temus

May 23, 2022

[918]:
```
import os
import sys
import datetime
```

[919]:
```
import warnings
from statsmodels.tools.sm_exceptions import ValueWarning, ConvergenceWarning
from tqdm.std import TqdmWarning

warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=ValueWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
warnings.simplefilter(action='ignore', category=ConvergenceWarning)
warnings.simplefilter(action='ignore', category=TqdmWarning)
```

[920]:
```
import IPython
import IPython.display

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker

import seaborn as sns

plt.rcParams['figure.figsize'] = (25, 5)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

[921]:
```
import numpy as np
import pandas as pd
from scipy import fft

import statsmodels.api as sm
import tensorflow as tf
```

[922]:
```
# from https://stackoverflow.com/questions/11130156/
↪suppress-stdout-stderr-print-from-python-functions
class suppress_stdout_stderr(object):
    '''
```

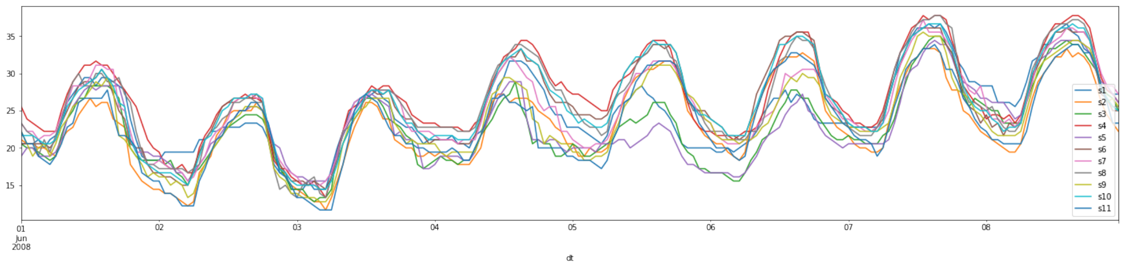
```
    A context manager for doing a "deep suppression" of stdout and stderr in
    Python, i.e. will suppress all print, even if the print originates in a
    compiled C/Fortran sub-function.
       This will not suppress raised exceptions, since exceptions are printed
    to stderr just before a script exits, and after the context manager has
    exited (at least, I think that is why it lets exceptions through).

    '''
    def __init__(self):
        # Open a pair of null files
        self.null_fds = [os.open(os.devnull, os.O_RDWR) for x in range(2)]
        # Save the actual stdout (1) and stderr (2) file descriptors.
        self.save_fds = (os.dup(1), os.dup(2))

    def __enter__(self):
        # Assign the null pointers to stdout and stderr.
        os.dup2(self.null_fds[0], 1)
        os.dup2(self.null_fds[1], 2)

    def __exit__(self, *_):
        # Re-assign the real stdout/stderr back to (1) and (2)
        os.dup2(self.save_fds[0], 1)
        os.dup2(self.save_fds[1], 2)
        # Close the null files
        os.close(self.null_fds[0])
        os.close(self.null_fds[1])
```

[923]:

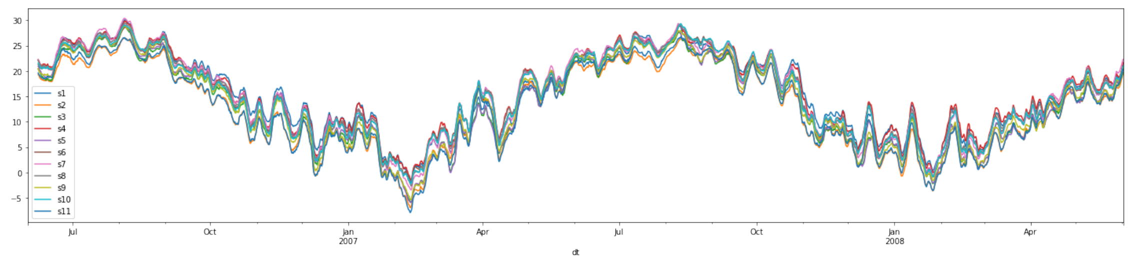
```
# load stations temps
df =  pd.read_csv('../data/raw/gef2012-load/temperature_history.csv')
df = df.set_index(['station_id','year','month','day'])
# convert h1..h24 to 'hours' columns 0..23
df.columns = [int(x[1:])-1 for x in df.columns]
# wide to long using melt
df = pd.melt(df.reset_index(), id_vars=['station_id','year','month','day'],␣
 ↪value_vars=range(24), var_name='hour', value_name='temp')
df['temp'] = (df['temp'].astype(float)-32)*5/9
df['dt'] = pd.to_datetime(df[['year','month', 'day', 'hour']])
df = df.sort_values(['station_id', 'dt']).set_index(['station_id', 'dt'])
df = df.drop(columns=['year','month','day', 'hour'])
df = df.unstack('station_id')
df.columns = [f's{x[1]}' for x in  df.columns]
df_stations = df.copy()
```

[931]:

```
# plot a week of hourly data
df_stations["2008-06-01":"2008-06-08"].plot();
```

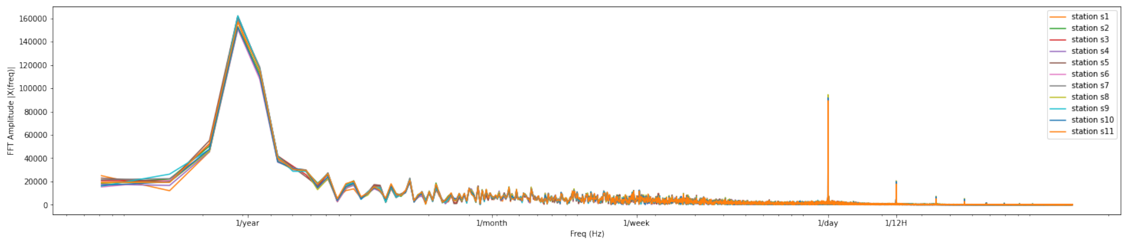


[932]:
```
# two years of data resampled weekly
df_stations["2006-06-01":"2008-06-01"].rolling(7*24).mean().plot();
```

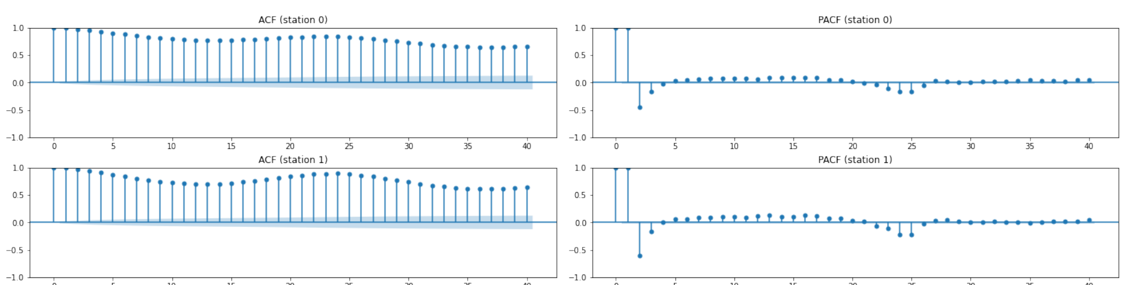


[933]:
```
# fft analysis
from scipy.fftpack import fft, ifft

fig1, ax1 = plt.subplots()

for i in range(1,12):
    x = df_stations[:"2008-06-01"][f's{i}'].values

    X = fft(x)
    N = len(X)
    n = np.arange(N)
    # get the sampling rate
    sr = 1 / (60*60)
    T = N/sr
    freq = n/T

    # Get the one-sided specturm
    n_oneside = N//2
    # get the one side frequency
    f_oneside = freq[1:n_oneside]

    ax1.plot(f_oneside, np.abs(X[1:n_oneside]),color=colors[i%10],
 ↪label=f'station s{i}')
```

```
    ax1.set_xlabel('Freq (Hz)')
    ax1.set_ylabel('FFT Amplitude |X(freq)|')
    ax1.set_xscale('log')
    ax1.get_xaxis().set_major_formatter(matplotlib.ticker.ScalarFormatter())
    ax1.set_xticks(ticks=[sr/(365*24), sr/(30.5*24),sr/(7*24), sr/24, sr/12], labels=['1/year', '1/month', '1/week', '1/day', '1/12H'])

ax1.legend()
plt.show()
```



[934]:
```
import statsmodels.api as sm
n = 2 # change to 11 to show all stations
fig, ax = plt.subplots(n, 2, figsize=(20, 5), constrained_layout=True)
for i in range(n):
    x = df_stations["2007-01-01":"2008-06-01"][f's{i+1}']
    sm.graphics.tsa.plot_acf(x.values.squeeze(), lags=40, ax=ax[i, 0], title=f'ACF (station {i})')
    sm.graphics.tsa.plot_pacf(x.values.squeeze(), lags=40, ax=ax[i, 1], title=f'PACF (station {i})')
plt.show()
```



[935]:
```
# # load zones load
df=  pd.read_csv('../data/raw/gef2012-load/Load_history.csv',thousands=',')

df = df.set_index(['zone_id','year','month','day'])
```

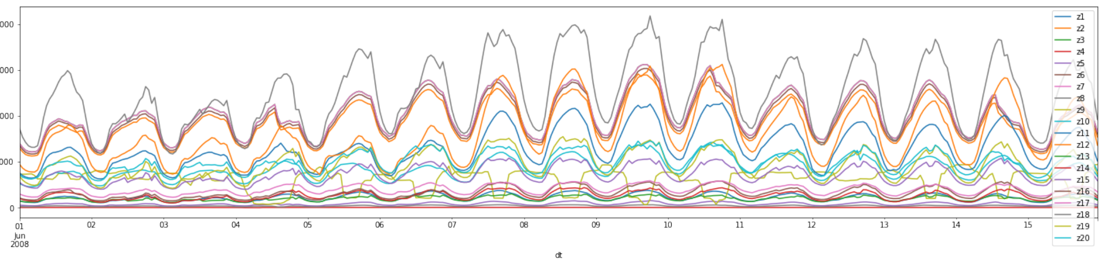
```python
# convert h1..h24 to 'hours' columns 0..23
df.columns = [int(x[1:])-1 for x in df.columns]

# wide to long using melt
df = pd.melt(df.reset_index(), id_vars=['zone_id','year','month','day'],␣
↪value_vars=range(24), var_name='hour', value_name='load')

df['load'] = df['load'].astype(float)

df['dt'] = pd.to_datetime(df[['year','month', 'day', 'hour']])
df = df.sort_values(['zone_id', 'dt']).set_index(['zone_id', 'dt'])
df = df.drop(columns=['year','month','day', 'hour'])
df =df.unstack('zone_id').copy()
df.columns = [f'z{x[1]}' for x in  df.columns]

df_load=df.copy()
df_load.head()
```

[935]:

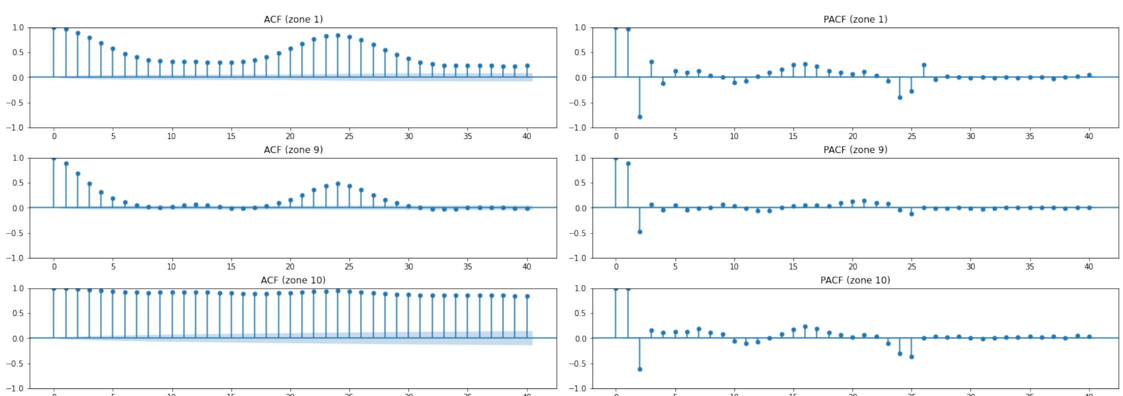
```
                          z1        z2        z3     z4      z5        z6  \
dt
2004-01-01 00:00:00  16853.0  126259.0  136233.0  484.0  6829.0  133088.0
2004-01-01 01:00:00  16450.0  123313.0  133055.0  457.0  6596.0  129909.0
2004-01-01 02:00:00  16517.0  119192.0  128608.0  450.0  6525.0  125717.0
2004-01-01 03:00:00  16873.0  117507.0  126791.0  448.0  6654.0  124162.0
2004-01-01 04:00:00  17064.0  118343.0  127692.0  444.0  6977.0  125320.0

                           z7      z8       z9      z10      z11       z12  \
dt
2004-01-01 00:00:00  136233.0  3124.0  75243.0  23339.0  90700.0  118378.0
2004-01-01 01:00:00  133055.0  2956.0  67368.0  22100.0  86699.0  112480.0
2004-01-01 02:00:00  128608.0  2953.0  64050.0  21376.0  84243.0  108435.0
2004-01-01 03:00:00  126791.0  2914.0  63861.0  21335.0  84285.0  107224.0
2004-01-01 04:00:00  127692.0  3221.0  75852.0  21564.0  86087.0  108870.0

