ExampleE

June 23, 2022

In this example we will set up a more complex dayahead power generation prediction, in order to test a backtesting operation of the dataset.

Note this example may take some time (up to 15 minutes) to run completly, and a fair amount of RAM (around 1.5GB) is required to load the historical data.

[1]:
```
import enda
import datetime
import os
import pandas as pd
import time

from enda.contracts import Contracts
from enda.scoring import Scoring
from enda.backtesting import BackTesting

from enda.feature_engineering.datetime_features import DatetimeFeature
from enda.power_stations import PowerStations
from enda.timeseries import TimeSeries

pd.options.display.max_columns = None
pd.options.display.max_colwidth = 30

import matplotlib.pyplot as plt
#%matplotlib notebook
```

[2]:
```
DIR = '.'
generation_source = ["wind", "solar", "river"]
```

1 1. Read and prepare data

[3]:
```
def get_example_e_dataset(source):

    if source not in ["wind", "solar", "river"]:
            raise NotImplementedError("unknown source argument")

    # get station portfolio
    stations = Contracts.read_contracts_from_file(os.path.join(
```

```python
    DIR, source, "stations_" + source + ".csv")
)

# display it as a multiindex with day as second index
stations = PowerStations.get_stations_daily(
    stations,
    station_col='station',
    date_start_col="date_start",
    date_end_exclusive_col="date_end_exclusive"
)

# between dates of interest
stations = PowerStations.get_stations_between_dates(
    stations,
    start_datetime=pd.to_datetime('2017-01-01'),
    end_datetime_exclusive=pd.to_datetime('2022-01-01')
)

# on a 30-minutes scale
stations = TimeSeries.interpolate_daily_to_sub_daily_data(
    stations,
    freq='30min',
    tz='Europe/Paris',
    index_name='time'
)

# integrate outages
outages = PowerStations.read_outages_from_file(
    os.path.join(DIR, "events.csv"),
    station_col='station',
    time_start_col="time_start",
    time_end_exclusive_col="time_end",
    pct_outages_col="impact_production_pct_kw",
    tzinfo="Europe/Paris"
)

stations = PowerStations.integrate_outages(
    df_stations=stations,
    df_outages=outages,
    station_col='station',
    time_start_col="time_start",
    time_end_exclusive_col="time_end",
    installed_capacity_col="installed_capacity_kw",
    pct_outages_col="impact_production_pct_kw"
)

# get production
```

```python
    production = pd.read_csv(
        os.path.join(DIR, source, "production_" + source + ".csv"),
        parse_dates=["time"],
        date_parser=lambda col: pd.to_datetime(col, utc=True)
    )
    production['time'] = TimeSeries.align_timezone(production['time'], tzinfo='Europe/Paris')
    production.set_index(["station", "time"], inplace=True)

    production = TimeSeries.average_to_upper_freq(
        production,
        freq='30min',
        tz='Europe/Paris',
        index_name='time',
        enforce_single_freq=False
    )

    dataset = pd.merge(stations, production, how='inner', left_index=True, right_index=True)
    dataset = dataset.dropna()

    # get weather for wind and solar
    if source in ["wind", "solar"]:
        weather = pd.read_csv(
            os.path.join(DIR, source, "weather_forecast_" + source + ".csv"),
            parse_dates=["time"],
            date_parser=lambda col: pd.to_datetime(col, utc=True)
        )
        weather['time'] = TimeSeries.align_timezone(weather['time'], tzinfo='Europe/Paris')
        weather.set_index(["station", "time"], inplace=True)

        weather = TimeSeries.interpolate_freq_to_sub_freq_data(
            weather,
            freq='30min',
            tz='Europe/Paris',
            index_name='time',
            method="linear"
        )

        dataset = pd.merge(dataset, weather, how='inner', left_index=True, right_index=True)

    # featurize for solar
    if source == "solar":
        dataset = DatetimeFeature.split_datetime(
            dataset, split_list=['minuteofday', 'dayofyear']
```

```
        )

        dataset = DatetimeFeature.encode_cyclic_datetime_index(
            dataset, split_list=['minuteofday', 'dayofyear']
        )

    return dataset
```

[4]:
```
%%time
dataset_wind = get_example_e_dataset("wind")
```

```
CPU times: user 35.4 s, sys: 1.44 s, total: 36.8 s
Wall time: 37.6 s
```

[5]:
```
%%time
dataset_solar = get_example_e_dataset("solar")
```

```
CPU times: user 1min 39s, sys: 7.54 s, total: 1min 47s
Wall time: 1min 50s
```

[6]:
```
%%time
dataset_river = get_example_e_dataset("river")
```

```
CPU times: user 3min 44s, sys: 14 s, total: 3min 58s
Wall time: 4min 6s
```

[7]:
```
dataset = dict(zip(generation_source, [dataset_wind, dataset_solar,␣
 ↪dataset_river]))
```

[8]:
```
# Compute load factor
# We drop the power_kw information during that step, not to bias the IA␣
 ↪algorithm afterwards.
def wrapper_compute_load_factor(df):
    return enda.PowerStations.compute_load_factor(
                df,
                installed_capacity_kw='installed_capacity_kw',
                power_kw='power_kw',
                drop_power_kw=True
            )

dataset_final = {source: wrapper_compute_load_factor(d) for source, d in␣
 ↪dataset.items()}
```

2 2. Make a basic prediction

2.0.1 Separe between training and forecasting dataset to backtest the data

We have here the full datasets which have been built using the Enda utilities function, and some historical information gathered from the TSO, diverse meteo information suppliers, and contracts data.

