ExampleE

June 1, 2022

1 Project enda : Example E

In this example we will set up a more complete dayahead power generation prediction.

[1]:

```python
import enda
import datetime
import os
import pandas as pd
import time

from enda.contracts import Contracts
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
```

[2]:

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

1.1 Read and prepare data

[3]:

```python
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(
        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'
    )

    # get production
    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 28.4 s, sys: 1.13 s, total: 29.5 s
Wall time: 29.7 s
```

[5]:

```
%%time
dataset_solar = get_example_e_dataset("solar")
```

```
CPU times: user 1min 9s, sys: 4.94 s, total: 1min 14s
Wall time: 1min 14s
```

[6]:

```
%%time
dataset_river = get_example_e_dataset("river")
```

```
CPU times: user 2min 37s, sys: 7.89 s, total: 2min 45s
Wall time: 2min 45s
```

[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()}
```

1.2 Separe between training and forecasting dataset to backtest the data

We have here the full datasets which have been built using the enda utilities functions, and some historical information gathered from the TSO, diverse weather forecast suppliers, and contracts data.

We will now distinguish within our full dataset a training and a forecasting datasets, as a representative example of what could be obtained in real life conditions.

[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)
```

2 Make a prediction

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

[11]:

```
# import ML backends
from enda.ml_backends.sklearn_estimator import EndaSklearnEstimator
from sklearn.linear_model import LinearRegression
from enda.estimators import EndaEstimatorRecopy

# import power predictors
from enda.power_predictor import PowerPredictor
```

2.0.1 Run of river prediction

[12]:

```
# build a PowerPredictor obejct
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                  425.0     0.525412
        2019-12-22 00:30:00+01:00                  425.0     0.555059
        2019-12-22 01:00:00+01:00                  425.0     0.601176
        2019-12-22 01:30:00+01:00                  425.0     0.607765
        2019-12-22 02:00:00+01:00                  425.0     0.606118
…                                                  …            …
hy_99   2021-11-30 21:30:00+01:00                   57.5     1.456812
        2021-11-30 22:00:00+01:00                   57.5     1.432464
        2021-11-30 22:30:00+01:00                   57.5     1.391884
        2021-11-30 23:00:00+01:00                   57.5     1.262029
        2021-11-30 23:30:00+01:00                   57.5     1.172754
```

```
[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      0.640737
        2021-12-31 22:00:00+01:00      0.640737
        2021-12-31 22:30:00+01:00      0.640737
        2021-12-31 23:00:00+01:00      0.640737
        2021-12-31 23:30:00+01:00      0.640737

[123504 rows x 1 columns]
```

2.0.2 Wind prediction

[16]:
```
# 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/tmpsar6tuub
  JVM stdout: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpsar6tuub/h2o_c
lement_jeannesson_started_from_python.out
  JVM stderr: /var/folders/pp/kyc80_js50g283hj0_c4yrhc0000gp/T/tmpsar6tuub/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:    1 month and 18 days
H2O_cluster_name:           H2O_from_python_clement_jeannesson_yslcj1
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

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_positive = 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     0.052628
        2021-12-31 20:30:00+01:00     0.048231
        2021-12-31 21:00:00+01:00     0.048231
        2021-12-31 21:30:00+01:00     0.048231
        2021-12-31 22:00:00+01:00     0.047366

[28215 rows x 1 columns]
```

2.0.3 Solar prediction

[22]:

```
# 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_positive=True)
```

[23]:

```
pred_solar
```

[23]:

```
                                   load_factor
station time
pv_0    2021-12-01 00:00:00+01:00     0.002614
        2021-12-01 00:30:00+01:00     0.002614
        2021-12-01 01:00:00+01:00     0.002506
        2021-12-01 01:30:00+01:00     0.002621
        2021-12-01 02:00:00+01:00     0.002614
…                                          …
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]
```

[24]:

```
# 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_a2c3 closed.
```

2.1 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)

[25]:

```
# 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
             }
```

[26]:

```
prediction["river"]
```

[26]:

```
                                   installed_capacity_kw  load_factor
station time
hy_0    2021-12-01 00:00:00+01:00                  425.0     0.000000
        2021-12-01 00:30:00+01:00                  425.0     0.000000
        2021-12-01 01:00:00+01:00                  425.0     0.000000
        2021-12-01 01:30:00+01:00                  425.0     0.000000
        2021-12-01 02:00:00+01:00                  425.0     0.000000
…                                                  …           …
hy_99   2021-12-31 21:30:00+01:00                   57.5     0.640737
        2021-12-31 22:00:00+01:00                   57.5     0.640737
        2021-12-31 22:30:00+01:00                   57.5     0.640737
        2021-12-31 23:00:00+01:00                   57.5     0.640737
        2021-12-31 23:30:00+01:00                   57.5     0.640737

[123504 rows x 2 columns]
```

[27]:

```
# 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()}
```

2.1.1 Plot predicted data along with the real production

[28]:

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

# 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()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```