                         z13      z14      z15      z16      z17       z18  \
dt
2004-01-01 00:00:00  20673.0  21791.0  65970.0  28752.0  30645.0  200946.0
2004-01-01 01:00:00  19666.0  21400.0  64600.0  27851.0  30461.0  195835.0
2004-01-01 02:00:00  19020.0  20998.0  63843.0  27631.0  30197.0  194093.0
2004-01-01 03:00:00  18841.0  21214.0  64023.0  27986.0  30264.0  194708.0
2004-01-01 04:00:00  19310.0  21830.0  65679.0  29160.0  30907.0  202458.0

                         z19      z20
dt
2004-01-01 00:00:00  82298.0  79830.0
2004-01-01 01:00:00  79827.0  77429.0
2004-01-01 02:00:00  77728.0  75558.0
```

```
2004-01-01 03:00:00  76433.0  75709.0
2004-01-01 04:00:00  78172.0  77475.0
```

[936]:
```
df_load["2008-06-01":"2008-06-15"].plot();
```



[937]:
```
import statsmodels.api as sm
sel = [1,9,10] # change to range(20) to show all stations
fig, ax = plt.subplots(len(sel), 2, figsize=(20, 7), constrained_layout=True)
for i, v in enumerate(sel):
    x = df_load["2007-01-01":"2008-06-01"][f'z{v}']
    sm.graphics.tsa.plot_acf(x.values.squeeze(), lags=40, ax=ax[i, 0],␣
 ↪title=f'ACF (zone {v})')
    sm.graphics.tsa.plot_pacf(x.values.squeeze(), lags=40, ax=ax[i, 1],␣
 ↪title=f'PACF (zone {v})')
plt.show()
```



[938]:
```
df = df_load.join(df_stations)
df = df[df.index < '2008-06-30 00:00:00']
desc = df.describe().T
desc['na'] = df.isna().sum()
desc
```

[938]:

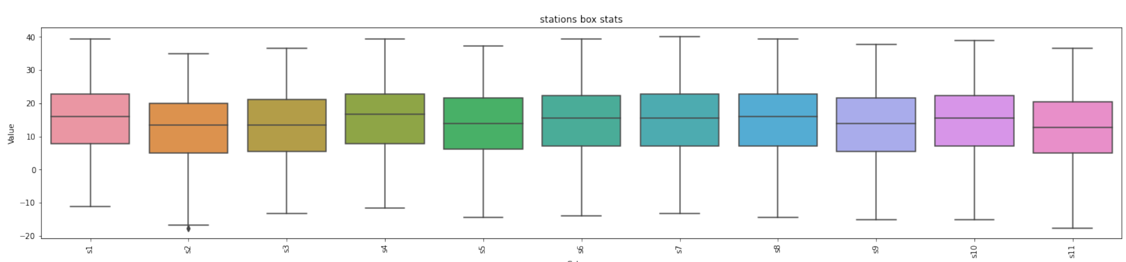
	count	mean	std	min	25%	\
z1	38064.0	18641.241488	5796.667521	7319.000000	14415.000000	
z2	38064.0	173748.222730	34996.374374	82672.000000	149160.500000	
z3	38064.0	187474.692781	37761.160010	89204.000000	160943.500000	
z4	38064.0	498.803804	121.665345	0.000000	414.000000	
z5	38064.0	7766.504387	2606.447348	1525.000000	5798.000000	
z6	38064.0	181513.962563	37294.490362	86652.000000	155603.750000	
z7	38064.0	187474.692781	37761.160010	89204.000000	160943.500000	
z8	38064.0	3773.782498	1010.179244	1720.000000	3026.000000	
z9	38064.0	67622.125473	18769.439340	0.000000	63777.000000	
z10	38064.0	32491.972467	17735.246736	7193.000000	22797.750000	
z11	38064.0	107829.890395	31016.602734	49085.000000	86069.750000	
z12	38064.0	132945.256647	43579.381147	56255.000000	101594.500000	
z13	38064.0	19665.332887	4983.234490	8516.000000	16080.000000	
z14	38064.0	20859.864807	7318.592495	6404.000000	15225.000000	
z15	38064.0	62258.531999	15550.458456	27819.000000	50738.750000	
z16	38064.0	29226.795660	10300.967857	7184.000000	21379.750000	
z17	38064.0	32871.816861	7941.881119	16177.000000	26985.000000	
z18	38064.0	213578.805669	65248.227626	90621.000000	165303.250000	
z19	38064.0	78275.519231	24922.155735	25536.000000	59375.750000	
z20	38064.0	88649.680223	18654.775828	46291.000000	75384.000000	
s1	39408.0	15.240518	9.291440	-11.111111	7.777778	
s2	39408.0	12.490611	9.671418	-17.777778	5.000000	
s3	39408.0	13.209613	9.768619	-13.333333	5.555556	
s4	39408.0	15.526106	9.658859	-11.666667	7.777778	
s5	39408.0	13.563236	9.774812	-14.444444	6.111111	
s6	39408.0	14.712650	9.483504	-13.888889	7.222222	
s7	39408.0	14.791244	9.972991	-13.333333	7.222222	
s8	39408.0	14.982871	9.864238	-14.444444	7.222222	
s9	39408.0	13.546220	9.953081	-15.000000	5.555556	
s10	39408.0	14.755506	9.668470	-15.000000	7.222222	
s11	39408.0	12.589237	9.949387	-17.777778	5.000000	

	50%	75%	max	na
z1	17349.500000	22024.250000	45547.000000	1344
z2	170451.500000	196107.750000	321509.000000	1344
z3	183917.500000	211600.750000	346909.000000	1344
z4	490.000000	575.000000	1104.000000	1344
z5	7343.000000	9448.000000	18706.000000	1344
z6	177697.500000	205354.750000	339927.000000	1344
z7	183917.500000	211600.750000	346909.000000	1344
z8	3649.000000	4463.000000	7817.000000	1344
z9	72135.000000	79212.000000	100128.000000	1344
z10	26884.000000	34157.500000	145023.000000	1344
z11	102027.500000	125451.250000	250134.000000	1344
z12	124182.500000	157368.250000	343526.000000	1344
z13	19054.500000	22879.250000	45880.000000	1344

```
z14    19375.500000   25671.250000   51385.000000  1344
z15    59143.000000   71839.250000  131843.000000  1344
z16    26892.000000   35702.000000   82114.000000  1344
z17    31330.000000   37833.500000   70247.000000  1344
z18   200690.500000  255882.750000  540393.000000  1344
z19    73423.500000   94506.000000  200744.000000  1344
z20    86771.000000  100715.000000  176705.000000  1344
s1        16.111111      22.777778      39.444444     0
s2        13.333333      20.000000      35.000000     0
s3        13.333333      21.111111      36.666667     0
s4        16.666667      22.777778      39.444444     0
s5        13.888889      21.666667      37.222222     0
s6        15.555556      22.222222      39.444444     0
s7        15.555556      22.777778      40.000000     0
s8        16.111111      22.777778      39.444444     0
s9        13.888889      21.666667      37.777778     0
s10       15.555556      22.222222      38.888889     0
s11       12.777778      20.555556      36.666667     0
```

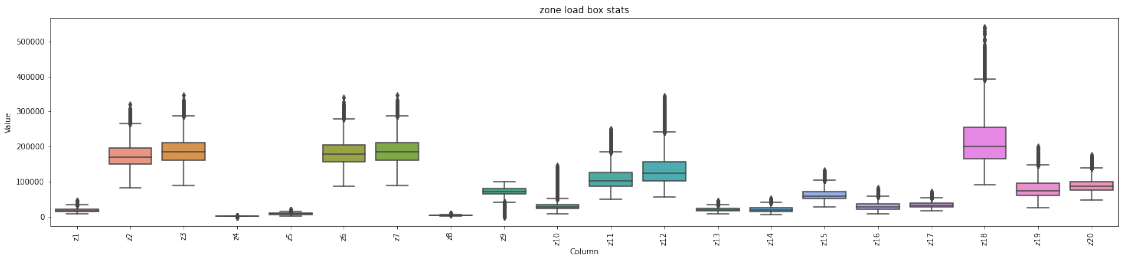
[939]:

```
sel = [x for x in df.columns if x[0]=='s' and x[1] in '0123456789']
df_box = df[sel]#(df - train_mean) / train_std
df_box = df_box.melt(var_name='Column', value_name='Value')
plt.figure()
ax = sns.boxplot(x='Column', y='Value', data=df_box)
ax.set_title('stations box stats')
_ = ax.set_xticklabels(sel, rotation=90)
```



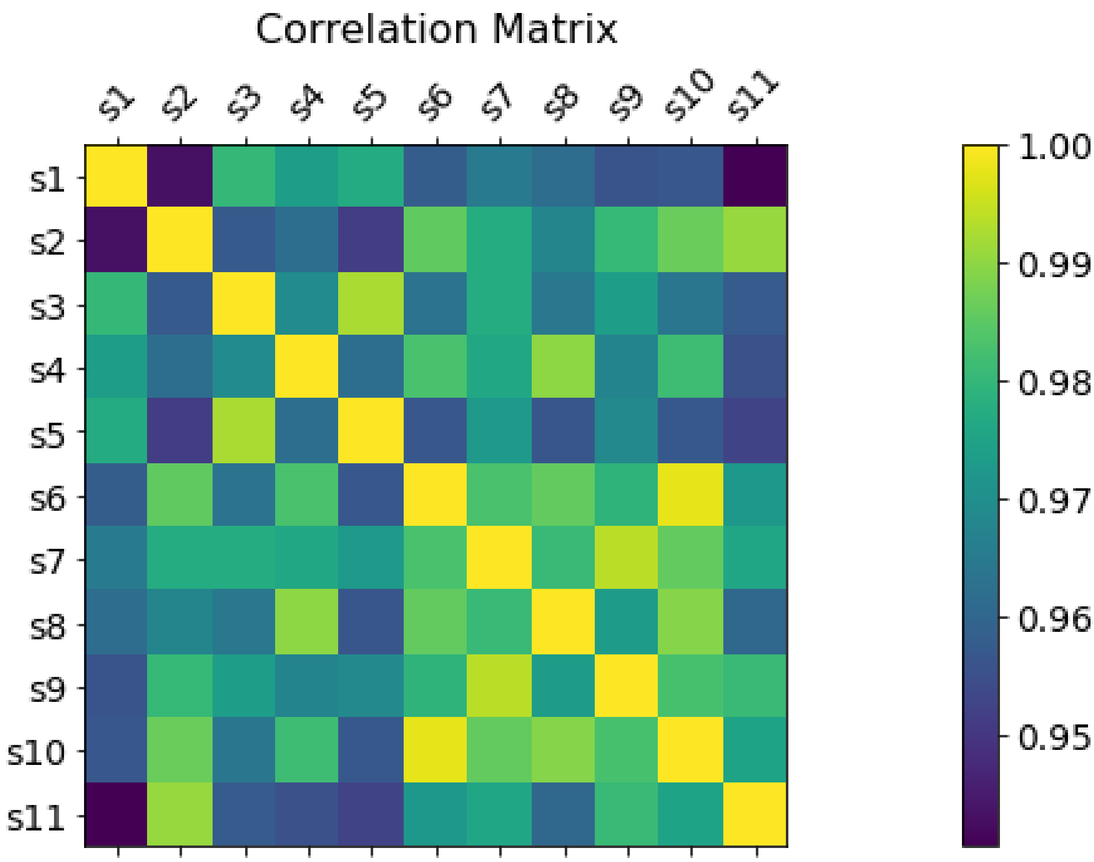
[940]:

```
sel = [x for x in df.columns if x[0]=='z' and x[1] in '0123456789']
df_box = df[sel]#(df - train_mean) / train_std
df_box = df_box.melt(var_name='Column', value_name='Value')
plt.figure()
ax = sns.boxplot(x='Column', y='Value', data=df_box)
ax.set_title('zone load box stats')
_ = ax.set_xticklabels(sel, rotation=90)
```



[941]:
```
# station cross-correlation
f = plt.figure()
d = df_stations
plt.matshow(d.corr(), fignum=f.number)
plt.xticks(range(d.select_dtypes(['number']).shape[1]), d.
 ↪select_dtypes(['number']).columns, fontsize=14, rotation=45)
plt.yticks(range(d.select_dtypes(['number']).shape[1]), d.
 ↪select_dtypes(['number']).columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Correlation Matrix', fontsize=16);
d.corr().min()
```

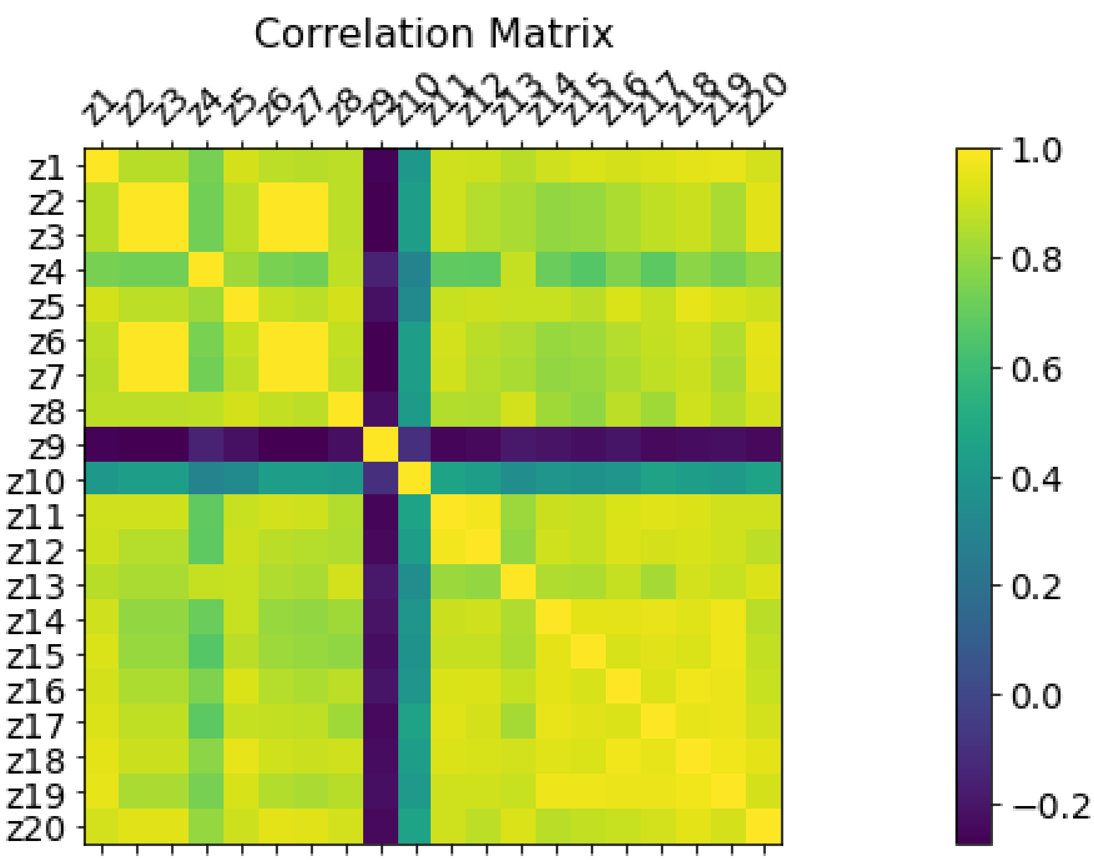
[941]:
```
s1     0.940581
s2     0.943141
s3     0.957310
s4     0.955277
s5     0.951411
s6     0.956629
s7     0.965370
s8     0.956580
s9     0.956002
s10    0.956611
s11    0.940581
dtype: float64
```



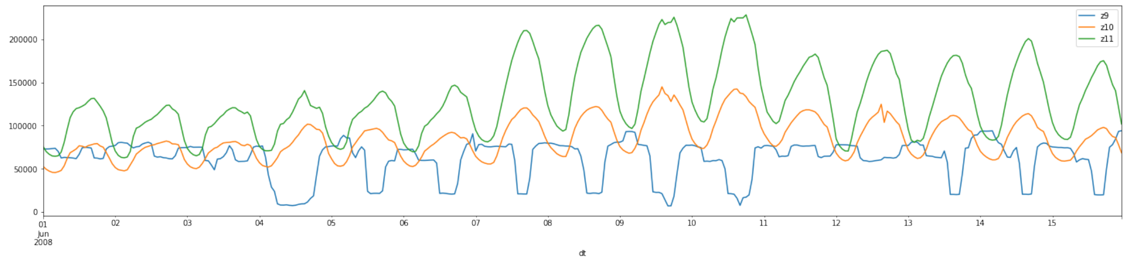
[942]:
```
#load cross-corrrelation
f = plt.figure()
d = df_load.copy()
plt.matshow(d.corr(), fignum=f.number)
plt.xticks(range(d.select_dtypes(['number']).shape[1]), d.
    ↪select_dtypes(['number']).columns, fontsize=14, rotation=45)
plt.yticks(range(d.select_dtypes(['number']).shape[1]), d.
    ↪select_dtypes(['number']).columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Correlation Matrix', fontsize=16);
d.corr().median()
```

[942]:
```
z1      0.904877
z2      0.868045
z3      0.868045
z4      0.733925
z5      0.891531
z6      0.878890
```

```
z7      0.868045
z8      0.874338
z9     -0.229298
z10     0.426437
z11     0.905915
z12     0.880397
z13     0.849485
z14     0.879523
z15     0.873775
z16     0.907042
z17     0.898647
z18     0.925051
z19     0.909205
z20     0.910531
dtype: float64
```

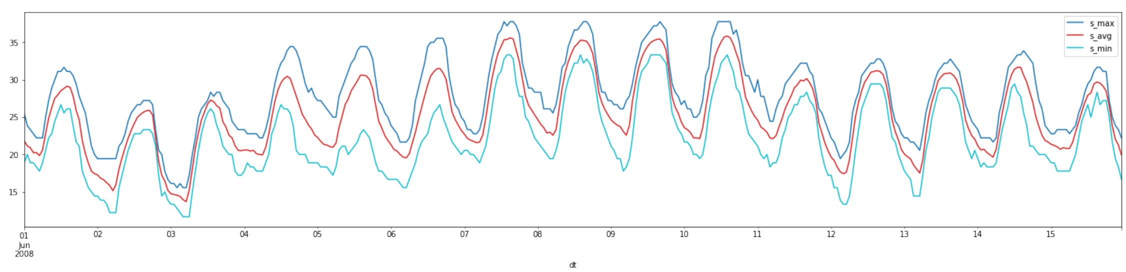


[943]:
```
# z9 and z10 are different
df_load["2008-06-01":"2008-06-15"][['z9','z10','z11']].plot();
```



[944]:
```
# stations are highly correlated: calculate station mean, min, max, std
df['s_avg'] = df[[f's{i}'for i in range(1,12)]].mean(axis=1)
df['s_min'] = df[[f's{i}'for i in range(1,12)]].min(axis=1)
df['s_max'] = df[[f's{i}'for i in range(1,12)]].max(axis=1)
df['s_std'] = df[[f's{i}'for i in range(1,12)]].std(axis=1)

# featurize time
df['year'] = df.index.year
df['month'] = df.index.month
df['day'] = df.index.day
df['hour'] = df.index.hour
df['dow'] = df.index.dayofweek
```

[945]:
```
df["2008-06-01":"2008-06-15"][['s_max','s_avg','s_min']].
↪plot(color=['#1f77b4','#d62728','#17becf']);
```

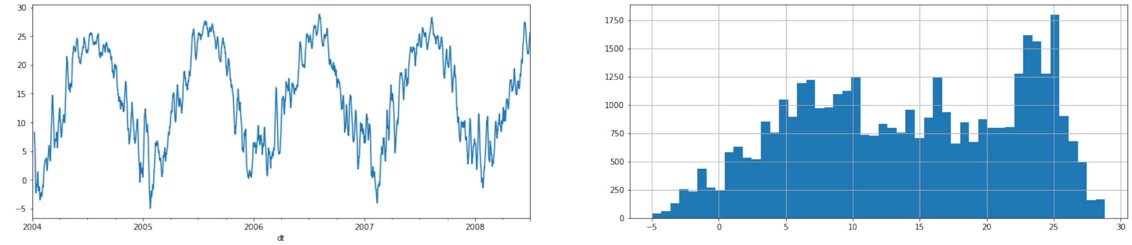


[946]:
```
# minimum load
minload_temp = df['s_avg'].median()
minload_temp
```

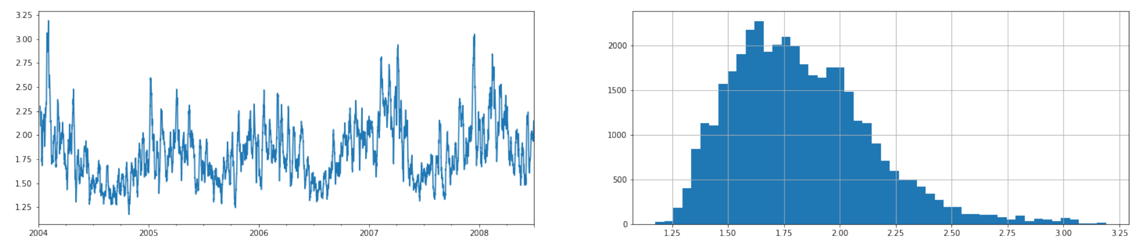
[946]: 14.696969696969699

[947]:
```
fig, ax = plt.subplots(1,2)
# stations rolling weekly std (averaged) over the years
df['s_avg'].rolling(24*7).mean().plot(ax = ax[0]);
# ... and its histogram
```

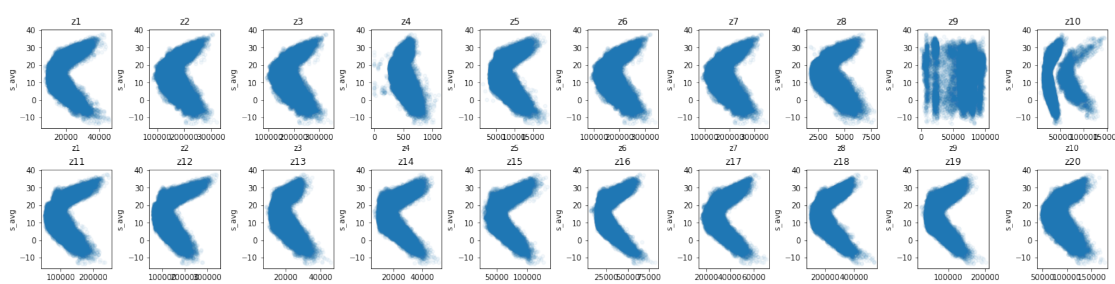
```
df['s_avg'].rolling(24*7).mean().hist(bins=50, ax = ax[1]);
```



[948]:
```
fig, ax = plt.subplots(1,2)
# stations rolling weekly std (averaged) over the years
df['s_std'].rolling(24*7).mean().plot(ax = ax[0]);
# ... and its histogram
df['s_std'].rolling(24*7).mean().hist(bins=50, ax = ax[1]);
```



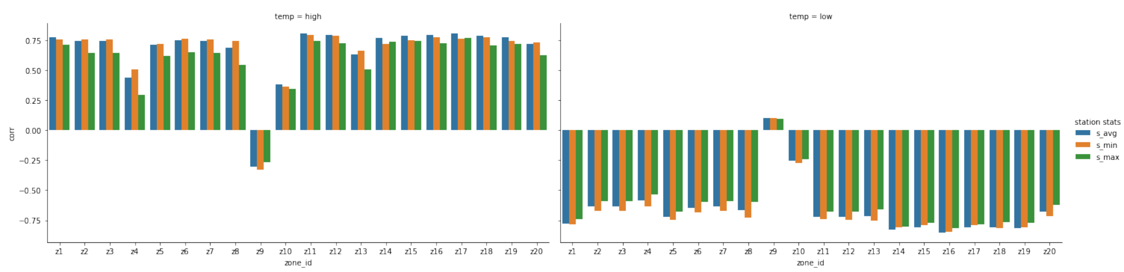
[949]:
```
fig, ax = plt.subplots(2,10, constrained_layout=True)
for i in range(2):
    for j in range(10):
        df.plot.scatter(f'z{i*10+j+1}', 's_avg', figsize=(20, 5), ax=ax[i,j],␣
 ↪alpha=0.05, title=f'z{i*10+j+1}')
```



[950]:
```
# correlate load with temp (averaged over  all stations)
d = dict()
```

```
for t in ["high", "low"]:
    d[t] = dict()
    for s in ['s_avg', 's_min', 's_max']:
        d[t][s] = dict()
        for i in range(1,21):
            f = df[s]<minload_temp if t=='low' else df[s]>=minload_temp
            d[t][s][f'z{i}'] = df[f][[f'z{i}', s]].corr().iloc[0,1]
```

[951]:
```
hitemp_corr = pd.DataFrame(d['high'])
hitemp_corr['temp'] = 'high'
lotemp_corr = pd.DataFrame(d['low'])
lotemp_corr['temp'] = 'low'

d = pd.concat([hitemp_corr,lotemp_corr], axis=0)
d.index.name = 'zone_id'
d = d.reset_index()
d = d.set_index(['zone_id', 'temp'])
d = d.stack()
d = d.reset_index()
d.columns = ['zone_id', 'temp', 'station stats', 'corr']

_ = sns.catplot(data=d, kind="bar", x='zone_id', y='corr', hue="station stats",␣
↪col="temp", height=5, aspect=2)
```



[952]:
```
#intuition: z4: winter usage (maybe montain area), z9: industrial, z10: mixed␣
↪use: residential/manufaturing
```

[953]:
```
df.head(3)
```

[953]:
```
                         z1        z2        z3     z4      z5        z6  \
dt
2004-01-01 00:00:00  16853.0  126259.0  136233.0  484.0  6829.0  133088.0
2004-01-01 01:00:00  16450.0  123313.0  133055.0  457.0  6596.0  129909.0
2004-01-01 02:00:00  16517.0  119192.0  128608.0  450.0  6525.0  125717.0

                         z7       z8        z9       z10  …        s11  \
dt                                                        …
```

```
2004-01-01 00:00:00  136233.0  3124.0  75243.0  23339.0  …  2.222222
2004-01-01 01:00:00  133055.0  2956.0  67368.0  22100.0  …  0.000000
2004-01-01 02:00:00  128608.0  2953.0  64050.0  21376.0  … -0.555556