We will now distinguish our full datasets in two, in order to obtain the training and forecasting datasets representative of what could be obtained in real life condition.

[9]:
```
# wrapper function around the
def separate_train_test_sets(df):

    # let's create the input train dataset
    train_set = df[df.index.get_level_values(1) < pd.to_datetime('2021-12-01 00:
 ↪00:00+01:00')]

    # let's create the input data for our forecast
    forecast_set = df[df.index.get_level_values(1) >= pd.
 ↪to_datetime('2021-12-01 00:00:00+01:00')]
    forecast_set = forecast_set.drop(columns="load_factor")

    # and let us keep the information of the real power generation for testing␣
 ↪purposes
    future_set = df[df.index.get_level_values(1) >= pd.to_datetime('2021-12-01␣
 ↪00:00:00+01:00')]

    return train_set, forecast_set, future_set

train_test_future_sets = {source: separate_train_test_sets(data) for source,␣
 ↪data in dataset_final.items()}

train_set = {source: train_test_future_sets[source][0] for source in␣
 ↪generation_source}
forecast_set = {source: train_test_future_sets[source][1] for source in␣
 ↪generation_source}
future_set = {source: train_test_future_sets[source][2] for source in␣
 ↪generation_source}
```

[10]:
```
train_set["wind"].shape
```

[10]:
```
(628798, 4)
```

Let's use the enda algorithms to make a simple power prediction.

[11]:
```
# import power predictors
from enda.power_predictor import PowerPredictor
```

2.0.2 Run of river prediction

[12]:
```
# import a dummy ML backend for river
from enda.estimators import EndaEstimatorRecopy

# build a PowerPredictor object
river_predictor = PowerPredictor(standard_plant=False)

# use PowerPredictor to train the estimator from the run of river data,
# and from a naive recopy estimator
river_predictor.train(train_set["river"],␣
↪estimator=EndaEstimatorRecopy(period='1D'), target_col="load_factor")
```

[13]:
```
train_set["river"]
```

[13]:
```
                                      installed_capacity_kw  load_factor
station time
hy_0    2019-12-22 00:00:00+01:00                     595.0     0.375294
        2019-12-22 00:30:00+01:00                     595.0     0.396471
        2019-12-22 01:00:00+01:00                     595.0     0.429412
        2019-12-22 01:30:00+01:00                     595.0     0.434118
        2019-12-22 02:00:00+01:00                     595.0     0.432941
…                                                     …          …
hy_99   2021-11-30 21:30:00+01:00                      80.5     1.040580
        2021-11-30 22:00:00+01:00                      80.5     1.023188
        2021-11-30 22:30:00+01:00                      80.5     0.994203
        2021-11-30 23:00:00+01:00                      80.5     0.901449
        2021-11-30 23:30:00+01:00                      80.5     0.837681

[3936142 rows x 2 columns]
```

[14]:
```
# Once it has been trained, we can predict the power for each power plant␣
↪individually, calling predict()
# from PowerPredictor()
pred_river = river_predictor.predict(forecast_set["river"],␣
↪target_col="load_factor")
```

[15]:
```
pred_river
```

[15]:
```
                                   load_factor
station time
hy_0    2021-12-01 00:00:00+01:00     0.000000
        2021-12-01 00:30:00+01:00     0.000000
        2021-12-01 01:00:00+01:00     0.000000
        2021-12-01 01:30:00+01:00     0.000000
        2021-12-01 02:00:00+01:00     0.000000
…                                          …
hy_99   2021-12-31 21:30:00+01:00     0.457669
```

```
2021-12-31 22:00:00+01:00     0.457669
2021-12-31 22:30:00+01:00     0.457669
2021-12-31 23:00:00+01:00     0.457669
2021-12-31 23:30:00+01:00     0.457669

[123504 rows x 1 columns]
```

2.0.3 Wind prediction

[25]:
```
# boot up an H2O server
import h2o
h2o.init(nthreads=-1)
h2o.no_progress()
```

```
Checking whether there is an H2O instance running at http://localhost:54321
… not found.
Attempting to start a local H2O server…
  Java Version: openjdk version "17.0.2" 2022-01-18 LTS; OpenJDK Runtime
Environment Zulu17.32+13-CA (build 17.0.2+8-LTS); OpenJDK 64-Bit Server VM
Zulu17.32+13-CA (build 17.0.2+8-LTS, mixed mode, sharing)
  Starting server from /Users/clement.jeannesson/.pyenv/versions/3.9.10/envs/end
a_test_007/lib/python3.9/site-packages/h2o/backend/bin/h2o.jar
  Ice root: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpnay1kl3g
  JVM stdout: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpnay1kl3g/h2o_c
lement_jeannesson_started_from_python.out
  JVM stderr: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpnay1kl3g/h2o_c
lement_jeannesson_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321 … successful.

--------------------------  -----------------------------------------
H2O_cluster_uptime:         01 secs
H2O_cluster_timezone:       Europe/Paris
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.36.1.1
H2O_cluster_version_age:    2 months and 9 days
H2O_cluster_name:           H2O_from_python_clement_jeannesson_ydomeq
H2O_cluster_total_nodes:    1
H2O_cluster_free_memory:    4 Gb
H2O_cluster_total_cores:    8
H2O_cluster_allowed_cores:  8
H2O_cluster_status:         locked, healthy
H2O_connection_url:         http://127.0.0.1:54321
H2O_connection_proxy:       {"http": null, "https": null}
H2O_internal_security:      False
Python_version:             3.9.10 final
--------------------------  -----------------------------------------
```

