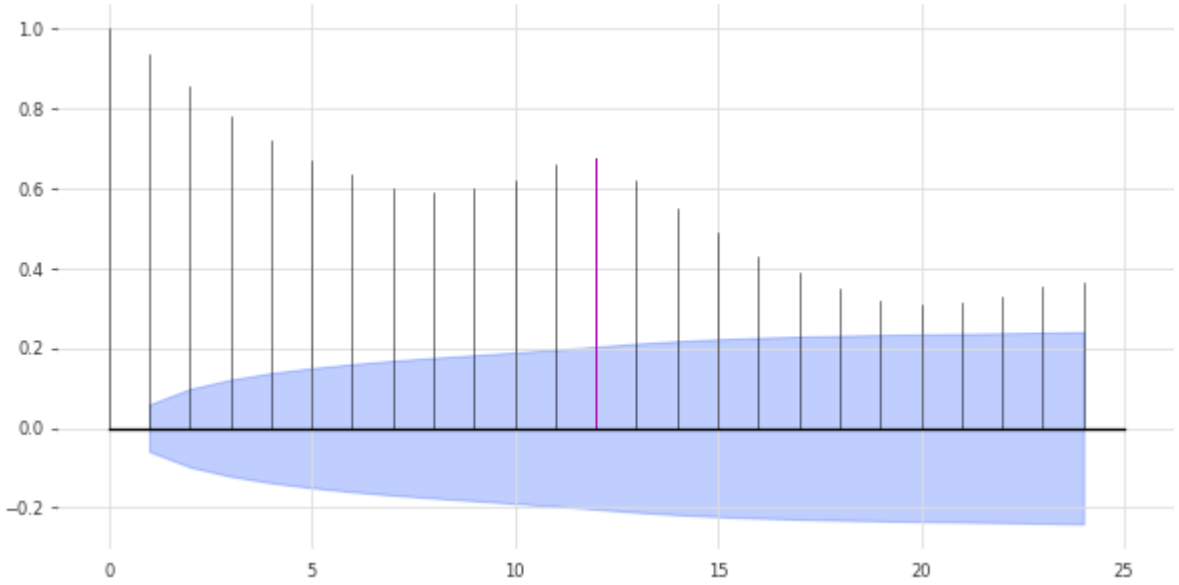


It's very easy to fit models and produce predictions on `TimeSeries`. All the models have a `fit()` and a `predict()` function. This is similar to https://scikit-learn.org/ (https://scikit-learn.org/), except that it is specific to time series. The `fit()` function takes in argument the training time series on which to fit the model, and the `predict()` function takes in argument the number of time steps (after the end of the training series) over which to forecast.

Inspect Seasonality

Our model above is perhaps a bit too naive. We can already improve by exploiting the seasonality in the data. It seems quite obvious that the data has a yearly seasonality, which we can confirm by looking at the auto-correlation function (ACF), and highlighting the lag `m=12`:

In [13]:
```
1 plot_acf(train, m = 12, alpha = .05)
```



The ACF presents a spike at x = 12, which suggests a yearly seasonality trend (highlighted in red). The blue zone determines the significance of the statistics for a confidence level of alpha = 5%. In cases where we are unsure, we can also run a statistical check of seasonality for each candidate period `m` :

In [14]:
```
1 for m in range(2, 25):
2     is_seasonal, period = check_seasonality(train, m=m, alpha=.05)
3     if is_seasonal:
4         print('There is seasonality of order {}.'.format(period))
```

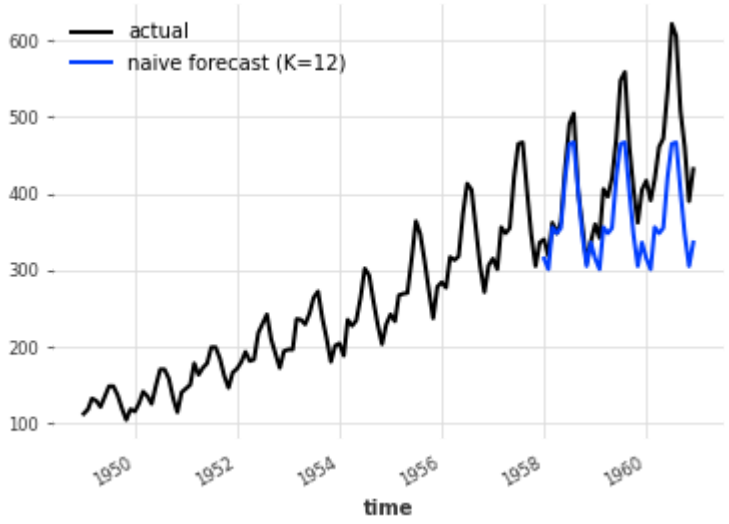
```
There is seasonality of order 12.
```

A less naive model

Let's try the `NaiveSeasonal` model again with a seasonality of 12:

In [15]:

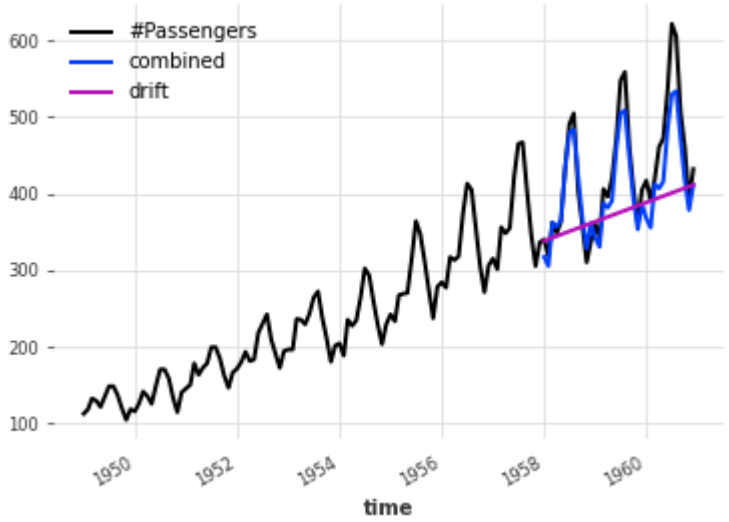
```
seasonal_model = NaiveSeasonal(K=12)
seasonal_model.fit(train)
seasonal_forecast = seasonal_model.predict(36)

series.plot(label='actual')
seasonal_forecast.plot(label='naive forecast (K=12)')
plt.legend();
```



This is better, but we are still missing the trend. Fortunately, there is also another naive baseline model capturing the trend, which is called `NaiveDrift`. This model will simply produce linear predictions, with a slope that is determined by the first and last values of the training set:

In [16]:

```
drift_model = NaiveDrift()
drift_model.fit(train)
drift_forecast = drift_model.predict(36)

combined_forecast = drift_forecast + seasonal_forecast - train.last_value()

series.plot()
combined_forecast.plot(label='combined')
drift_forecast.plot(label='drift')
plt.legend();
```



What happened in the last cell? We simply fit a naive drift model, and add its forecast to the seasonal forecast we had previously. We

also substract the last value of the training set to the result, so that the resulting combined forecast starts off with the right offset.

This looks already like a fairly decent forecast, and we did not use any non-naive model yet! In fact - any model should be able to beat this. But hey, what's the error we are getting here? Let's see what we'll have to beat:

In [17]:
```
print("Mean absolute percentage error for the combined naive drift + seasonal: {:.2f}%.".format(
      mape(series, combined_forecast)))
```

```
Mean absolute percentage error for the combined naive drift + seasonal: 5.66%.
```

Quickly try a few more models

`darts` is built to make it easy to train and validate several models in a unified way. Let's train a few more and compute their respective MAPE on the validation set:

In [18]:
```
def eval_model(model):
    model.fit(train)
    forecast = model.predict(len(val))
    print('model {} obtains MAPE: {:.2f}%'.format(model, mape(val, forecast)))

eval_model(ExponentialSmoothing())
eval_model(Prophet())
eval_model(AutoARIMA())
eval_model(Theta())
```

```
model Exponential smoothing obtains MAPE: 5.11%
model Prophet obtains MAPE: 9.87%
model Auto-ARIMA obtains MAPE: 11.65%
model Theta(2) obtains MAPE: 8.15%
```

Here, we did only built these models with their default parameters. We can probably do better if we fine-tune to our problem. Let's try with the Theta method.

The Theta method

The model `Theta` contains an implementation of Assimakopoulos and Nikolopoulos' Theta method. This method has known great

success, particularly in the M3-competition.

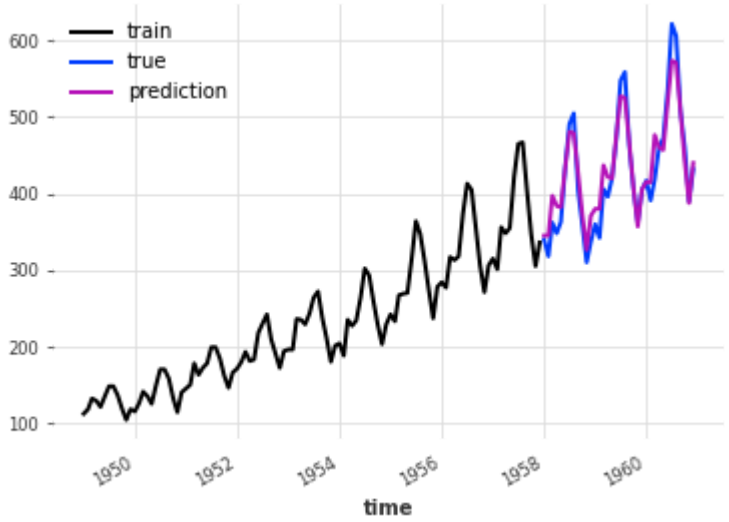
Though the value of the Theta parameter is often set to 0 in applications, our implementation supports a variable value for parameter tuning purposes. Let's try to find a good value for Theta:

In [19]:
```
# Search for the best theta parameter, by trying 50 different values
thetas = 2 - np.linspace(-10, 10, 50)

best_mape = float('inf')
best_theta = 0

for theta in thetas:
    model = Theta(theta)
    model.fit(train)
    pred_theta = model.predict(len(val))
    res = mape(val, pred_theta)

    if res < best_mape:
        best_mape = res
        best_theta = theta
```

In [20]:
```
best_theta_model = Theta(best_theta)
best_theta_model.fit(train)
pred_best_theta = best_theta_model.predict(len(val))

print('The MAPE is: {:.2f}, with theta = {}.'.format(mape(val, pred_best_theta), best_theta))
```

```
The MAPE is: 4.40, with theta = -3.5102040816326543.
```

In [21]:
```
train.plot(label='train')
val.plot(label='true')
pred_best_theta.plot(label='prediction')
plt.legend();
```



We can observe that the model with `best_theta` is so far the best we have, in terms of MAPE.

Backtesting: simulate historical forecasting

So at this point we have a model that performs well on our validation set, and that's good. But, how can we know the performance we *would have obtained* if we *had been using this model* historically?

Backtesting simulates predictions that would have been obtained historically with a given model. It can take a while to produce, since the model is re-fit every time the simulated prediction time advances.