                        s_avg     s_min     s_max     s_std  year  month  day  \
dt
2004-01-01 00:00:00  5.757576  2.222222  7.777778  1.708236  2004      1    1
2004-01-01 01:00:00  5.151515  0.000000  7.777778  2.264745  2004      1    1
2004-01-01 02:00:00  4.242424 -0.555556  7.222222  2.253567  2004      1    1

                     hour  dow
dt
2004-01-01 00:00:00     0    3
2004-01-01 01:00:00     1    3
2004-01-01 02:00:00     2    3

[3 rows x 40 columns]
```

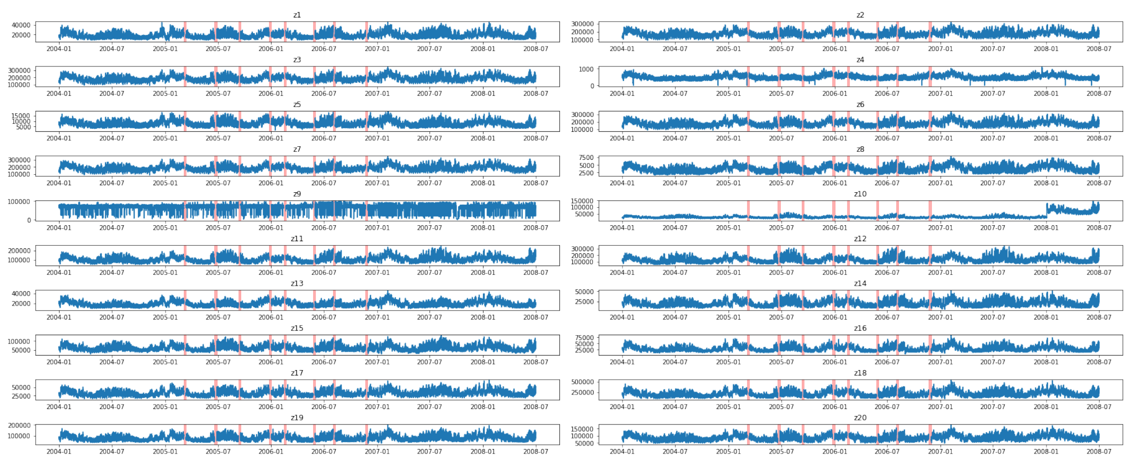
[954]:

```
def detect_gaps(df):
    cnt_gap = 0
    max_gap = pd.Timedelta(0)
    min_gap2gap = [pd.Timedelta('365 days') for x in range(1,21)]

    gaps = dict()
    for z in range(1,21):
        prev_gap_end = None
        gaps[f'z{z}'] = dict()

        for k, g in df[['year', 'month', f'z{z}']].groupby(['year', 'month']):
            s = g[f'z{z}']
            if not isinstance(s[s.isna()].index.min(), type(pd.NaT)):
                cnt_gap += 1
                ts_min = s[s.isna()].index.min()
                ts_max = s[s.isna()].index.max()
                ts_gap = ts_max - ts_min

                if prev_gap_end is not None:
                    gap2gap = ts_min - prev_gap_end
                    min_gap2gap[z-1] = gap2gap if gap2gap<min_gap2gap[z-1] else␣
↪min_gap2gap[z-1]

                prev_gap_end = ts_max
                max_gap = ts_gap if ts_gap>max_gap else max_gap
                gaps[f'z{z}'][k] = [ ts_min, ts_max]

    print(f'Total gaps over 20 zones: {cnt_gap}')
    print(f'Max value gap over 20 zones: {max_gap}')
```

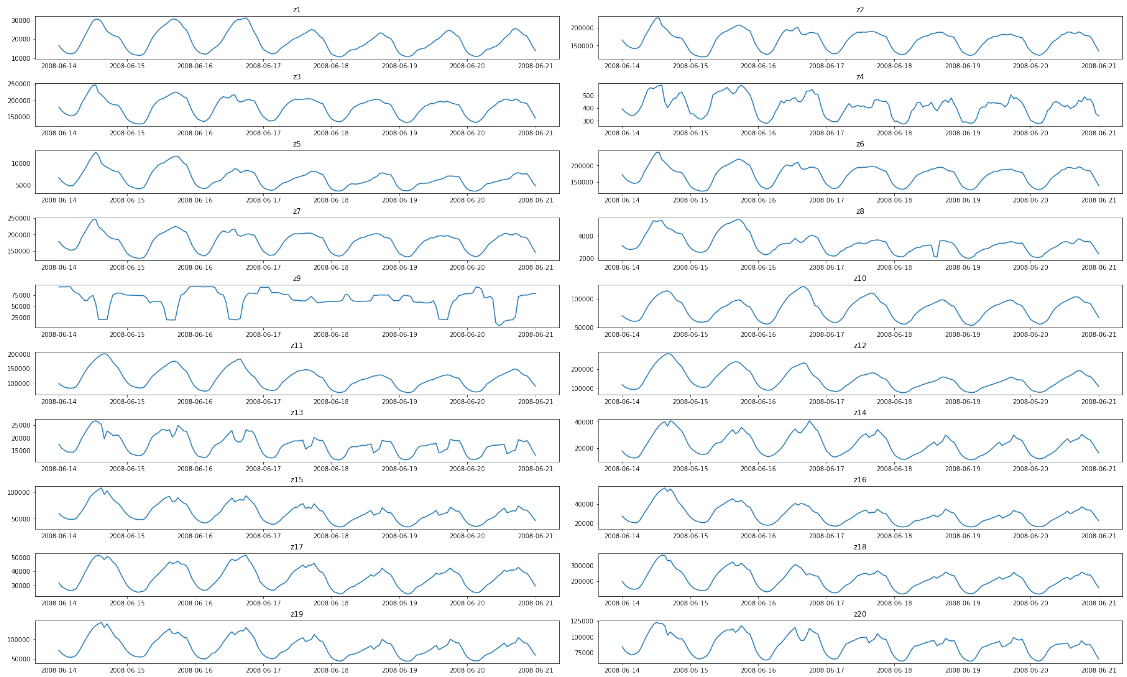
```
    print(f'Min inter-gap distance: {min(min_gap2gap)}')
    return gaps

gaps = detect_gaps(df)
```

```
Total gaps over 20 zones: 160
Max value gap over 20 zones: 6 days 23:00:00
Min inter-gap distance: 43 days 01:00:00
```

[955]:
```
from matplotlib.dates import date2num

fig, ax = plt.subplots(10, 2, figsize=(25,10),constrained_layout=True)
for i in range(20):
    axis =ax[i//2,i%2]
    axis.plot(df.index,df[f'z{i+1}'])
    axis.set_title(f'z{i+1}')
    for k, v in gaps[f'z{i+1}'].items():
        axis.axvspan(date2num(v[0]), date2num(v[1]), color="red", alpha=0.3)
```



[956]:
```
fig, ax = plt.subplots(10, 2, figsize=(25,15),constrained_layout=True)
for i in range(20):
    axis =ax[i//2,i%2]
    axis.plot(df["2008-06-14 00:00:00":"2008-06-21 00:00:00"].
 ↪index,df["2008-06-14 00:00:00":"2008-06-21 00:00:00"][f'z{i+1}'])
    axis.set_title(f'z{i+1}')
```



[957]:
```
# can safely fill with snaive(24*7) 1 week ago, hourly

# note that this step is only required for statistical forecasting methods
# which rely directly on auto-regression values not being NaN (such as arima,␣
↪hw, ses, etc)
# methods such as regression trees, prophet, boosted ensembles
# can deal with NaN/NA skipping those values will learn just fine
# neural networks rely on (automatic) differentiation and cannot deal with NaN␣
↪neither
```

[958]:
```
d7 = pd.Timedelta('7 days')
for z, g in gaps.items():
    for k, v in g.items():
        # slice to slice copy
        # https://stackoverflow.com/questions/64022977/
↪how-do-i-replace-a-slice-of-a-dataframe-column-with-values-from-another-datafram
        df.iloc[
            df.index.get_loc(v[0]):df.index.get_loc(v[1]+pd.Timedelta('1H')),
            df.columns.get_loc(z)
        ] = df.loc[v[0]-d7:v[1]-d7, z]
```

[959]:
```
# recheck gaps
gaps = detect_gaps(df)
```

```
Total gaps over 20 zones: 0
```

```
Max value gap over 20 zones: 0 days 00:00:00
Min inter-gap distance: 365 days 00:00:00
```

[960]:
```
from IPython.display import display, display_markdown
from prophet import Prophet
```

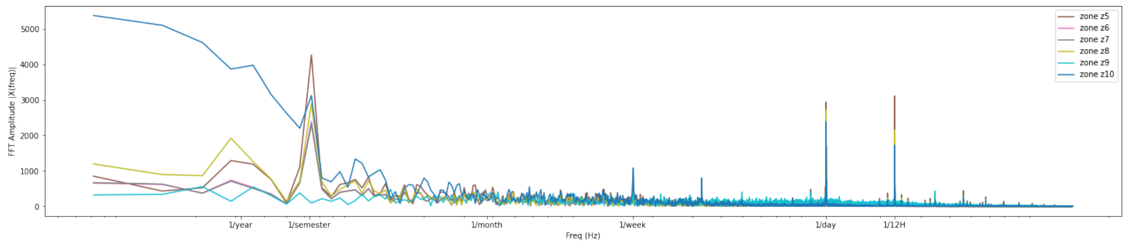
[961]:
```
def run_prophet(df, target, regressors=None, check_anomalies=True,␣
↪fill_missing_inplace=True):
    regressors = [] if regressors is None else regressors
    regressors = regressors if type(regressors)==list else [regressors]

    # init prophet
    m = Prophet()
    m.add_country_holidays(country_name='US')
    for reg in regressors:
        m.add_regressor(reg)

    # prep dataframe
    ts = df[[target]+regressors].copy()
    ts = ts.reset_index()
    ts.columns = ['ds', 'y']+ regressors

    with suppress_stdout_stderr():
            m.fit(ts)

            if check_anomalies:
                # quick check to verify that the actual is within the upper/
↪lower confidence interval
                pred = m.predict(ts)
                ad = ts.join(pred[['yhat', 'yhat_lower','yhat_upper']])
                anomalies = ad[(ad.y>ad.yhat_upper) & (ad.y<ad.yhat_lower)]
                if len(anomalies)>0:
                    display_markdown(f'Anomalies on ** target {target} **')
                    display(ad[(ad.y>ad.yhat_upper) & (ad.y<ad.yhat_lower)])

            if fill_missing_inplace:
                # backfill gaps
                # note that this step is only required for statistical␣
↪forecasting methods
                # which rely directly on auto-regression values not being NaN␣
↪(such as arima, hw, ses, etc)
                # methods such as regression trees, prophet, boosted ensembles
                # and all SGD solution trained skipping those values will learn␣
↪just fine
                f = df[target].isna()
                if f.sum()>0:
                    ts = df.loc[f, regressors].reset_index()
```

```
                    ts.columns = ['ds']+ regressors
                    pred = m.predict(ts)
                    index = df.reset_index().index
                    df.iloc[index[f], df.columns.get_loc(target)] = pred.yhat
    return m
```

[962]:
```
# alternative way of filling (first attempt, replaced with snaive(24*7) here␣
 ↪above
# dm = dict()
# for i in range(1,21):
#     m[i] = run_prophet(df, f'z{i}', 's_min')

# debug prophet plot components for z1
# i=1
# ts = df[[f'z{i}', 's_min']].copy()
# ts = ts.reset_index()
# ts.columns = ['ds', 'y', 's_min']
# pred = m[i].predict(ts)
# m[i].plot_components(pred);
```

[963]:
```
# fft analysis
from scipy.fftpack import fft, ifft

fig1, ax1 = plt.subplots()

# plot zones 5 to 11: those are the most interesting ...
for i in range(5,11):
    x = df[:"2008-06-01"][f'z{i}'].values
    x = x/x.mean()

    X = fft(x)
    N = len(X)
    n = np.arange(N)
    # get the sampling rate
    sr = 1 / (60*60)
    T = N/sr
    freq = n/T

    # Get the one-sided specturm
    n_oneside = N//2
    # get the one side frequency
    f_oneside = freq[1:n_oneside]

    ax1.plot(f_oneside, np.abs(X[1:n_oneside]),color=colors[i%10], label=f'zone␣
 ↪z{i}')
    ax1.set_xlabel('Freq (Hz)')
    ax1.set_ylabel('FFT Amplitude |X(freq)|')
```

```
    ax1.set_xscale('log')
    ax1.get_xaxis().set_major_formatter(matplotlib.ticker.ScalarFormatter())
    ax1.set_xticks(ticks=[sr/(1*365*24), sr/(182.5*24), sr/(30.5*24),sr/(7*24),␣
↪sr/24, sr/12], labels=['1/year', '1/semester', '1/month', '1/week', '1/day',␣
↪'1/12H'])

ax1.legend()
plt.show()
```



[964]:
```
# intuition: winter/summer high peaks -> 6month cycle (high consumption, but␣
↪for different reasons too cold/hot resp.)
```

[965]:
```
# save uptill here
# cleaned data
# stations are highly correlated: calculate station mean, min, max, std

df['z_avg'] = df[[f'z{i}'for i in range(1,21)]].mean(axis=1)
df['z_min'] = df[[f'z{i}'for i in range(1,21)]].min(axis=1)
df['z_max'] = df[[f'z{i}'for i in range(1,21)]].max(axis=1)
df['z_std'] = df[[f'z{i}'for i in range(1,21)]].std(axis=1)

df.to_parquet('../data/processed/load.parquet')
```

[966]:
```
df = pd.read_parquet('../data/processed/load.parquet')
```

[967]:
```
from patsy import dmatrix, dmatrices

def lag(col, start, end=None):
    s = col

    # create the range of lags
    end = end or start + 1   # if no end provided, we only iterate on start
    r = range(start, end)

