ExampleB

March 24, 2021

1 Project enda : Example B

If you haven't already, read Example A first, it is not long. Run this notebook in the correct python environment.

In this example we will go more in depth, with realistic data and more historical data (~4-5 years). This example is divided in 7 parts: 1. Read and prepare data, check for missing values and gaps 2. Visualize data 3. Feature engineering : datetime and calendar features 4. Portfolio forecast & basic prediction 5. Benchmark with simple evaluation 6. Benchmark with Backtesting 7. Make the prediction

We set ourselves in a setup as if we were **exactly on 2020-11-30**. We want to predict the total consumption of customers for the next few days starting 2020-12-01 at a 30min time-step. We have: - our customer contracts until 2020-11-30 included. - historical load data from 2015-01-01 until 2020-11-15 included. - weather forecast until 2020-12-11 (11 days). - our TSO's network load forecast until 2020-12-7 (7 days).

In here (example B), we will put all our customers in only 1 group and forecast the next 7 days. We will first construct the dataset and the forecast input data and test it with a basic linear regressor. We will then try various algorithms and compare them. Finally we will give an example of backtesting on the data.

[1]:
```
import enda
import pandas as pd
import os
import enda.ml_backends.sklearn_linreg
from enda.ml_backends.sklearn_linreg import SKLearnLinearRegression
import joblib
```

1.1 1. Read and prepare data, check for missing values and gaps

[2]:
```
# download and unzip example_b.zip then replace this with the path to you␣
↪example-b directory.
DIR = '/Users/emmanuel.charon/Documents/CodeProjects/enercoop/enda/data/
↪example_b'
```

[3]:
```
# get the 30min time-step data just like in Example A (columns are a bit␣
↪different and there is more data)
# here we consider all customers in one big group
```

```
def read_data():
    contracts = enda.Contracts.read_contracts_from_file(os.path.join(DIR,␣
 ↪"contracts.csv"))
    contracts["contracts_count"] = 1
    portfolio_by_day = enda.Contracts.compute_portfolio_by_day(
        contracts,
        columns_to_sum = ["contracts_count", "kva"],
        date_start_col="date_start",
        date_end_exclusive_col="date_end_exclusive",
    )
    portfolio = enda.TimeSeries.interpolate_daily_to_sub_daily_data(
        portfolio_by_day,
        freq='30min',
        tz='Europe/Paris'
    )

    historic_load_measured = pd.read_csv(os.path.join(DIR,␣
 ↪"historic_load_measured.csv"))
    weather_and_tso_forecasts = pd.read_csv(os.path.join(DIR,␣
 ↪"weather_and_tso_forecasts.csv"))
    # correctly format 'time' as a pandas.DatetimeIndex of dtype: datetime[ns,␣
 ↪tzinfo]
    for df in [historic_load_measured, weather_and_tso_forecasts]:
        df['time'] = pd.to_datetime(df['time'])
        df['time'] = enda.TimeSeries.align_timezone(df['time'], tzinfo =␣
 ↪'Europe/Paris')
        df.set_index('time', inplace=True)

    # keep only where both loads are known
    historic_load_measured = historic_load_measured.dropna()
    historic_load_measured["load_kw"] =␣
 ↪historic_load_measured["smart_metered_kw"] + historic_load_measured["slp_kw"]
    # keep only the full load
    historic_load_measured = historic_load_measured[["load_kw"]]


    return contracts, portfolio, historic_load_measured,␣
 ↪weather_and_tso_forecasts
```

[4]:

```
contracts, portfolio, historic_load_measured, weather_and_tso_forecasts =␣
 ↪read_data()
# remove data where tso is not available
weather_and_tso_forecasts = weather_and_tso_forecasts.
 ↪dropna(subset=["tso_forecast_load_mw"])
```

[5]:

```
contracts
```

[5]:
```
        date_start date_end_exclusive   kva meter_reading_type  contracts_count
0       2006-08-09                NaT  12.0            PROFILE                1
1       2006-09-01         2006-11-23   6.0            PROFILE                1
2       2006-09-01         2007-11-01   3.0            PROFILE                1
3       2006-09-01         2007-12-19  12.0            PROFILE                1
4       2006-09-01         2008-06-28  12.0            PROFILE                1
…              …                  …   …                  …                …
162598  2020-11-30                NaT   6.0            PROFILE                1
162599  2020-11-30                NaT   6.0            PROFILE                1
162600  2020-11-30                NaT   6.0            PROFILE                1
162601  2020-11-30                NaT   6.0            PROFILE                1
162602  2020-11-30                NaT   6.0            PROFILE                1

[162603 rows x 5 columns]
```

[6]:
```
portfolio
```

[6]:
```
                           contracts_count       kva
time
2006-08-09 00:00:00+02:00              1.0      12.0
2006-08-09 00:30:00+02:00              1.0      12.0
2006-08-09 01:00:00+02:00              1.0      12.0
2006-08-09 01:30:00+02:00              1.0      12.0
2006-08-09 02:00:00+02:00              1.0      12.0
…                                        …        …
2020-11-30 21:30:00+01:00          96134.0  820005.7
2020-11-30 22:00:00+01:00          96134.0  820005.7
2020-11-30 22:30:00+01:00          96134.0  820005.7
2020-11-30 23:00:00+01:00          96134.0  820005.7
2020-11-30 23:30:00+01:00          96134.0  820005.7

[250946 rows x 2 columns]
```

[7]:
```
historic_load_measured
```

[7]:
```
                               load_kw
time
2015-01-01 00:00:00+01:00  2490.925806
2015-01-01 00:30:00+01:00  2412.623113
2015-01-01 01:00:00+01:00  2365.611276
2015-01-01 01:30:00+01:00  2336.141065
2015-01-01 02:00:00+01:00  2300.935642
…                                  …
2020-11-15 21:30:00+01:00  7657.293444
2020-11-15 22:00:00+01:00  7317.540759
2020-11-15 22:30:00+01:00  7580.051439
2020-11-15 23:00:00+01:00  7496.273993
```

```
2020-11-15 23:30:00+01:00  7376.005701

[97198 rows x 1 columns]
```

[8]:
```
# t_weighted is the average french temperature weighted by population density
# t_smooth is a smoothing computed over t_weighted to take into account␣
↪building calorific inertia
weather_and_tso_forecasts
```

[8]:
```
                           tso_forecast_load_mw  t_weighted  t_smooth
time
2015-01-01 00:00:00+01:00               72900.0       -0.41      1.17
2015-01-01 00:30:00+01:00               71600.0       -0.48      1.17
2015-01-01 01:00:00+01:00               69900.0       -0.55      1.15
2015-01-01 01:30:00+01:00               70600.0       -0.66      1.14
2015-01-01 02:00:00+01:00               70500.0       -0.78      1.11
…                                           …          …        …
2020-12-07 21:30:00+01:00               68400.0        4.20      4.13
2020-12-07 22:00:00+01:00               66900.0        4.12      4.10
2020-12-07 22:30:00+01:00               67600.0        4.03      4.08
2020-12-07 23:00:00+01:00               70200.0        3.94      4.07
2020-12-07 23:30:00+01:00               69600.0        3.94      4.07

[104064 rows x 3 columns]
```

[9]:
```
# lets create the train set with historical data
historic = pd.merge(
    portfolio,
    historic_load_measured, # here we select only the load of the desired group
    how='inner', left_index=True, right_index=True
)

historic = pd.merge(
    historic,
    weather_and_tso_forecasts,
    how='inner', left_index=True, right_index=True
)
```

[10]:
```
historic
```

[10]:
```
                           contracts_count       kva      load_kw  \
time
2015-01-01 00:00:00+01:00          21261.0  167416.4  2490.925806
2015-01-01 00:30:00+01:00          21261.0  167416.4  2412.623113
2015-01-01 01:00:00+01:00          21261.0  167416.4  2365.611276
2015-01-01 01:30:00+01:00          21261.0  167416.4  2336.141065
2015-01-01 02:00:00+01:00          21261.0  167416.4  2300.935642
```

```
…                                      …        …          …
2020-11-15 21:30:00+01:00          95475.0  813328.8  7657.293444
2020-11-15 22:00:00+01:00          95475.0  813328.8  7317.540759
2020-11-15 22:30:00+01:00          95475.0  813328.8  7580.051439
2020-11-15 23:00:00+01:00          95475.0  813328.8  7496.273993
2020-11-15 23:30:00+01:00          95475.0  813328.8  7376.005701

                           tso_forecast_load_mw  t_weighted  t_smooth
time
2015-01-01 00:00:00+01:00               72900.0       -0.41      1.17
2015-01-01 00:30:00+01:00               71600.0       -0.48      1.17
2015-01-01 01:00:00+01:00               69900.0       -0.55      1.15
2015-01-01 01:30:00+01:00               70600.0       -0.66      1.14
2015-01-01 02:00:00+01:00               70500.0       -0.78      1.11
…                                           …          …        …
2020-11-15 21:30:00+01:00               46200.0       12.05     12.01
2020-11-15 22:00:00+01:00               45200.0       11.92     11.97
2020-11-15 22:30:00+01:00               46400.0       11.84     11.96
2020-11-15 23:00:00+01:00               48600.0       11.75     11.94
2020-11-15 23:30:00+01:00               49400.0       11.64     11.92

[97198 rows x 6 columns]
```

