Feature Engineering for Cambridge UK Weather Forecasting

Feature engineering for time series analysis of Cambridge UK temperature measurements taken at the University computer lab weather station.

This notebook is being developed on Google Colab. Initially I was most interested in short term temperature forecasts (less than 2 hours) but now mostly produce results up to 24 hours in the future for comparison with earlier baselines.

See my previous notebooks, web apps etc:

- Cambridge UK temperature forecast python notebooks
- Cambridge UK temperature forecast R models
- Bayesian optimisation of prophet temperature model
- Cambridge University Computer Laboratory weather station R shiny web app

The linked notebooks, web apps etc contain further details including:

- data description
- data cleaning and preparation
- data exploration

In particular, see the notebooks:

- cammet_baselines_2021 including persistent, simple exponential smoothing, Holt Winter's exponential smoothing and vector autoregression
- keras_mlp_fcn_resnet_time_series, which uses a streamlined version of data preparation from Tensorflow time series forecasting tutorial
- lstm_time_series with stacked LSTMs, bidirectional LSTMs and ConvLSTM1D networks
- cnn_time_series with Conv1D, multi-head Conv1D, Conv2D and Inception-style models
- encoder_decoder which includes autoencoder with attention, encoder decoder with teacher forcing, transformer with teacher forcing and padding, encoder only with MultiHeadAttention
- gradient_boosting lightGBM models with darts time series framework and Borota-style shadow variables for feature selection
- tsfresh_feature_engineering automated feature engineering and selection for time series analysis of Cambridge UK weather measurements

Most of the above repositories, notebooks, web apps etc were built on both less data and less thoroughly cleaned data.

Table of Contents

TODO Add internal links before "final" commits

Some sections may get added/deleted during development.

Don't want any broken links, so finish later.

Code Setup

- Library Imports
- Environment Variables
- Custom Functions

Data Setup

- Import ComLab Data
- Meteorological Feature Calculations
- Solar Feature Calculations
- Seasonal Decomposition - statsmodels
    - unobserved componets model
    - experimental
- Split Data
- Seasonal Decomposition - prophet

Exploratory Data Analysis

- Auto-correlation and Partial Autocorrelation Plots
- Stationarity Tests

Feature Engineering

- Rolling Statistics
- Cross-correlation Statistics
- catch22
- tsfeatures
    - plus additional 'intervals' and 'pacf' features
- Bivariate Features
- Pairwise Features

Conclusion

- Features Summary
- What Worked
- What Failed
- Rejected Ideas
- Future Work

Metadata

Load Libraries

Load most of the required packages.

```
import re
import sys
import math
import timeit
import datetime
import itertools
import subprocess
import pkg_resources

import numpy as np
import pandas as pd
from google.colab import files
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
from tqdm import tqdm
from scipy import stats, signal, special
from prophet import Prophet
from itertools import product
import statsmodels.api as sm
from statsmodels.tsa.stattools import acf, pacf, lagmat, coint
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.nonparametric.smoothers_lowess import lowess
from sklearn.utils import check_X_y
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler, SplineTransformer
from sklearn.feature_selection import f_regression, mutual_info_regression, r_regı


# Reduces variance in results but won't eliminate it :-(
%env PYTHONHASHSEED=0
import random

# set seed to make all processes deterministic
seed = 0
random.seed(seed)
np.random.seed(seed)


%matplotlib inline


# Prevent warnings in sanity_check_before_after_dfs and related functions
import warnings


from google.colab import drive
drive.mount('/content/drive')
```

```
env: PYTHONHASHSEED=0
Mounted at /content/drive
```

Environment Variables

Set some environment variables:

```
HORIZON = 48
Y_COL   = 'y_des' # 'y_des_fft' 'y_res' 'y'
CORE_FEATS = [Y_COL, 'dew.point_des', 'humidity', 'pressure']
FUT_FEATS  = ['irradiance', 'za_rad', 'azimuth_cos']

DAY  = 24 * 60 * 60
YEAR = 365.2425 * DAY

DAILY_OBS  = 48
YEARLY_OBS = int(365.2425 * DAILY_OBS)  # annual observations
DAY_SECS_STEP = int(DAY / DAILY_OBS)

VALID_YEAR = 2021
TEST_YEAR  = 2022
```

Custom Functions

Next, define some utility functions:

- rmse_
- mse_
- mae_
- summarise_backtest
- print_rmse_mae
- drop_cols_correlated_with_feat_cols
- drop_problem_cols
- summarise_historic_comparison
- plot_lagged_feat_imp_subplot
- get_pastcov_features
- get_pastcov_lags
- plot_lagged_feature_importances
- plot_feature_importances
- get_feature_importances
- expand_grid
- keep_key
- get_historic_comparison
- _plot_xy_for_label