    # apply .shift to get lags on every timeseries which are segregated using␣
↪groupby
```

```
    # list comprehension performs the above step for the entire list of lags␣
↪(start to end)
    ss = [ s.shift(n)for n in r ]

    # concat all the lags together into one dataframe and return it
    df =  pd.concat(ss, axis=1)
    df.columns = [f'{s.name}.lag{x}' for x in r]
    return df.bfill()

def roll(col, window, aggfunc=None):
    if aggfunc is None:
        return col.rolling(window).mean()
    else:
        return col.rolling(window).apply(aggfunc)
```

[968]:
```
from sklearn.model_selection import TimeSeriesSplit

ts_cv = TimeSeriesSplit(test_size=24*7, max_train_size=24*365*1, n_splits=3)
```

[969]:
```
index = df.index
all_splits = list(ts_cv.split(df))

for i in range(ts_cv.n_splits):
    print(f'fold {i}')
    print('   train:', index[all_splits[i][0]].min(), ' - ',␣
↪index[all_splits[i][0]].max())
    print('   test :', index[all_splits[i][1]].min(), ' - ',␣
↪index[all_splits[i][1]].max())
```

```
fold 0
   train: 2007-06-10 00:00:00  -  2008-06-08 23:00:00
   test : 2008-06-09 00:00:00  -  2008-06-15 23:00:00
fold 1
   train: 2007-06-17 00:00:00  -  2008-06-15 23:00:00
   test : 2008-06-16 00:00:00  -  2008-06-22 23:00:00
fold 2
   train: 2007-06-24 00:00:00  -  2008-06-22 23:00:00
   test : 2008-06-23 00:00:00  -  2008-06-29 23:00:00
```

[970]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import HistGradientBoostingRegressor
from lightgbm import LGBMRegressor
from sklearn.model_selection import cross_validate
```

[971]:

```
import math
import cloudpickle

from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error,␣
↪mean_absolute_error

def evaluate_cv(model, X, y, cv):
    cv_results = []
    all_splits = list(cv.split(X,y))
    for i in range(cv.n_splits):
        m = cloudpickle.loads(cloudpickle.dumps(model))
        m.fit(X.iloc[all_splits[i][0]], y.iloc[all_splits[i][0]])
        y_pred = m.predict(X.iloc[all_splits[i][1]])
        y_true = y.iloc[all_splits[i][1]]
        cv_results.append({
            'model': m,
            'scores': {
                'rmse': math.sqrt(mean_squared_error(y_true, y_pred)),
                'mape': mean_absolute_percentage_error(y_true, y_pred),
                'mae': mean_absolute_error(y_true, y_pred),
            }
        })

    mae = np.array([x['scores']['mae'] for x in cv_results])
    mape = np.array([x['scores']['mape'] for x in cv_results])
    rmse = np.array([x['scores']['rmse'] for x in cv_results])


    print(
        f"Mean Absolute Error     (MAE) : {mae.mean():.2f} +/- {mae.std():.
↪2f}\n"
        f"Root Mean Squared Error (RMSE): {rmse.mean():.2f} +/- {rmse.std():.
↪2f}\n"
        f"Mean Absolute % Error   (MAPE): {mape.mean():.2f} +/- {mape.std():.
↪2f}\n"
    )

    model_performance = {
        'mae': mae.mean(),
        'mape': mape.mean(),
        'rmse': rmse.mean()
    }

    return cv_results, model_performance
```

[976]:
```
def plot_cv(cv_results, X, y, cv):
    all_splits = list(cv.split(X,y))
    df_cv = None
    for i in range(cv.n_splits):

        # build a small dataframe ts, y and yhat
        d = pd.DataFrame(data={
                'y': y.iloc[all_splits[i][1]],
                'yhat':cv_results[i]['model'].predict(X.iloc[all_splits[i][1]])
            },
            index=y.iloc[all_splits[i][1]].index
        )

        # concatenate time cross validation
        df_cv = d if df_cv is None else pd.concat([df_cv, d])

    # plot everything,
    # separate the cross validation blocks with vertical bars
    fig = df_cv.plot()
    for i in range(cv.n_splits):
        fig.vlines(y.iloc[all_splits[i][1]].index.max(), df_cv.min().min(),␣
 ↪df_cv.max().max(), color='green', linestyles ="dashed")

def del_cv(cv_results):
    for result in results:
        try:
            del result['model']
        except:
            pass

model_performance = dict()
```

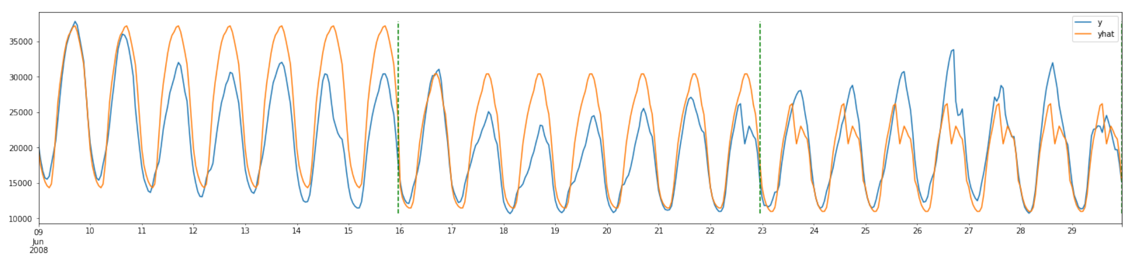
[977]:
```
import re
import patsy

def transform(formula, df):
    X_formula = patsy.dmatrix(formula_like=formula, data=pd.DataFrame(df))
    columns = X_formula.design_info.column_names
    columns = [re.sub(r'[[)\(\].,]', '_', x) for x in columns]
    columns = [re.sub(r'_+', '_', x) for x in columns]
    columns = [x.strip('_').lower() for x in columns]
    return pd.DataFrame(X_formula, columns=columns, index=df.index)
```

[978]:
```
class SarimaSK:
    def __init__(self, model, params):
        self.model = model
        self.modelresult = None
```

```
        self.params = params
        self.samples = 0
        self.ts_max = None

    def fit(self, X, y):
        self.samples = len(y)
        self.ts_max = y.index.max()
        with suppress_stdout_stderr():
            self.modelresult = self.model(y, **self.params).fit()

    def predict(self, X, y=None):
        return self.modelresult.predict(self.samples, self.samples+len(X)-1,␣
↪dynamic=True)
```

[980]:
```
import statsmodels.api as sm

X = df[[]]
y = df['z1']

ts_cv = TimeSeriesSplit(test_size=24*7, max_train_size=24*28*1, n_splits=3)

snaive = SarimaSK(sm.tsa.statespace.SARIMAX, {"order": (0, 0, 0),␣
↪"seasonal_order": (0, 1, 0, 24), "trend": "n"})
results, model_performance['snaive(7*24)'] = evaluate_cv(snaive, X, y, cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

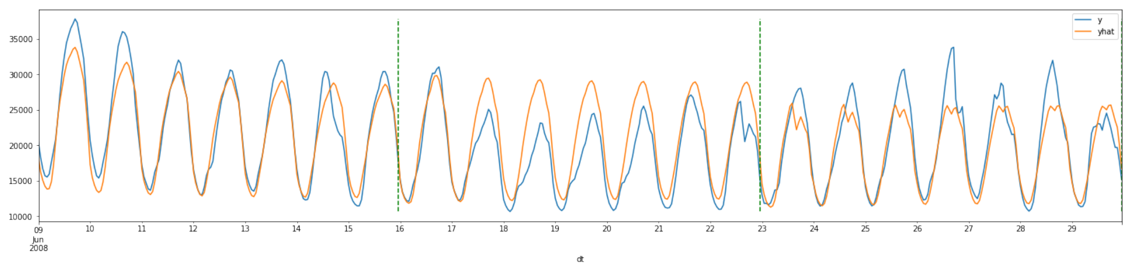
```
Mean Absolute Error     (MAE) : 3135.44 +/- 709.42
Root Mean Squared Error (RMSE): 4096.03 +/- 690.77
Mean Absolute % Error   (MAPE): 0.15 +/- 0.03
```



[982]:
```
import statsmodels.api as sm

X = df[[]]
y = df['z1']
```

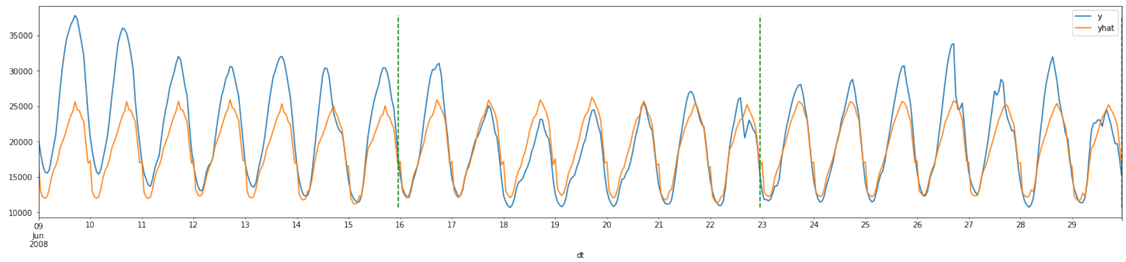
```
ts_cv = TimeSeriesSplit(test_size=24*7, max_train_size=24*365, n_splits=3)

sarima = SarimaSK(sm.tsa.statespace.SARIMAX, {"order": (0, 0, 0),␣
↪"seasonal_order": (1, 1, 1, 24), "trend": "n"})
results, model_performance['sarima((0,0,0), (1,1,1,24)'] = evaluate_cv(sarima,␣
↪X, y, cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

```
Mean Absolute Error     (MAE) : 2131.85 +/- 523.82
Root Mean Squared Error (RMSE): 2737.56 +/- 609.71
Mean Absolute % Error   (MAPE): 0.10 +/- 0.04
```



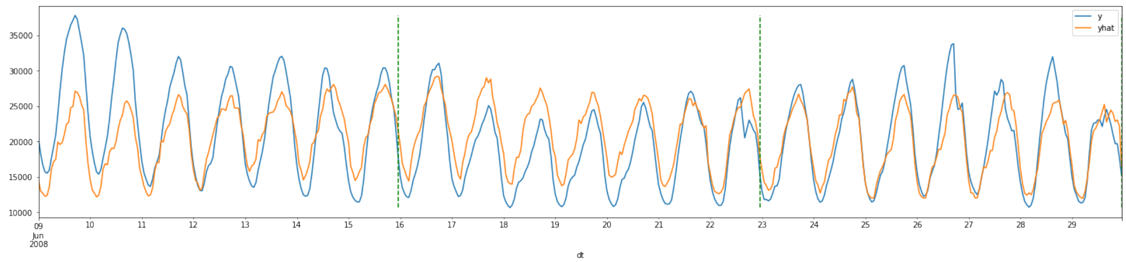
[983]:
```
# stateless

X = transform("""
    0 +
    C(month) +
    C(hour) +
    C(dow)
    """, df)

ts_cv = TimeSeriesSplit(test_size=24*7, n_splits=3)

results, model_performance['BoostReg stateless'] =␣
↪evaluate_cv(HistGradientBoostingRegressor(random_state=42), X, y, cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

```
Mean Absolute Error     (MAE) : 2642.54 +/- 1179.86
Root Mean Squared Error (RMSE): 3452.93 +/- 1445.65
Mean Absolute % Error   (MAPE): 0.12 +/- 0.04
```



[985]:
```
X = transform("""
    0 +
    lag(z1, 168, 172) +
    lag(z_avg, 168, 172) +
    lag(s_min, 168, 172) +
    lag(s_avg, 168, 172) +
    C(month) +
    C(hour)
    """, df)

ts_cv = TimeSeriesSplit(test_size=24*7, n_splits=3)
results, model_performance['BoostReg last week avgs'] =␣
 ↪evaluate_cv(HistGradientBoostingRegressor(random_state=42), X, y, cv=ts_cv)
plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

```
Mean Absolute Error     (MAE) : 2994.55 +/- 783.76
Root Mean Squared Error (RMSE): 3693.34 +/- 929.09
Mean Absolute % Error   (MAPE): 0.15 +/- 0.04
```



[986]:
```
class ProphetSK:
    def __init__(self, model):
        self.model = model

    def fit(self, X, y):
```