[11]:
```
# check that there is no NaN value
historic.isna().sum()
```

[11]:
```
contracts_count         0
kva                     0
load_kw                 0
tso_forecast_load_mw    0
t_weighted              0
t_smooth                0
dtype: int64
```

[12]:
```
# check missing data in the timeseries (based on the time index only)
freq, missing_periods, extra_points = enda.TimeSeries.
 ↪find_missing_and_extra_periods(
    dti=historic.index,
    expected_freq = '30min',
    expected_start_datetime = pd.to_datetime('2015-01-01 00:00:00+01:00').
 ↪astimezone('Europe/Paris'),
    expected_end_datetime = pd.to_datetime('2020-11-30 23:30:00+01:00').
 ↪astimezone('Europe/Paris')
)
for missing_period in missing_periods:
    print("Missing data from {} to {}.".format(missing_period[0],␣
 ↪missing_period[1]))
```

```
if len(extra_points) > 0 :
    print("Extra points found: {}".format(extra_points))
```

```
Missing data from 2015-09-01 00:00:00+02:00 to 2015-11-30 23:30:00+01:00.
Missing data from 2018-06-01 00:00:00+02:00 to 2018-06-30 23:30:00+02:00.
Missing data from 2020-11-16 00:00:00+01:00 to 2020-11-30 23:30:00+01:00.
```

We expected the missing data from 2020-11-16 to 2020-11-30, but not from the rest.

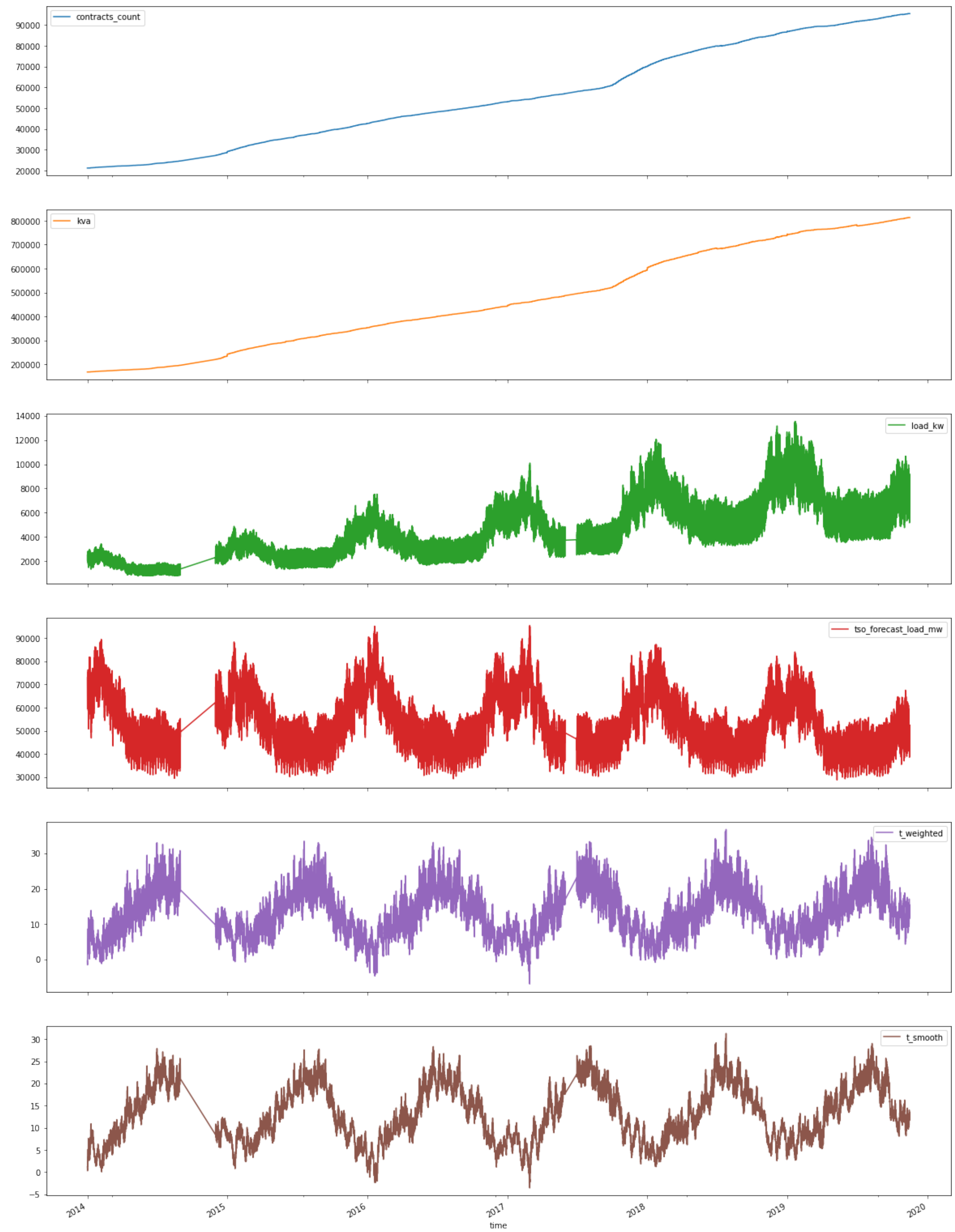
1.2 2. Visualize data

To visualise using pandas, you need matplotlib

```
pip install matplotlib
```

[13]:
```
# Show full data set
historic.plot(figsize=(20, 30), subplots=True)
```

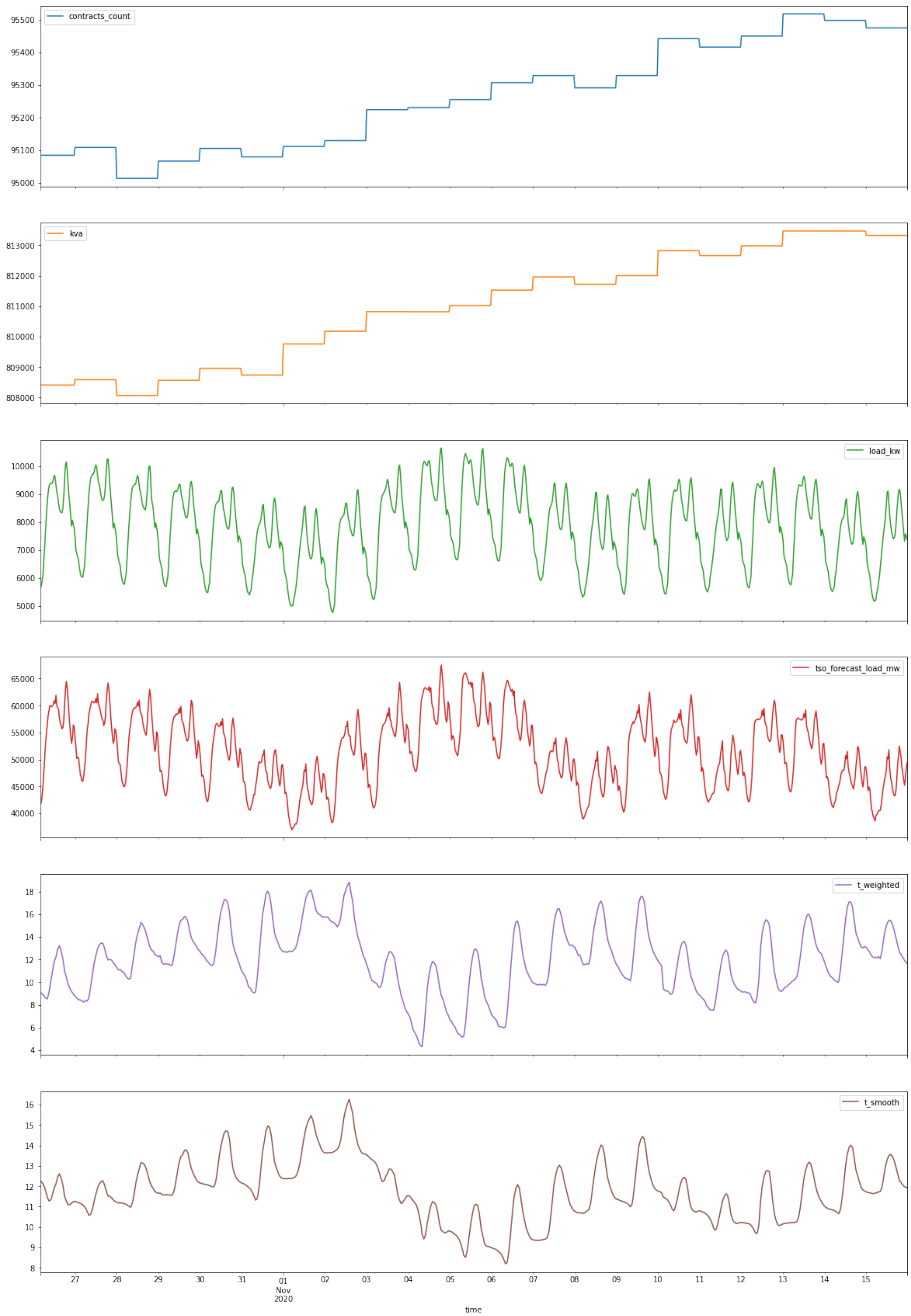
[13]:
```
array([<AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
       <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
       <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>],
      dtype=object)
```