[17]:
```python
# enda's wrapper around H2O models
from enda.ml_backends.h2o_estimator import EndaH2OEstimator
from h2o.estimators import H2OGradientBoostingEstimator
from h2o.estimators import H2OGeneralizedLinearEstimator

# define an estimator
gradboost_estimator = EndaH2OEstimator(H2OGradientBoostingEstimator(
    ntrees=500,
    max_depth=5,
    sample_rate=0.5,
    min_rows=5,
    seed=17
))
```

[18]:
```python
# build a PowerPredictor object
wind_predictor = PowerPredictor(standard_plant=True)
```

[19]:
```python
# train the estimator
wind_predictor.train(train_set["wind"], estimator=gradboost_estimator,␣
↪target_col="load_factor")
```

[20]:
```python
# predict
pred_wind = wind_predictor.predict(forecast_set['wind'],␣
↪target_col="load_factor", is_normally_clamped=True)
```

[21]:
```python
pred_wind
```

[21]:
```
                                   load_factor
station time
eo_0    2021-12-01 00:00:00+01:00     0.000000
        2021-12-01 00:30:00+01:00     0.000000
        2021-12-01 01:00:00+01:00     0.000000
        2021-12-01 01:30:00+01:00     0.000000
        2021-12-01 02:00:00+01:00     0.000000
…                                          …
eo_9    2021-12-31 20:00:00+01:00     0.041006
        2021-12-31 20:30:00+01:00     0.041006
        2021-12-31 21:00:00+01:00     0.041006
        2021-12-31 21:30:00+01:00     0.041006
        2021-12-31 22:00:00+01:00     0.040764

[28215 rows x 1 columns]
```

2.0.4 Solar prediction

[26]:
```
# keep the best estimator from h2o

# build a PowerPredictor object
solar_predictor = PowerPredictor(standard_plant=True)

# use the same good estimator
gradboost_estimator = EndaH2OEstimator(H2OGradientBoostingEstimator(
    ntrees=500,
    max_depth=5,
    sample_rate=0.5,
    min_rows=5,
    seed=17
))

# train the estimator
solar_predictor.train(train_set["solar"], estimator=gradboost_estimator,␣
 ↪target_col="load_factor")

# predict
pred_solar= solar_predictor.predict(forecast_set["solar"],␣
 ↪target_col="load_factor", is_normally_clamped=True)
```

[27]:
```
pred_solar
```

[27]:
```
                                          load_factor
station time
pv_0    2021-12-01 00:00:00+01:00     0.000278
        2021-12-01 00:30:00+01:00     0.000278
        2021-12-01 01:00:00+01:00     0.001461
        2021-12-01 01:30:00+01:00     0.000667
        2021-12-01 02:00:00+01:00     0.000276
…                                          …
pv_9    2021-12-31 20:00:00+01:00     0.000000
        2021-12-31 20:30:00+01:00     0.000000
        2021-12-31 21:00:00+01:00     0.000000
        2021-12-31 21:30:00+01:00     0.000000
        2021-12-31 22:00:00+01:00     0.000000

[65244 rows x 1 columns]
```

[28]:
```
# shutdown your h2o local server
h2o.cluster().shutdown()
# wait for h2o to really finish shutting down
time.sleep(5)
```

```
H2O session _sid_8b45 closed.
```

2.0.5 Getting back to power prediction

To get back to power prediction, we simply need to use the installed capacity field and multiply it by the load factor to find again the power (kw)

[29]:
```
# we start by merging again the installed_capacity (kw) field

def merge_stations_and_features(df1, df2):
    df = pd.merge(df1, df2, how='inner', left_index=True, right_index=True)
    return df.dropna()

pred = dict(zip(generation_source, [pred_wind, pred_solar, pred_river]))
prediction = {source: merge_stations_and_features(
                          forecast_set[source].loc[:,
 ↪["installed_capacity_kw"]],
                          pred[source])
              for source in generation_source
             }
```

[30]:
```
# We drop the load_factor information during that step.
def wrapper_compute_power_kw_from_load_factor(df):
    return enda.PowerStations.compute_power_kw_from_load_factor(
               df,
               installed_capacity_kw='installed_capacity_kw',
               load_factor='load_factor',
               drop_load_factor=True
           )

prediction = {source: wrapper_compute_power_kw_from_load_factor(p)
             for source, p in prediction.items()}
```

[31]:
```
prediction["river"]
```

[31]:
```
                                   installed_capacity_kw   power_kw
station time
hy_0    2021-12-01 00:00:00+01:00                  595.0   0.000000
        2021-12-01 00:30:00+01:00                  595.0   0.000000
        2021-12-01 01:00:00+01:00                  595.0   0.000000
        2021-12-01 01:30:00+01:00                  595.0   0.000000
        2021-12-01 02:00:00+01:00                  595.0   0.000000
…                                                      …          …
hy_99   2021-12-31 21:30:00+01:00                   80.5  36.842361
        2021-12-31 22:00:00+01:00                   80.5  36.842361
        2021-12-31 22:30:00+01:00                   80.5  36.842361
        2021-12-31 23:00:00+01:00                   80.5  36.842361
        2021-12-31 23:30:00+01:00                   80.5  36.842361

[123504 rows x 2 columns]
```