- plot_multistep_obs_vs_preds
- plot_multistep_obs_vs_mean_preds_by_step
- plot_multistep_obs_preds_dists
- plot_multistep_residuals
- plot_multistep_residuals_dist
- plot_multistep_residuals_vs_predicted
- se_
- metric_ci_vals
- plot_horizon_metrics
- plot_horizon_metrics_boxplots
- plot_multistep_diagnostics
- _filter_out_missing
- plot_multistep_forecast_examples
- get_rmse_mae_from_backtest
- plot_catboost_learning_curve
- plot_lgb_learning_curve
- drop_correlated_cols
- get_feature_selection_scores
- plot_observation_examples
- sanity_check_df_rows_cols_labels
- sanity_check_before_after_dfs
- sanity_check_train_valid_test
- print_train_valid_test_shapes
- plot_feature_history
- plot_feature_history_separately
- check_high_low_thresholds
- get_features_filename
- merge_data_and_aggs
- get_rolling_features
- finalise_rolling_features
- print_null_columns
- print_na_locations
- get_features
- get_darts_series
- plot_short_term_acf
- plot_long_term_acf
- plot_periodogram
- plot_periodograms

```
def _check_obs_preds_lens_eq(obs, preds):
    obs_preds_lens_eq = 1
```

```
    if len(obs) != len(preds):
        print("obs:  ", len(obs))
        print("preds:", len(preds))
        obs_preds_lens_eq = 0

    return obs_preds_lens_eq


def rmse_(obs, preds):
    if _check_obs_preds_lens_eq(obs, preds) == 0:
        stop()
    else:
        return np.sqrt(np.mean((obs - preds) ** 2))


def mse_(obs, preds):
    if _check_obs_preds_lens_eq(obs, preds) == 0:
        stop()
    else:
        return np.mean((obs - preds) ** 2)


def mae_(obs, preds):
    "mean absolute error - equivalent to the keras loss function"
    if _check_obs_preds_lens_eq(obs, preds) == 0:
        stop()
    else:
        return np.mean(np.abs(obs - preds))      # keras loss
        # return np.median(np.abs(obs - preds))  # earlier baselines


def print_rmse_mae(obs, preds, postfix_str, prefix_str = '', digits = 6):
    print(prefix_str, "Backtest RMSE ", postfix_str, ": ",
          round(rmse_(obs, preds), digits),
          sep='')
    print(prefix_str, "Backtest MAE ",  postfix_str, ":  ",
          round( mae_(obs, preds), digits),
          sep='')
    print()


def drop_cols_correlated_with_feat_cols(df, feats_df, threshold=0.95):
  for feat_col in feats_df.columns:
    corrs = df.corrwith(feats_df[feat_col])
    drop_cols = corrs[(corrs > threshold) & (corrs != 1.0)]

    for i in range(len(drop_cols)):
      drop_col = drop_cols.index[i]
      if drop_col in df.columns:  # and drop_col not in feats_df.columns:
        del df[drop_col]

  return df
```

```
def drop_problem_cols(df, lag, drop_cor=True,
                      var_cutoff=0.05, cor_cutoff=0.95, na_cutoff=0.05,
                      verbose = False):

  if verbose:
    print('drop_problem_cols - start:', df.shape)


  # drop all NA columns
  df = df.dropna(axis = 1, how = 'all')

  if verbose:
    print('drop_problem_cols - after dropna:', df.shape)


  # drop single value columns
  df = df.loc[:, (df != df.iloc[lag]).any()]

  if verbose:
    print('drop_problem_cols - after drop single value cols:', df.shape)


  # drop low variance columns
  if 'ds' in df.columns:
    df = df.drop(['ds'], axis=1)
    df = df.loc[:, df.std() > var_cutoff]
    df['ds'] = df.index
  else:
    df = df.loc[:, df.std() > var_cutoff]

  if verbose:
    print('drop_problem_cols - after drop low var cols:', df.shape)


  # drop highly correlated columns
  if drop_cor:
    df = drop_correlated_cols(df, cor_cutoff)

  if verbose:
    print('drop_problem_cols - after drop correlated cols:', df.shape)


  # drop cols with high % of NA values
  pc_thresh = int(na_cutoff * df.shape[0])
  #print('five_pc_thresh:', five_pc_thresh)
  print('columns with null values:')
  display(df.isnull().sum())
  df = df.loc[:, df.isnull().sum() < pc_thresh]

  if verbose:
    print('drop_problem_cols - after drop high % of NAs:', df.shape)

  return df
```

```
def expand_grid(dictionary):
   return pd.DataFrame([row for row in product(*dictionary.values())],
                       columns = dictionary.keys())


def keep_key(d, k):
  """ models = keep_key(models, 'datasets') """
  return {k: d[k]}


def plot_one_step_abs_err_boxplot(one_step, title):
  one_step['abs_err'] = np.abs(one_step['res'])
  one_step[['abs_err']].boxplot(meanline  = False,
                                showmeans = True,
                                showcaps  = True,
                                showbox   = True,
                                # showfliers = False,
                                )
  plt.title(title + '\nboxplot with mean and median')
  plt.suptitle('')
  plt.ylabel('absolute error')
  plt.show()


def plot_one_step_residuals_dist(one_step, title):
  plt.figure(figsize = (12, 16))
  plt.subplot(5, 1, 5)
  pd.Series(one_step['res']).plot(kind = 'density', label='residuals')
  plt.xlim(-10, 10)
  plt.title(title)
  plt.show()


def plot_one_step_residuals(one_step, title):
  x_miss = one_step.loc[one_step['missing'] == 1.0, 'obs'].index
  y_miss = one_step.loc[one_step['missing'] == 1.0, 'res']

  plt.figure(figsize = (12, 16))
  plt.subplot(5, 1, 4)
  plt.scatter(x = one_step.index, y = one_step['res'])
  plt.scatter(x_miss, y_miss, color='red', label='missing')
  plt.axhline(y = 0, color = 'grey')
  plt.xlabel('Index position')
  plt.ylabel('Residuals')
  plt.legend(loc='lower right')
  plt.title(title)
  plt.show()


def plot_one_step_obs_preds_dists(one_step, title):
  obs   = one_step['obs']
  preds = one_step['preds']
```

```
  r2score = r2_score(obs, preds)

  plt.figure(figsize = (12, 16))
  plt.subplot(5, 1, 3)
  pd.Series(obs).plot(kind = 'density', label='observations')
  pd.Series(preds).plot(kind = 'density', label='predictions')
  plt.xlim(-10, 40)
  plt.title(title)
  plt.legend()
  plt.annotate("$R^2$ = {:.3f}".format(r2score), (-7.5, 0.055))
  # plt.tight_layout()
  plt.show()


def plot_one_step_obs_vs_preds(one_step, title):

  obs   = one_step['obs']
  preds = one_step['preds']
  x_miss = one_step.loc[one_step['missing'] == 1.0, 'obs']
  y_miss = one_step.loc[one_step['missing'] == 1.0, 'preds']

  r2score = r2_score(obs, preds)

  plt.figure(figsize = (12, 16))
  plt.subplot(5, 1, 1)
  plt.scatter(x = obs, y = preds)
  plt.scatter(x_miss, y_miss, color='red', label='missing')
  plt.axline((0, 0), slope=1.0, color="grey")
  plt.xlabel('Observations')
  plt.ylabel('Predictions')
  plt.legend(loc='lower right')
  plt.annotate("$R^2$ = {:.3f}".format(r2score), (-9, 31))
  plt.title(title)
  plt.xlim((-10, 35))
  plt.ylim((-10, 35))
  plt.show()


def plot_one_step_diagnostics(model, data, val_series, val_pastcov_series, title,
  plot_feature_importances(model)

  # re-seasonalise observations
  if Y_COL == 'y_des':
    obs = data['y_des'] + data['y_yearly'] + data['y_daily'] + data['y_trend']
  elif Y_COL == 'y_des_fft':
    obs = data['y_des_fft'] + data['y_fft']
  else:
    obs = data[Y_COL]

  # print('data:', data.shape)
  # display(data[['y_des_fft', 'y_fft']])
  # print('obs:', obs.shape)
  # display(obs)
```

```
  if val_fut_cov is None:
    res = model.residuals(series = val_series,
                          past_covariates = val_pastcov_series,
                          retrain = False).pd_series()
  else:
    res = model.residuals(series = val_series,
                          past_covariates = val_pastcov_series,
                          future_covariates = val_fut_cov,
                          retrain = False).pd_series()

  preds = obs + res
  preds = preds.dropna()
  obs  = obs[preds.index]
  res  = res[preds.index]
  miss = data.loc[preds.index, 'missing']

  print_rmse_mae(obs, preds, '1st', '# ')

  one_step = pd.concat([obs, preds, res, miss], axis=1)
  one_step.columns = ['obs', 'preds', 'res', 'missing']

  title = 'step = 1 ' + title
  plot_one_step_obs_vs_preds(one_step, title)
  # plot_obs_vs_mean_preds_by_step(hist, title)
  plot_one_step_obs_preds_dists(one_step, title)
  plot_one_step_residuals(one_step, title + ' residuals')
  plot_one_step_residuals_dist(one_step, title + ' residuals density')
  plot_one_step_residuals_acf(one_step, title + ' residuals acf')
  plot_one_step_residuals_qq(one_step, title + ' residuals qq-plot')
  plot_one_step_abs_err_boxplot(one_step, title)


def plot_one_step_residuals_qq(one_step, title_):
  fig, axs = plt.subplots(figsize=(6, 6))
  sm.qqplot(one_step['res'], line='q', ax=axs)
  axs.set_title(title_)
  plt.show()


def plot_one_step_residuals_acf(one_step, title_, max_lags = 300):
  plt.figure(figsize = (6, 6))

  acf = pd.DataFrame()
  acf_feat = 'res'

  acf[acf_feat] = [one_step[acf_feat].autocorr(l) for l in range(1, max_lags)]
  plt.plot(acf[acf_feat], label='residual')

  plt.axhline(0, linestyle='--', c='black')
  plt.ylabel('autocorrelation')
  plt.xlabel('time lags')
  plt.title(title_)
  plt.show()
```

```
def _plot_xy_for_label(data, label, x_feat, y_feat, color):
    x = data.loc[data[label] == 1.0, x_feat]
    y = data.loc[data[label] == 1.0, y_feat]

    if len(x) > 0:
        plt.scatter(x = x, y = y, color=color, alpha=0.5, label=label)


def se_(obs, preds, metric):
    '''Standard error of sum of squared residuals or sum of absolute residuals'''

    if _check_obs_preds_lens_eq(obs, preds) == 0:
        stop()

    if metric == 'rmse':
        se = np.sqrt(np.sum((obs - preds) ** 2) / len(obs))
    elif metric == 'mae':
        se = np.sqrt(np.sum(np.abs(obs - preds)) / len(obs))
    else:
        print('Unrecognised metric:', metric)
        print("metric should be 'rmse' or 'mae'")
        stop()

    return se


def metric_ci_vals(test_val, se, z_val = 1.95996):
    cil = z_val * se
    # print('cil:', cil)
    metric_cil = test_val - cil
    metric_ciu = test_val + cil

    return metric_cil, metric_ciu


# TODO: Remove unused confidence intervals
# NOTE: VAR baseline metrics cvar_rmse and cvar_mae hardcoded to 48 steps
def plot_horizon_metrics(hist, title, y_col=Y_COL, horizon = HORIZON, ci=False):
    steps = [i for i in range(1, horizon+1)]

    # calculate metrics
    z_val_95 = 1.95996
    z_val_50 = 0.674
    rmse_h,   mae_h    = np.zeros(horizon), np.zeros(horizon)
    res_se_h, abs_se_h = np.zeros(horizon), np.zeros(horizon)
    rmse_ciu, rmse_cil = np.zeros(horizon), np.zeros(horizon)
    mae_ciu,  mae_cil  = np.zeros(horizon), np.zeros(horizon)

    for i in range(1, horizon+1):
      obs   = hist.loc[hist['step'] == i, y_col]
      preds = hist.loc[hist['step'] == i, 'pred']
      rmse_h[i-1] = rmse_(obs, preds)
      mae_h[i-1]  =  mae_(obs, preds)
```

```
  res_se_h[i-1] = se_(obs, preds, 'rmse')
  abs_se_h[i-1] = se_(obs, preds, 'mae')
  # mae_h[i]  = np.median(np.abs(obs - preds))  # for comparison with baseline
  rmse_cil[i-1], rmse_ciu[i-1] = metric_ci_vals(rmse_h[i-1], res_se_h[i-1], z_
  mae_cil[i-1],  mae_ciu[i-1]  = metric_ci_vals(mae_h[i-1],  abs_se_h[i-1], z_

# print('rmse_h:', rmse_h)
# print('mae_h:',  mae_h)

# plot metrics for horizons
fig, axs = plt.subplots(1, 2, figsize = (14, 7))
fig.suptitle(title + ' forecast horizon errors')
axs = axs.ravel()


mean_val_lab = title + ' mean value'
axs[0].plot(steps, rmse_h, color='green', label=title)

if ci is True:
  axs[0].fill_between(steps, rmse_cil, rmse_ciu, color='green', alpha=0.25)

# i - initial, u - updated, c - corrected
#ivar_rmse = np.array([0.39, 0.52, 0.64, 0.75, 0.86, 0.96, 1.06, 1.15, 1.23,
#                      1.31, 1.38, 1.45, 1.51, 1.57, 1.63, 1.68, 1.73, 1.77,
#                      1.81, 1.85, 1.89, 1.92, 1.96, 1.99, 2.02, 2.05, 2.08,
#                      2.1 , 2.13, 2.15, 2.18, 2.2 , 2.22, 2.24, 2.26, 2.28,
#                      2.3 , 2.31, 2.33, 2.35, 2.36, 2.38, 2.39, 2.4 , 2.42,
#                      2.43, 2.44, 2.45])
# NOTE: uvar_rmse tested on test_df
#uvar_rmse = np.array([0.36, 0.49, 0.6, 0.7, 0.8, 0.89, 0.98, 1.06, 1.14,
#                       1.21, 1.28, 1.35, 1.41, 1.47, 1.52, 1.57, 1.62, 1.66,
#                       1.7, 1.74, 1.78, 1.81, 1.84, 1.87, 1.9, 1.93, 1.96,
#                       1.99, 2.01, 2.03, 2.06, 2.08, 2.1, 2.12, 2.14, 2.16,
#                       2.18,  2.19, 2.21, 2.23, 2.24, 2.26, 2.27, 2.29, 2.3,
#                       2.31, 2.33, 2.34])
cvar_rmse = np.array([0.49318888, 0.70222546, 0.88570688, 1.05495349,
1.21081157, 1.34945832, 1.46844034, 1.57779714, 1.67754323, 1.7665827,
1.84567039, 1.91561743, 1.97899766, 2.03616174, 2.08661944, 2.13396441,
2.17809725, 2.21946156, 2.25780078, 2.29370568, 2.3272055,  2.35760153,
2.38520845, 2.41076185, 2.43404716, 2.45466806, 2.47361784, 2.49117761,