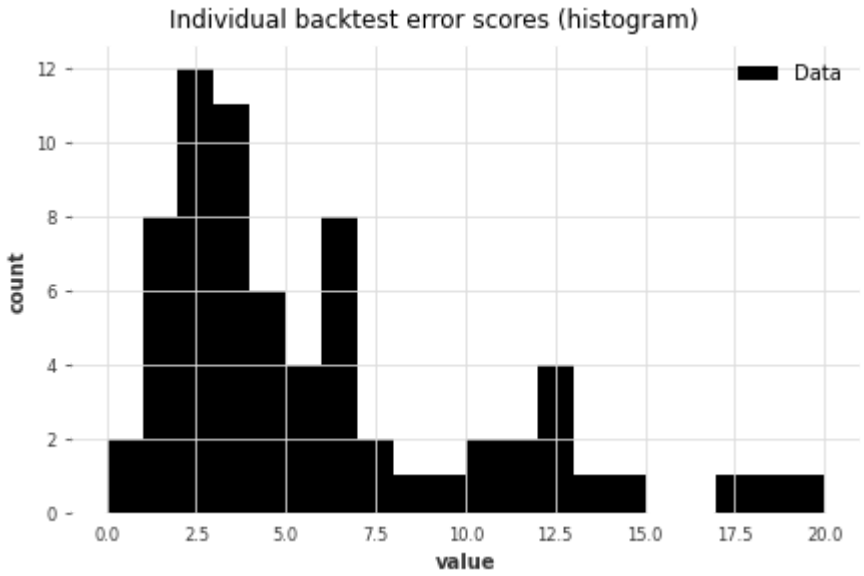
Such simulated forecasts are always defined with respect to a *forecast horizon*, which is the number of time steps that separate the prediction time from the forecast time. In the example below, we simulate forecasts done for 3 months in the future (compared to prediction time).

Using the `backtest()` method, you can either look at the performance of the model evaluated over the whole forecasts it produces (each point in each forecast is used to compute an error score) or only the last point of each historical forecast. In the latter case, you can get a time series representation of those points by calling `historical_forecasts()` with the default setting (`last_points_only=True`)

In [22]:

```
best_theta_model = Theta(best_theta)

average_error = best_theta_model.backtest(series, start=pd.Timestamp('19550101'), forecast_horiz
median_error = best_theta_model.backtest(series, start=pd.Timestamp('19550101'), forecast_horizo
print("Average error (MAPE) over all historical forecasts: {}".format(average_error))
print("Median error (MAPE) over all historical forecasts: {}".format(median_error))

raw_errors = best_theta_model.backtest(series, start=pd.Timestamp('19550101'), forecast_horizon=
plot_hist(raw_errors, bins=np.arange(0, max(raw_errors), 1), title='Individual backtest error sc

historical_fcast_theta = best_theta_model.historical_forecasts(series, start=pd.Timestamp('19550
```

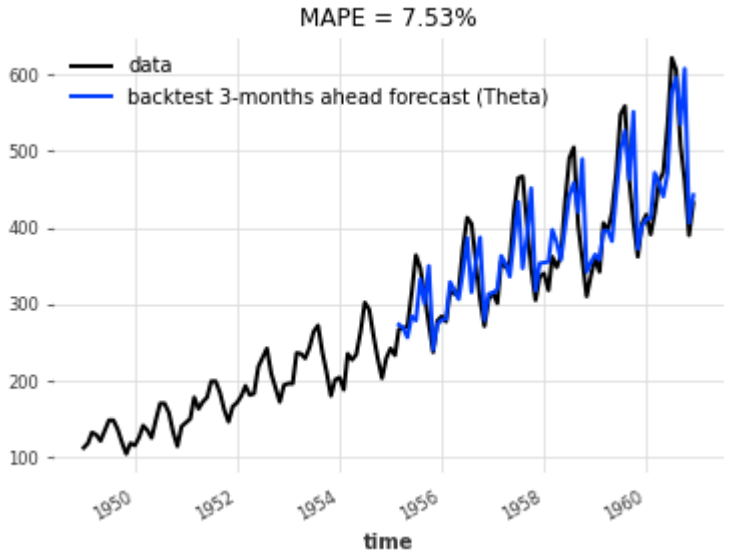
```
  0%|          | 0/70 [00:00<?, ?it/s]

  0%|          | 0/70 [00:00<?, ?it/s]

Average error (MAPE) over all historical forecasts: 6.1338415502368635
Median error (MAPE) over all historical forecasts: 4.205415091459073

  0%|          | 0/70 [00:00<?, ?it/s]

  0%|          | 0/70 [00:00<?, ?it/s]
```



Let's see what this backtest forecast looks like. You can see it produces more accurate predictions at a 3 months horizon than the one-