[14]:

```
# Show recent data
historic[-1000:].plot(figsize=(20, 30), subplots=True)
```

```
[14]: array([<AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
              <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>,
              <AxesSubplot:xlabel='time'>, <AxesSubplot:xlabel='time'>],
             dtype=object)
```



contracts_count
95000
95100
95200
95300
95400
95500
kva
808000
809000
810000
811000
812000
813000
load_kw
5000
6000
7000
8000
9000
10000
tso_forecast_load_mw
40000
45000
50000
55000
60000
65000
t_weighted
4
6
8
10
12
14
16
18
t_smooth
8
9
10
11
12
13
14
15
16
27
28
29
30
31
01
Nov
2020
02
03
04
05
06
07
08
09
10
11
12
13
14
15
time

Don't hesitate to add your own visualisations!

```
[ ]:
```

1.3 3. Feature engineering

Before we train, we will add some features based on the `datetime`, and some calendar features related to national holidays or school holydays.

We use some packages for the holidays, which are used in **enda.feature_engineering.calendar** :

`pip install jours-feries-france vacances-scolaires-france Unidecode`

```
[15]: import enda.feature_engineering.calendar
```

```
[16]: # define the features we want to add before training/predicting
def featurize(df):
    # put datetime features to capture the data frequencies: daily, weekly and␣
↪yearly periods.
    df = enda.DatetimeFeature.split_datetime(
        df, split_list = ['minuteofday', 'dayofweek', 'month']
    )
    df = enda.DatetimeFeature.encode_cyclic_datetime_index(
        df, split_list = ['minuteofday', 'dayofweek', 'dayofyear']
    )
    special_days = enda.feature_engineering.calendar.Calendar().
↪get_french_special_days()
    df = pd.merge(
        df, special_days,
        how='left', left_index=True, right_index=True
    )
    return df
```

```
[17]: full_train_set = featurize(historic)
```

```
[18]: full_train_set
```

```
[18]:                            contracts_count       kva      load_kw  \
time
2015-01-01 00:00:00+01:00          21261.0  167416.4  2490.925806
2015-01-01 00:30:00+01:00          21261.0  167416.4  2412.623113
2015-01-01 01:00:00+01:00          21261.0  167416.4  2365.611276
2015-01-01 01:30:00+01:00          21261.0  167416.4  2336.141065
2015-01-01 02:00:00+01:00          21261.0  167416.4  2300.935642
…                                      …         …          …
2020-11-15 21:30:00+01:00          95475.0  813328.8  7657.293444
```

```
2020-11-15 22:00:00+01:00          95475.0  813328.8  7317.540759
2020-11-15 22:30:00+01:00          95475.0  813328.8  7580.051439
2020-11-15 23:00:00+01:00          95475.0  813328.8  7496.273993
2020-11-15 23:30:00+01:00          95475.0  813328.8  7376.005701

                           tso_forecast_load_mw  t_weighted  t_smooth  \
time
2015-01-01 00:00:00+01:00               72900.0       -0.41      1.17
2015-01-01 00:30:00+01:00               71600.0       -0.48      1.17
2015-01-01 01:00:00+01:00               69900.0       -0.55      1.15
2015-01-01 01:30:00+01:00               70600.0       -0.66      1.14
2015-01-01 02:00:00+01:00               70500.0       -0.78      1.11
…                                          …          …       …
2020-11-15 21:30:00+01:00               46200.0       12.05     12.01
2020-11-15 22:00:00+01:00               45200.0       11.92     11.97
2020-11-15 22:30:00+01:00               46400.0       11.84     11.96
2020-11-15 23:00:00+01:00               48600.0       11.75     11.94
2020-11-15 23:30:00+01:00               49400.0       11.64     11.92

                           minuteofday  dayofweek  month  minuteofday_cos  \
time
2015-01-01 00:00:00+01:00            0          3      1         1.000000
2015-01-01 00:30:00+01:00           30          3      1         0.991445
2015-01-01 01:00:00+01:00           60          3      1         0.965926
2015-01-01 01:30:00+01:00           90          3      1         0.923880
2015-01-01 02:00:00+01:00          120          3      1         0.866025
…                                  …         …    …               …
2020-11-15 21:30:00+01:00         1290          6     11         0.793353
2020-11-15 22:00:00+01:00         1320          6     11         0.866025
2020-11-15 22:30:00+01:00         1350          6     11         0.923880
2020-11-15 23:00:00+01:00         1380          6     11         0.965926
2020-11-15 23:30:00+01:00         1410          6     11         0.991445

                           minuteofday_sin  dayofweek_cos  dayofweek_sin  \
time
2015-01-01 00:00:00+01:00         0.000000      -0.900969       0.433884
2015-01-01 00:30:00+01:00         0.130526      -0.900969       0.433884
2015-01-01 01:00:00+01:00         0.258819      -0.900969       0.433884
2015-01-01 01:30:00+01:00         0.382683      -0.900969       0.433884
2015-01-01 02:00:00+01:00         0.500000      -0.900969       0.433884
…                                      …             …             …
2020-11-15 21:30:00+01:00        -0.608761       0.623490      -0.781831
2020-11-15 22:00:00+01:00        -0.500000       0.623490      -0.781831
2020-11-15 22:30:00+01:00        -0.382683       0.623490      -0.781831
2020-11-15 23:00:00+01:00        -0.258819       0.623490      -0.781831
2020-11-15 23:30:00+01:00        -0.130526       0.623490      -0.781831
```

```
                           dayofyear_cos  dayofyear_sin  lockdown  \
time
2015-01-01 00:00:00+01:00       1.000000       0.000000       0.0
2015-01-01 00:30:00+01:00       1.000000       0.000000       0.0
2015-01-01 01:00:00+01:00       1.000000       0.000000       0.0
2015-01-01 01:30:00+01:00       1.000000       0.000000       0.0
2015-01-01 02:00:00+01:00       1.000000       0.000000       0.0
…                                  …             …          …
2020-11-15 21:30:00+01:00       0.691771      -0.722117       0.0
2020-11-15 22:00:00+01:00       0.691771      -0.722117       0.0
2020-11-15 22:30:00+01:00       0.691771      -0.722117       0.0
2020-11-15 23:00:00+01:00       0.691771      -0.722117       0.0
2020-11-15 23:30:00+01:00       0.691771      -0.722117       0.0

                           public_holiday  nb_school_areas_off  \
time
2015-01-01 00:00:00+01:00             1.0                  3.0
2015-01-01 00:30:00+01:00             1.0                  3.0
2015-01-01 01:00:00+01:00             1.0                  3.0
2015-01-01 01:30:00+01:00             1.0                  3.0
2015-01-01 02:00:00+01:00             1.0                  3.0
…                                    …                  …
2020-11-15 21:30:00+01:00             0.0                  0.0
2020-11-15 22:00:00+01:00             0.0                  0.0
2020-11-15 22:30:00+01:00             0.0                  0.0
2020-11-15 23:00:00+01:00             0.0                  0.0
2020-11-15 23:30:00+01:00             0.0                  0.0

                           extra_long_weekend
time
2015-01-01 00:00:00+01:00                 0.0
2015-01-01 00:30:00+01:00                 0.0
2015-01-01 01:00:00+01:00                 0.0
2015-01-01 01:30:00+01:00                 0.0
2015-01-01 02:00:00+01:00                 0.0
…                                       …
2020-11-15 21:30:00+01:00                 0.0
2020-11-15 22:00:00+01:00                 0.0
2020-11-15 22:30:00+01:00                 0.0
2020-11-15 23:00:00+01:00                 0.0
2020-11-15 23:30:00+01:00                 0.0

[97198 rows x 19 columns]
```

[19]:

```
# train a basic SKLearnLinearRegression
lin_reg = SKLearnLinearRegression()
lin_reg.train(full_train_set, target_col='load_kw')
```

1.4 4. Portfolio forecast & basic prediction

We need an estimate of our portfolio in the next few days, the tso_load and weather forecasts.

In order to get our portfolio in the next few days, here we will just consider the latest trends in our portfolio.

In another setup, you might want to connect to your sales software or ERP and take into account contracts that will end or start soon.