3 3. Plots and KPI

3.0.1 Plot predicted data along with the real production

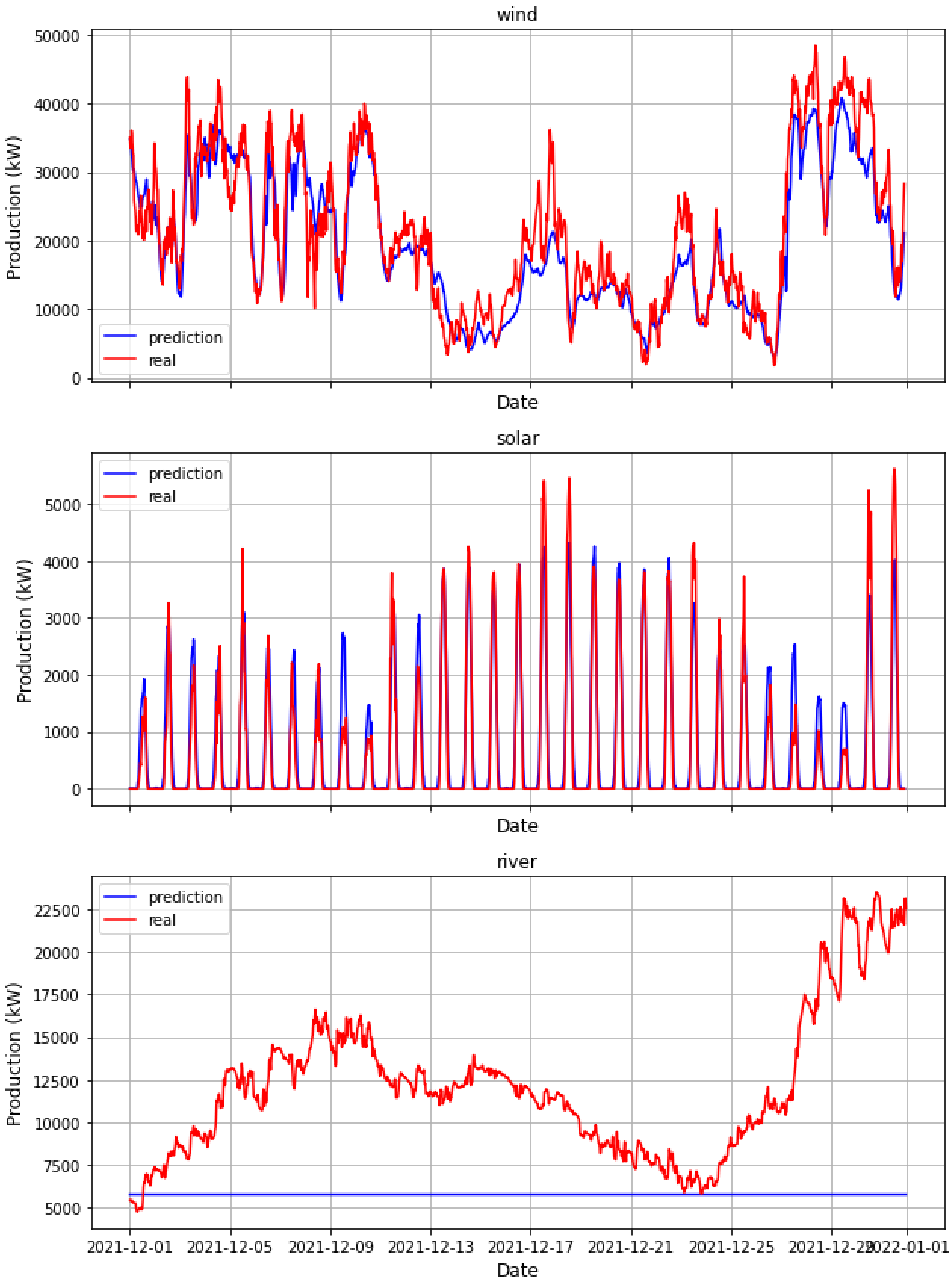
[32]:
```
# Get back to the power_kw
real = {source: wrapper_compute_power_kw_from_load_factor(r)
             for source, r in future_set.items()}


fig, axis = plt.subplots(3, 1, figsize=(9, 12), sharex=True, sharey=False)

i = 0
for source, data in prediction.items():
    axis[i].grid(True)
    axis[i].plot(data["power_kw"].groupby(level=1).agg("sum"),␣
 ↪label="prediction", c="blue")
    axis[i].set_xlabel('Date', fontsize=12)
    axis[i].set_ylabel('Production (kW)', fontsize=12)
    axis[i].set_title(source)
    i+=1

i = 0
for source, data in real.items():
    axis[i].plot(data["power_kw"].groupby(level=1).agg("sum"), label="real",␣
 ↪c="red")
    axis[i].set_xlabel('Date', fontsize=12)
    axis[i].set_ylabel('Production (kW)', fontsize=12)
    axis[i].legend()
    i +=1

fig.tight_layout()
```


wind
Production (kW)
50000
40000
30000
20000
10000
0
prediction
real
Date
solar
Production (kW)
5000
4000
3000
2000
1000
0
prediction
real
Date
river
Production (kW)
22500
20000
17500
15000
12500
10000
7500
5000
prediction
real
2021-12-01
2021-12-05
2021-12-09
2021-12-13
2021-12-17
2021-12-21
2021-12-25
2021-12-29
2022-01-01
Date

3.0.2 Compute the nMAPE

[33]:
```
# create the benchmark dataframe for wind power plant

# we keep the active capacity, the actual power injection, and the power␣
↪prediction
def build_becnhmark(source):
    benchmark = pd.merge(real[source][["installed_capacity_kw", "power_kw"]],
                         prediction[source]["power_kw"].to_frame(),
                         how="inner", left_index=True, right_index=True)
    benchmark = benchmark.rename({"power_kw_x": "actual",
                                  "power_kw_y": "enda",
                                 },
                                 axis=1)
    return benchmark

benchmark = {source: build_becnhmark(source)for source in generation_source}

benchmark['wind']
```