2.50625606, 2.52023589, 2.53319205, 2.54566125, 2.55764924, 2.56870554,
2.57976955, 2.59102429, 2.6018822, 2.61242356, 2.62280045, 2.63353767,
2.64410312, 2.65458709, 2.66532837, 2.67609086, 2.68675178, 2.69745108,
2.71002892, 2.72445726])
#axs[0].plot(steps, ivar_rmse, color='black', label='Initial VAR')
axs[0].plot(steps, cvar_rmse, color='blue', label='Updated VAR')
axs[0].hlines(np.mean(rmse_h), xmin=1, xmax=horizon,
              color='green', linestyles='dotted', label=mean_val_lab)
axs[0].hlines(np.mean(cvar_rmse), xmin=1, xmax=horizon,
              color='blue', linestyles='dotted', label='Updated VAR mean value
axs[0].set_xlabel("horizon - half hour steps")
axs[0].set_ylabel("rmse")
```

```
    axs[1].plot(steps, mae_h, color='green', label=title)

    if ci is True:
      axs[1].fill_between(steps, mae_cil, mae_ciu, color='green', alpha=0.25)

    # NOTE: ivar_mae tested on test_df
    #ivar_mae = np.array([0.39, 0.49, 0.57, 0.66, 0.74, 0.83, 0.91, 0.98, 1.05,
    #                     1.12, 1.18, 1.24, 1.29, 1.34, 1.39, 1.43, 1.47, 1.5 ,
    #                     1.53, 1.56, 1.59, 1.62, 1.64, 1.66, 1.68, 1.7 , 1.72,
    #                     1.73, 1.75, 1.76, 1.77, 1.78, 1.8 , 1.81, 1.82, 1.83,
    #                     1.83, 1.84, 1.85, 1.85, 1.86, 1.86, 1.87, 1.87, 1.88,
    #                     1.88, 1.89, 1.89])
    #uvar_mae = np.array([0.36, 0.45, 0.53, 0.61, 0.69, 0.76, 0.83, 0.9, 0.97,
    #                      1.03, 1.09, 1.14, 1.19, 1.24, 1.28, 1.32, 1.36, 1.4,
    #                      1.43, 1.46, 1.49, 1.52, 1.54, 1.56, 1.58, 1.6, 1.62,
    #                      1.63, 1.65, 1.66, 1.68, 1.69, 1.7, 1.71, 1.72, 1.73,
    #                      1.74, 1.74, 1.75, 1.75, 1.76, 1.76, 1.77, 1.77, 1.78,
    #                      1.78, 1.78, 1.78])
    cvar_mae = np.array([0.34694645, 0.50765333, 0.65132003, 0.78584432,
    0.9077075,  1.01705088, 1.11113622, 1.19759807, 1.27696634, 1.34941444,
    1.4134705,  1.47180058, 1.52304802, 1.56961154, 1.60903759, 1.64763418,
    1.68391297, 1.71690735, 1.74787094, 1.77721642, 1.80442554, 1.82951782,
    1.85358226, 1.87488643, 1.89346337, 1.91069565, 1.92613218, 1.94071845,
    1.95245349, 1.96323923, 1.9736734,  1.98370815, 1.99367508, 2.00204077,
    2.00992601, 2.01796976, 2.02747736, 2.03477489, 2.04173317, 2.04985428,
    2.05843847, 2.06731348, 2.07606609, 2.08533656, 2.09560914, 2.10668272,
    2.1183637,  2.13164371])
    #axs[1].plot(steps, ivar_mae, color='black', label='Initial VAR')
    axs[1].plot(steps, cvar_mae, color='blue', label='Updated VAR')
    axs[1].hlines(np.mean(mae_h), xmin=1, xmax=horizon,
                  color='green', linestyles='dotted', label=mean_val_lab)
    axs[1].hlines(np.mean(cvar_mae), xmin=1, xmax=horizon,
                  color='blue', linestyles='dotted', label='Updated VAR mean value
    axs[1].set_xlabel("horizon - half hour steps")
    axs[1].set_ylabel("mae")

    plt.legend(bbox_to_anchor=(1.04, 0.5), loc="center left", borderaxespad=0)
    plt.show()


def plot_horizon_metrics_boxplots(hist, title):

  hist['abs_err'] = np.abs(hist['res'])
  hist[['abs_err', 'step']].boxplot(by='step',
                                    meanline=False,
                                    showmeans=True,
                                    showcaps=True,
                                    showbox=True,
                                    showfliers=False,
                                    )
  plt.title(title + '\nboxplots with mean and median')
  plt.suptitle('')
  plt.xlabel("horizon - half hour steps")
  plt.ylabel("absolute error")
```

```
  x_step = 10.0
  x_max  = np.ceil(np.max(hist.step) / x_step) * int(x_step)
  plt.xticks(np.arange(0, x_max, int(x_step)))
  plt.show()


# TODO Refactor this
#      miss, preds, obs, res, err, dates etc "family" of variables
#      is a warning sign
#      try-catch around lagged_miss is clear indication of upsteam issues
#      Consider using a better data structure
#      See also: plot_forecast_examples immediately below
def _filter_out_missing(pos_neg_rmse_all, miss, lags, subplots):
    '''Check if obs (lags and horizon) missing == 1.0
    and
    Avoid contiguous indices'''

    # print("pos_neg_rmse_all:", pos_neg_rmse_all)

    pos_neg_rmse = pd.Series(subplots)
    subplot_count = j = 0

    while subplot_count < subplots:
      restart = False
      idx = pos_neg_rmse_all.index[j]
      # print(j, idx, pos_neg_rmse_all.loc[pos_neg_rmse_all.index[j]])

      # Avoid indices in the first few observations
      # Would be incomplete
      if idx < lags:
        # print('idx < lags:', idx)
        j += 1
        continue

      # Avoid contiguous indices - don't want 877, 878, 879
      if subplot_count > 0:
        for i in range(subplot_count):
          if abs(idx - pos_neg_rmse[i]) < lags:
            # print('contiguous indices - idx, pos_neg_rmse[i]:', idx, pos_neg_rms
            restart = True
            break

      if restart is False:
        try:
            lagged_miss = (miss.loc[idx - lags, :] == 1.0).any()
        except KeyError:
            lagged_miss = True

        horizon_miss = (miss.loc[idx, :] == 1.0).any()
        missing = lagged_miss or horizon_miss
        # print("\nlagged_miss:", lagged_miss)
        # print("horizon_miss:",  horizon_miss)
        # print("missing:", missing)
```

```
        if missing is False:
          pos_neg_rmse[subplot_count] = idx
          subplot_count += 1
        #else:
        #  print('missing')

      j += 1

    return pos_neg_rmse


def get_main_plot_title(pre_str, lag_params, mod_params):
  lag_params_str = ', '.join([f"{' '.join(map(str, v))}" for v in lag_params.items
  mod_params_str = ', '.join([f"{' '.join(map(str, v))}" for v in mod_params.items
  plot_title = pre_str + lag_params_str + '\n' + mod_params_str

  return plot_title


def drop_correlated_cols(dataset, threshold=0.95):
  '''Adapted from https://stackoverflow.com/a/44674459/100129'''

  col_corr = set()  # Set of all the names of deleted columns
  corr_matrix = dataset.corr(numeric_only=True).abs()

  for i in range(len(corr_matrix.columns)):
    for j in range(i):
      if (corr_matrix.iloc[i, j] >= threshold) and (corr_matrix.columns[j] not in
        colname = corr_matrix.columns[i]
        col_corr.add(colname)
        if colname in dataset.columns:
          del dataset[colname]

  return dataset


def get_feature_selection_scores(df, sel_cols, y_col=Y_COL, sort_col='f_test'):
  '''WARNING: These tests assume a linear model.  This may not be optimal.

     Don't draw any hasty conclusions from these scores.
  '''

  dfcols  = df.columns
  matches = ['_window_', '_zscore_', '_scaled_']
  feat_cols = [dfcol for dfcol in dfcols if any([x in dfcol for x in matches])]
  # feat_cols = [df_col for df_col in df_cols if '_window_' in df_col ]
  feat_cols.extend(sel_cols)

  cols = [*feat_cols, y_col]
  # df_nona = df.dropna()
  # df_nona = df.loc[:, cols].dropna()
  df_nona = df[cols].dropna()
  X_df = df_nona[feat_cols]
  y_df = df_nona[y_col]
```

```
  mi_feats = mutual_info_regression(X_df, y_df)
  mi_feats /= np.sum(mi_feats)

  f_tests, _ = f_regression(X_df, y_df)
  f_tests /= np.sum(f_tests)

  r_tests = r_regression(X_df, y_df)
  r_tests /= np.sum(r_tests)

  # Correlations with Y_COL
  # corrs = []
  # for feat in feat_cols:
  #   corrs.append(X_df[feat].corr(y_df))

  fs_df = pd.DataFrame({#'correlation': corrs,
                        'r_test': r_tests.round(6),
                        'f_test': f_tests.round(6),
                        'mi':     mi_feats,
                       })
  fs_df.index = feat_cols

  if sort_col == 'f_test':
    fs_df = fs_df.sort_values('f_test', ascending=False)
  elif sort_col == 'r_test':
    fs_df = fs_df.sort_values('r_test', ascending=False)
  elif sort_col == 'mi':
    fs_df = fs_df.sort_values('mi', ascending=False)

  return fs_df


def plot_observation_examples(df, cols, num_plots = 9):
    """Plot 9 sets of observations in 3 * 3 matrix"""

    num_plots_sqrt = int(np.sqrt(num_plots))
    assert num_plots_sqrt ** 2 == num_plots

    days = df.ds.dt.date.sample(n = num_plots).sort_values()
    p_data = [df[df.ds.dt.date.eq(days[i])] for i in range(num_plots)]

    fig, axs = plt.subplots(num_plots_sqrt, num_plots_sqrt, figsize = (15, 10))
    axs = axs.ravel()  # apl for the win :-)

    for i in range(num_plots):
        for col in cols:
            axs[i].plot(p_data[i]['ds'], p_data[i][col])
            axs[i].xaxis.set_tick_params(rotation = 20, labelsize = 10)

    fig.suptitle('Observation examples')
    fig.legend(cols, loc = 'lower center',  ncol = len(cols))

    return None
```

```
def sanity_check_df_rows_cols_labels(before, after,
                                     row_var_cutoff=0.005, col_var_cutoff=0.05,
                                     col_corr_cutoff=0.,
                                     fast=True, verbose=False):
  '''Sanity check dataframes before and after modifications

  WARN: default row_var_cutoff, col_var_cutoff, col_corr_cutoff are fairly arbitra
        there is some redundancy between these tests

  '''

  print_v = print if verbose else lambda *a, **k: None

  df = pd.DataFrame(columns = ['before', 'after', 'diff'])
  df_labels = []

  label = 'rows'
  # start_time = timeit.default_timer()
  i = 0
  df.loc[len(df), df.columns] = before.shape[i], after.shape[i], 0
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'cols'
  # start_time = timeit.default_timer()
  i = 1
  df.loc[len(df), df.columns] = before.shape[i], after.shape[i], 0
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'missing_rows'
  # start_time = timeit.default_timer()
  i = 0
  before_after = pd.merge(before, after, left_index=True, right_index=True, how='o
  missing_rows = before_after.loc[before_after['_merge'] == 'left_only', :]
  df.loc[len(df), df.columns] = 0, missing_rows.shape[i], 0
  if missing_rows.shape[i] > 0:
    print_v('\n', label, ':')
    print_v(missing_rows)
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'missing_cols'
  # start_time = timeit.default_timer()
  i = 1
  common_cols = before.columns.intersection(after.columns)
  missing_cols = before.shape[i] - len(common_cols)
  df.loc[len(df), df.columns] = 0, missing_cols, 0
  if missing_cols > 0:
    print_v('\n', label, ':')
    print_v(set(before.columns) - set(common_cols))
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))
```

```
label = 'total_nas'
# start_time = timeit.default_timer()
df.loc[len(df), df.columns] = before.isna().sum().sum(), \
                              after.isna().sum().sum(), 0
df_labels.append(label)
# print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'rows_with_nas'
# start_time = timeit.default_timer()
before_rows_nas = before.isnull().any(axis=1).sum()
after_rows_nas  = after.isnull().any(axis=1).sum()
df.loc[len(df), df.columns] = before_rows_nas, after_rows_nas, 0
if before_rows_nas != after_rows_nas:
  print_v('\n', label, ':')
  print_v(before[before.isnull().any(axis=1)])
  print_v(after[after.isnull().any(axis=1)])
df_labels.append(label)
# print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'cols_with_nas'
# start_time = timeit.default_timer()
before_cols_nas = before.isnull().any().sum()
after_cols_nas  = after.isnull().any().sum()
df.loc[len(df), df.columns] = before_cols_nas, after_cols_nas, 0
if before_cols_nas != after_cols_nas:
  print_v('\n', label, ':')
  print_v(before.isnull().any().index.values)
  print_v(after.isnull().any().index.values)
df_labels.append(label)
# print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'single_value_rows'
if not fast:
  # start_time = timeit.default_timer()
  before_single_value_rows = np.sum(before.nunique(axis=1) <= 1)
  after_single_value_rows  = np.sum(after.nunique(axis=1) <= 1)
  df.loc[len(df), df.columns] = before_single_value_rows, \
                                after_single_value_rows, 0
  if before_single_value_rows != after_single_value_rows:
    print_v('\n', label, ':')
    print_v(before[before.nunique(axis=1) <= 1])
    print_v(after[after.nunique(axis=1) <= 1])
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'single_value_cols'
# start_time = timeit.default_timer()
before_single_value_cols = np.sum(before.nunique() <= 1)
after_single_value_cols  = np.sum(after.nunique() <= 1)
df.loc[len(df), df.columns] = before_single_value_cols, \
                              after_single_value_cols, 0
if before_single_value_cols != after_single_value_cols:
  print_v('\n', label, ':')