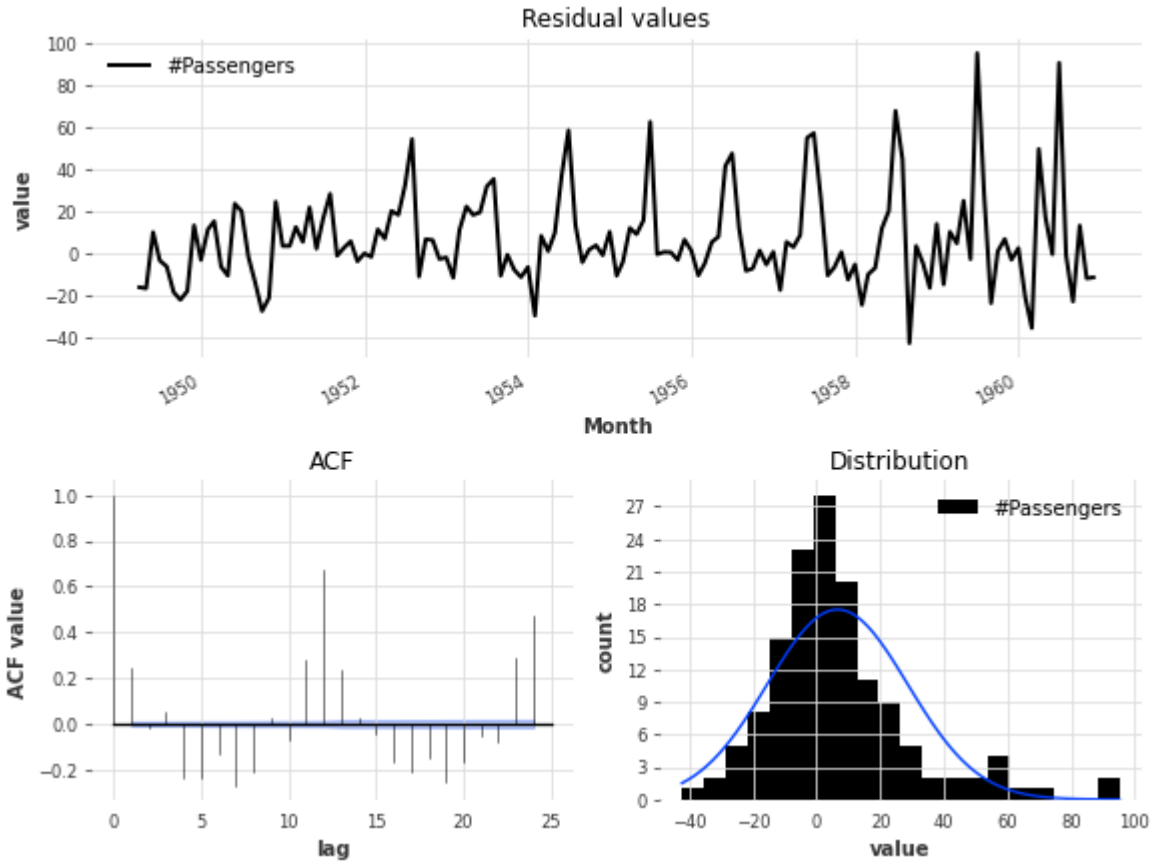
off prediction done above, because here the model is re-fit every month.

In [23]:
```
series.plot(label='data')
historical_fcast_theta.plot(label='backtest 3-months ahead forecast (Theta)')
plt.title('MAPE = {:.2f}%'.format(mape(historical_fcast_theta, series)))
plt.legend();
```



Let's look at the fitted value residuals of our current `Theta` model, i.e. the difference between the 1-step forecasts at every point in time obtained by fitting the model on all previous points, and the actual observed values.

In [24]:
```
1 plot_residuals_analysis(best_theta_model.residuals(series))
```

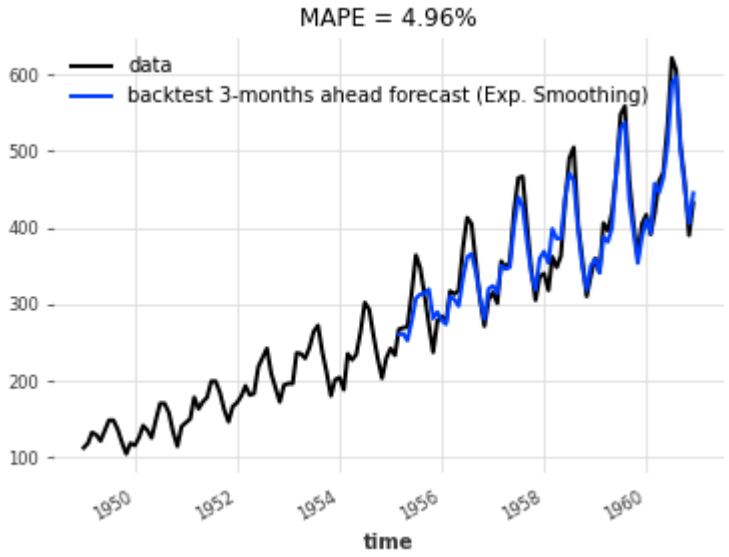


We can see that the distribution has a mean that is slightly larger than 0. This means that our `Theta` model is biased. We can also make out a large ACF value at lag equal to 12, which indicates that the residuals contain information that was not used by the model.

Could we maybe do better with a simple `ExponentialSmoothing` model?