We will use `enda.Contracts.forecast_using_trend`which requires the `statsmodel` package :

`pip install statsmodels`

[20]:

```
# we will forecast the portfolio using holt method
forecast_portfolio =  enda.Contracts.forecast_using_trend(
    portfolio_df=portfolio,
    start_forecast_date=pd.to_datetime("2020-12-01 00:00:00+01:00"),
    nb_days=7,
    past_days=150  # only use recent portfolio trend to forecast the next few␣
 ↪days
)
forecast_portfolio
```

```
/Users/emmanuel.charon/Documents/CodeProjects/enercoop/enda/venv/lib/python3.7/s
ite-packages/statsmodels/tsa/holtwinters/model.py:922: ConvergenceWarning:
Optimization failed to converge. Check mle_retvals.
  ConvergenceWarning,
```

[20]:

```
                           contracts_count       kva
time
2020-12-01 00:00:00+01:00          96134.6  820008.8
2020-12-01 00:30:00+01:00          96135.3  820011.8
2020-12-01 01:00:00+01:00          96135.9  820014.9
2020-12-01 01:30:00+01:00          96136.5  820017.9
2020-12-01 02:00:00+01:00          96137.1  820021.0
…                                      …         …
2020-12-07 21:30:00+01:00          96341.2  821020.6
2020-12-07 22:00:00+01:00          96341.8  821023.7
2020-12-07 22:30:00+01:00          96342.4  821026.7
2020-12-07 23:00:00+01:00          96343.1  821029.8
2020-12-07 23:30:00+01:00          96343.7  821032.8

[336 rows x 2 columns]
```

[21]:

```
# add weather_and_tso_forecasts
forecast_input_data = pd.merge(
    forecast_portfolio,
    weather_and_tso_forecasts,
    how='inner', left_index=True, right_index=True
```

```
)
# add feature engineering
forecast_input_data = featurize(forecast_input_data)
forecast_input_data
```

[21]:

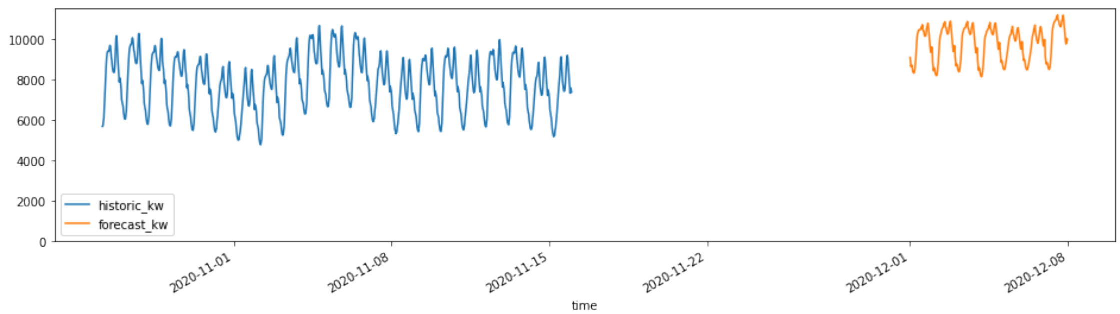
```
                           contracts_count       kva  tso_forecast_load_mw  \
time
2020-12-01 00:00:00+01:00          96134.6  820008.8               66100.0
2020-12-01 00:30:00+01:00          96135.3  820011.8               64200.0
2020-12-01 01:00:00+01:00          96135.9  820014.9               61900.0
2020-12-01 01:30:00+01:00          96136.5  820017.9               62800.0
2020-12-01 02:00:00+01:00          96137.1  820021.0               62300.0
…                                      …         …                    …
2020-12-07 21:30:00+01:00          96341.2  821020.6               68400.0
2020-12-07 22:00:00+01:00          96341.8  821023.7               66900.0
2020-12-07 22:30:00+01:00          96342.4  821026.7               67600.0
2020-12-07 23:00:00+01:00          96343.1  821029.8               70200.0
2020-12-07 23:30:00+01:00          96343.7  821032.8               69600.0

                           t_weighted  t_smooth  minuteofday  dayofweek  \
time
2020-12-01 00:00:00+01:00        4.69      5.08            0          1
2020-12-01 00:30:00+01:00        4.82      5.10           30          1
2020-12-01 01:00:00+01:00        4.96      5.12           60          1
2020-12-01 01:30:00+01:00        5.04      5.13           90          1
2020-12-01 02:00:00+01:00        5.13      5.14          120          1
…                                   …         …            …          …
2020-12-07 21:30:00+01:00        4.20      4.13         1290          0
2020-12-07 22:00:00+01:00        4.12      4.10         1320          0
2020-12-07 22:30:00+01:00        4.03      4.08         1350          0
2020-12-07 23:00:00+01:00        3.94      4.07         1380          0
2020-12-07 23:30:00+01:00        3.94      4.07         1410          0

                           month  minuteofday_cos  minuteofday_sin  \
time
2020-12-01 00:00:00+01:00     12         1.000000         0.000000
2020-12-01 00:30:00+01:00     12         0.991445         0.130526
2020-12-01 01:00:00+01:00     12         0.965926         0.258819
2020-12-01 01:30:00+01:00     12         0.923880         0.382683
2020-12-01 02:00:00+01:00     12         0.866025         0.500000
…                              …                …                …
2020-12-07 21:30:00+01:00     12         0.793353        -0.608761
2020-12-07 22:00:00+01:00     12         0.866025        -0.500000
2020-12-07 22:30:00+01:00     12         0.923880        -0.382683
2020-12-07 23:00:00+01:00     12         0.965926        -0.258819
2020-12-07 23:30:00+01:00     12         0.991445        -0.130526
```

```
                           dayofweek_cos  dayofweek_sin  dayofyear_cos  \
time
2020-12-01 00:00:00+01:00        0.62349       0.781831       0.861702
2020-12-01 00:30:00+01:00        0.62349       0.781831       0.861702
2020-12-01 01:00:00+01:00        0.62349       0.781831       0.861702
2020-12-01 01:30:00+01:00        0.62349       0.781831       0.861702
2020-12-01 02:00:00+01:00        0.62349       0.781831       0.861702
…                                    …            …             …
2020-12-07 21:30:00+01:00        1.00000       0.000000       0.909308
2020-12-07 22:00:00+01:00        1.00000       0.000000       0.909308
2020-12-07 22:30:00+01:00        1.00000       0.000000       0.909308
2020-12-07 23:00:00+01:00        1.00000       0.000000       0.909308
2020-12-07 23:30:00+01:00        1.00000       0.000000       0.909308

                           dayofyear_sin  lockdown  public_holiday  \
time
2020-12-01 00:00:00+01:00      -0.507415       0.0             0.0
2020-12-01 00:30:00+01:00      -0.507415       0.0             0.0
2020-12-01 01:00:00+01:00      -0.507415       0.0             0.0
2020-12-01 01:30:00+01:00      -0.507415       0.0             0.0
2020-12-01 02:00:00+01:00      -0.507415       0.0             0.0
…                                    …       …             …
2020-12-07 21:30:00+01:00      -0.416125       0.0             0.0
2020-12-07 22:00:00+01:00      -0.416125       0.0             0.0
2020-12-07 22:30:00+01:00      -0.416125       0.0             0.0
2020-12-07 23:00:00+01:00      -0.416125       0.0             0.0
2020-12-07 23:30:00+01:00      -0.416125       0.0             0.0

                           nb_school_areas_off  extra_long_weekend
time
2020-12-01 00:00:00+01:00                  0.0                 0.0
2020-12-01 00:30:00+01:00                  0.0                 0.0
2020-12-01 01:00:00+01:00                  0.0                 0.0
2020-12-01 01:30:00+01:00                  0.0                 0.0
2020-12-01 02:00:00+01:00                  0.0                 0.0
…                                        …                  …
2020-12-07 21:30:00+01:00                  0.0                 0.0
2020-12-07 22:00:00+01:00                  0.0                 0.0
2020-12-07 22:30:00+01:00                  0.0                 0.0
2020-12-07 23:00:00+01:00                  0.0                 0.0
2020-12-07 23:30:00+01:00                  0.0                 0.0

[336 rows x 18 columns]
```

[22]:
```
# do the prediction
lin_reg_prediction = lin_reg.predict(forecast_input_data, target_col="load_kw")
```

[23]:
```python
# visualize recent load along with our forecast; remember we don't have recent␣
↪actual load so there is a time-gap.
to_plot = pd.merge(
    historic["load_kw"][-1000:].to_frame("historic_kw"),
    lin_reg_prediction.rename(columns={"load_kw": "forecast_kw"}),
    how='outer', left_index=True, right_index=True
)
to_plot.plot(ylim=0, figsize=(16, 4))
```

[23]:
```
<AxesSubplot:xlabel='time'>
```



1.5 5. Benchmark with simple evaluation

We can see that the previous prediction is pretty poor, so we can try and use a better algorithm.

For that we will use h2o as a backend for classic machine learning algorithms, and enda's models on top of them.