```

```
  print_v(before.columns[before.nunique() <= 1].values)
  print_v(after.columns[after.nunique() <= 1].values)
df_labels.append(label)
# print('\t', label, round(timeit.default_timer() - start_time, 2))

# warnings.resetwarnings()

with warnings.catch_warnings():
  warnings.simplefilter('ignore')

  label = 'low_var_rows'
  # start_time = timeit.default_timer()
  before_low_var_rows = (before.select_dtypes(include=[np.number]).std(axis=1) <
  after_low_var_rows  = (after.select_dtypes(include=[np.number]).std(axis=1) <=
  df.loc[len(df), df.columns] = before_low_var_rows, after_low_var_rows, 0
  if before_low_var_rows != after_low_var_rows:
    print_v('\n', label, ':')
    print_v(before.select_dtypes(include=[np.number]).std(axis=1) <= row_var_cut
    print_v(after.select_dtypes(include=[np.number]).std(axis=1)  <= row_var_cut
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'low_var_cols'
  # start_time = timeit.default_timer()
  before_low_var_cols = (before.select_dtypes(include=[np.number]).std() <= col_
  after_low_var_cols  = (after.select_dtypes(include=[np.number]).std() <= col_v
  df.loc[len(df), df.columns] = before_low_var_cols, after_low_var_cols, 0
  if before_low_var_cols != after_low_var_cols:
    print_v('\n', label, ':')
    s = before.select_dtypes(include=[np.number]).std() <= col_var_cutoff
    t = after.select_dtypes(include=[np.number]).std()  <= col_var_cutoff
    print_v(s[s].index.values)
    print_v(t[t].index.values)
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'duplicate_rows'
# start_time = timeit.default_timer()
before_dup_rows = before.shape[0] - before.drop_duplicates().shape[0]
after_dup_rows  = after.shape[0]  - after.drop_duplicates().shape[0]
df.loc[len(df), df.columns] = before_dup_rows, after_dup_rows, 0
if before_dup_rows != after_dup_rows:
  print_v('\n', label, ':')
  print_v(before[before.duplicated(keep=False)])
  print_v(after[after.duplicated(keep=False)])
df_labels.append(label)
# print('\t', label, round(timeit.default_timer() - start_time, 2))

label = 'highly_correlated_cols'
# .copy() so we don't modify the original dataframe
if not fast:
  # start_time = timeit.default_timer()
  before_high_corr_cols = before.shape[1] - drop_correlated_cols(before.copy(),
  after_high_corr_cols  = after.shape[1]  - drop_correlated_cols(after.copy(), c
```

```
    df.loc[len(df), df.columns] = before_high_corr_cols, after_high_corr_cols, 0
    if before_high_corr_cols != after_high_corr_cols:
      print_v('\n', label, ':')
      print_v(set(before.columns) - set(drop_correlated_cols(before.copy(), col_c
      print_v(set(after.columns) - set(drop_correlated_cols(after.copy(), col_cor
    df_labels.append(label)
    # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'duplicate_index_labels'
  # start_time = timeit.default_timer()
  before_idx_labels = before.index.duplicated().sum()
  after_idx_labels  = after.index.duplicated().sum()
  df.loc[len(df), df.columns] = before_idx_labels, after_idx_labels, 0
  if before_idx_labels != after_idx_labels:
    print_v('\n', label, ':')
    print_v(before.index.duplicated())
    print_v(after.index.duplicated())
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  label = 'duplicate_col_labels'
  # start_time = timeit.default_timer()
  before_dup_col_labels = before.columns.duplicated().sum()
  after_dup_col_labels  = after.columns.duplicated().sum()
  df.loc[len(df), df.columns] = before_dup_col_labels, after_dup_col_labels, 0
  if before_dup_col_labels != after_dup_col_labels:
    print_v('\n', label, ':')
    print_v(before.columns.duplicated())
    print_v(after.columns.duplicated())
  df_labels.append(label)
  # print('\t', label, round(timeit.default_timer() - start_time, 2))

  # TODO Find renamed columns from before in after?


  df['diff'] = df['after'] - df['before']
  df.index = df_labels

  return df


def sanity_check_before_after_dfs(before_, after_, ds_name, fast=True, verbose=Fal

  print('\n', ds_name, sep='')

  # Reasons I HATE pandas number Inf a neverending series:
  # PerformanceWarning: DataFrame is highly fragmented.  This is usually the
  # result of calling `frame.insert` many times, which has poor performance.
  # Consider joining all columns at once using pd.concat(axis=1) instead. To
  # get a de-fragmented frame, use `newframe = frame.copy()`
  before = before_.copy(deep=True)
  after  = after_.copy(deep=True)

  # start_time = timeit.default_timer()
```

```
sanity_df = sanity_check_df_rows_cols_labels(before, after, fast=fast, verbose=v
# print('\t sanity_check_df_rows_cols_labels', round(timeit.default_timer() - st


# start_time = timeit.default_timer()
print('before.index.equals(after.index):', before.index.equals(after.index))

# check index freq is set and are equal
print('before.index.freq == after.index.freq:', before.index.freq == after.index
if verbose:
  print('before.index.freq:', before.index.freq)
  print('after.index.freq:',  after.index.freq)


# check if common column dtypes have changed
common_cols = before.columns.intersection(after.columns)
print('before[common_cols].dtypes == after[common_cols].dtypes:',
      (before[common_cols].dtypes == after[common_cols].dtypes).all())
if verbose:
  print('before[common_cols].dtypes:', before[common_cols].dtypes)
  print('after[common_cols].dtypes:',  after[common_cols].dtypes)

# check if describe() summaries are equal
print('before[common_cols].describe() == after[common_cols].describe():',
      (before[common_cols].describe() == after[common_cols].describe()).all().al
if verbose:
  print(before[common_cols].describe() == after[common_cols].describe())

# check after subsetted by before equals before
print('\nbefore[common_cols].equals(after[common_cols]):',
before[common_cols].dropna().drop_duplicates().equals(after[common_cols].dropna(
)
if verbose:
  print('before.isin(after):',
  before[common_cols].dropna().drop_duplicates().isin(after[common_cols].dropna(
  )
  print(before.dropna().drop_duplicates().isin(after.dropna().drop_duplicates())
  print(before.dropna().drop_duplicates().isin(after.dropna().drop_duplicates())


# Reasons I HATE pandas number Inf a neverending series:
# PerformanceWarning: DataFrame is highly fragmented.  This is usually the
# result of calling `frame.insert` many times, which has poor performance.
# Consider joining all columns at once using pd.concat(axis=1) instead. To
# get a de-fragmented frame, use `newframe = frame.copy()`
# calculate duplicate row counts then find mean duplicate count
# for each column and finally find mean of means aka redundancy
# warnings.resetwarnings()
with warnings.catch_warnings():
  warnings.simplefilter('ignore')
  before_red = before.dropna().groupby(before.select_dtypes(include=np.number).c
  after_red  = after.dropna().groupby(after.select_dtypes(include=np.number).col
  print('redundancy before > after:', before_red > after_red)
  print('mean before feature redundancy:', round(before_red, 3))
```

```
    print('mean after feature redundancy: ', round(after_red,  3))

  # Check all data is numeric, finite (but allow NAs) and reasonably shaped
  # If any problems then this will error out
  # Only checking 'after' dataframe
  # https://scikit-learn.org/stable/modules/generated/sklearn.utils.check_X_y.html
  if Y_COL in after.columns:
    _, _ = check_X_y(after.drop(columns=[Y_COL, 'ds']),
                     after[Y_COL],
                     y_numeric = True,
                     force_all_finite = 'allow-nan')

  print()
  # print('\t end sanity_check_before_after_dfs', round(timeit.default_timer() - s

  display(sanity_df)

  return sanity_df


def compare_train_valid_test_sanity_dfs(train_sanity, valid_sanity, test_sanity, e
  '''...'''

  if ex_labels is None:
    ex_labels = ['rows']

  train_sanity = train_sanity.loc[~train_sanity.index.isin(ex_labels)]
  valid_sanity = valid_sanity.loc[~valid_sanity.index.isin(ex_labels)]
  test_sanity  =  test_sanity.loc[~test_sanity.index.isin(ex_labels)]

  if not train_sanity.equals(valid_sanity):
    print('WARN: train_sanity != valid_sanity')
    display(pd.concat([train_sanity, valid_sanity]).drop_duplicates(keep=False))

  if not train_sanity.equals(test_sanity):
    print('WARN: train_sanity != test_sanity')
    display(pd.concat([train_sanity, test_sanity]).drop_duplicates(keep=False))

  if not test_sanity.equals(valid_sanity):
    print('WARN: test_sanity != valid_sanity')
    display(pd.concat([test_sanity, valid_sanity]).drop_duplicates(keep=False))

  return None


def sanity_check_train_valid_test(train_df, valid_df, test_df):

  # Check number of columns is equal
  if (train_df.shape[1] != valid_df.shape[1]) or \
     (train_df.shape[1] != test_df.shape[1])  or \
     (valid_df.shape[1] != test_df.shape[1]):
    print('ERROR: Inconsistent number of columns!')
    print('train_df.shape[1]:', train_df.shape[1])
    print('valid_df.shape[1]:', valid_df.shape[1])
```

```
  print('test_df.shape[1]:',  test_df.shape[1])


# Check column names are equal
if not (train_df.columns == valid_df.columns).all():
  print('ERROR: Inconsistent train_df, valid_df column names!')
  print('train_df.columns:', train_df.columns)
  print('valid_df.columns:', valid_df.columns)

if not (train_df.columns == test_df.columns).all():
  print('ERROR: Inconsistent train_df, test_df column names!')
  print('train_df.columns:', train_df.columns)
  print('test_df.columns:',  test_df.columns)

if not (valid_df.columns == test_df.columns).all():
  print('ERROR: Inconsistent valid_df, test_df column names!')
  print('valid_df.columns:', valid_df.columns)
  print('test_df.columns:',  test_df.columns)


# Check column dtypes are equal
if not (train_df.dtypes == valid_df.dtypes).all():
  print('ERROR: Inconsistent train_df, valid_df dtypes!')
  print('train_df.dtypes:', train_df.dtypes)
  print('valid_df.dtypes:', valid_df.dtypes)

if not (train_df.dtypes == test_df.dtypes).all():
  print('ERROR: Inconsistent train_df, test_df dtypes!')
  print('train_df.dtypes:', train_df.dtypes)
  print('test_df.dtypes:',  test_df.dtypes)

if not (valid_df.dtypes == test_df.dtypes).all():
  print('ERROR: Inconsistent valid_df, test_df dtypes!')
  print('valid_df.dtypes:', valid_df.dtypes)
  print('test_df.dtypes:',  test_df.dtypes)


# Check index freqs are equal
if train_df.index.freq != valid_df.index.freq:
  print('ERROR: Inconsistent train_df, valid_df index frequencies!')
  print('train_df.index.freq:', train_df.index.freq)
  print('valid_df.index.freq:', valid_df.index.freq)

if train_df.index.freq != test_df.index.freq:
  print('ERROR: Inconsistent train_df, test_df index frequencies!')
  print('train_df.index.freq:', train_df.index.freq)
  print('test_df.index.freq:',   test_df.index.freq)

if valid_df.index.freq != test_df.index.freq:
  print('ERROR: Inconsistent valid_df, test_df index frequencies!')
  print('valid_df.index.freq:', valid_df.index.freq)
  print('test_df.index.freq:',   test_df.index.freq)
```

```
# Verify dataframes are different!
if train_df.equals(valid_df):
  print('ERROR: train_df == valid_df!')

if train_df.equals(test_df):
  print('ERROR: train_df == test_df!')

if valid_df.equals(test_df):
  print('ERROR: valid_df == test_df!')


# Check no overlap between train_df.index and valid_df.index
# train_df.index strictly before valid_df.index and test_df.index
if max(train_df.index) >= min(valid_df.index):
  print('ERROR: Overlap between train_df, valid_df indices!')
  print('max(train_df.index):', max(train_df.index))
  print('min(valid_df.index):', max(valid_df.index))

# Check no overlap between train_df.index and test_df.index
# train_df.index strictly before valid_df.index and test_df.index
if max(train_df.index) >= min(test_df.index):
  print('ERROR: Overlap between train_df, test_df indices!')
  print('max(train_df.index):', max(train_df.index))
  print('min(test_df.index):',  max(test_df.index))


# Check no overlap between valid_df.index and test_df.index
# valid_df.index can be before or after test_df.index
if (max(valid_df.index) >= min(test_df.index)) and \
   (max(valid_df.index) <= max(test_df.index)):
  print('ERROR: Overlap between valid_df, test_df indices!')
  print('valid_df.index:', max(valid_df.index), '-', max(valid_df.index))
  print('test_df.index:',  max(test_df.index),  '-', max(test_df.index))

if (min(valid_df.index) >= min(test_df.index)) and \
   (min(valid_df.index) <= max(test_df.index)):
  print('ERROR: Overlap between valid_df, test_df indices!')
  print('valid_df.index:', max(valid_df.index), '-', max(valid_df.index))