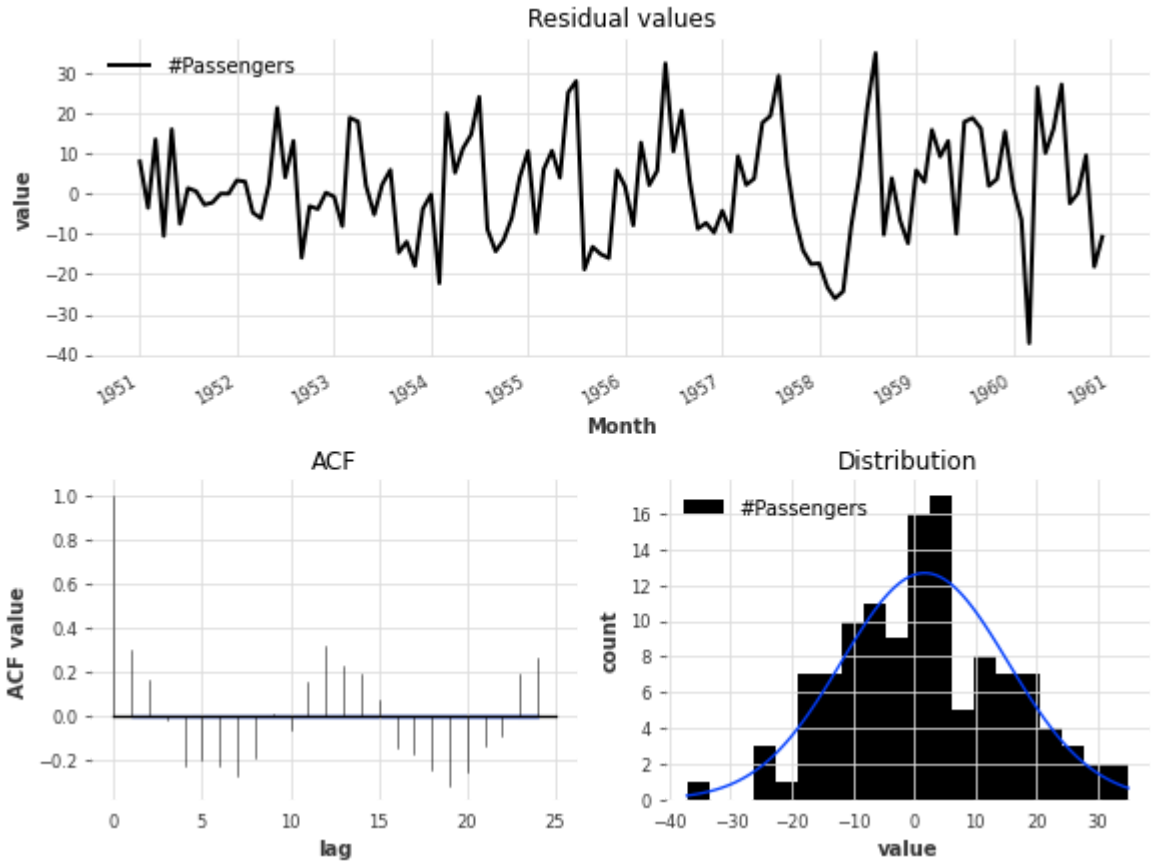
In [25]:
```
model_es = ExponentialSmoothing()
historical_fcast_es = model_es.historical_forecasts(series, start=pd.Timestamp('19550101'), fore

series.plot(label='data')
historical_fcast_es.plot(label='backtest 3-months ahead forecast (Exp. Smoothing)')
plt.title('MAPE = {:.2f}%'.format(mape(historical_fcast_es, series)))
plt.legend()
plt.show()
```

```
0%|          | 0/70 [00:00<?, ?it/s]
```



This much better! We get a mean absolute percentage error of about 4-5% when backtesting with a 3-months forecast horizon in this case.

In [26]:
```
1 plot_residuals_analysis(model_es.residuals(series))
```



The residual analysis also reflects an improved performance in that we now have a distribution of the residuals centred at value 0, and the ACF values, although not insignificant, have lower magnitudes.

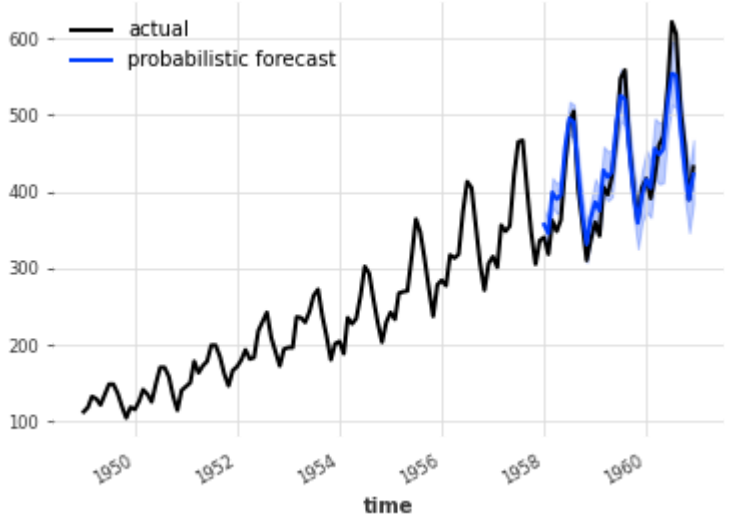
Probabilistic Forecasts

Some models can produce probabilistic forecasts. At the time of writing, this is the case for `RNNModel`, `TCNModel`, `ARIMA` and `ExponentialSmoothing`. For `RNNModel` and `TCNModel` (as well as other neural network based models), one has to specify a `Likelihood` (e.g. `darts.utils.likelihood models.GaussianLikelihood` for Gaussian likelihood) - we refer the readers

to notebooks 09 and 10 for examples.