For enda's `H2OModel` to work, we need the `h2o` package:

`pip install h2o`

[24]:
```python
import h2o
from enda.ml_backends.h2o_model import H2OModel
import time
```

[25]:
```python
# Lets define some algorithms then train and predict with them
all_models = dict()
```

[26]:
```python
# keep the basic one for the benchmark
all_models['sk_lin_reg'] = SKLearnLinearRegression()

# H2O's linear regressor
all_models['h2o_lin_reg'] = H2OModel(algo_name="glm", model_id="h2o_glm",␣
↪target="load_kw", algo_param_dict={})

# a random forest
```

```
all_models['h2o_rf'] = H2OModel(
    algo_name="randomforest",
    model_id="h2o_randomforest",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [300],
        "max_depth": [15],
        "sample_rate": [0.8],
        "min_rows": [10],
        "nbins": [52],
        "mtries": [3]
    }
)

# a GBM
all_models['h2o_gbm'] = H2OModel(
    algo_name="gbm",
    model_id="h2o_gbm",  # H2O requires a unique model_id for each algo we␣
↪train with it
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.5],
        "min_rows": [5]
    }
)

# an XGBoost
all_models['h2o_xgboost'] = H2OModel(
    algo_name="xgboost",
    model_id="h2o_xgboost",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.8],
        "min_rows": [10]
    },
)
```

[27]:
```
# an algorithm using Enda : "Normlized xgboost using kva"
enda_n = enda.models.NormalizedModel(
    normalized_model = H2OModel(
        algo_name="xgboost",
        model_id="enda_n_xgboost",
        target="load_kw",
```

```
        algo_param_dict= {
            "ntrees": [500],
            "max_depth": [5],
            "sample_rate": [0.8],
            "min_rows": [10]
        },
    ),
    target_col = "load_kw",
    normalization_col = "kva",
    columns_to_normalize = ["contracts_count"]
)
# all_models['enda_n'] = enda_n
```

[28]:

```
# another algorithm using Enda : "glm-stacking of [randomforest, gbm, xgboost]"
_m_randomforest = H2OModel(
    algo_name="randomforest",
    model_id="enda_s_randomforest",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [300],
        "max_depth": [15],
        "sample_rate": [0.8],
        "min_rows": [10],
        "nbins": [52],
        "mtries": [3]
    }
)

_m_gbm = H2OModel(
    algo_name="gbm",
    model_id="enda_s_gbm",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.5],
        "min_rows": [5]
    }
)

_m_xgboost = H2OModel(
    algo_name="xgboost",
    model_id="enda_s_xgboost",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
```

```
        "sample_rate": [0.8],
        "min_rows": [10]
    }
)

enda_s = enda.models.StackingModel(
    base_models = {
        "randomforest": _m_randomforest,
        "gbm": _m_gbm,
        "xgboost": _m_xgboost
    },
    final_model = H2OModel(
        algo_name="xgboost",
        model_id="enda_s_xgboost",
        target="load_kw",
        algo_param_dict= {
            "ntrees": [500],
            "max_depth": [5],
            "sample_rate": [0.8],
            "min_rows": [10]
        }
    )
)
# all_models['enda_s'] = enda_s
```

[29]:
```
# another Enda algorithm : "normalized glm-stacking of [randomforest, gbm,␣
↪xgboost]"

_m_randomforest = H2OModel(
    algo_name="randomforest",
    model_id="enda_ns_randomforest",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [300],
        "max_depth": [15],
        "sample_rate": [0.8],
        "min_rows": [10],
        "nbins": [52],
        "mtries": [3]
    }
)

_m_gbm = H2OModel(
    algo_name="gbm",
    model_id="enda_ns_gbm",
    target="load_kw",
    algo_param_dict= {
```

```
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.5],
        "min_rows": [5]
    }
)

_m_xgboost = H2OModel(
    algo_name="xgboost",
    model_id="enda_ns_xgboost",
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.8],
        "min_rows": [10]
    }
)

_m_stacking = enda.models.StackingModel(
    base_models = {
        "randomforest": _m_randomforest,
        "gbm": _m_gbm,
        "xgboost": _m_xgboost
    },
    final_model = H2OModel(algo_name="glm", model_id="enda_ns_stacking_glm",␣
↪target="load_kw", algo_param_dict={})
)

m_enda_ns = enda.models.NormalizedModel(
    normalized_model = _m_stacking,
    target_col = "load_kw",
    normalization_col = "kva",
    columns_to_normalize = ["contracts_count"]
)

# all_models["enda_ns"] = m_enda_ns
```

[30]:
```
# here we do a benchmark, we want to compare with actual data, lets says from␣
↪2020-11-01 to 2020-11-15
benchmark_train = full_train_set[full_train_set.index < '2020-11-01']
benchmark_test = full_train_set[full_train_set.index >= '2020-11-01']

benchmark = benchmark_test["load_kw"].to_frame("actual_load_kw")

benchmark_test = benchmark_test.drop(columns=["load_kw"])
```

[31]:
```
# use the same method as before to predict a portfolio for 2020-11-01 ->␣
↪2020-11-15
benchmark_test_portfolio = enda.Contracts.forecast_using_trend(
    portfolio_df=portfolio[portfolio.index < '2020-11-01'],
    start_forecast_date=pd.to_datetime("2020-11-01 00:00:00+01:00"),
    nb_days=15,
    past_days=150  # only use recent portfolio trend to forecast the next few␣
↪days
)
benchmark_test['kva'] = benchmark_test_portfolio['kva']
benchmark_test['contracts_count'] = benchmark_test_portfolio['contracts_count']
benchmark_test
```

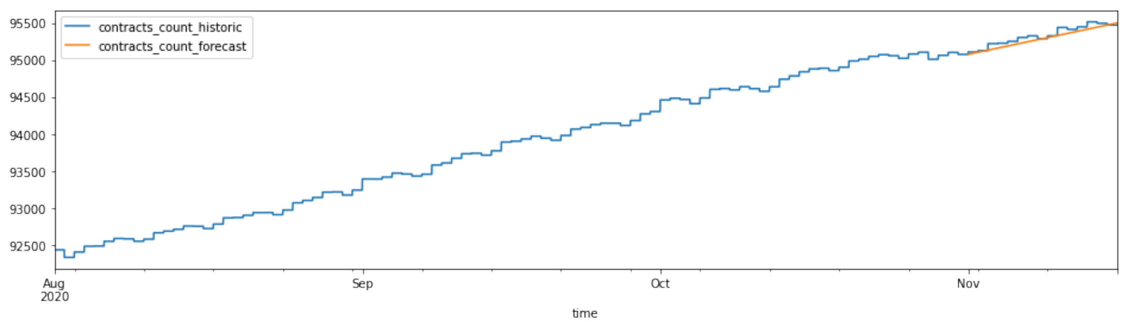
```
/Users/emmanuel.charon/Documents/CodeProjects/enercoop/enda/venv/lib/python3.7/s
ite-packages/statsmodels/tsa/holtwinters/model.py:922: ConvergenceWarning:
Optimization failed to converge. Check mle_retvals.
  ConvergenceWarning,
```