  print('test_df.index:',  max(test_df.index),  '-', max(test_df.index))


# TODO: Consider enforcing a gap of 1 day to 1 week between
#       train_df.index and {valid_df,test_df}.index to avoid data leakage?


# Check train_df has more observations than valid_df and test_df
if valid_df.shape[0] > train_df.shape[0]:
  print('ERROR: valid_df more observations than train_df!')
  print('train_df observations:', train_df.shape[0])
  print('valid_df observations:', valid_df.shape[0])

if test_df.shape[0] > train_df.shape[0]:
  print('ERROR: test_df more observations than train_df!')
  print('train_df observations:', train_df.shape[0])
```

```
    print('test_df observations:',  test_df.shape[0])


  # Check valid_df and test_df have equal number of observations
  # valid_df and test_df may be different sizes but
  # large size difference may indicate an issue
  # TODO: Use calendar.isleap() to check if leap year
  if valid_df.shape[0] != test_df.shape[0]:
    print('WARN: Inconsistent number of valid_df, test_df rows.  Leap year?')


  # Check valid_df and test_df are each 1 year long
  YEAR_OBS_MIN = 48 * 365
  YEAR_OBS_MAX = 48 * 366
  if (valid_df.shape[0] < YEAR_OBS_MIN) or \
     (valid_df.shape[0] > YEAR_OBS_MAX):
    print('ERROR: valid_df should be 1 year long [',
          YEAR_OBS_MIN, ',', YEAR_OBS_MAX, ']!')
    print('valid_df observations:', valid_df.shape[0])

  if (test_df.shape[0] < YEAR_OBS_MIN) or \
     (test_df.shape[0] > YEAR_OBS_MAX):
    print('ERROR: test_df should be 1 year long [',
          YEAR_OBS_MIN, ',', YEAR_OBS_MAX, ']!')
    print('test_df observations:', test_df.shape[0])

  return None


def print_train_valid_test_shapes(df, train_df, valid_df, test_df):
  print("df shape: ",            df.shape)
  print("train shape:   ", train_df.shape)
  print("valid shape:   ", valid_df.shape)
  print("test shape:    ",  test_df.shape)

  return None


def plot_feature_history_single_df(data, var, missing=False):
    plt.figure(figsize = (12, 6))
    plt.scatter(data.index, data[var],
                label='train', color='black', s=3)
    if missing:
      label = 'missing'
      x_lab = data.loc[data[label] == 1.0, 'ds']
      y_lab = data.loc[data[label] == 1.0, var]
      plt.scatter(x_lab, y_lab, color='red', label=label, s=3)

    plt.title(var)
    plt.show()


def plot_feature_history(train, valid, test, var, missing=False):
    label = 'missing'
```

```
    plt.figure(figsize = (12, 6))
    plt.scatter(train.index, train[var],
                label='train', color='black', s=3)
    if missing:
      x_lab = train.loc[train[label] == 1.0, 'ds']
      y_lab = train.loc[train[label] == 1.0, var]
      plt.scatter(x_lab, y_lab, color='red', label=label, s=3)

    plt.scatter(valid.index, valid[var],
                label='valid', color='blue', s=3)
    if missing:
      x_lab = valid.loc[valid[label] == 1.0, 'ds']
      y_lab = valid.loc[valid[label] == 1.0, var]
      plt.scatter(x_lab, y_lab, color='red', label=label, s=3)

    plt.scatter(test.index,  test[var],
                label='test', color='purple', s=3)
    if missing:
      x_lab = test.loc[test[label] == 1.0, 'ds']
      y_lab = test.loc[test[label] == 1.0, var]
      plt.scatter(x_lab, y_lab, color='red', label=label, s=3)

    plt.title(var)
    #ax = plt.gca()
    #leg = ax.get_legend()
    #leg.legendHandles[0].set_color('black')
    #leg.legendHandles[1].set_color('red')
    #leg.legendHandles[2].set_color('blue')
    #leg.legendHandles[3].set_color('red')
    #leg.legendHandles[4].set_color('purple')
    #leg.legendHandles[5].set_color('red')
    #hl_dict = {handle.get_label(): handle for handle in leg.legendHandles}
    #hl_dict['train'].set_color('black')
    #hl_dict['valid'].set_color('blue')
    #hl_dict['test'].set_color('purple')
    #hl_dict[label].set_color('red')
    #plt.legend(['train', 'valid', 'test', label])
    plt.show()


def plot_feature_history_separately(train, valid, test, var):
    fig, axs = plt.subplots(1, 3, figsize = (14, 7))

    axs[0].plot(train.index, train[var])
    axs[0].set_title('train')

    axs[1].plot(valid.index, valid[var])
    axs[1].set_title('valid')
    axs[1].set_xticks(axs[1].get_xticks(), axs[1].get_xticklabels(), rotation=45,

    axs[2].plot(test.index,  test[var])
    axs[2].set_title('test')
    axs[2].set_xticks(axs[2].get_xticks(), axs[2].get_xticklabels(), rotation=45,
```

```
    fig.suptitle(var)
    plt.show()


def check_high_low_thresholds(df):
  '''Check main features from dataframe are within reasonable thresholds'''

  all_ok = True
  feats = ['y', 'dew.point', 'humidity', 'pressure',
           'wind.speed.mean', 'wind.speed.max']
  highs = [ 45,  25, 100, 1060, 35, 70]
  lows  = [-20, -20,   5,  950,  0,  0]

  thresh = pd.DataFrame({'feat': feats,
                         'high': highs,
                         'low':  lows,})
  thresh.index = feats

  for feat in feats:
    feat_high = thresh.loc[feat, 'high']
    feat_low  = thresh.loc[feat, 'low']

    if not df[feat].between(feat_low, feat_high).all():
      all_ok = False
      print('%15s [%3d, %3d] - % 7.3f, % 7.3f' %
            (feat, feat_low, feat_high,
            round(min(df[feat]), 3), round(max(df[feat]), 3)))

  # check if dew.point ever greater than temperature
  if df.loc[df['dew.point'] > df['y'], ['y', 'dew.point']].shape[0] != 0:
    all_ok = False
    print('dew.point > y:')
    display(df.loc[df['dew.point'] > df['y'], ['y', 'dew.point']])

  return None


def get_features_filename(feat_name, data_name, date_str, file_ext='.csv.xz'):
    return feat_name + data_name + date_str + file_ext


def merge_data_and_aggs(data, aggs):
  data = pd.concat((data, aggs), axis=1)
  # data = data.join(aggs)

  # data.set_index('ds', drop = False, inplace = True)
  data['ds'] = data.index
  data = data[~data.index.duplicated(keep = 'first')]
  data = data.asfreq(freq = '30min')

  # Reasons I HATE pandas number Inf a neverending series:
  # PerformanceWarning: DataFrame is highly fragmented.  This is usually the
  # result of calling `frame.insert` many times, which has poor performance.
```

```
  # Consider joining all columns at once using pd.concat(axis=1) instead. To
  # get a de-fragmented frame, use `newframe = frame.copy()`
  # data_ = data.copy()

  return data


def get_rolling_features(data, params):
  agg_func = params['agg_func']
  # aggs = pd.DataFrame(index=data.index)

  print('\ndataset:', params['dataset'])

  if params['regenerate']:
    if params['agg_func'].__name__ == 'get_bivariate_dists_kerns':
      aggs = agg_func(data,
                      params['feat_cols'],
                      params['aggs'],
                      scale   = params['scale'],
                      z_score = params['z_score'],
                      verbose = params['verbose'],)
    elif params['agg_func'].__name__ == 'get_rolling_tsfeatures':
      aggs = agg_func(data,
                      params['feat_cols'],
                      params['windows'],
                      params['aggs'],
                      params['shifts'],
                      params['verbose'])
    else:
      aggs = agg_func(data,
                      params['feat_cols'],
                      params['windows'],
                      params['aggs'],
                      params['verbose'])

    # save new aggs to file ...
    fn = get_features_filename(params['feat_name'],
                               params['dataset'],
                               params['date_str'])
    # aggs.to_csv(fn)
    # files.download(fn)
  else:
    print('load aggs from file ...')
    fn = get_features_filename(params['feat_name'],
                               params['dataset'],
                               params['date_str'])
    fn = 'data/' + fn
    # aggs = pd.read_csv(fn, parse_dates=['ds'], compression='xz')
    # aggs.set_index('ds', drop=False, inplace=True)
    ## aggs = aggs[~aggs.index.duplicated(keep='first')]
    # aggs = aggs.asfreq(freq='30min')

  # merge data and aggs ...
  data_plus_aggs = merge_data_and_aggs(data, aggs)
```

```
  print('before:', data.shape)
  print('after: ', data_plus_aggs.shape)
  display(data_plus_aggs)
  display(data_plus_aggs.describe())

  return data_plus_aggs


def finalise_rolling_features(df, train_df_aggs, valid_df_aggs, test_df_aggs):
  print_train_valid_test_shapes(df, train_df_aggs, valid_df_aggs, test_df_aggs)

  common_cols = train_df_aggs.columns.intersection(valid_df_aggs.columns).intersec
  train_df_aggs = train_df_aggs[common_cols].copy()
  valid_df_aggs = valid_df_aggs[common_cols].copy()
  test_df_aggs  =  test_df_aggs[common_cols].copy()

  print_train_valid_test_shapes(df, train_df_aggs, valid_df_aggs, test_df_aggs)
  sanity_check_train_valid_test(train_df_aggs, valid_df_aggs, test_df_aggs)

  return train_df_aggs, valid_df_aggs, test_df_aggs


def print_null_columns(df, df_name):
  print('\n', df_name, 'null columns:')
  display(df[df.columns[df.isnull().any()]].isnull().sum())


def print_na_locations(df):
  '''Print index row and column labels for NA in dataframe'''

  for index, row in df[df.isna().any(axis=1)].items():
    for col_name, row_item in row.items():
      if pd.isnull(df.loc[index, col_name]):
        print(index, col_name)

  return None


def save_data_and_download_files(data, ds, params):

  fn = get_features_filename(params['feat_name'],
                             ds,
                             params['date_str'],
                             file_ext = '.parquet',
                             # file_ext = '.feather',
                             # file_ext = '.hdf5',
                            )

  data.to_parquet(fn, index = False, compression = 'zstd')

  # data.reset_index(drop=True, inplace=True)
  # data.to_feather(fn,
  #                 # header = True,
```

```
  #                 # index = False,
  #                 # complevel = 9,
  #                 compression = 'zstd',
  #                 # encoding = 'utf-8')
  # data.to_csv(fn,
  #             header = True,
  #             index = False,
  #             # complevel = 9,
  #             compression = 'xz',
  #             encoding = 'utf-8')
  # data.to_hdf(fn,
  #             key = ds,
  #             complevel = 9,
  #             complib = 'blosc:zstd',
  #             mode = 'w')

  if params['save_and_download']:
    files.download(fn)
  elif params['save_to_gdrive']:
    gdrive_path = '/content/drive/MyDrive/data/CambridgeTemperatureNotebooks/featu
    subprocess.run(['cp', '-f', fn, gdrive_path])

  return None


def max_na_len(s):
    isna = s.isna()
    blocks = (~isna).cumsum()
    return isna.groupby(blocks).sum().max()


def remove_nas(data, fillna_limit=12, verbose=False):
  '''Fill short sequences of NAs or drop features with longer NA sequences'''

  na_feats = data.apply(max_na_len)
  na_feats = na_feats[na_feats > 0]

  if verbose:
    print('max NA seq len:')
    display(na_feats)

  for na_feat in na_feats.index:
    if na_feats[na_feat] <= fillna_limit:
      data[na_feat] = data[na_feat].interpolate(method='linear')
      if verbose:
        print('interpolate:', na_feat)
    else:
      data.drop(na_feat, axis=1, inplace=True)
      if verbose:
        print('drop:', na_feat)

  return data
```

```
def get_features(train, valid, test, params):
  params.update({'dataset': 'valid'})
  valid_feats = get_rolling_features(valid, params)
  valid_feats = valid_feats.loc[valid_feats.ds.dt.year == VALID_YEAR]
  valid_feats = remove_nas(valid_feats, verbose=True)

  # WARN: Temporarily disabling this to speed up feature engineering
  # test_feats = valid_feats
  params.update({'dataset': 'test'})
  test_feats = get_rolling_features(test, params)
  test_feats = test_feats.loc[test_feats.ds.dt.year == TEST_YEAR]
  test_feats = remove_nas(test_feats, verbose=True)

  params.update({'dataset': 'train'})
  train = train.loc[train.index.year >= 2016]
  train_feats = get_rolling_features(train, params)
  train_feats = train_feats.loc[train_feats.index >= '2016-01-12']
  train_feats = remove_nas(train_feats, verbose=True)

  # 2017 is an arbitrary selection
  sel_cols = ['pressure', 'humidity', 'dew.point_des', 'irradiance',
              'za_rad', 'azimuth']
  train_feats['year'] = train_feats['ds'].dt.year
  train_feats_2017 = train_feats.loc[train_feats.year == 2017, :]
  fs_df = get_feature_selection_scores(train_feats_2017, sel_cols)
  display(fs_df.head(40))