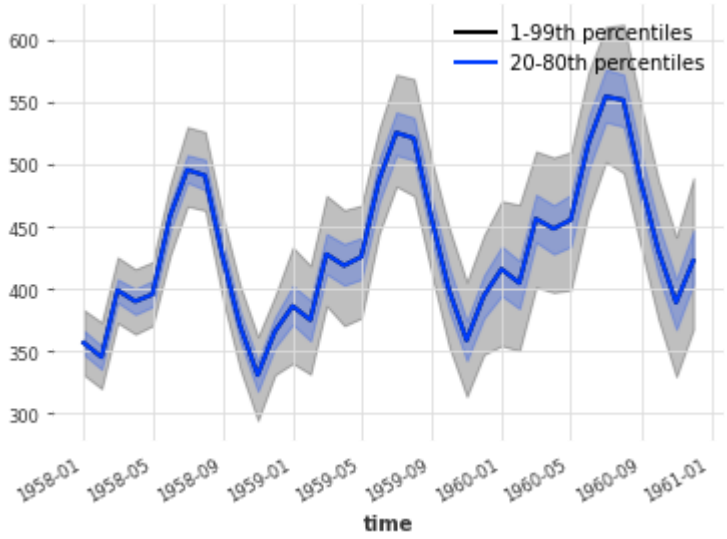
For `ARIMA` and `ExponentialSmoothing`, one can simply specify a `num_samples` parameter to the `predict()` function. The returned `TimeSeries` will then be composed of `num_samples` samples describing the distribution of the time series' values, which can for instance be used to obtain quantiles:

In [27]:
```
model_es = ExponentialSmoothing()
model_es.fit(train)
probabilistic_forecast = model_es.predict(len(val), num_samples=500)

series.plot(label='actual')
probabilistic_forecast.plot(label='probabilistic forecast')
plt.legend()
plt.show()
```



By default `TimeSeries.plot()` shows the median as well as the 5th and 95th percentiles (of the marginal distributions, if the `TimeSeries` is multivariate). It is possible to control this:

In [28]:
```
probabilistic_forecast.plot(low_quantile=0.01, high_quantile=0.99, label='1-99th percentiles')
probabilistic_forecast.plot(low_quantile=0.2, high_quantile=0.8, label='20-80th percentiles')
```



Ensembling several predictions

Ensembling is about combining the forecasts produced by several models, in order to obtain a final -- and hopefully better forecast.

For instance, in our example of a "less naive" model above, we manually combined a naive seasonal model with a naive drift model. Here, we will try to find such combinations in an automated way, first, using direclty a `RegressionModel`, and then, using the convenient `RegressionEnsembleModel`.

If you are in a hurry, jump to the **`RegressionEnsembleModel` approach**.

`RegressionModel` approach

A regression model is a model that predicts the target time series using some lags of this series (and possibly other series). It is also possible to use this approach to combine the forecasts of several models into one final forecast.

Here, we will first compute the historical predictions from two naive seasonal models (with 6 and 12 months seasonality), and naive drift model. To compute the historical forecasts, we can simply reuse the `historical_forecasts()` method:

In [29]:
```
models = [NaiveSeasonal(6), NaiveSeasonal(12), NaiveDrift()]

model_predictions = [m.historical_forecasts(series,
                                            start=pd.Timestamp('19561201'),
                                            forecast_horizon=12,
                                            stride=12,
                                            last_points_only=False,
                                            verbose=True)
                     for m in models]

model_predictions = [reduce((lambda a, b: a.append(b)), model_pred) for model_pred in model_pred
```

```
0%|          | 0/4 [00:00<?, ?it/s]

0%|          | 0/4 [00:00<?, ?it/s]

0%|          | 0/4 [00:00<?, ?it/s]
```

In [30]:
```
model_predictions_stacked = model_predictions[0]
for model_prediction in model_predictions[1:]:
    model_predictions_stacked = model_predictions_stacked.stack(model_prediction)
```

Now that we have the historical forecasts *that we would have obtained* from a couple of models, we can train a `RegressionModel`, in order to learn in a supervised way how to best combine the features time series (our 3 forecasts) into the target series that we are trying to predict.

By default the `RegressionModel` will fit a linear regression for predicting the target series from some features series. If you want something different than linear regression, `RegressionModel` can wrap around any scikit-learn regression model.

In [31]:

```
""" We build the regression model, and tell it to use the current predictions
"""
regr_model = RegressionModel(lags=None, lags_future_covariates=[0])

""" Our target series is what we want to predict (the actual data)
    It has to have the same time index as the features series:
"""
series_target = series.slice_intersect(model_predictions[0])

""" Here we backtest our regression model
"""
ensemble_pred = regr_model.historical_forecasts(
    series=series_target, future_covariates=model_predictions_stacked,
    start=pd.Timestamp('19580101'), forecast_horizon=3, verbose=True
)
```