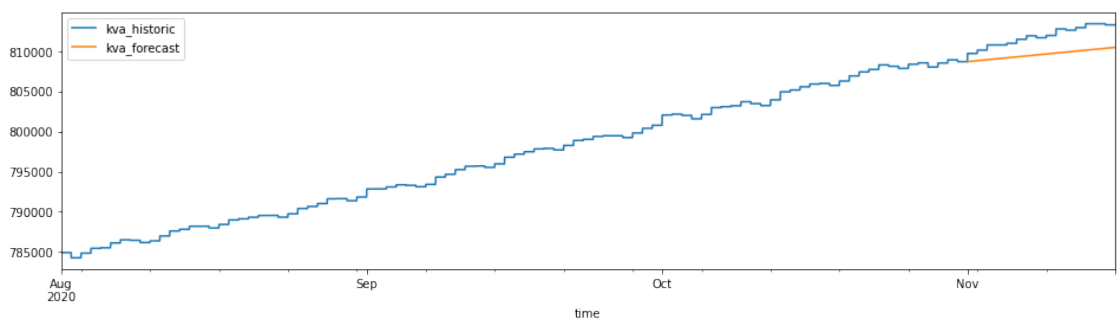
[31]:
```
                           contracts_count       kva  tso_forecast_load_mw  \
time
2020-11-01 00:00:00+01:00          95079.6  808742.3               47900.0
2020-11-01 00:30:00+01:00          95080.2  808744.8               45800.0
2020-11-01 01:00:00+01:00          95080.8  808747.2               43700.0
2020-11-01 01:30:00+01:00          95081.4  808749.7               43900.0
2020-11-01 02:00:00+01:00          95081.9  808752.2               43200.0
…                                      …         …                   …
2020-11-15 21:30:00+01:00          95500.0  810510.5               46200.0
2020-11-15 22:00:00+01:00          95500.6  810512.9               45200.0
2020-11-15 22:30:00+01:00          95501.2  810515.4               46400.0
2020-11-15 23:00:00+01:00          95501.8  810517.9               48600.0
2020-11-15 23:30:00+01:00          95502.4  810520.4               49400.0

                           t_weighted  t_smooth  minuteofday  dayofweek  \
time
2020-11-01 00:00:00+01:00       12.67     12.37            0          6
2020-11-01 00:30:00+01:00       12.68     12.37           30          6
2020-11-01 01:00:00+01:00       12.70     12.37           60          6
2020-11-01 01:30:00+01:00       12.66     12.37           90          6
2020-11-01 02:00:00+01:00       12.63     12.36          120          6
…                                 …       …          …        …
2020-11-15 21:30:00+01:00       12.05     12.01         1290          6
2020-11-15 22:00:00+01:00       11.92     11.97         1320          6
2020-11-15 22:30:00+01:00       11.84     11.96         1350          6
2020-11-15 23:00:00+01:00       11.75     11.94         1380          6
2020-11-15 23:30:00+01:00       11.64     11.92         1410          6

                           month  minuteofday_cos  minuteofday_sin  \
```

```
time
2020-11-01 00:00:00+01:00     11          1.000000         0.000000
2020-11-01 00:30:00+01:00     11          0.991445         0.130526
2020-11-01 01:00:00+01:00     11          0.965926         0.258819
2020-11-01 01:30:00+01:00     11          0.923880         0.382683
2020-11-01 02:00:00+01:00     11          0.866025         0.500000
…                          …               …              …
2020-11-15 21:30:00+01:00     11          0.793353        -0.608761
2020-11-15 22:00:00+01:00     11          0.866025        -0.500000
2020-11-15 22:30:00+01:00     11          0.923880        -0.382683
2020-11-15 23:00:00+01:00     11          0.965926        -0.258819
2020-11-15 23:30:00+01:00     11          0.991445        -0.130526

                           dayofweek_cos  dayofweek_sin  dayofyear_cos  \
time
2020-11-01 00:00:00+01:00        0.62349      -0.781831       0.500000
2020-11-01 00:30:00+01:00        0.62349      -0.781831       0.500000
2020-11-01 01:00:00+01:00        0.62349      -0.781831       0.500000
2020-11-01 01:30:00+01:00        0.62349      -0.781831       0.500000
2020-11-01 02:00:00+01:00        0.62349      -0.781831       0.500000
…                                  …             …              …
2020-11-15 21:30:00+01:00        0.62349      -0.781831       0.691771
2020-11-15 22:00:00+01:00        0.62349      -0.781831       0.691771
2020-11-15 22:30:00+01:00        0.62349      -0.781831       0.691771
2020-11-15 23:00:00+01:00        0.62349      -0.781831       0.691771
2020-11-15 23:30:00+01:00        0.62349      -0.781831       0.691771

                           dayofyear_sin  lockdown  public_holiday  \
time
2020-11-01 00:00:00+01:00      -0.866025       0.0             1.0
2020-11-01 00:30:00+01:00      -0.866025       0.0             1.0
2020-11-01 01:00:00+01:00      -0.866025       0.0             1.0
2020-11-01 01:30:00+01:00      -0.866025       0.0             1.0
2020-11-01 02:00:00+01:00      -0.866025       0.0             1.0
…                                  …       …              …
2020-11-15 21:30:00+01:00      -0.722117       0.0             0.0
2020-11-15 22:00:00+01:00      -0.722117       0.0             0.0
2020-11-15 22:30:00+01:00      -0.722117       0.0             0.0
2020-11-15 23:00:00+01:00      -0.722117       0.0             0.0
2020-11-15 23:30:00+01:00      -0.722117       0.0             0.0

                           nb_school_areas_off  extra_long_weekend
time
2020-11-01 00:00:00+01:00                  3.0                 0.0
2020-11-01 00:30:00+01:00                  3.0                 0.0
2020-11-01 01:00:00+01:00                  3.0                 0.0
2020-11-01 01:30:00+01:00                  3.0                 0.0
```

```
2020-11-01 02:00:00+01:00                    3.0                   0.0
…                                          …                  …
2020-11-15 21:30:00+01:00                    0.0                   0.0
2020-11-15 22:00:00+01:00                    0.0                   0.0
2020-11-15 22:30:00+01:00                    0.0                   0.0
2020-11-15 23:00:00+01:00                    0.0                   0.0
2020-11-15 23:30:00+01:00                    0.0                   0.0

[720 rows x 18 columns]
```

[32]:
```python
# compare portfolio forecast to reality
for c in ["contracts_count", "kva"]:
    to_plot = pd.merge(
        portfolio[(portfolio.index >= '2020-08-01') & (portfolio.index <
 ↪'2020-11-16')][c].to_frame(c+"_historic"),
        benchmark_test[c].to_frame(c+"_forecast"),
        how='outer', left_index=True, right_index=True
    )

    to_plot.plot(figsize=(16, 4))
```





[33]:
```
# to train or predict with H2O models, we boot up a local h2o server
h2o.init(nthreads=-1)
h2o.remove_all()  # in case these were left-overs from a previous run
```

```
Checking whether there is an H2O instance running at http://localhost:54321
… not found.
Attempting to start a local H2O server…
  Java Version: java version "12.0.1" 2019-04-16; Java(TM) SE Runtime
Environment (build 12.0.1+12); Java HotSpot(TM) 64-Bit Server VM (build
12.0.1+12, mixed mode, sharing)
  Starting server from /Users/emmanuel.charon/Documents/CodeProjects/enercoop/en
da/venv/lib/python3.7/site-packages/h2o/backend/bin/h2o.jar
  Ice root: /var/folders/5x/409ks2012xxch_pmbs6qpzfh0000gp/T/tmp1h852xwu
  JVM stdout: /var/folders/5x/409ks2012xxch_pmbs6qpzfh0000gp/T/tmp1h852xwu/h2o_e
mmanuel_charon_started_from_python.out
  JVM stderr: /var/folders/5x/409ks2012xxch_pmbs6qpzfh0000gp/T/tmp1h852xwu/h2o_e
mmanuel_charon_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:         02 secs
H2O_cluster_timezone:       Europe/Paris
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.32.0.4
H2O_cluster_version_age:    1 month and 22 days
H2O_cluster_name:           H2O_from_python_emmanuel_charon_nt2mlp
H2O_cluster_total_nodes:    1
H2O_cluster_free_memory:    4 Gb
H2O_cluster_total_cores:    4
H2O_cluster_allowed_cores:  4
H2O_cluster_status:         accepting new members, healthy
H2O_connection_url:         http://127.0.0.1:54321
H2O_connection_proxy:       {"http": null, "https": null}
H2O_internal_security:      False
H2O_API_Extensions:         Amazon S3, XGBoost, Algos, AutoML, Core V3,␣
↪TargetEncoder, Core V4
Python_version:             3.7.6 final
-------------------------- ␣
↪-----------------------------------------------------------------
```

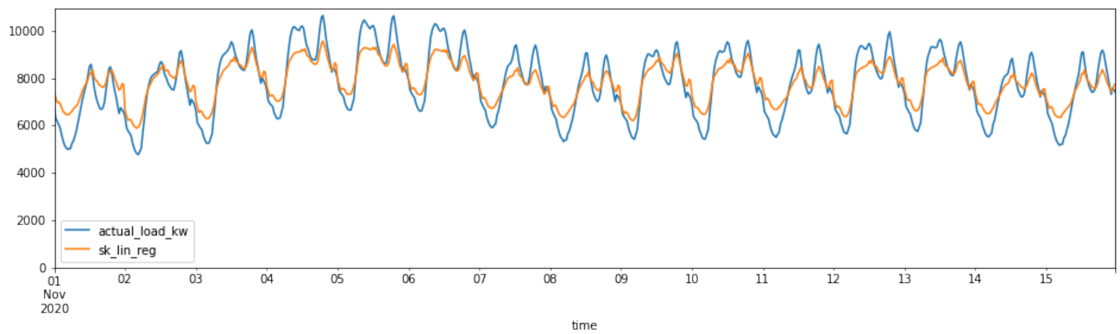
[34]:
```
for model_name, model in all_models.items():
    print("Training {} before predicting with it".format(model_name))
    model.train(benchmark_train, target_col='load_kw')
    model_prediction = model.predict(benchmark_test, target_col='load_kw')
    benchmark[model_name] = model_prediction
```

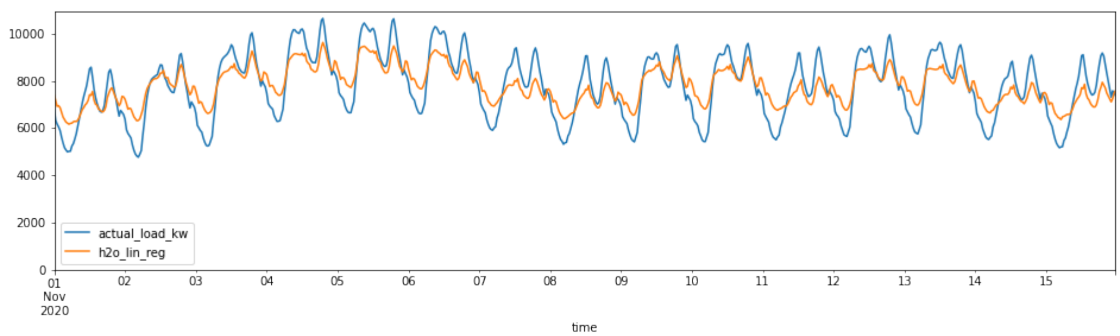
```
Training sk_lin_reg before predicting with it
Training h2o_lin_reg before predicting with it
Training h2o_rf before predicting with it
Training h2o_gbm before predicting with it
Training h2o_xgboost before predicting with it
```