  # SLOW - approx 5 mins :-(
  # fs_df = get_feature_selection_scores(train_df_pair, sel_cols)
  # display(fs_df.head(40))

  # Make train, valid and test columns consistent
  print_train_valid_test_shapes(df, train, valid, test)
  train_feats, valid_feats, test_feats = finalise_rolling_features(df,
                                                                   train_feats,
                                                                   valid_feats,
                                                                   test_feats)

  sanity_check_train_valid_test(train_feats, valid_feats, test_feats)

  train_sanity = sanity_check_before_after_dfs(train, train_feats, 'train_df')
  valid_sanity = sanity_check_before_after_dfs(valid, valid_feats, 'valid_df')
  test_sanity  = sanity_check_before_after_dfs(test,  test_feats,  'test_df')

  compare_train_valid_test_sanity_dfs(train_sanity, valid_sanity, test_sanity)

  check_high_low_thresholds(train_feats)
  check_high_low_thresholds(valid_feats)
  check_high_low_thresholds(test_feats)

  save_data_and_download_files(train_feats, 'train', params)
  save_data_and_download_files(valid_feats, 'valid', params)
  save_data_and_download_files(test_feats,   'test', params)
```

```
  return train_feats, valid_feats, test_feats


def get_darts_series(data, data_params):
  series   = TimeSeries.from_dataframe(data, value_cols=data_params['y_col'])
  past_cov = TimeSeries.from_dataframe(data, value_cols=data_params['past_cov_cols

  if data_params['fut_cov_cols'] is not None:
    fut_cov = TimeSeries.from_dataframe(data, value_cols=data_params['fut_cov_cols
  else:
    fut_cov = None

  return series, past_cov, fut_cov


def plot_short_term_acf(data, acf_feats, acf_cols, title_,
                        mean_feat = False, max_lags = 300):
  plt.figure(figsize = (12, 6))

  acf = pd.DataFrame()

  for acf_feat, acf_col in zip(acf_feats, acf_cols):
    acf[acf_feat] = [data[acf_feat].autocorr(l) for l in range(1, max_lags)]
    plt.plot(acf[acf_feat], label=acf_feat, c=acf_col)

  if mean_feat:
    acf['mean_acf'] = acf.mean(axis=1)
    plt.plot(acf['mean_acf'], label='mean_acf', c='black')

  plt.axhline(0, linestyle='--', c='black')
  plt.axhline(0.875, linestyle=':', c='lightgrey')
  plt.ylabel('autocorrelation')
  plt.xlabel('time lags')
  plt.title(title_)
  plt.legend()
  plt.show()


def plot_long_term_acf(data, var, num_years=3):
  # WARN: Slow function :-(
  #       Results are more useful when displaying more years of data
  pd.plotting.autocorrelation_plot(data[var].head(17532 * num_years))
  plt.title(var)
  plt.show()


def add_transmit_heuristic_feature(data):
  '''Add atmospheric transmittance heuristic feature (tau)
  See Table 1 and Equation 10 from:
  Estimating Hourly Incoming Solar Radiation from Limited Meteorological Data
  Kurt Spokas, Frank Forcella
  Weed Science, Vol. 54, No. 1 (Jan. - Feb., 2006), pp. 182-189
  https://www.jstor.org/stable/4539375?seq=2
  '''
```

```
  data['tau'] = 0.7  # 1.0

  # No rainfall at delta_temperature > 10C
  data.loc[(data['rainfall'] == 0.0) & (data['y_window_48_min_max_diff'] > 10.0),

  # No rainfall today, but rainfall the previous day
  data.loc[(data['rain_prev_24_hours_binary'] == 0) & (data['rain_prev_48_hours_bi

  # Rainfall occuring on present day
  data.loc[data['rain_prev_24_hours_binary'] == 1, 'tau'] = 0.4

  # Rainfall today and also the previous day
  data.loc[(data['rain_prev_24_hours_binary'] == 1) & (data['rain_prev_48_hours_bi

  # Tau was modified if delta_temperature <= 10C (from Equation 10)
  data.loc[data['y_window_48_min_max_diff'] <= 10.0, 'tau'] = data['tau'] / (11.0

  return data


def convert_wind_to_xy(data, speed, bearing, var = ''):
  # Convert wind direction and speed to x and y vectors, so the model can more eas
  wd_rad = data[bearing] * np.pi / 180  # Convert to radians
  wv = data[speed + var]

  # Calculate the wind x and y components
  data[speed + var + '.x'] = wv * np.cos(wd_rad)
  data[speed + var + '.y'] = wv * np.sin(wd_rad)

  return data


def c_to_k(c):
    '''
    Convert temperature in Celsius to Kelvin
    '''

    if c is not None:
        k = c + 273.15
    else:
        k = None

    return k


def mbar_to_pa(mbar):
    '''
    Convert pressure in mBar to Pa
    '''
    return mbar * 100.0


def relative_humidity(dp, temperature):
```

```
    '''From https://carnotcycle.wordpress.com/2012/08/04/how-to-convert-relative-h
       Neither ah (absolute humidity) nor rh (relative humidity) proved useful
       for forecasting but they may have utility for imputation.
       See also: https://carnotcycle.wordpress.com/2017/08/01/compute-dewpoint-ter
                 https://carnotcycle.wordpress.com/tag/formula/
    '''
    rh = 100 * (np.exp((18.678 * dp) / (257.14 + dp)) / np.exp((18.678 * temperatu
    rh = 100.0 if rh > 100.0 else rh
    rh =  15.0 if rh <  15.0 else rh
    return rh


def absolute_humidity(humidity, temperature):
    '''Absolute humidity in g / m^3
    From https://carnotcycle.wordpress.com/2012/08/04/how-to-convert-relative-humi
    '''
    ah = 13.24715 * humidity * (np.exp((17.67 * temperature / (243.5 + temperature
    return ah


def mixing_ratio(humidity, temperature, pressure):
    '''Mixing ratio in g / Kg dry air
    From https://carnotcycle.wordpress.com/2020/06/01/how-to-calculate-mixing-rati
    '''
    mr = 6.112 * 6.2218 * humidity * np.exp(17.67 * temperature / (temperature + 2
          (pressure - 0.06112 * humidity * np.exp(17.67 * temperature / (temperatu
    return mr


def saturation_vapour_pressure(temperature):
    '''Saturation vapour pressure in KPa using Teten's equation
    See also https://en.wikipedia.org/wiki/Vapour_pressure_of_water#Approximation_
             Eqn 1 from https://cran.r-project.org/web/packages/humidity/vignettes
    '''
    # svp = 6.1078 * np.exp(21.8745584 * (temperature - 273.16) / (temperature - 3
    # temp = (1 / 273.15) - (1 / temperature)
    # svp = 6.11 * np.exp( 2.5e6 * temp / 461.52 )

    # Teten's eqn from https://en.wikipedia.org/wiki/Vapour_pressure_of_water#Appr
    svp = 0.61078 * np.exp( 17.27 * temperature / (temperature + 237.3) )
    return svp


def air_density(temperature, pressure, p_v):
    '''Air density in Kg / m^3
    For humid air, not dry air
    p_v actual vapour pressure Pa
    https://en.wikipedia.org/wiki/Density_of_air#Humid_air
    '''
    t_k    = c_to_k(temperature)
    p_pa   = mbar_to_pa(pressure)
    p_v_pa = mbar_to_pa(p_v)
    R_d = 287.058  # specific gas constant for dry air J/(kg·K)
    R_v = 461.495  # specific gas constant for water vapor J/(kg·K)
```

```
    p_d_pa = p_pa - p_v_pa  # partial pressure of dry air Pa
    rho = p_d_pa / (R_d * t_k) + p_v_pa / (R_v * t_k)

    return rho


def water_vapour_concentration(rh, p_s, p_a):
    '''Atmospheric water vapour concentration in ppmv
    ppmv is parts per million by volume which I beleive is equivalent to mole
    fraction - mmol mol^{-1} (micromole per mole).
    From: Equation 3 in Gregory J. McRae (1980)
          A Simple Procedure for Calculating Atmospheric Water Vapor Concentratior
          Journal of the Air Pollution Control Association, 30:4, 394-394,
          DOI:10.1080/00022470.1980.10464362
    '''
    ppmv = rh * 10 ** 4 * p_s / p_a
    return ppmv


def specific_humidity(pressure, vapour_pressure):
    '''Specific humidity
    Eqn 5 from https://cran.r-project.org/web/packages/humidity/vignettes/humidity
    '''
    q = 0.622 * vapour_pressure / (pressure - 0.378 * vapour_pressure)
    return q


def potential_temperature(temperature, pressure):
    '''Potential temperature in K
    https://en.wikipedia.org/wiki/Potential_temperature
    '''
    p_0 = 1000
    t_k = c_to_k(temperature)
    theta = t_k * (p_0 / pressure) ** 0.286
    return theta


def dew_point_approx(T, RH):
    '''https://carnotcycle.wordpress.com/2017/08/01/compute-dewpoint-temperature-f
    frac = 17.67 * T / (243.5 + T)
    numer = 243.5 * (np.log(RH / 100.0) + frac)
    denom = 17.67 - np.log(RH / 100.0) - frac
    dp_approx = numer / denom

    return dp_approx


def temperature_approx(TD, RH):
    '''https://earthscience.stackexchange.com/q/14899/18379'''
    frac = 17.625 * TD / (243.04 + TD)
    numer = 243.04 * (frac - np.log(RH / 100.0))
    denom = 17.625 + np.log(RH / 100.0) - frac
    temp_approx = numer / denom
```

```
    return temp_approx


def humidity_approx(TD, T):
    '''https://earthscience.stackexchange.com/q/16570/18379'''
    numer = 243.04 * 17.625 * (TD - T)
    denom = (243.04 + T) * (243.04 + TD)
    frac  = numer / denom

    return 100 * np.exp(frac)


def ground_heat_flux(surface_temp):
    '''
    Calculate ground heat flux

    This is an approximation based on last term in only equation in question 6
    Page 61 of Parameterization Schemes: Keys to Understanding Numerical
    Weather Prediction Models by David J. Stensrud.
    '''

    # "Constants"
    K = 11            # J m^-2 K^-1 s^-1 - Thermal diffusivity of air
    ground_temp = 10  # Celcius - strictly speaking will vary seasonally

    T_g = c_to_k(ground_temp)   # K - Ground reservoir temperature
    T_s = c_to_k(surface_temp)  # K - Surface air temperature

    Q_G = K * (T_s - T_g)

    return Q_G


def sinusoidal_arg(timestamp_s, denominator):
    return 2 * np.pi * timestamp_s / denominator


# Add daily spline-based time terms
def periodic_spline_transformer(period, n_splines=None, degree=3):
    if n_splines is None:
        n_splines = period

    n_knots = n_splines + 1  # periodic and include_bias is True

    return SplineTransformer(
        degree  = degree,
        n_knots = n_knots,
        knots   = np.linspace(0, period, n_knots).reshape(n_knots, 1),
        extrapolation = "periodic",
        include_bias  = True,
    )


def simple_daily_yearly_res_decomp(data, var):
```

```
  # NOTE: Potential data leak here
  #       Seasonality should be calculated on train data only

  df_des = data[[var, 'ds', 'secs_since_midnight', 'doy', 'secs_elapsed']].dropna(
  df_des.set_index('ds', drop=False, inplace=True)
  data.set_index('ds', drop=False, inplace=True)

  # df_des['secs_since_midnight'] = ((df_des['ds'] - df_des['ds'].dt.normalize())
  # df_des['doy'] = df_des['ds'].apply(lambda x: x.dayofyear - 1)
  # display(df_des.info())
  # df_des['secs_elapsed'] = df_des['secs_since_midnight'] + df_des['doy'] * DAY
  # df_des['y_det'] = df_des.y - df_des.y.mean()

  df_yearly = df_des[[var, 'doy']].groupby('doy').mean(var)
  df_yearly = df_yearly.rename(columns={var: var+'_yearly'})
  df_des = pd.merge(df_des, df_yearly, on='doy')
  df_des.set_index('ds', drop=False, inplace=True)
  # display(df_des)

  df_des[var+'_des_1'] = df_des[var] - df_des[var+'_yearly']

  df_daily = df_des[[var+'_des_1', 'secs_since_midnight']].groupby('secs_since_mid
  df_daily = df_daily.rename(columns={var+'_des_1': var+'_daily'})
  df_des = pd.merge(df_des, df_daily, on='secs_since_midnight')
  df_des.set_index('ds', drop=False, inplace=True)

  df_des[var+'_res'] = df[var] - df_des[var+'_yearly'] - df_des[var+'_daily']

  del df_des[var+'_des_1']
  df_des_cols = [var+'_yearly', var+'_daily', var+'_res']
  data = pd.merge(data, df_des[df_des_cols], right_index=True, left_index=True)
  # display(data)
  data.set_index('ds', drop=False, inplace=True)

  return data


def print_df_summary(df):
  print("Shape:")
  display(df.shape)

  total_nas = df.isna().sum().sum()
  rows_nas  = df.isnull().any(axis=1).sum()
  cols_nas  = df.isnull().any().sum()
  print('\nTotal NAs:', total_nas)
  print('Rows with NAs:', rows_nas)
  print('Cols with NAs:', cols_nas)

  print("\nInfo:")
  display(df.info())

  print("\nSummary stats:")
  display(df.describe())
```

```
  print("\nRaw data:")
  display(df)
  print("\n")


def plot_periodogram(data, feature, periods, xlim=None, ylim=None):
  x = data[feature].to_numpy()

  if xlim is None and ylim is None:
    plt.plot(x)
    plt.title(feature)
    plt.show()

  fs = 1
  f, Pxx_spec = signal.periodogram(x, fs, 'flattop', scaling='spectrum')

  plt.figure()

  for i in periods:
    plt.axvline(x=i/48, c='pink')

  plt.semilogy(f, np.sqrt(Pxx_spec))

  if ylim is not None:
    plt.ylim(ylim)
    plt.axhline(y=1e-1, c='gray')
  if xlim is not None:
    plt.xlim(xlim)

  plt.xlabel('frequency [Hz]')
  plt.ylabel('Linear spectrum [V RMS]')
  plt.title('Power spectrum (scipy.signal.periodogram) - ' + feature)
  plt.show()


def plot_periodograms(data, feat, periods, xlim=[0, 0.1], ylim=[1e-2, 1e1]):
  plot_periodogram(data, feat, periods)
  plot_periodogram(data, feat, periods, xlim=xlim, ylim=ylim)
```