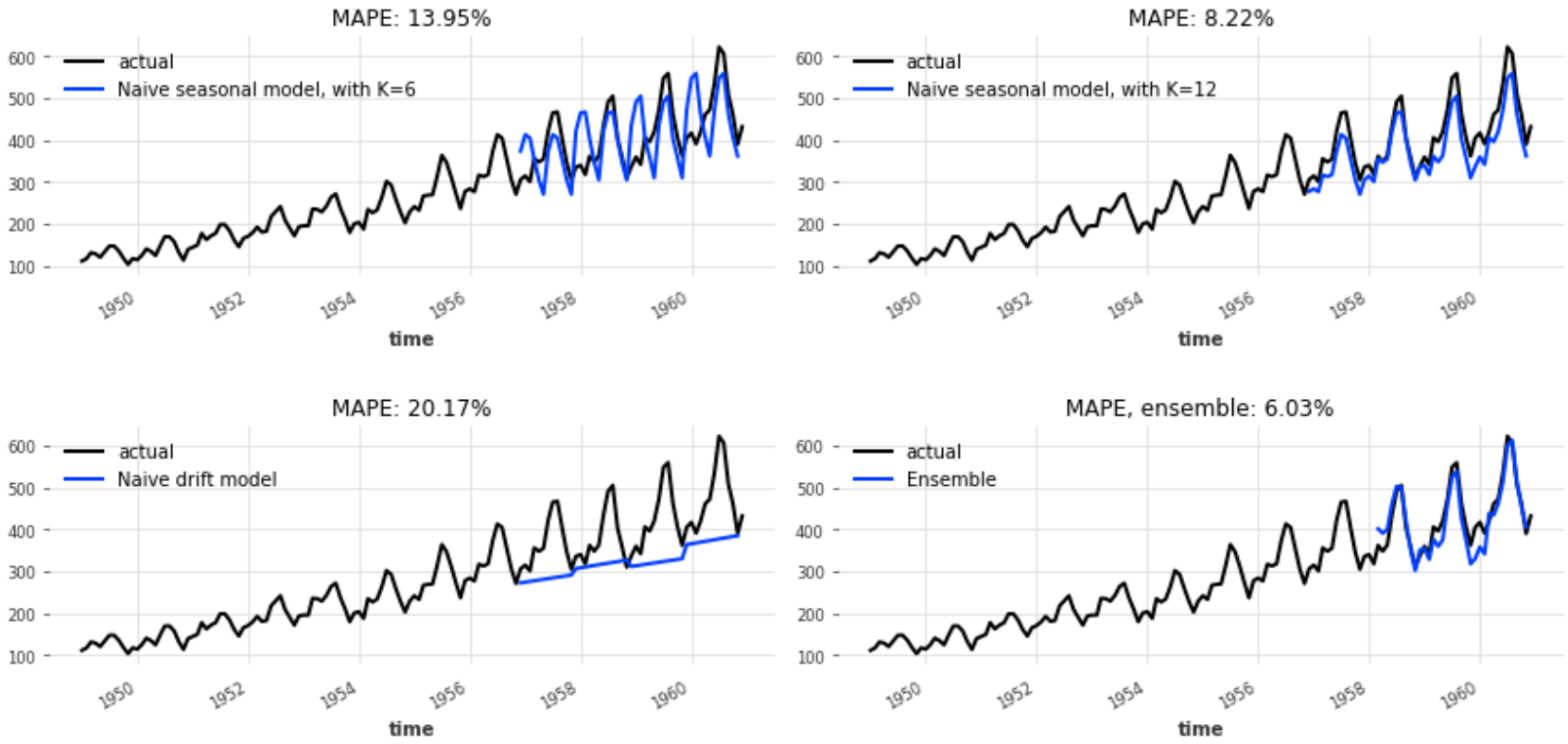
```
  0%|          | 0/33 [00:00<?, ?it/s]
```

Finally, let's see how good the regression performs, compared to the original forecasts:

In [32]:

```
fig, ax = plt.subplots(2,2,figsize=(12,6))
ax = ax.ravel()

for i, m in enumerate(models):
    series.plot(label='actual', ax=ax[i])
    model_predictions[i].plot(label=str(m), ax=ax[i])

    # intersect last part, to compare all the methods over the duration of the ensemble forecast
    model_pred = model_predictions[i].slice_intersect(ensemble_pred)

    mape_model = mape(series, model_pred)
    ax[i].set_title('\nMAPE: {:.2f}%'.format(mape_model))
    ax[i].legend()

series.plot(label='actual', ax=ax[3])
ensemble_pred.plot(label='Ensemble', ax=ax[3])
ax[3].set_title('\nMAPE, ensemble: {:.2f}%'.format(mape(series, ensemble_pred)))
ax[3].legend()

print('\nRegression coefficients for the individual models:')
for i, m in enumerate(models):
    print('Learned coefficient for {}: {:.2f}'.format(m, regr_model.model.coef_[i]))
plt.tight_layout();
```

```
Regression coefficients for the individual models:
Learned coefficient for Naive seasonal model, with K=6: -0.01
Learned coefficient for Naive seasonal model, with K=12: 1.09
Learned coefficient for Naive drift model: 0.10
```



That's quite nice: by just combining 3 naive models (two seasonal repetitions and a linear trend) using a linear regression, we get a decent-looking ensemble model, which is better than any of the sub-model, with a MAPE of 6.03%.

A couple of interesting things to observe:

- Note how the seasonal model with `K=6` and the naive drift model have an incorrect phase compared to the original signal (due to the original signal having a true seasonality of 12). Despite this, the ensembling is able to learn to ignore the seasonal model with `K=6` (by assigning a coefficient of 0), and learns a coefficient of only 0.09 for the drift model.

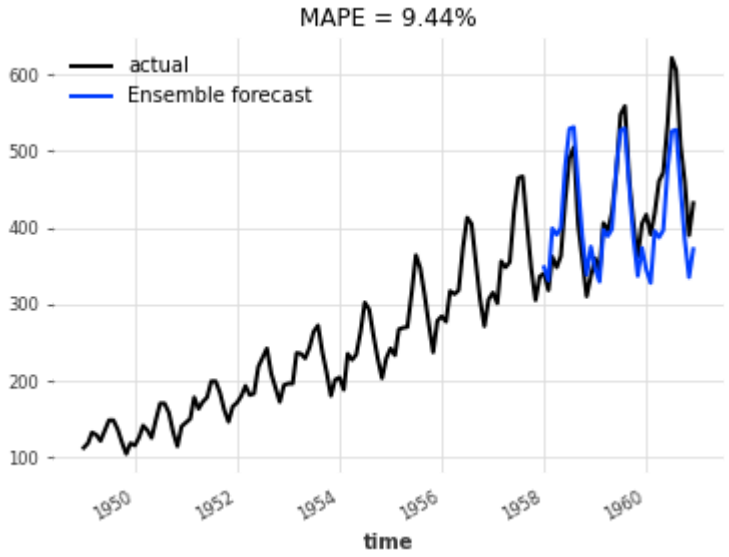
- Note how the regression (ensemble) forecast starts off 12 months after the individual models forecasts -- that is because the regression model needs 12 data points to fit the weights coefficients of the linear regression.