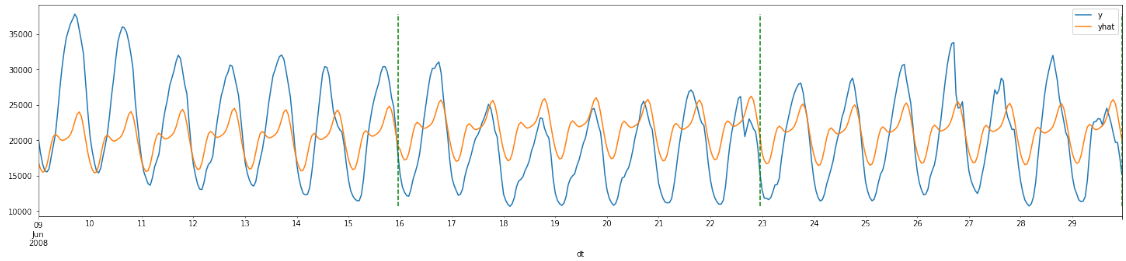
```
        for c in X.columns:
            self.model.add_regressor(c)
        if X.shape[1]>0:
            ts = pd.DataFrame(y).join(X).reset_index()
            ts.columns = ['ds', 'y']+ list(X.columns)
        else:
            ts = y.reset_index()
            ts.columns =['ds', 'y']

        with suppress_stdout_stderr():
            self.model.fit(ts)

    def predict(self, X, y=None):
        ts_pred = X.copy()
        ts_pred.index.name = 'ds'
        ts_pred=ts_pred.reset_index()
        res = self.model.predict(ts_pred)
        res = res.set_index('ds')
        return res['yhat']
```

[989]:
```
from prophet import Prophet

X = df[[]]
y = df['z1']
```

[990]:
```
m = Prophet()
m.add_country_holidays(country_name='US')
prophet = ProphetSK(m)
```

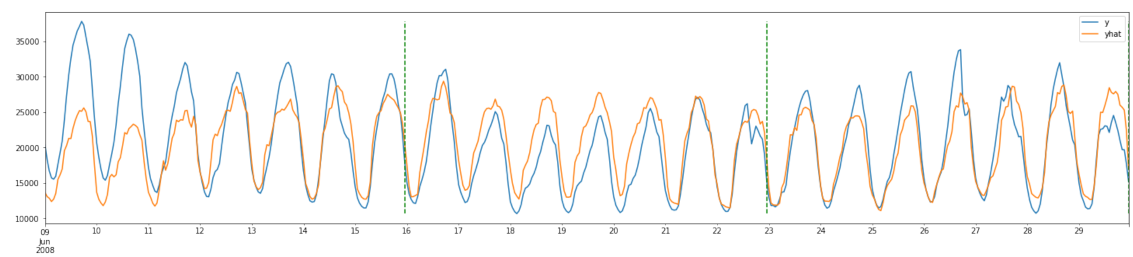
[991]:
```
ts_cv = TimeSeriesSplit(test_size=24*7, max_train_size=24*365*3, n_splits=3)
results, model_performance['Prophet'] = evaluate_cv(prophet, X, y, cv=ts_cv)
plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

```
Mean Absolute Error     (MAE) : 4433.60 +/- 383.53
Root Mean Squared Error (RMSE): 5310.07 +/- 694.36
Mean Absolute % Error   (MAPE): 0.24 +/- 0.03
```



```
[993]: X = transform("""
    0 +
    lag(z1, 168, 168+6) +
    lag(z1, 168*2) +
    lag(z1, 168*3) +
    lag(z_avg, 168*1) +
    lag(z_avg, 168*2) +
    lag(z_avg, 168*3) +
    lag(roll(s_min, 24), 168*1) +
    lag(roll(s_max, 24), 168*1) +
    lag(roll(s_std, 24), 168*1) +
    lag(roll(s_min, 24), 168*2) +
    lag(roll(s_max, 24), 168*2) +
    lag(roll(s_std, 24), 168*2) +
    lag(roll(s_avg, 3), 168) +
    C(month) +
    C(hour) +
    C(dow)
    """, df)

ts_cv = TimeSeriesSplit(test_size=24*7, n_splits=3)

results, model_performance['LGBM avgs'] =␣
 ↪evaluate_cv(LGBMRegressor(random_state=42), X, y, cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

```
Mean Absolute Error      (MAE) : 2879.69 +/- 733.82
Root Mean Squared Error (RMSE): 3677.08 +/- 1030.73
Mean Absolute % Error   (MAPE): 0.14 +/- 0.03
```



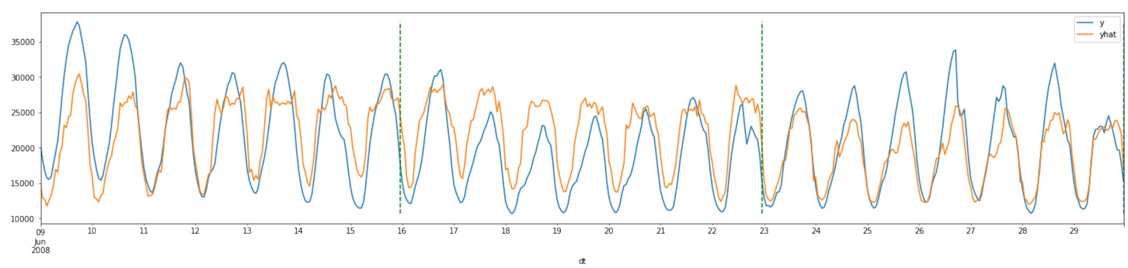

```
[891]: lags_zones = ' + '.join([f'lag(z{i}, 168, 172)' for i in range(1,21)])
lags_stations = ' + '.join([f'lag(s{i}, 168, 172)' for i in range(1,11)])
```

```
formula = f" 0 + {lags_zones} + {lags_stations} + C(month) + C(hour) + C(dow)"
formula
```

[891]: ' 0 + lag(z1, 168, 168+3) + lag(z2, 168, 168+3) + lag(z3, 168, 168+3) + lag(z4, 168, 168+3) + lag(z5, 168, 168+3) + lag(z6, 168, 168+3) + lag(z7, 168, 168+3) + lag(z8, 168, 168+3) + lag(z9, 168, 168+3) + lag(z10, 168, 168+3) + lag(z11, 168, 168+3) + lag(z12, 168, 168+3) + lag(z13, 168, 168+3) + lag(z14, 168, 168+3) + lag(z15, 168, 168+3) + lag(z16, 168, 168+3) + lag(z17, 168, 168+3) + lag(z18, 168, 168+3) + lag(z19, 168, 168+3) + lag(z20, 168, 168+3) + lag(s1, 168, 168+3) + lag(s2, 168, 168+3) + lag(s3, 168, 168+3) + lag(s4, 168, 168+3) + lag(s5, 168, 168+3) + lag(s6, 168, 168+3) + lag(s7, 168, 168+3) + lag(s8, 168, 168+3) + lag(s9, 168, 168+3) + lag(s10, 168, 168+3) + C(month) + C(hour) + C(dow)'

[995]:
```
X = transform(formula, df)

ts_cv = TimeSeriesSplit(test_size=24*7, n_splits=3)

results, model_performance['LGBM all signals'] =
↪evaluate_cv(LGBMRegressor(random_state=42), X, y, cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
del_cv(results)
```

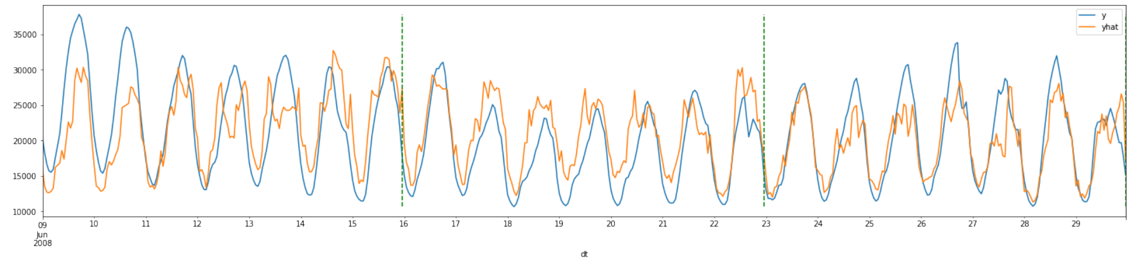
```
Mean Absolute Error     (MAE) : 3389.42 +/- 813.85
Root Mean Squared Error (RMSE): 4132.54 +/- 671.63
Mean Absolute % Error   (MAPE): 0.17 +/- 0.06
```



[996]:
```
from flaml import AutoML
automl = AutoML(task="regression", estimator_list=["lgbm"], verbose=-1)

ts_cv = TimeSeriesSplit(test_size=24*7, n_splits=3)

results, model_performance['Auto LGBM all signals'] = evaluate_cv(automl, X, y,
↪cv=ts_cv)

plot_cv(results, X, y, cv=ts_cv)
```

```
del_cv(results)
```

```
Mean Absolute Error     (MAE) : 3288.99 +/- 809.36
Root Mean Squared Error (RMSE): 4054.93 +/- 835.25
Mean Absolute % Error   (MAPE): 0.16 +/- 0.04
```



[997]:
```
import os
import datetime

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
```

[998]:
```
class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df=train_df, val_df=val_df, test_df=test_df, label_columns=None):

        # Store the raw data.
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Work out the label column indices.
        self.label_columns = label_columns
        if label_columns is not None:
            self.label_columns_indices = {name: i for i, name in enumerate(label_columns)}

        self.column_indices = {name: i for i, name in enumerate(train_df.columns)}
```

```
        # Work out the window parameters.
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[self.
↪labels_slice]

    def split_window(self, features):
        inputs = features[:, self.input_slice, :]
        labels = features[:, self.labels_slice, :]
        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:, :, self.column_indices[name]] for name in self.
↪label_columns],
                axis=-1)

        # Slicing doesn't preserve static shape information, so set the shapes
        # manually. This way the `tf.data.Datasets` are easier to inspect.
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])

        return inputs, labels

    def plot(self, model=None, plot_col='z1', max_subplots=3):
        inputs, labels = self.example
        plt.figure(figsize=(12, 8))
        plot_col_index = self.column_indices[plot_col]
        max_n = min(max_subplots, len(inputs))
        for n in range(max_n):
            plt.subplot(max_n, 1, n+1)
            plt.ylabel(f'{plot_col} [normed]')
            plt.plot(self.input_indices, inputs[n, :, plot_col_index],␣
↪label='Inputs', c=colors[0], zorder=-10)

            if self.label_columns:
                label_col_index = self.label_columns_indices.get(plot_col, None)
            else:
                label_col_index = plot_col_index
```

```
            if label_col_index is None:
                continue

            plt.plot(
                self.label_indices,
                labels[n, :, label_col_index],
                label='Labels',
                marker='.',
                c=colors[0])

            if model is not None:
                predictions = model(inputs)
                plt.plot(
                    self.label_indices,
                    predictions[n, :, label_col_index],
                    label='Predictions',
                    marker='.',
                    c=colors[1])

        if n == 0:
            plt.legend()

        plt.xlabel('Time [h]')

    def make_dataset(self, data):
        data = np.array(data, dtype=np.float32)
        ds = tf.keras.utils.timeseries_dataset_from_array(
          data=data,
          targets=None,
          sequence_length=self.total_window_size,
          sequence_stride=1,
          shuffle=True,
          batch_size=32,)

        ds = ds.map(self.split_window)

        return ds

    @property
    def train(self):
        return self.make_dataset(self.train_df)

    @property
    def val(self):
        return self.make_dataset(self.val_df)

    @property
```

```python
    def test(self):
        return self.make_dataset(self.test_df)

    @property
    def example(self):
        """Get and cache an example batch of `inputs, labels` for plotting."""
        result = getattr(self, '_example', None)

        if result is None:
            # No example batch was found, so get one from the `.train` dataset
            result = next(iter(self.train))
            # And cache it for next time
            self._example = result

        return result

    def __repr__(self):
        return '\n'.join([
            f'Total window size: {self.total_window_size}',
            f'Input indices: {self.input_indices}',
            f'Label indices: {self.label_indices}',
            f'Label column name(s): {self.label_columns}'])
```

[999]:
```python
MAX_EPOCHS = 20

def compile_and_fit(model, window, patience=2):
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                      patience=patience,
                                                      mode='min')

    model.compile(loss=tf.losses.MeanSquaredError(),
                  optimizer=tf.optimizers.Adam(),
                  metrics=[tf.metrics.MeanAbsoluteError()])

    history = model.fit(window.train, epochs=MAX_EPOCHS,
                        validation_data=window.val,
                        callbacks=[early_stopping])
    return history
```