Data Setup

Import ComLab Data

The measurements are relatively noisy and there are usually several hundred missing values every year; often across multiple variables. Observations have been extensively cleaned but may still have issues. Interpolation and missing value imputation have been used to fill all missing values. See the cleaning section in the Cambridge Temperature Model repository for details. Observations start in August 2008 and end in August 2023 and occur every 30 mins.

```
url_date_filex = "2023.08.08.csv"
if 'google.colab' in str(get_ipython()):
    data_url = "https://github.com/makeyourownmaker/CambridgeTemperatureNotebooks/
                url_date_filex + ".xz?raw=true"
else:
    data_url = "../data/CamMetCleanishMissAnnotated" + url_date_filex + '.xz'

df = pd.read_csv(data_url, parse_dates=['ds'], compression='xz')
df.set_index('ds', drop=False, inplace=True)
df = df[~df.index.duplicated(keep='first')]
df = df.asfreq(freq='30min')
df_orig = df.copy()

# Unusable - Mostly NAs
drop_cols = ['sunshine', 'ceil_hgt', 'visibility']
df.drop(drop_cols, axis=1, inplace=True)


# Data reformating - https://www.cl.cam.ac.uk/research/dtg/weather/weather-raw-for
df['rainfall'] /= 1000
for column in ['temperature', 'dew.point', 'wind.speed.mean', 'wind.speed.max']:
    df[column] /= 10

df['y'] = df['temperature']


df['rain_prev_6_hours']  = df['rainfall'].rolling(12, min_periods=1).sum()
df['rain_prev_12_hours'] = df['rainfall'].rolling(24, min_periods=1).sum()
df['rain_prev_24_hours'] = df['rainfall'].rolling(48, min_periods=1).sum()
df['rain_prev_48_hours'] = df['rainfall'].rolling(96, min_periods=1).sum()
df['rain_prev_24_hours_binary'] = (df['rain_prev_24_hours'] > 0.0) * 1
df['rain_prev_48_hours_binary'] = (df['rain_prev_48_hours'] > 0.0) * 1


# Faster than np.ptp - https://stackoverflow.com/a/40184053/100129
# df['y_window_48_min_max_diff'] required in add_transmit_heuristic_feature
df['y_window_48_min_max_diff'] = df['y'].rolling(48, min_periods=1).agg(['min', 'r
df = add_transmit_heuristic_feature(df)

# Deep copy avoids SettingWithCopyWarning
df = df.loc['2008-08-01 00:00:00':'2022-12-31 23:30:00', :].copy(deep=True)


# Remove extreme outliers
humidity_min = 5.00
df.loc[df['humidity'] < humidity_min, 'humidity'] = humidity_min

pressure_min = 950
pressure_max = 1060
df.loc[df['pressure'] < pressure_min, 'pressure'] = pressure_min
df.loc[df['pressure'] > pressure_max, 'pressure'] = pressure_max


# Remove obviously bad temperature spike in '2016-01-08 23:00:00':'2016-01-09 07:3
# display(df.loc['2016-01-08 21:00:00':'2016-01-09 12:00:00', ['y', 'missing']])
```

```
# display(df.loc['2016-01-08 23:00:00':'2016-01-09 07:30:00', 'y'])
df.loc['2016-01-08 23:00:00':'2016-01-09 07:30:00', 'y'] = np.linspace(3.7, 5.9, 1


df['pressure.log'] = np.log(df['pressure'])
df['wind.speed.mean.sqrt'] = np.sqrt(df['wind.speed.mean'])
df['wind.speed.max.sqrt']  = np.sqrt(df['wind.speed.max'])


# Convert wind direction and speed to x and y vectors, so the model can more easil
df = convert_wind_to_xy(df, 'wind.speed.mean', 'wind.bearing.mean')
df = convert_wind_to_xy(df, 'wind.speed.mean.sqrt', 'wind.bearing.mean')
df = convert_wind_to_xy(df, 'wind.speed.max', 'wind.bearing.mean')
df = convert_wind_to_xy(df, 'wind.speed.max.sqrt', 'wind.bearing.mean')


# df['rh'] = relative_humidity(df['dew.point'], df['y'])
df['ah'] = absolute_humidity(df['humidity'], df['y'])
df['mixing_ratio'] = mixing_ratio(df['humidity'], df['temperature'], df['pressure
df['svp'] = saturation_vapour_pressure(df['temperature'])

# actual water vapour pressure
df['vapour_pressure'] = saturation_vapour_pressure(df['dew.point'])

# vapour pressure deficit
#   https://en.wikipedia.org/wiki/Vapour-pressure_deficit
#   https://physics.stackexchange.com/a/4553/243807
#   TODO Check alt. vp_def calculation methods in above stackexchange question
df['vp_def'] = df['vapour_pressure'] - df['svp']

df['air_density'] = air_density(df['y'], df['pressure'], df['vapour_pressure'])
df['H2OC'] = water_vapour_concentration(df['humidity'], df['svp'], df['pressure'])
df['specific_humidity'] = specific_humidity(df['pressure'], df['vapour_pressure'])
df['t_pot'] = potential_temperature(df['y'], df['pressure'])

# df['dew.point_approx'] = dew_point_approx(df['y'], df['humidity'])
# df['y_approx'] = temperature_approx(df['dew.point'], df['humidity'])
# df['humidity_approx'] = humidity_approx(df['dew.point'], df['y'])

df['ground_hf'] = ground_heat_flux(df['y'])


# Convert to secs and add daily and yearly sinusoidal time terms
date_time   = pd.to_datetime(df['ds'], format = '%Y.%m.%d %H:%M:%S')
timestamp_s = date_time.map(datetime.datetime.timestamp)


# ps - phase shift
for i, ps in enumerate([0, np.pi], start=1):
    df['day.sin.'  + str(i)] = np.sin(sinusoidal_arg(timestamp_s, DAY)  + ps)
    df['day.cos.'  + str(i)] = np.cos(sinusoidal_arg(timestamp_s, DAY)  + ps)
    df['year.sin.' + str(i)] = np.sin(sinusoidal_arg(timestamp_s, YEAR) + ps)
    df['year.cos.' + str(i)] = np.cos(sinusoidal_arg(timestamp_s, YEAR) + ps)
```

```
hour_df = pd.DataFrame(
    np.linspace(0, DAY, DAILY_OBS + 1).reshape(-1, 1),
    columns=["secs"],
)

month_df = pd.DataFrame(
    np.linspace(0, YEAR, YEARLY_OBS + 1).reshape(-1, 1),
    columns=["secs"],
)

# 12 splines approximating 12 month-like time components
day_splines = periodic_spline_transformer(DAY, n_splines=12).fit_transform(hour_df
day_splines_df = pd.DataFrame(
    day_splines,
    columns=[f"day_spline_{i}" for i in range(day_splines.shape[1])],
)
day_splines_df['secs_since_midnight'] = range(0, DAY + DAY_SECS_STEP, DAY_SECS_STE


year_splines = periodic_spline_transformer(YEAR, n_splines=12).fit_transform(month
year_splines_df = pd.DataFrame(
    year_splines,
    columns=[f"year_spline_{i}" for i in range(year_splines.shape[1])],
)
year_splines_df['secs_elapsed'] = range(0, int(YEAR), DAY_SECS_STEP)



# Add seasonal mean temperature (y_seasonal), humidity and dew.point
df['secs_since_midnight'] = ((df['ds'] - df['ds'].dt.normalize()) / pd.Timedelta(
df['doy'] = df['ds'].apply(lambda x: x.dayofyear - 1)
df['secs_elapsed'] = df['secs_since_midnight'] + df['doy'] * DAY

# NOTE: Potential data leak here
#       Seasonality should be calculated on train data only
#for var in ['y', 'humidity', 'dew.point', 'pressure', 'wind.speed.mean']:
#  df_seasonal_var = df[[var, 'secs_elapsed']].groupby('secs_elapsed').mean(var)
#  df_seasonal_var.rename(columns={var: var + '_seasonal'}, inplace=True)
#  df = pd.merge(df, df_seasonal_var,   on='secs_elapsed')
#  df[var + '_des'] = df[var] - df[var + '_seasonal']  # des - deseasonal

#df = pd.merge(df, day_splines_df,  on='secs_since_midnight')
#df = pd.merge(df, year_splines_df, on='secs_elapsed')

# for var in ['y', 'humidity', 'dew.point', 'pressure', 'wind.speed.mean']:
#   df = simple_daily_yearly_res_decomp(df, var)
#   # display(df.info())


# TODO: Potential data leaks here?
# df['y_diff_1']  = df['y'].diff(1)
# df['dew.point_diff_1']  = df['dew.point'].diff(1)
# df['humidity_diff_1']   = df['humidity'].diff(1)
```

```
# df['pressure_diff_1'] = df['pressure'].diff(1)  # unusable

# TODO: Investiage using np.gradient() here
df['dT_dH'] = df['y'].diff(1) / df['humidity'].diff(1)
df['dT_dP'] = df['y'].diff(1) / df['pressure'].diff(1)
df['dT_dTdp'] = df['y'].diff(1) / df['dew.point'].diff(1)

# WARNING WARNING Danger Will Robinson! Definite data leak here :-(
# inf introduced due to diff(1) == 0 when no change
df = df.replace([np.inf, -np.inf], np.nan)
df['dT_dH']   = df['dT_dH'].interpolate(method='linear')
df['dT_dP']   = df['dT_dP'].interpolate(method='linear')
df['dT_dTdp'] = df['dT_dTdp'].interpolate(method='linear')

for col in ['y', 'dew.point', 'humidity', 'pressure']:
  df[col+'_grad'] = np.gradient(df[col])


df.set_index('ds', drop=False, inplace=True)
df = df.asfreq(freq='30min')


# Reorder and drop temporary calculation columns
inc_cols = ['ds', 'y', #'y_daily', 'y_yearly', 'y_res',
            'humidity', #'humidity_daily', 'humidity_yearly', 'humidity_res',
            'dew.point', #'dew.point_daily', 'dew.point_yearly', 'dew.point_res',
            'pressure', #'pressure_daily', 'pressure_yearly', 'pressure_res',
            'pressure.log', 'y_window_48_min_max_diff',
           #'humidity_diff_1',
           #'humidity_diff_48', 'pressure_diff_48', 'humidity_diff_1_48',
           #'wind.speed.mean_daily', 'wind.speed.mean_yearly', 'wind.speed.mean_re
            'wind.speed.mean.sqrt', 'wind.speed.mean',
            'wind.bearing.mean', 'wind.speed.mean.x', 'wind.speed.mean.y',
            'wind.speed.mean.sqrt.x', 'wind.speed.mean.sqrt.y',
            'wind.speed.max', 'wind.speed.max.sqrt',
            'wind.speed.max.sqrt.x', 'wind.speed.max.sqrt.y',
            'ground_hf', 'rainfall',
            'tau', 'rain_prev_6_hours', 'rain_prev_12_hours',
            'rain_prev_24_hours', 'rain_prev_24_hours_binary',
            'rain_prev_48_hours', 'rain_prev_48_hours_binary',
            'mixing_ratio', 'ah', 'specific_humidity', 'svp', 'vapour_pressure',
            'vp_def', 't_pot', 'air_density', 'H2OC',
            'dT_dH', 'dT_dP', 'dT_dTdp',
            'y_grad', 'dew.point_grad', 'humidity_grad', 'pressure_grad',
           #'y_approx', 'humidity_approx', 'dew.point_approx',
           #'y_seasonal', 'y_des',
           #'humidity_seasonal', 'humidity_des',
           #'dew.point_seasonal', 'dew.point_des',
           #'pressure_seasonal', 'pressure_des',
           #'wind.speed.mean_seasonal', 'wind.speed.mean_des',
            'day.sin.1', 'day.cos.1', 'year.sin.1', 'year.cos.1',
           #'y_shadow', 'humidity_shadow', 'pressure_shadow', 'dew.point_shadow',
            'missing', 'known_inaccuracy', 'isd_outlier', 'long_run', 'spike',
            'cooksd_out', 'isd_3_sigma', 'isd_filled', 'tsclean_filled',
```

```
            'tsclean_filled_temperature', 'tsclean_filled_dew.point',
            'tsclean_filled_humidity', 'tsclean_filled_pressure',
            'tsclean_filled_wind.speed.mean', 'tsclean_filled_wind.speed.max',
            'tsclean_filled_wind.bearing.mean', 'tsclean_filled_rainfall',
           #'hist_average', 'mi_filled', 'mi_spike_interp', 'lin_interp'
           ]
df = df[inc_cols]

# df = df.loc[df['missing'] == 0.0, :]
# df = df.loc[(df['mi_filled'] != 1.0) & (df['hist_average'] != 1.0), :]



# For use in other notebooks
if not 'google.colab' in str(get_ipython()):
    data_loc = "../data/CamMetPrepped" + url_date_filex
    df.to_csv(data_loc)


df = df.fillna(method='bfill')  # Small number of NAs in first row
print_df_summary(df)


plot_cols = ['y', 'humidity', 'dew.point', 'wind.speed.mean.x',
             'wind.speed.mean.y']  # 'pressure',
plot_observation_examples(df, plot_cols)


df_sanity = sanity_check_before_after_dfs(df_orig, df, 'df')
check_high_low_thresholds(df)
```

```
Shape:
(252768, 64)

Total NAs: 0
Rows with NAs: 0
Cols with NAs: 0

Info:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 252768 entries, 2008-08-01 00:00:00 to 2022-12-31 23:30:00
Freq: 30T
Data columns (total 64 columns):
 #   Column                              Non-Null Count   Dtype
---  ------                              --------------   -----
 0   ds                                  252768 non-null  datetime64[ns]
 1   y                                   252768 non-null  float64
 2   humidity                            252768 non-null  float64
 3   dew.point                           252768 non-null  float64
 4   pressure                            252768 non-null  float64
 5   pressure.log                        252768 non-null  float64
 6   y_window_48_min_max_diff            252768 non-null  float64
 7   wind.speed.mean.sqrt                252768 non-null  float64
 8   wind.speed.mean                     252768 non-null  float64
 9   wind.bearing.mean                   252768 non-null  float64
 10  wind.speed.mean.x                   252768 non-null  float64
 11  wind.speed.mean.y                   252768 non-null  float64
 12  wind.speed.mean.sqrt.x              252768 non-null  float64
 13  wind.speed.mean.sqrt.y              252768 non-null  float64
 14  wind.speed.max                      252768 non-null  float64
 15  wind.speed.max.sqrt                 252768 non-null  float64
 16  wind.speed.max.sqrt.x               252768 non-null  float64
 17  wind.speed.max.sqrt.y               252768 non-null  float64
 18  ground_hf                           252768 non-null  float64
 19  rainfall                            252768 non-null  float64
 20  tau                                 252768 non-null  float64
 21  rain_prev_6_hours                   252768 non-null  float64
 22  rain_prev_12_hours                  252768 non-null  float64
 23  rain_prev_24_hours                  252768 non-null  float64
 24  rain_prev_24_hours_binary           252768 non-null  int64
 25  rain_prev_48_hours                  252768 non-null  float64
 26  rain_prev_48_hours_binary           252768 non-null  int64
 27  mixing_ratio                        252768 non-null  float64
 28  ah                                  252768 non-null  float64
 29  specific_humidity                   252768 non-null  float64
 30  svp                                 252768 non-null  float64
 31  vapour_pressure                     252768 non-null  float64
 32  vp_def                              252768 non-null  float64
 33  t_pot                               252768 non-null  float64
 34  air_density                         252768 non-null  float64
 35  H2OC                                252768 non-null  float64
 36  dT_dH                               252768 non-null  float64
 37  dT_dP                               252768 non-null  float64
 38  dT_dTdp                             252768 non-null  float64
 39  y_grad                              252768 non-null  float64
 40  dew.point_grad                      252768 non-null  float64
 41  humidity_grad                       252768 non-null  float64
 42  pressure_grad                       252768 non-null  float64
 43  day.sin.1                           252768 non-null  float64
 44  day.cos.1                           252768 non-null  float64
 45  year.sin.1                          252768 non-null  float64
```

```
 46  year.cos.1                          252768 non-null  float64
 47  missing                             252768 non-null  int64
 48  known_inaccuracy                    252768 non-null  int64
 49  isd_outlier                         252768 non-null  int64
 50  long_run                            252768 non-null  int64
 51  spike                               252768 non-null  int64
 52  cooksd_out                          252768 non-null  int64
 53  isd_3_sigma                         252768 non-null  int64
 54  isd_filled                          252768 non-null  int64
 55  tsclean_filled                      252768 non-null  int64
 56  tsclean_filled_temperature          252768 non-null  int64
 57  tsclean_filled_dew.point            252768 non-null  int64
 58  tsclean_filled_humidity             252768 non-null  int64
 59  tsclean_filled_pressure             252768 non-null  int64
 60  tsclean_filled_wind.speed.mean      252768 non-null  int64
 61  tsclean_filled_wind.speed.max       252768 non-null  int64
 62  tsclean_filled_wind.bearing.mean    252768 non-null  int64
 63  tsclean_filled_rainfall             252768 non-null  int64
dtypes: datetime64[ns](1), float64(44), int64(19)
memory usage: 125.4 MB
None
```

Summary stats:

	ds	y	humidity	dew.point	pressure	p
count	252768	252768.000000	252768.000000	252768.000000	252768.000000	2
mean	2015-10-16 23:45:00.000001024	10.109653	78.171399	5.842545	1014.440231	
min	2008-08-01 00:00:00	-7.144690	5.000000	-11.164754	950.000000	
25%	2012-03-09 11:52:30	5.200000	68.000000	1.900000	1008.000000	
50%	2015-10-16 23:45:00	9.600000	82.000000	5.900000	1016.000000	
75%	2019-05-25 11:37:30	14.500000	92.000000	9.700000	1023.000000	
max	2022-12-31 23:30:00	37.200000	100.000000	20.900000	1060.000000	
std	NaN	6.516515	17.554887	5.154348	11.916806	

8 rows × 64 columns

Raw data:

	ds	y	humidity	dew.point	pressure	pressure.log	y_window_4
ds							
2008-08-01 00:00:00	2008-08-01 00:00:00	20.0	89.221997	1.610001	1009.926805	6.917633	
2008-08-01 00:30:00	2008-08-01 00:30:00	19.5	88.290807	1.657847	1009.931495	6.917638	

2008-08-01 01:00:00	2008-08-01 01:00:00	19.1	85.755790	1.432873	1010.092861	6.917798
2008-08-01 01:30:00	2008-08-01 01:30:00	19.1	85.572379	1.397409	1009.902342	6.917609
2008-08-01 02:00:00	2008-08-01 02:00:00	19.1	84.345642	1.278219	1010.133476	6.917838
...	...	...	...	...	...	...
2022-12-31 21:30:00	2022-12-31 21:30:00	9.6	80.000000	6.300000	998.000000	6.905753
2022-12-31 22:00:00	2022-12-31 22:00:00	10.0	88.000000	8.100000	998.000000	6.905753
2022-12-31 22:30:00	2022-12-31 22:30:00	10.0	80.000000	6.700000	999.000000	6.906755
2022-12-31 23:00:00	2022-12-31 23:00:00	9.6	79.000000	6.100000	1000.000000	6.907755
2022-12-31 23:30:00	2022-12-31 23:30:00	9.2	82.000000	6.300000	1000.000000	6.907755