[33]:
```
                                   installed_capacity_kw  actual       enda
station time
eo_0    2021-12-01 00:00:00+01:00                   28.0     0.0   0.000000
        2021-12-01 00:30:00+01:00                   28.0     0.0   0.000000
        2021-12-01 01:00:00+01:00                   28.0     0.0   0.000000
        2021-12-01 01:30:00+01:00                   28.0     0.0   0.000000
        2021-12-01 02:00:00+01:00                   28.0     0.0   0.000000
…                                                      …       …          …
eo_9    2021-12-31 20:00:00+01:00                 1190.0   102.2  48.797439
        2021-12-31 20:30:00+01:00                 1190.0    21.0  48.797439
        2021-12-31 21:00:00+01:00                 1190.0     2.1  48.797439
        2021-12-31 21:30:00+01:00                 1190.0     0.0  48.797439
        2021-12-31 22:00:00+01:00                 1190.0     0.0  48.509588

[28215 rows x 3 columns]
```

[34]:
```
# sum over all power plants
benchmark_portfolio = {source: benchmark[source].groupby(level="time").sum()␣
↪for source in generation_source}
```

[35]:
```
# define a scoring
scoring_benchmark = {source: Scoring(benchmark_portfolio[source],
                                     target="actual",
                                     normalizing_col="installed_capacity_kw")␣
↪for source in generation_source}
```

[36]:
```
# compute the nAE
nAE = {source: scoring_benchmark[source].normalized_absolute_error() for source␣
 ↪in generation_source}
nAE['wind']
```

[36]:
```
                                enda
time
2021-12-01 00:00:00+01:00  0.002808
2021-12-01 00:30:00+01:00  0.021117
2021-12-01 01:00:00+01:00  0.010076
2021-12-01 01:30:00+01:00  0.053388
2021-12-01 02:00:00+01:00  0.059667
…                               …
2021-12-31 20:00:00+01:00  0.018362
2021-12-31 20:30:00+01:00  0.026846
2021-12-31 21:00:00+01:00  0.076548
2021-12-31 21:30:00+01:00  0.086034
2021-12-31 22:00:00+01:00  0.108073

[1485 rows x 1 columns]
```

[37]:
```
nMAPE = {source: nAE[source].mean() for source in generation_source}
nMAPE
```

[37]:
```
{'wind': enda    0.052625
 dtype: float64,
 'solar': enda    0.016584
 dtype: float64,
 'river': enda    0.103418
 dtype: float64}
```

It is a 5% difference (not exactly a percentage) for wind, 1% for solar, and 10% for run of river.

4 4. Perform a benchmark with backtesting

As in example B_load, we will perform a backtesting of the data we gathered, week after week. With the given dataset, this means : - for each week w from early 2020 until the end of the dataset : train using data from the beginning of the dataset (early 2018) until a few days before week w, then eval on w. - the first iteration will train an algorithm using data from 2018 to 2019, then eval on the first week of 2020 - the second iteration will train using data from 2018 to a bit before the first week of 2020, then eval on the second week of 2020 - and so on… - keep the predictions of each time-step using this method, from early 2020 to december 2021.
- then compare these predictions to the historic data to evaluate the quality of each algorithm.

This makes most sense if in your production environment, you plan to retrain the algorithm regularly with recent data.

We'll just perform it for wind turbines

[38]:
```python
# boot up an H2O server
import h2o
h2o.init(nthreads=-1)
h2o.no_progress()
```

```
Checking whether there is an H2O instance running at http://localhost:54321
… not found.
Attempting to start a local H2O server…
  Java Version: openjdk version "17.0.2" 2022-01-18 LTS; OpenJDK Runtime
Environment Zulu17.32+13-CA (build 17.0.2+8-LTS); OpenJDK 64-Bit Server VM
Zulu17.32+13-CA (build 17.0.2+8-LTS, mixed mode, sharing)
  Starting server from /Users/clement.jeannesson/.pyenv/versions/3.9.10/envs/end
a_test_007/lib/python3.9/site-packages/h2o/backend/bin/h2o.jar
  Ice root: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpyf03s1tp
  JVM stdout: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpyf03s1tp/h2o_c
lement_jeannesson_started_from_python.out
  JVM stderr: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpyf03s1tp/h2o_c
lement_jeannesson_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321 … successful.

--------------------------  -----------------------------------------
H2O_cluster_uptime:         01 secs
H2O_cluster_timezone:       Europe/Paris
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.36.1.1
H2O_cluster_version_age:    2 months and 9 days
H2O_cluster_name:           H2O_from_python_clement_jeannesson_n4gs60
H2O_cluster_total_nodes:    1
H2O_cluster_free_memory:    4 Gb
H2O_cluster_total_cores:    8
H2O_cluster_allowed_cores:  8
H2O_cluster_status:         locked, healthy
H2O_connection_url:         http://127.0.0.1:54321
H2O_connection_proxy:       {"http": null, "https": null}
H2O_internal_security:      False
Python_version:             3.9.10 final
--------------------------  -----------------------------------------
```