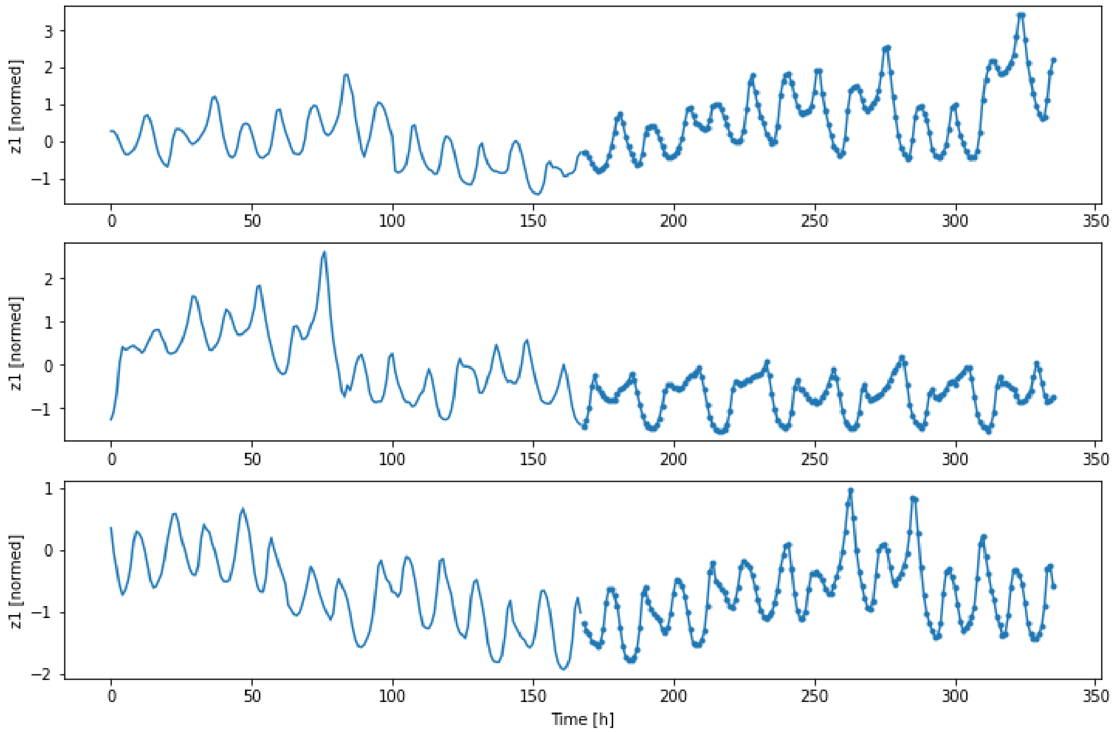
[1038]:
```python
WINDOW_SIZE = 2*7*24
NORMALIZE = True

# qualify the dataframe
n = len(df)
column_indices = {name: i for i, name in enumerate(df.columns)}
num_features = df.shape[1]
```

```
# train, validate, test
train_df = df[:-2*WINDOW_SIZE]
val_df = df[-2*WINDOW_SIZE:-1*WINDOW_SIZE]
test_df = df[-1*WINDOW_SIZE:]

if NORMALIZE:
    train_mean = train_df.mean()
    train_std = train_df.std()

    train_df = (train_df - train_mean) / train_std
    val_df = (val_df - train_mean) / train_std
    test_df = (test_df - train_mean) / train_std
```

[1039]:
```
OUT_STEPS = 7*24
multi_window = WindowGenerator(
    input_width=OUT_STEPS,
    label_width=OUT_STEPS,
    shift=OUT_STEPS,
    train_df=train_df,
    val_df=val_df,
    test_df=test_df,
    label_columns=['z1'])

multi_window.plot()
multi_window
```

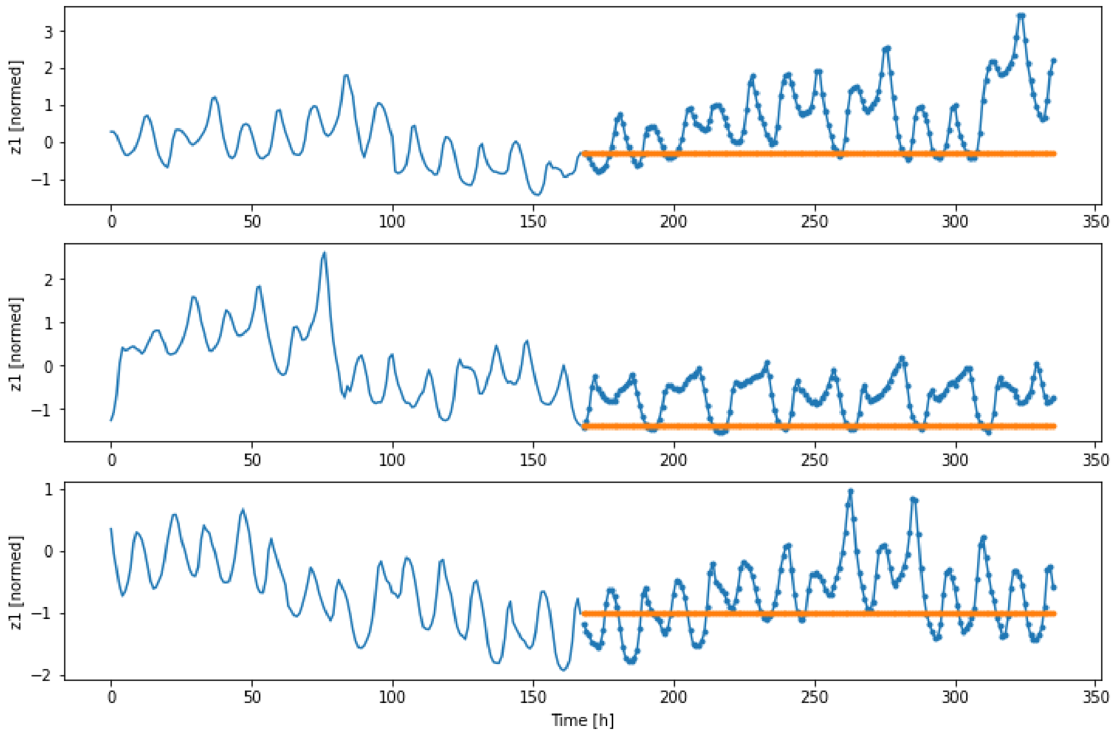
[1039]:
```
Total window size: 336
Input indices: [  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167]
Label indices: [168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183
184 185
 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203
 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221
 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257
 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275
 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293
 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311
```

```
 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329
 330 331 332 333 334 335]
Label column name(s): ['z1']
```



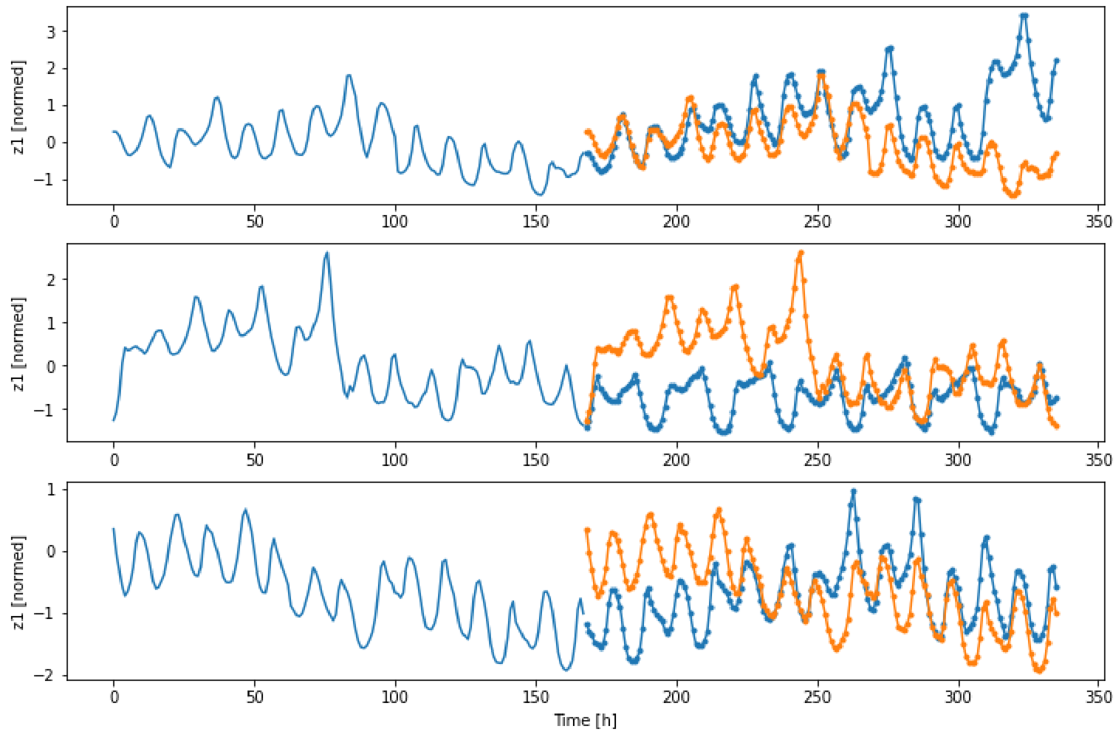
[1054]:
```python
class MultiStepLastBaseline(tf.keras.Model):
    def call(self, inputs):
        return tf.tile(inputs[:, -1:, :], [1, OUT_STEPS, 1])

last_baseline = MultiStepLastBaseline()
last_baseline.compile(loss=tf.losses.MeanSquaredError(),
                      metrics=[tf.metrics.MeanAbsoluteError()])

mae = np.array([
    last_baseline.evaluate(multi_window.val, verbose=0),
    last_baseline.evaluate(multi_window.test, verbose=0)
])[:,1]

print(f"Mean Absolute Error     (MAE) : {mae.mean():.2f} +/- {mae.std():.2f}")
model_performance['Keras Naive'] = {'mae': mae.mean()}

multi_window.plot(last_baseline)
```

```
Mean Absolute Error     (MAE) : 1.16 +/- 0.03
```



[1055]:
```
class RepeatBaseline(tf.keras.Model):
    def call(self, inputs):
        return inputs

repeat_baseline = RepeatBaseline()
repeat_baseline.compile(loss=tf.losses.MeanSquaredError(), metrics=[tf.metrics.
 ↪MeanAbsoluteError()])

mae = np.array([
    repeat_baseline.evaluate(multi_window.val, verbose=0),
    repeat_baseline.evaluate(multi_window.test, verbose=0)
])[:,1]

print(f"Mean Absolute Error     (MAE) : {mae.mean():.2f} +/- {mae.std():.2f}")
model_performance['Keras sNaive(7*24)'] = {'mae': mae.mean()}

multi_window.plot(repeat_baseline)
```

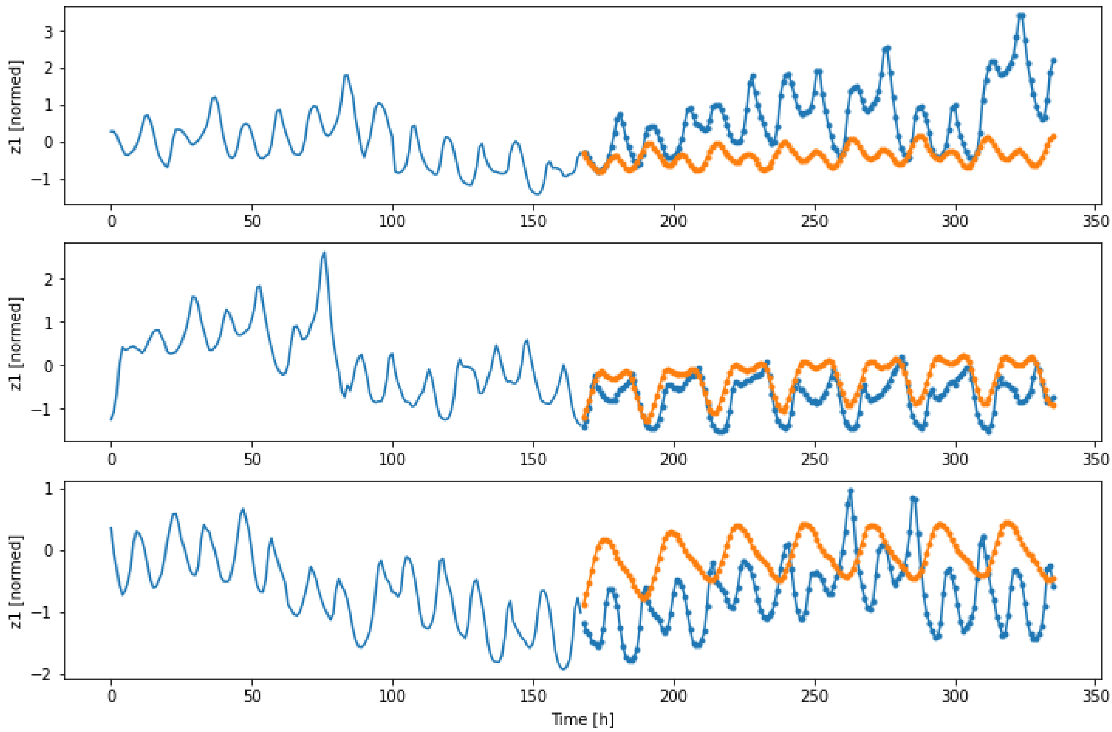
```
Mean Absolute Error     (MAE) : 0.95 +/- 0.12
```




```
[1057]: multi_linear_model = tf.keras.Sequential([
    # Take the last time-step.
    # Shape [batch, time, features] => [batch, 1, features]
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    # Shape => [batch, 1, out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features, kernel_initializer=tf.
 ↪initializers.zeros()),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_linear_model, multi_window)

IPython.display.clear_output()

mae = np.array([
    multi_linear_model.evaluate(multi_window.val, verbose=0),
    multi_linear_model.evaluate(multi_window.test, verbose=0)
])[:,1]

print(f"Mean Absolute Error     (MAE) : {mae.mean():.2f} +/- {mae.std():.2f}")
model_performance['Keras LinRegression'] = {'mae': mae.mean()}
```

```
multi_window.plot(multi_linear_model)
```