`RegressionEnsembleModel` approach

Wouldn't it be great if there would be a way to get an ensemble prediction in fewer lines of code?

Actually, there is: using the `RegressionEnsembleModel`. It is as simple as instantiating the ensemble model, calling the method `fit` on the training set, and then, calling the method `predict` with the number of time steps to forecast:

In [33]:
```
ensemble_model = RegressionEnsembleModel(
    forecasting_models=[NaiveSeasonal(6), NaiveSeasonal(12), NaiveDrift()],
    regression_train_n_points=12)

ensemble_model.fit(train)
ensemble_pred = ensemble_model.predict(36)

series.plot(label='actual')
ensemble_pred.plot(label='Ensemble forecast')
plt.title('MAPE = {:.2f}%'.format(mape(ensemble_pred, series)))
plt.legend();
```

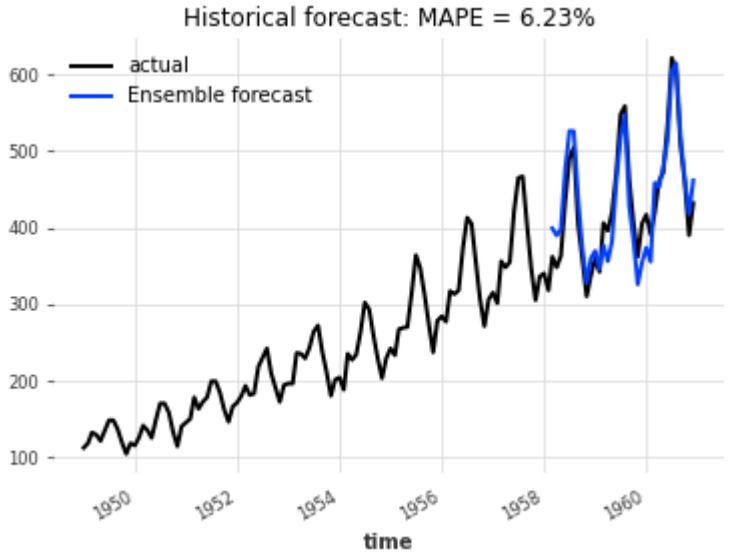


Quite easy, isn't it?

Now, to get a result similar to the one obtained with the `RegressionModel` approach, just use the `historical_forecasts` method of your ensemble model:

In [34]:
```
ensemble_pred_hist = ensemble_model.historical_forecasts(series,
                                                         start=pd.Timestamp('19580101'),
                                                         forecast_horizon=3,
                                                         verbose=True)
series.plot(label='actual')
ensemble_pred_hist.plot(label='Ensemble forecast')
plt.title('Historical forecast: MAPE = {:.2f}%'.format(mape(ensemble_pred_hist, series)))
plt.legend();
```

```
0%|          | 0/34 [00:00<?, ?it/s]
```



Note that the when we use `RegressionEnsembleModel`, by default, we are using the `LinearRegression` model from scikit-learn. To change this, you can pass to the `regression_model` argument any regression model with `predict()` and `fit()` methods, for example `RidgeCV()`.

FFT and RNNs

If you'd like to try models based on Fast Fourier Transform or Recurrent Neural Networks, we recommend that you go over the `03-FFT-examples.ipynb` and `04-RNN-examples.ipynb` notebooks, respectively. Notebooks after that give more examples (essentially of neural nets based models), and notebooks 09 (DeepAR) and 10 (DeepTCN) showcase probabilistic versions of RNN and TCN models.

A final word of caution

So is Theta, exponential smoothing, or a linear regression of naive models the best approach for predicting the future number of airline passengers? Well, at this point it's still hard to say exactly which one is best. Our time series is small, and our validation set is even smaller. In such cases, it's very easy to overfit the whole forecasting exercise to such a small validation set. That's especially true if the number of available models and their degrees of freedom is high; so always take results with a grain of salt (especially on small datasets), and apply the scientific method before making any kind of forecast!

In [1]:
```
1 import sys
```

In [2]:
```
1 sys.path
```

Out[2]:
```
['/home/wengong/projects/time-series/darts/examples',
 '/usr/lib/python38.zip',
 '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '',
 '/home/wengong/.local/lib/python3.8/site-packages',
 '/home/wengong/projects/yahoofinancials',
 '/home/wengong/projects/tsmoothie',
 '/home/wengong/projects/yfinance/yfinance',
 '/home/wengong/projects/NLP/BERTopic',
 '/home/wengong/projects/NLP/textnets',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/local/lib/python3.8/dist-packages/pandas-1.2.3-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/numpy-1.20.1-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/alpha_vantage-2.2.0-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/aiohttp-4.0.0a1-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/yarl-1.6.3-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/typing_extensions-3.7.4.3-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/multidict-4.7.6-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/attrs-20.3.0-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/async_timeout-3.0.1-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/mplfinance-0.12.7a9-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/matplotlib-3.4.0rc3-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/pyparsing-3.0.0b2-py3.8.egg',
 '/usr/local/lib/python3.8/dist-packages/kiwisolver-1.3.1-py3.8-linux-x86_64.egg',
 '/usr/local/lib/python3.8/dist-packages/cycler-0.10.0-py3.8.egg',
 '/home/wengong/projects/iexfinance/iexfinance',
 '/usr/lib/python3/dist-packages',
 '/home/wengong/.local/lib/python3.8/site-packages/IPython/extensions',
 '/home/wengong/.ipython']
```

In [3]:
```
1 import darts
```

In [ ]:
```
1
```