[39]:
```python
# we'll test simple estimators from Sklearn we haven't tried yet, for␣
↪demonstration purposes.
from enda.ml_backends.sklearn_estimator import EndaSklearnEstimator
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, SGDRegressor

# define a dict of estimators
all_estimators= dict()
```

```python
all_estimators['sklearn_lin_reg'] = EndaSklearnEstimator(LinearRegression())

all_estimators['sklearn_sgd'] = EndaSklearnEstimator(
    Pipeline([('standard_scaler', StandardScaler()),
              ('sgd', SGDRegressor())
             ]
            )
)

all_estimators['h2o_gboost'] = EndaH2OEstimator(H2OGradientBoostingEstimator(
    ntrees=500,
    max_depth=5,
    sample_rate=0.5,
    min_rows=5,
    seed=17
))
```

[40]:
```python
# create a PowerPredictor
predictor = PowerPredictor(standard_plant=True)
```

[41]:
```python
# run the backtesting and fill a benchmark_wind dataframe with the results from␣
↪the different algorithms

start_backtesting_dt = pd.to_datetime('2020-01-01 00:00:00+01:00').
↪tz_convert('Europe/Paris')
benchmark_wind = dataset_final['wind'][dataset_final['wind'].index.
↪get_level_values('time') \
                                        >= start_backtesting_dt]["load_factor"].
↪to_frame("actual_load_factor")

days_in_each_iteration = 28

for estimator_name, estimator in all_estimators.items():

    count_iterations = 0

    estimator_predictions = []

    for train_set, test_set in BackTesting.yield_train_test(
        dataset_final['wind'],
        start_eval_datetime=start_backtesting_dt,
        days_between_trains=days_in_each_iteration,
        gap_days_between_train_and_eval=14
    ):
        count_iterations += 1
```

```
        if count_iterations <= 2 or count_iterations % 10 == 0:
            print("Model {}, backtesting iteration {}, train set {}->{}, test␣
↪set {}->{}\n".format(
                    estimator_name, count_iterations,
                    train_set.index.get_level_values('time').min(),
                    train_set.index.get_level_values('time').max(),
                    test_set.index.get_level_values('time').min(),
                    test_set.index.get_level_values('time').max()))

        # featurize
        test_set = test_set.drop(columns=["load_factor"])

        # train and predict
        predictor.train(train_set, estimator=estimator,␣
↪target_col='load_factor')
        estimator_predictions.append(predictor.predict(test_set,␣
↪target_col='load_factor', is_normally_clamped=True))

    benchmark_wind[estimator_name] = pd.concat(estimator_predictions)
```

```
Model sklearn_lin_reg, backtesting iteration 1, train set 2018-12-22
00:00:00+01:00->2019-12-17 23:30:00+01:00, test set 2020-01-01
00:00:00+01:00->2020-01-28 23:30:00+01:00

Model sklearn_lin_reg, backtesting iteration 2, train set 2018-12-22
00:00:00+01:00->2020-01-14 23:30:00+01:00, test set 2020-01-29
00:00:00+01:00->2020-02-25 23:30:00+01:00

Model sklearn_lin_reg, backtesting iteration 10, train set 2018-12-22
00:00:00+01:00->2020-08-25 23:30:00+02:00, test set 2020-09-09
00:00:00+02:00->2020-10-06 23:30:00+02:00

Model sklearn_lin_reg, backtesting iteration 20, train set 2018-12-22
00:00:00+01:00->2021-06-01 23:30:00+02:00, test set 2021-06-16
00:00:00+02:00->2021-07-13 23:30:00+02:00

Model sklearn_sgd, backtesting iteration 1, train set 2018-12-22
00:00:00+01:00->2019-12-17 23:30:00+01:00, test set 2020-01-01
00:00:00+01:00->2020-01-28 23:30:00+01:00

Model sklearn_sgd, backtesting iteration 2, train set 2018-12-22
00:00:00+01:00->2020-01-14 23:30:00+01:00, test set 2020-01-29
00:00:00+01:00->2020-02-25 23:30:00+01:00

Model sklearn_sgd, backtesting iteration 10, train set 2018-12-22
00:00:00+01:00->2020-08-25 23:30:00+02:00, test set 2020-09-09
00:00:00+02:00->2020-10-06 23:30:00+02:00
```

```
Model sklearn_sgd, backtesting iteration 20, train set 2018-12-22
00:00:00+01:00->2021-06-01 23:30:00+02:00, test set 2021-06-16
00:00:00+02:00->2021-07-13 23:30:00+02:00

Model h2o_gboost, backtesting iteration 1, train set 2018-12-22
00:00:00+01:00->2019-12-17 23:30:00+01:00, test set 2020-01-01
00:00:00+01:00->2020-01-28 23:30:00+01:00

Model h2o_gboost, backtesting iteration 2, train set 2018-12-22
00:00:00+01:00->2020-01-14 23:30:00+01:00, test set 2020-01-29
00:00:00+01:00->2020-02-25 23:30:00+01:00

Model h2o_gboost, backtesting iteration 10, train set 2018-12-22
00:00:00+01:00->2020-08-25 23:30:00+02:00, test set 2020-09-09
00:00:00+02:00->2020-10-06 23:30:00+02:00

Model h2o_gboost, backtesting iteration 20, train set 2018-12-22
00:00:00+01:00->2021-06-01 23:30:00+02:00, test set 2021-06-16
00:00:00+02:00->2021-07-13 23:30:00+02:00
```

[42]:
```
# don't forget to shutdown your h2o local server
h2o.cluster().shutdown()
# wait for h2o to really finish shutting down
time.sleep(5)
```

```
H2O session _sid_98d7 closed.
```