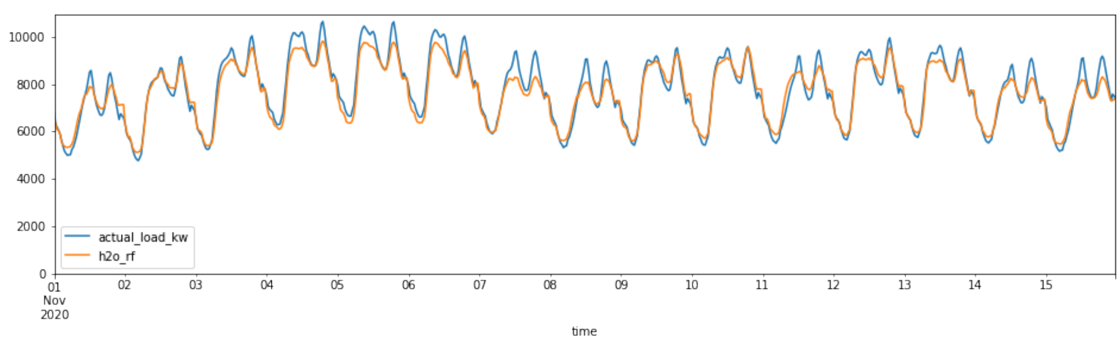
[35]:
```
benchmark
```

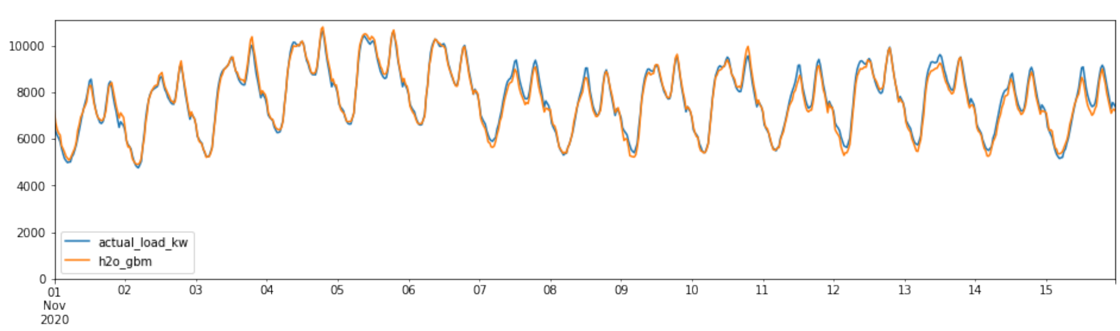
[35]:
```
                           actual_load_kw   sk_lin_reg  h2o_lin_reg  \
time
2020-11-01 00:00:00+01:00     6817.332090  7416.477529  7402.432609
2020-11-01 00:30:00+01:00     6326.667322  7192.916989  7164.175176
2020-11-01 01:00:00+01:00     6172.223671  6974.931684  6925.916729
2020-11-01 01:30:00+01:00     6050.575318  6993.462928  6948.635651
2020-11-01 02:00:00+01:00     5898.881230  6928.525322  6869.233390
…                                     …            …            …
2020-11-15 21:30:00+01:00     7657.293444  7554.874780  7227.449423
2020-11-15 22:00:00+01:00     7317.540759  7422.568321  7114.005755
2020-11-15 22:30:00+01:00     7580.051439  7510.802466  7250.192657
2020-11-15 23:00:00+01:00     7496.273993  7702.238171  7499.847540
2020-11-15 23:30:00+01:00     7376.005701  7759.626894  7590.647251

                                h2o_rf      h2o_gbm  h2o_xgboost
time
2020-11-01 00:00:00+01:00  6546.351367  7212.077053  6940.511719
2020-11-01 00:30:00+01:00  6301.282248  6718.050101  6525.155762
2020-11-01 01:00:00+01:00  6076.009499  6425.362441  6374.268555
2020-11-01 01:30:00+01:00  6040.864937  6255.951603  6200.651855
2020-11-01 02:00:00+01:00  5868.837508  6185.681897  6068.275879
…                                  …            …            …
2020-11-15 21:30:00+01:00  7451.094927  7387.095907  7337.505371
2020-11-15 22:00:00+01:00  7288.085474  7116.853558  7007.531738
2020-11-15 22:30:00+01:00  7316.264338  7257.872271  7203.997070
2020-11-15 23:00:00+01:00  7367.514113  7295.385239  7045.631836
2020-11-15 23:30:00+01:00  7379.755615  7199.281325  7114.983398

[720 rows x 6 columns]
```

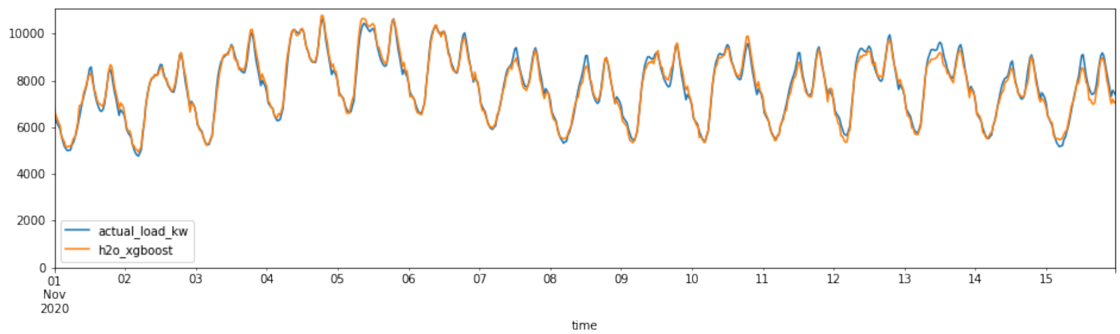
[36]:
```
# visualize predictions
for c in benchmark.columns:
    if c != "actual_load_kw":
        to_plot = benchmark[["actual_load_kw", c]]
        to_plot.plot(ylim=0, figsize=(16, 4))
```


10000
8000
6000
4000
2000
0
actual_load_kw
sk_lin_reg
01
Nov
2020
02
03
04
05
06
07
08
09
10
11
12
13
14
15
time


10000
8000
6000
4000
2000
0
actual_load_kw
h2o_lin_reg
01
Nov
2020
02
03
04
05
06
07
08
09
10
11
12
13
14
15
time


10000
8000
6000
4000
2000
0
actual_load_kw
h2o_rf
01
Nov
2020
02
03
04
05
06
07
08
09
10
11
12
13
14
15
time


10000
8000
6000
4000
2000
0
actual_load_kw
h2o_gbm
01
Nov
2020
02
03
04
05
06
07
08
09
10
11
12
13
14
15
time



[37]:
```
# compute absolute percentage error
benchmark_ape = benchmark.copy(deep=True).drop(columns=["actual_load_kw"])
for c in benchmark_ape.columns:
    benchmark_ape[c] = (benchmark_ape[c] - benchmark["actual_load_kw"]).abs()/
 ↪benchmark["actual_load_kw"]*100
benchmark_ape.mean()
```

[37]:
```
sk_lin_reg      7.055730
h2o_lin_reg     9.057864
h2o_rf          3.358170
h2o_gbm         2.093172
h2o_xgboost     2.006471
dtype: float64
```

[ ]:
```

```

1.6 6. Benchmark with Backtesting

In traditional machine learning, we need more than just 1 evaluation to test an algorithm. We typically use cross-validation to see if the algorithm is not biased and if it can be expected to work well in most cases. For time-series predictions we cannot do a regular cross-validation because it is not realistic : we always want to train using historical data that happened before the prediction.

Here we will do **backtesting** week after week. With the given dataset, this means : - for each week w from early 2019 until the end of the dataset : train using data from the beginning of the dataset (early 2015) until a few days before week w, then eval on w. - the first iteration will train an algorithm using data from 2015 to 2018, then eval on the first week of 2019 - the second iteration will train using data from 2015 to a bit before the first week of 2019, then eval on the second week of 2019 - and so on… - keep the predictions of each time-step using this method, from early 2019 to november 2020.
- 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.