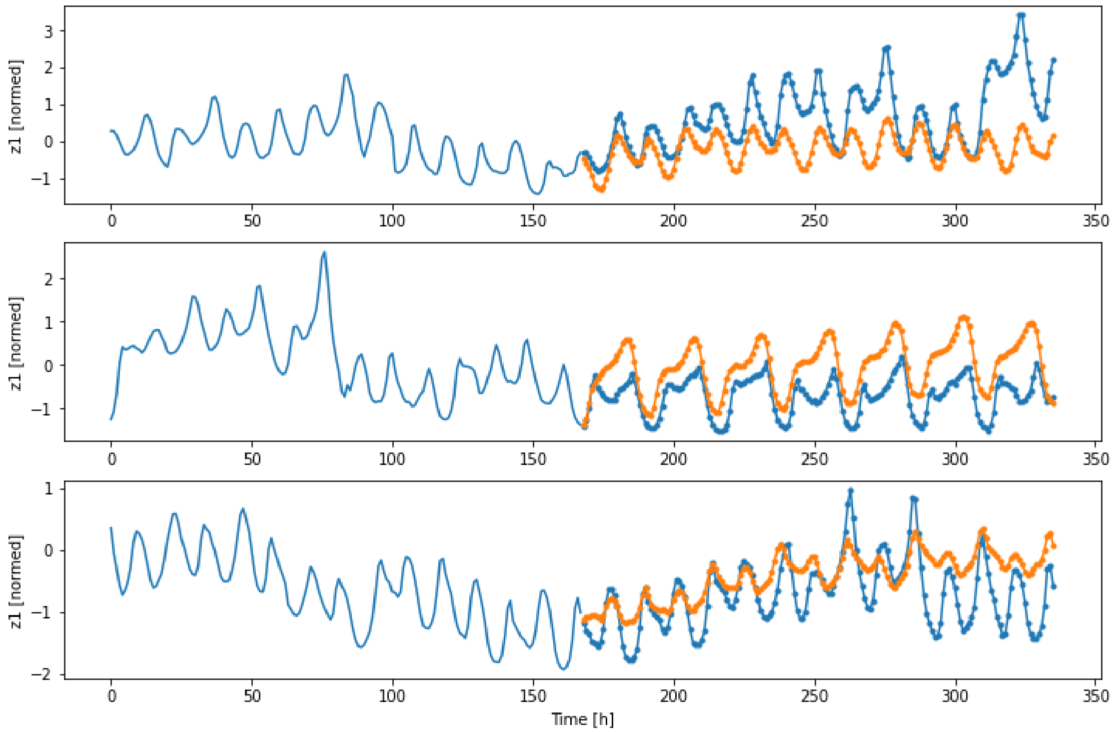
```
Mean Absolute Error     (MAE) : 0.78 +/- 0.16
```



[1058]:
```
CONV_WIDTH = 3
multi_conv_model = tf.keras.Sequential([
    # Shape [batch, time, features] => [batch, CONV_WIDTH, features]
    tf.keras.layers.Lambda(lambda x: x[:, -CONV_WIDTH:, :]),
    # Shape => [batch, 1, conv_units]
    tf.keras.layers.Conv1D(256, activation='relu', kernel_size=(CONV_WIDTH)),
    # Shape => [batch, 1,  out_steps*features]
    tf.keras.layers.Dense(OUT_STEPS*num_features, kernel_initializer=tf.
 ↪initializers.zeros()),
    # Shape => [batch, out_steps, features]
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])

history = compile_and_fit(multi_conv_model, multi_window)

IPython.display.clear_output()

mae = np.array([
    multi_conv_model.evaluate(multi_window.val, verbose=0),
```

```
    multi_conv_model.evaluate(multi_window.test, verbose=0)
])[:,1]

print(f"Mean Absolute Error     (MAE) : {mae.mean():.2f} +/- {mae.std():.2f}")
model_performance['Keras Conv1D'] = {'mae': mae.mean()}

multi_window.plot(multi_conv_model)
```

```
Mean Absolute Error     (MAE) : 0.74 +/- 0.01
```

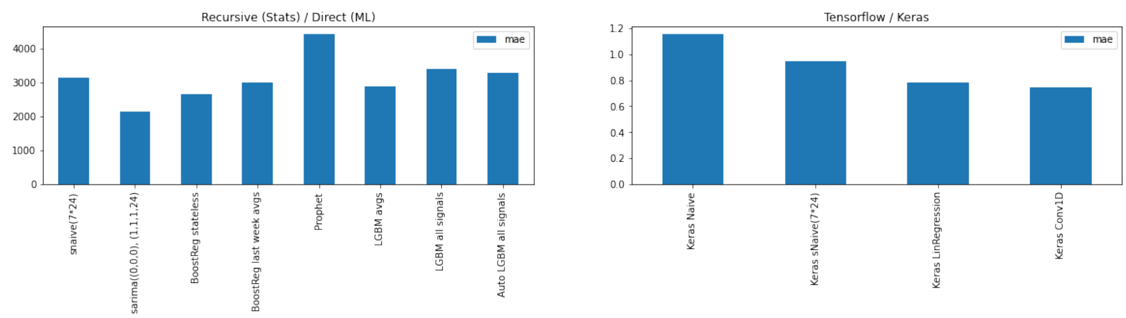


[1059]:
```
multi_conv_model.summary()
```

```
Model: "sequential_66"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lambda_56 (Lambda)          (None, 3, 44)             0

 conv1d_45 (Conv1D)          (None, 1, 256)            34048

 dense_75 (Dense)            (None, 1, 7392)           1899744

 reshape_53 (Reshape)        (None, 168, 44)           0
```

```
=================================================================
Total params: 1,933,792
Trainable params: 1,933,792
Non-trainable params: 0
_________________________________________________________________
```

[1080]:
```python
fig, ax = plt.subplots(1,2)

perf_sel ={k:v['mae'] for k, v in model_performance.items() if not k.startswith('Keras')}
pd.DataFrame(perf_sel.values(), perf_sel.keys(), columns=['mae']).plot.bar(figsize=(20,3), title='Recursive (Stats) / Direct (ML)', ax=ax[0]);

perf_sel ={k:v['mae'] for k, v in model_performance.items() if k.startswith('Keras')}
pd.DataFrame(perf_sel.values(), perf_sel.keys(), columns=['mae']).plot.bar(figsize=(20,3), title='Tensorflow / Keras', ax=ax[1]);
```



[1160]:
```python
# Forecasting:

# Current best: sarima
# Best next candidates: 'Keras Convolution', and 'LGBM avgs'

# augment LGBM
#   ... direct approach but separate models for each step
#     - create a model separately for each prediction step (168 models)

#   ... with a recursive approach (autoregressive methodology)
#     - feed forecasted model and forecasted regressors to the next model
```

[1164]:
```python
lags_zones = ' + '.join([f'lag(z{i}, 168, {168+3})' for i in range(1,21)])
lags_stations = ' + '.join([f'lag(s{i}, 168, {168+3})' for i in range(1,11)])
formula = f" 0 + {lags_zones} + {lags_stations} + C(month) + C(hour) + C(dow)"
print('Formula:' , formula)
X = transform(formula, df)
```

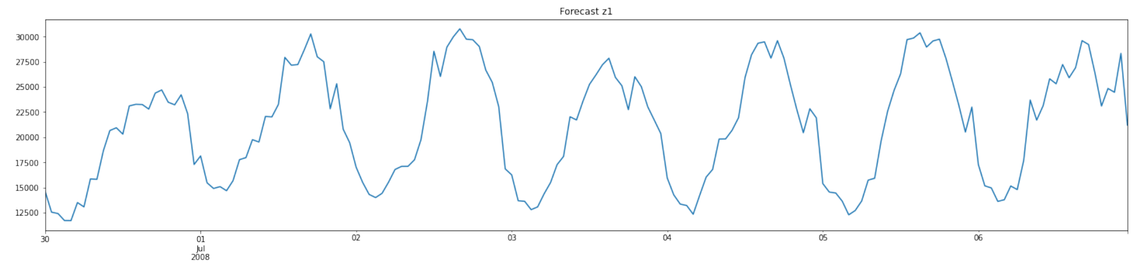
```
from flaml import AutoML
automl = AutoML(task="regression", estimator_list=["lgbm"], verbose=-1)
automl.fit(X, y)
```

```
Formula:  0 + lag(z1, 168, 171) + lag(z2, 168, 171) + lag(z3, 168, 171) +
lag(z4, 168, 171) + lag(z5, 168, 171) + lag(z6, 168, 171) + lag(z7, 168, 171) +
lag(z8, 168, 171) + lag(z9, 168, 171) + lag(z10, 168, 171) + lag(z11, 168, 171)
+ lag(z12, 168, 171) + lag(z13, 168, 171) + lag(z14, 168, 171) + lag(z15, 168,
171) + lag(z16, 168, 171) + lag(z17, 168, 171) + lag(z18, 168, 171) + lag(z19,
168, 171) + lag(z20, 168, 171) + lag(s1, 168, 171) + lag(s2, 168, 171) + lag(s3,
168, 171) + lag(s4, 168, 171) + lag(s5, 168, 171) + lag(s6, 168, 171) + lag(s7,
168, 171) + lag(s8, 168, 171) + lag(s9, 168, 171) + lag(s10, 168, 171) +
C(month) + C(hour) + C(dow)
```

[1174]:
```
df_pred = pd.concat([df, pd.DataFrame(index=pd.date_range(start = '2008-06-30␣
↪00:00:00', periods=7*24, freq='H'))])

# featurize time
df_pred['year'] = df_pred.index.year
df_pred['month'] = df_pred.index.month
df_pred['day'] = df_pred.index.day
df_pred['hour'] = df_pred.index.hour
df_pred['dow'] = df_pred.index.dayofweek
```

[1175]:
```
X_pred = transform(formula, df_pred)[-7*24:]
```

[1185]:
```
pd.Series(automl.predict(X_pred), index=X_pred.index).plot(title='Forecast z1');
```



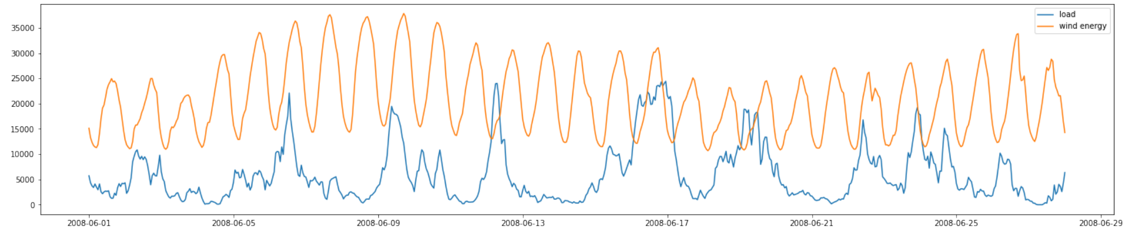
[1186]:
```
df_energy = pd.read_csv('../data/raw/gef2012-wind/train.csv')

df_energy.index = pd.to_datetime(df_energy.date, format='%Y%m%d%H')
df_energy = df_energy.drop(columns=['date'])
energy = df_energy.sum(axis=1)
```

[1187]:
```
energy
```

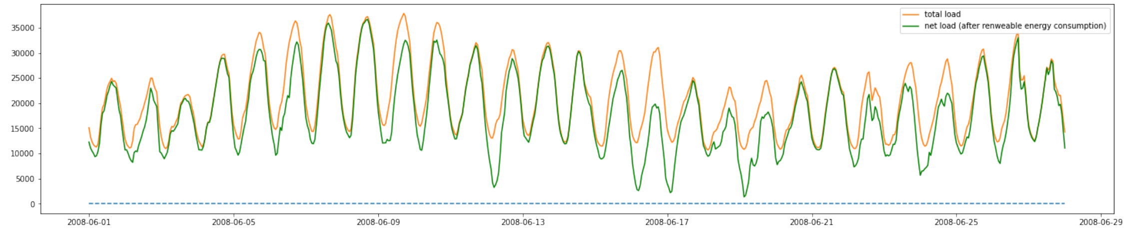
[1187]:
```
date
2009-07-01 00:00:00    1.102
2009-07-01 01:00:00    0.879
2009-07-01 02:00:00    0.447
2009-07-01 03:00:00    0.299
2009-07-01 04:00:00    0.205
                       …
2012-06-26 08:00:00    1.148
2012-06-26 09:00:00    1.285
2012-06-26 10:00:00    1.257
2012-06-26 11:00:00    1.250
2012-06-26 12:00:00    0.941
Length: 18757, dtype: float64
```

[1188]:
```
a = df["2008-06-01 00:00:00":"2008-06-28 00:00:00"]['z1'].values
b = energy["2010-06-01 00:00:00":"2010-06-28 00:00:00"].values
```

[1220]:
```
plt.plot(df["2008-06-01 00:00:00":"2008-06-28 00:00:00"].index, b*5000,␣
↪label='load');
plt.plot(df["2008-06-01 00:00:00":"2008-06-28 00:00:00"].index, a, label='wind␣
↪energy');
plt.legend();
```



[1221]:
```
plt.plot(df["2008-06-01 00:00:00":"2008-06-28 00:00:00"].index, a,␣
↪color=colors[1], label='total load');
plt.plot(df["2008-06-01 00:00:00":"2008-06-28 00:00:00"].index, a - b*2500,␣
↪color='green', label='net load (after renweable energy consumption)')
plt.hlines(0, date2num(pd.Timestamp("2008-06-01 00:00:00")),date2num(pd.
↪Timestamp("2008-06-28 00:00:00")), color=colors[0], linestyles ="dashed");
plt.legend();
```



[1215]:
```
(a - b*2500).sum()/a.sum()
```

[1215]: 0.8579964799425028

[ ]:
```
```