[43]:
```
benchmark_wind
```

[43]:
```
                                    actual_load_factor  sklearn_lin_reg  \
station time
eo_0    2020-01-01 00:00:00+01:00             0.000000         0.000847
        2020-01-01 00:30:00+01:00             0.000000         0.001620
        2020-01-01 01:00:00+01:00             0.000000         0.002394
        2020-01-01 01:30:00+01:00             0.000000         0.000575
        2020-01-01 02:00:00+01:00             0.000000         0.000000
…                                                    …                …
eo_9    2021-12-31 20:00:00+01:00             0.085882         0.136678
        2021-12-31 20:30:00+01:00             0.017647         0.136934
        2021-12-31 21:00:00+01:00             0.001765         0.137190
        2021-12-31 21:30:00+01:00             0.000000         0.137446
        2021-12-31 22:00:00+01:00             0.000000         0.137702

                                    sklearn_sgd  h2o_gboost
station time
eo_0    2020-01-01 00:00:00+01:00      0.000515    0.000000
```

```
         2020-01-01 00:30:00+01:00     0.001284    0.000000
         2020-01-01 01:00:00+01:00     0.002053    0.000000
         2020-01-01 01:30:00+01:00     0.000029    0.000000
         2020-01-01 02:00:00+01:00     0.000000    0.000000
…                                           …         …
eo_9     2021-12-31 20:00:00+01:00     0.137477    0.055904
         2021-12-31 20:30:00+01:00     0.137685    0.055904
         2021-12-31 21:00:00+01:00     0.137892    0.055515
         2021-12-31 21:30:00+01:00     0.138100    0.054639
         2021-12-31 22:00:00+01:00     0.138307    0.054639

[628741 rows x 4 columns]
```

[44]:

```
# add the installed_capacity
benchmark_wind_kw = pd.merge(benchmark_wind,␣
 ↪dataset_final['wind']["installed_capacity_kw"],
                          how='inner', left_index=True, right_index=True)

benchmark_wind_kw = (benchmark_wind_kw.
 ↪multiply(benchmark_wind_kw["installed_capacity_kw"], axis=0)
                                       .drop(columns="installed_capacity_kw")
                                       .rename({"actual_load_factor":␣
 ↪"actual_power_kw"}, axis=1)
        )

benchmark_wind_kw = pd.merge(benchmark_wind_kw,␣
 ↪dataset_final['wind']["installed_capacity_kw"],
                          how='inner', left_index=True, right_index=True)
benchmark_wind_kw
```

[44]:

```
                                   actual_power_kw  sklearn_lin_reg  \
station time
eo_0    2020-01-01 00:00:00+01:00              0.0         0.023703
        2020-01-01 00:30:00+01:00              0.0         0.045373
        2020-01-01 01:00:00+01:00              0.0         0.067043
        2020-01-01 01:30:00+01:00              0.0         0.016094
        2020-01-01 02:00:00+01:00              0.0         0.000000
…                                                …                …
eo_9    2021-12-31 20:00:00+01:00            102.2       162.646693
        2021-12-31 20:30:00+01:00             21.0       162.951460
        2021-12-31 21:00:00+01:00              2.1       163.256228
        2021-12-31 21:30:00+01:00              0.0       163.560995
        2021-12-31 22:00:00+01:00              0.0       163.865762

                                   sklearn_sgd  h2o_gboost  \
station time
eo_0    2020-01-01 00:00:00+01:00     0.014425    0.000000
```

```
        2020-01-01 00:30:00+01:00     0.035953    0.000000
        2020-01-01 01:00:00+01:00     0.057482    0.000000
        2020-01-01 01:30:00+01:00     0.000816    0.000000
        2020-01-01 02:00:00+01:00     0.000000    0.000000
…                                          …           …
eo_9    2021-12-31 20:00:00+01:00   163.597614   66.526294
        2021-12-31 20:30:00+01:00   163.844638   66.526294
        2021-12-31 21:00:00+01:00   164.091661   66.063337
        2021-12-31 21:30:00+01:00   164.338684   65.020300
        2021-12-31 22:00:00+01:00   164.585707   65.020300

                                   installed_capacity_kw
station time
eo_0    2020-01-01 00:00:00+01:00                   28.0
        2020-01-01 00:30:00+01:00                   28.0
        2020-01-01 01:00:00+01:00                   28.0
        2020-01-01 01:30:00+01:00                   28.0
        2020-01-01 02:00:00+01:00                   28.0
…                                                    …
eo_9    2021-12-31 20:00:00+01:00                 1190.0
        2021-12-31 20:30:00+01:00                 1190.0
        2021-12-31 21:00:00+01:00                 1190.0
        2021-12-31 21:30:00+01:00                 1190.0
        2021-12-31 22:00:00+01:00                 1190.0

[628741 rows x 5 columns]
```

[45]:

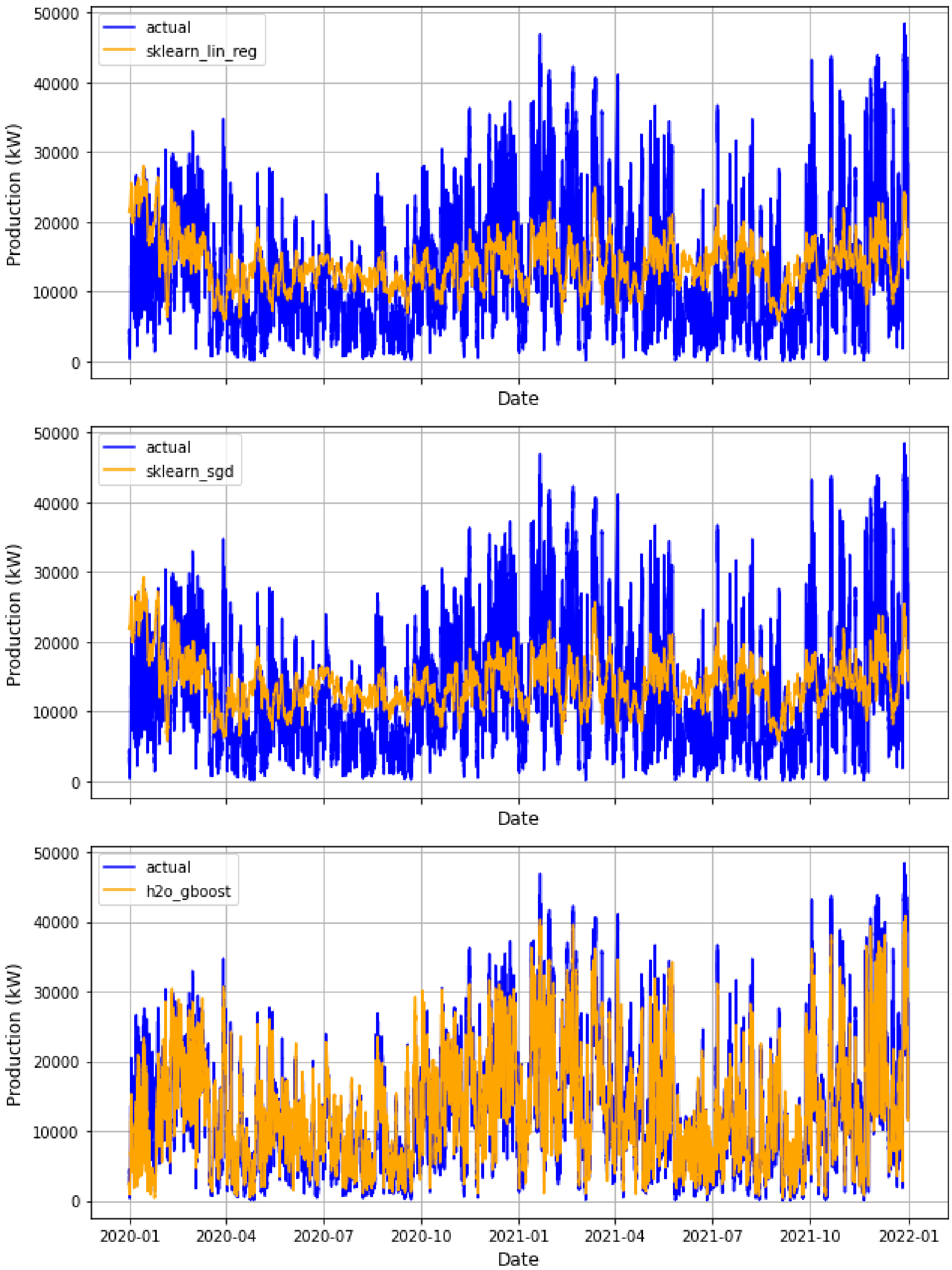
```
# visualize predictions

n_estimators = len(all_estimators) # = len(benchmark_wind.columns)-1

fig, axis = plt.subplots(n_estimators , 1, figsize=(9, 4 * n_estimators),␣
↪sharex=True, sharey=False)

i = 0
for c in benchmark_wind.columns:
    if c != "actual_load_factor":
        axis[i].grid(True)
        axis[i].plot(benchmark_wind_kw["actual_power_kw"].groupby(level=1).
↪agg("sum"), label="actual", c="blue")
        axis[i].plot(benchmark_wind_kw[c].groupby(level=1).agg("sum"), label=c,␣
↪c="orange")
        axis[i].set_xlabel('Date', fontsize=12)
        axis[i].set_ylabel('Production (kW)', fontsize=12)
        #axis[i].set_title(source)
        axis[i].legend()
        i+=1
```

```
fig.tight_layout()
```



[46]:
```
# sum over all power plants
benchmark_wind_portfolio = benchmark_wind_kw.groupby(level="time").sum()
```

[47]:
```
# define a scoring
scoring_benchmark = Scoring(benchmark_wind_portfolio ,␣
↪target="actual_power_kw", normalizing_col="installed_capacity_kw")
```

[48]:
```
# compute the nAE
nAE = scoring_benchmark.normalized_absolute_error()
nAE
```

[48]:
```
                           sklearn_lin_reg  sklearn_sgd  h2o_gboost
time
2020-01-01 00:00:00+01:00         0.348391     0.356722    0.034527
2020-01-01 00:30:00+01:00         0.348430     0.356680    0.035228
2020-01-01 01:00:00+01:00         0.339901     0.348071    0.038276
2020-01-01 01:30:00+01:00         0.352671     0.360762    0.028830
2020-01-01 02:00:00+01:00         0.370592     0.378604    0.020328
…                                      …            …          …
2021-12-31 20:00:00+01:00         0.023152     0.024953    0.018877
2021-12-31 20:30:00+01:00         0.046755     0.048689    0.020735
2021-12-31 21:00:00+01:00         0.123928     0.125996    0.080764
2021-12-31 21:30:00+01:00         0.161337     0.163538    0.089289
2021-12-31 22:00:00+01:00         0.208618     0.210952    0.116479

[35085 rows x 3 columns]
```

[50]:
```
nMAPE = nAE.mean()
nMAPE
```

[50]:
```
sklearn_lin_reg    0.118663
sklearn_sgd        0.119341
h2o_gboost         0.041352
dtype: float64
```

5 Conclusion

Do it with solar or run of river !