[38]:
```
# backtesting takes time so here we just show an example using models that␣
↪train fast.
all_models = dict()

# keep the basic one for the benchmark
all_models['bt_sk_lin_reg'] = SKLearnLinearRegression()

# H2O's linear regressor
all_models['bt_h2o_lin_reg'] = H2OModel(algo_name="glm", model_id="bt_h2o_glm",␣
↪target="load_kw", algo_param_dict={})
```

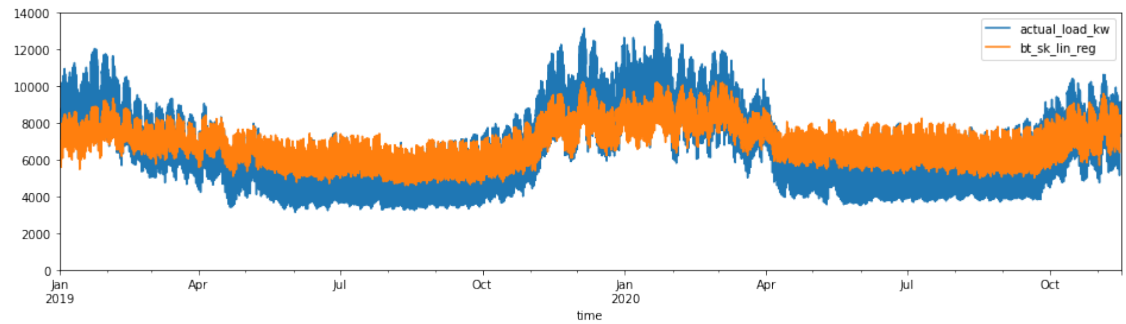
[39]:
```
# some parts give ConvergenceWarnings here and we'll ignore them.
import warnings
warnings.filterwarnings('ignore')
```

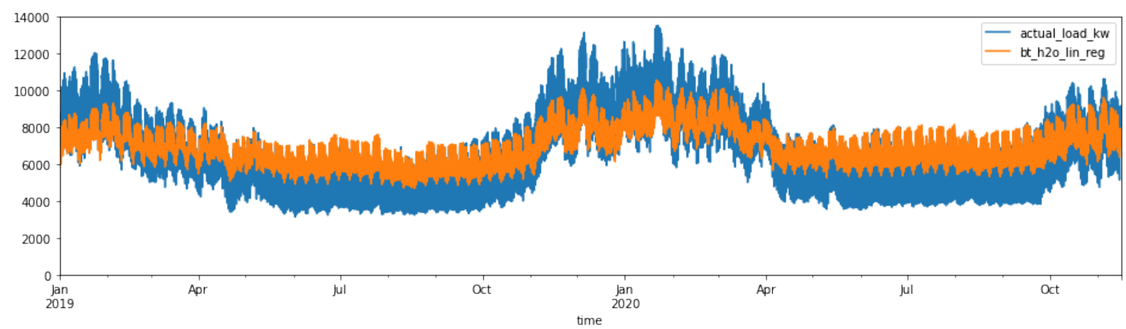
[40]:
```
start_backtesting_dt = pd.to_datetime('2019-01-01 00:00:00+01:00').
↪tz_convert('Europe/Paris')
benchmark = historic[historic.index>=start_backtesting_dt]["load_kw"].
↪to_frame("actual_load_kw")
days_in_each_iteration = 28


for model_name, model in all_models.items():

    count_iterations = 0

    model_predictions = []

    for train_set, test_set in enda.BackTesting.yield_train_test(
        historic,
        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 == 1 or count_iterations % 10 == 0:
            print("Model {}, backtesting iteration {}, train set {}->{}, test␣
↪set {}->{}".format(
                    model_name, count_iterations,
                    train_set.index.min(), train_set.index.max(),
                    test_set.index.min(), test_set.index.max()))

        # featurize
        train_set = featurize(train_set)
        test_set = test_set.drop(columns=["load_kw"])
```

```
        test_set = featurize(test_set)

        # use forecast porfolio in test_set
        forecast_portfolio = enda.Contracts.forecast_using_trend(
            portfolio_df=portfolio[portfolio.index<test_set.index.min()],
            start_forecast_date=test_set.index.min(),
            nb_days=days_in_each_iteration,
            past_days=150)  # recent portfolio trend

        test_set['kva'] = forecast_portfolio['kva']
        test_set['contracts_count'] = forecast_portfolio['contracts_count']

        # train and predict
        model.train(train_set, target_col='load_kw')
        model_predictions.append(model.predict(test_set, target_col='load_kw'))

    benchmark[model_name] = pd.concat(model_predictions)
```

```
Model bt_sk_lin_reg, backtesting iteration 1, train set 2015-01-01
00:00:00+01:00->2018-12-17 23:30:00+01:00, test set 2019-01-01
00:00:00+01:00->2019-01-28 23:30:00+01:00
Model bt_sk_lin_reg, backtesting iteration 10, train set 2015-01-01
00:00:00+01:00->2019-08-26 23:30:00+02:00, test set 2019-09-10
00:00:00+02:00->2019-10-07 23:30:00+02:00
Model bt_sk_lin_reg, backtesting iteration 20, train set 2015-01-01
00:00:00+01:00->2020-06-01 23:30:00+02:00, test set 2020-06-16
00:00:00+02:00->2020-07-13 23:30:00+02:00
Model bt_h2o_lin_reg, backtesting iteration 1, train set 2015-01-01
00:00:00+01:00->2018-12-17 23:30:00+01:00, test set 2019-01-01
00:00:00+01:00->2019-01-28 23:30:00+01:00
Model bt_h2o_lin_reg, backtesting iteration 10, train set 2015-01-01
00:00:00+01:00->2019-08-26 23:30:00+02:00, test set 2019-09-10
00:00:00+02:00->2019-10-07 23:30:00+02:00
Model bt_h2o_lin_reg, backtesting iteration 20, train set 2015-01-01
00:00:00+01:00->2020-06-01 23:30:00+02:00, test set 2020-06-16
00:00:00+02:00->2020-07-13 23:30:00+02:00
```

[41]:
```
# visualize predictions
for c in benchmark.columns:
    if c != "actual_load_kw":
        to_plot = benchmark[["actual_load_kw", c]]
        to_plot.plot(ylim=0, figsize=(16, 4))
```





[42]:
```
# compute absolute percentage error
benchmark_ape = benchmark.copy(deep=True).drop(columns=["actual_load_kw"])
for c in benchmark_ape.columns:
    benchmark_ape[c] = (benchmark_ape[c] - benchmark["actual_load_kw"]).abs()/
 ↪benchmark["actual_load_kw"]*100
benchmark_ape.mean()
```

[42]:
```
bt_sk_lin_reg     13.164438
bt_h2o_lin_reg    14.279924
dtype: float64
```

If you have time/computing power: - try more algorithms in the backtesting benchmark, this is more reliable than a simple benchmark - reduce the "days_in_each_iteration" down to 7 if you think you can have a weekly training in your production environment.

1.7 7. Make the prediction

Seeing the results from just the basic benchmark, we here decide to predict using h2o's xgboost. We now need to train it on the full dataset and make the prediction.

[43]:
```
xgboost = H2OModel(
    algo_name="xgboost",
    model_id="final_h2o_xgboost",
```

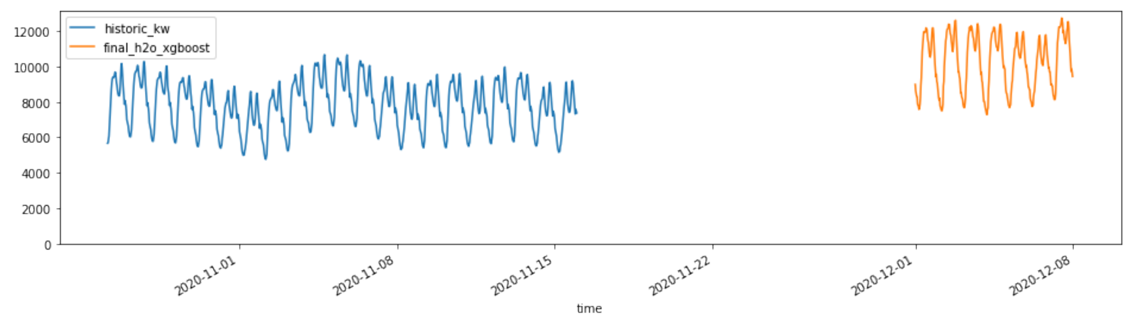
```
    target="load_kw",
    algo_param_dict= {
        "ntrees": [500],
        "max_depth": [5],
        "sample_rate": [0.8],
        "min_rows": [10]
    },
)
```

[44]:
```
xgboost.train(full_train_set, target_col='load_kw')
```

[45]:
```
xgboost_prediction = xgboost.predict(forecast_input_data, target_col="load_kw")
```

[46]:
```
# visualize recent load along with our forecast; remember we don't have recent␣
↪actual load so there is a time-gap.
# (remember that the prediction takes weather forecast and more information␣
↪into account)
to_plot = pd.merge(
    historic["load_kw"][-1000:].to_frame("historic_kw"),
    xgboost_prediction.rename(columns={"load_kw": "forecast_kw"}),
    how='outer', left_index=True, right_index=True
)
to_plot.plot(ylim=0, figsize=(16, 4))
```

[46]: <AxesSubplot:xlabel='time'>



[47]:
```
# don't forget to shutdown your h2o local server
h2o.cluster().shutdown()
time.sleep(5)  # wait for h2o to finish shutting down
```

H2O session _sid_8347 closed.

1.8 Conclusion

Thats all for Example B. Check out Example C next. Thanks for reading and don't hesitate to send feeback at: emmanuel.charon@enercoop.org !