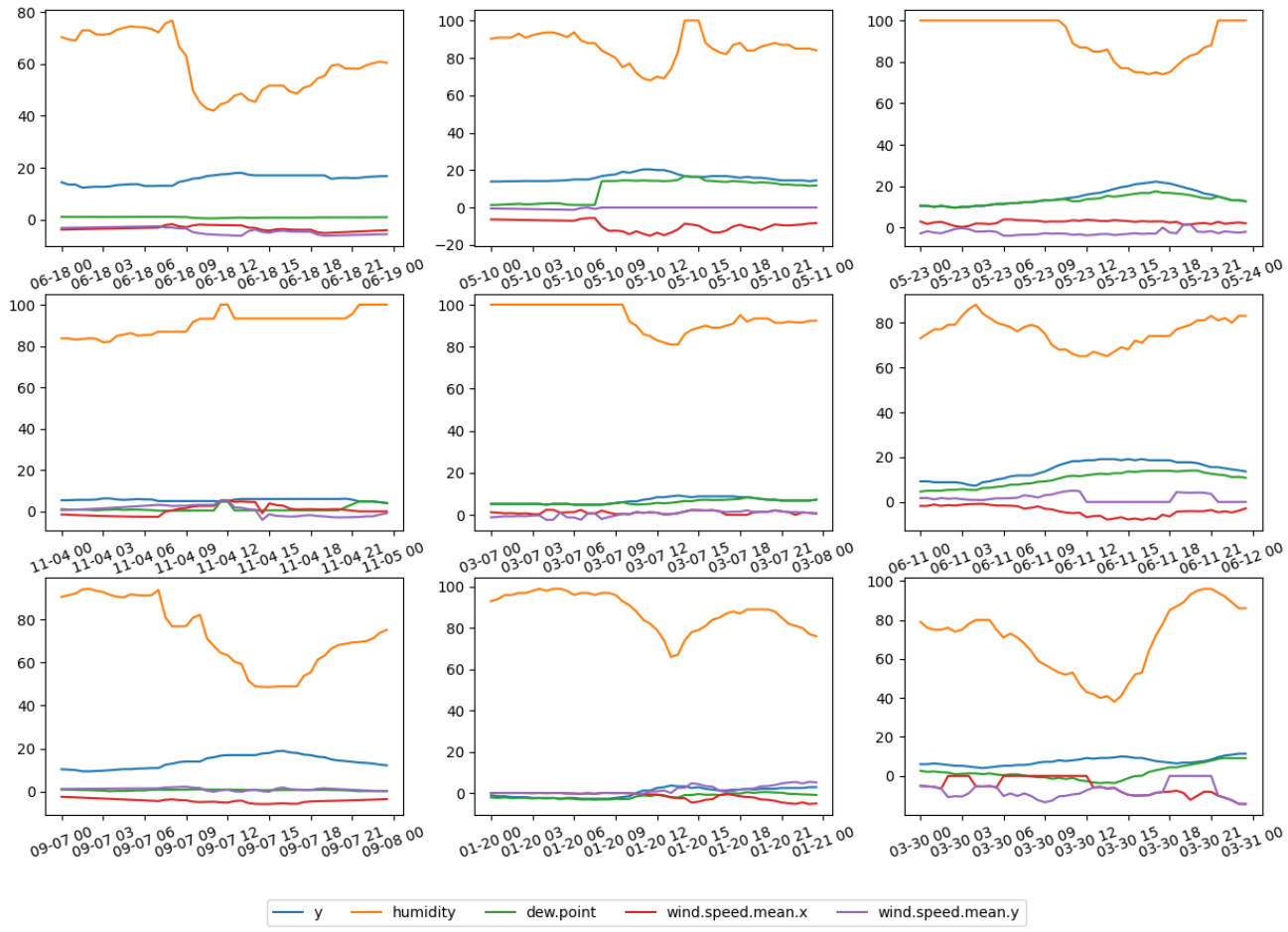
252768 rows × 64 columns

```
df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 616.985
mean after feature redundancy:  40.862
```

	before	after	diff
rows	263284	252768	-10516
cols	29	64	35
missing_rows	0	10516	10516
missing_cols	0	4	4
total_nas	438360	0	-438360
rows_with_nas	227103	0	-227103

cols_with_nas	3	0	-3
single_value_cols	0	0	0
low_var_rows	0	0	0
low_var_cols	3	6	3
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

Observation examples



I didn't include `pressure` in example observation plots because those values are an order of magnitude higher than the other features.

A few other things to note:

- features which did not prove useful this time
    - absolute humidity
    - mixing ratio
    - spline-based time components
    - higher frequency sinusoidal time components
    - phase-shifted sinusoidal time components
    - I leave the code for generating these features incase it is useful later
        - for example with the inclusion of the rainfall feature

---

Calculate Solar Features

Irradiance, zenith angle, azimuth and declination:

Solar irradiance is the power per unit area (surface power density) received from the Sun. Irradiance plays a part in weather forecasting. I calculate solar irradiance for Cambridge using the python solarpy module. I suspect forecasts could be substantially improved if solar irradiance could be combined with a measure of cloud cover. It can also be used as a future covariate with models built with the darts package.

The solar zenith angle is the angle between the sun's rays and the vertical direction. I use the pysolar python package for zenith angle calculations. Similarly to irradiance, it can be used as a future covariate. See this stackoverflow question and the source code for calculation details. There is a refraction correction term which assumes 'standard' pressure and temperature values. Zenith angle is used to calculate solar irradiance.

Solar azimuth angle is the angle between the projection of sun rays and a line due south or north. Again, I use the pysolar python package for azimuth angle calculations. Similarly to irradiance and zenith angle, it can be used as a future covariate.

Solar declination is the angle between the equator and a line drawn from the centre of the Earth to the centre of the sun. It can be calculated with the solarpy module. It did not prove useful in the lightgbm models. I've commented it out for now.

Calculations:

- just calculate irradiance etc for a single year (arbitrarily 2020)
- then repeat these values for the other years

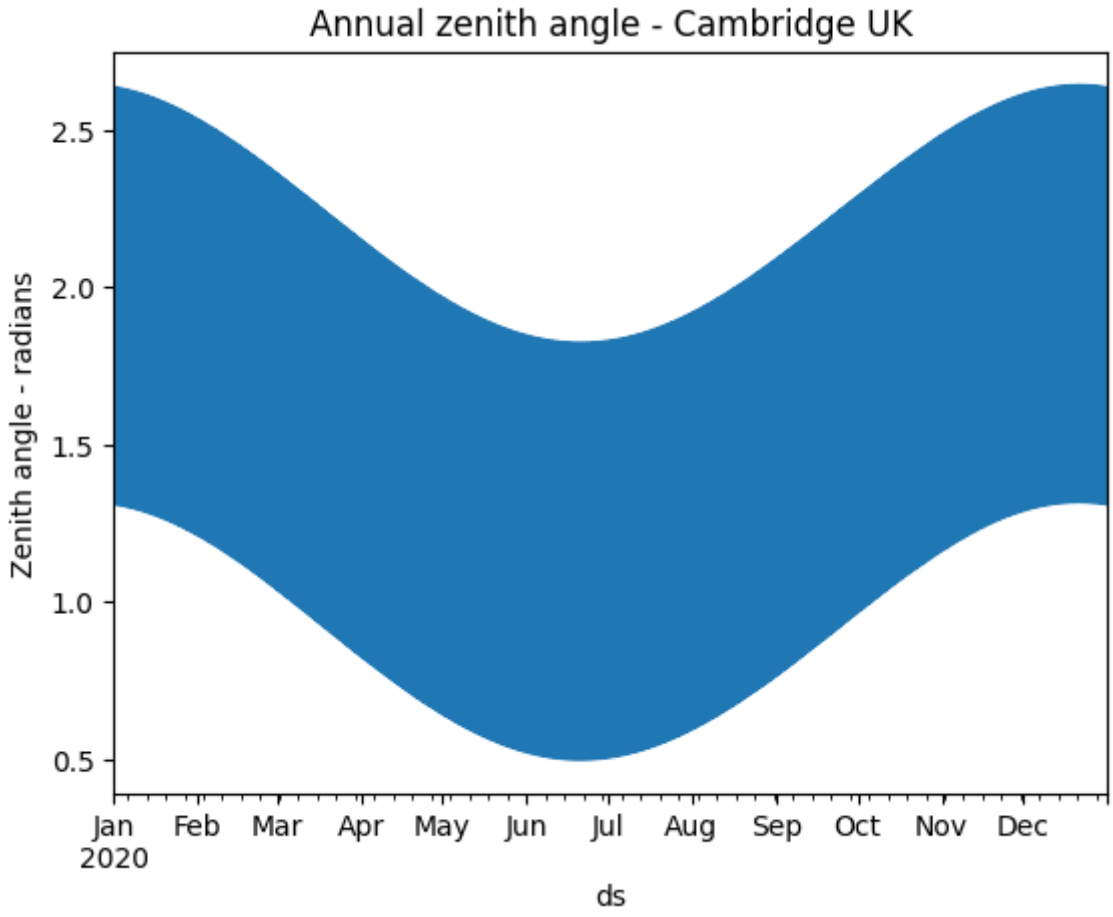
```
required  = {'pysolar', 'solarpy'}
installed = {pkg.key for pkg in pkg_resources.working_set}
missing   = required - installed

if missing == required:
    print('Installing', missing, '...', sep=' ', end='')
    python = sys.executable
    subprocess.check_call([python, '-m', 'pip', 'install', *missing])  #, stdout=s
    print(' Done')


from pysolar.solar import get_altitude, get_azimuth
from solarpy import irradiance_on_plane, declination


def calc_solar_data(df, solar_calc):
  IYEAR = 2020  # arbitrary year
  LAT   = 52.210922
  LON   =  0.091964

  df['year'] = df['ds'].dt.year
  data = pd.DataFrame()
  data.index = df.loc[df['year'] == IYEAR, 'ds']
  data.index = pd.to_datetime(data.index)

  print(solar_calc.title())

  if solar_calc == 'declination':
    declinations = [0] * len(data)
    i = 0

    for d in tqdm(data.index, desc='Calculating ' + solar_calc):
      declinations[i] = declination(d)
      i += 1

    data['declination'] = declinations
    display(data['declination'].describe())
  elif solar_calc == 'irradiance':
    HEIGHT = 6  # height above sea level
    VNORM  = np.array([0, 0, -1])  # plane pointing zenith
    irradiances  = [0] * len(data)
    i = 0

    for d in tqdm(data.index, desc='Calculating ' + solar_calc):
```

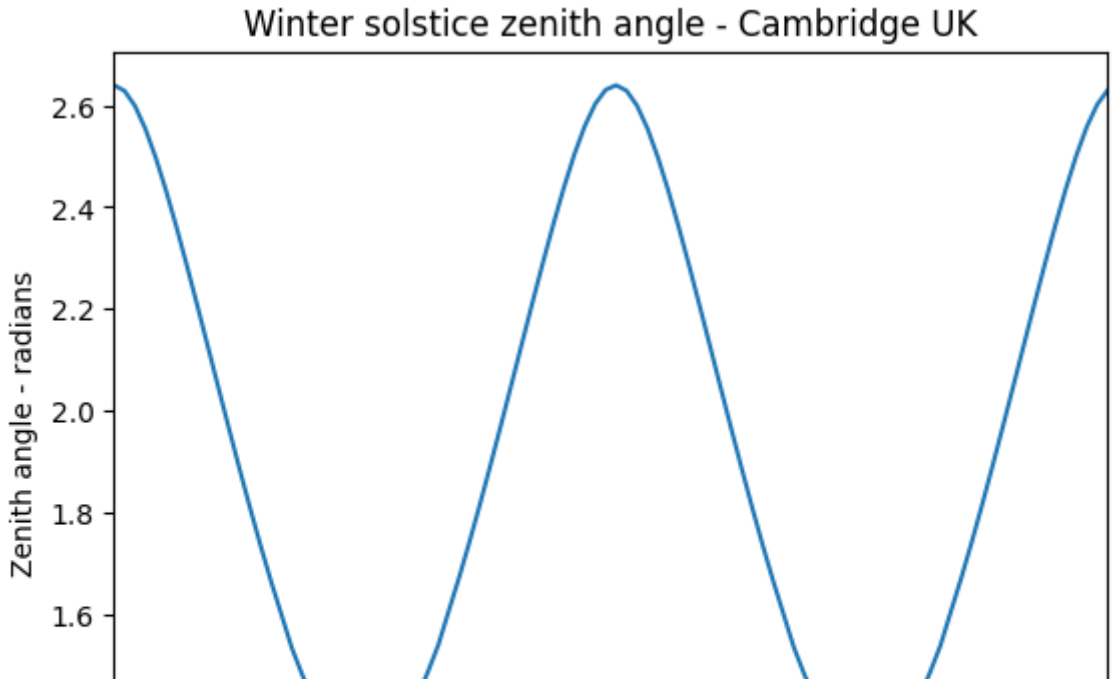
```
    irradiances[i] = irradiance_on_plane(VNORM, HEIGHT, d, LAT)
    i += 1

  data['irradiance'] = irradiances
  display(data['irradiance'].describe())
elif solar_calc == 'zenith':
  za = [0] * len(data)
  za_rad = [0] * len(data)
  i = 0

  for d in tqdm(data.index, desc='Calculating ' + solar_calc):
    ts = pd.Timestamp(d, tz='UTC').to_pydatetime()
    za[i] = float(90) - get_altitude(LAT, LON, ts)
    za_rad[i] = np.radians(za[i])
    i += 1

  # TODO rename za to zenith
  data['za'] = za
  data['za_rad'] = za_rad
  display(data['za'].describe())
elif solar_calc == 'azimuth':
  az = [0] * len(data)
  az_rad = az
  az_rad_cos = az
  az_rad_sin = az
  i = 0

  for d in tqdm(data.index, desc='Calculating ' + solar_calc):
    ts = pd.Timestamp(d, tz='UTC').to_pydatetime()
    az[i] = get_azimuth(LAT, LON, ts)
    az_rad[i] = np.radians(az[i])
    az_rad_cos[i] = np.cos(az_rad[i])
    az_rad_sin[i] = np.sin(az_rad[i])
    i += 1

  data['azimuth'] = az
  data['azimuth_rad'] = az_rad
  data['azimuth_cos'] = az_rad_cos
  data['azimuth_sin'] = az_rad_sin
  display(data['azimuth'].describe())
else:
  print("Unknown solar_calc parameter:", solar_calc,
        "\nUse 'declination', 'irradiance' or 'zenith'")

print()
data['month']  = data.index.month
data['day']    = data.index.day
data['hour']   = data.index.hour
data['minute'] = data.index.minute

df['month']  = df.index.month
df['day']    = df.index.day
df['hour']   = df.index.hour
df['minute'] = df.index.minute
```

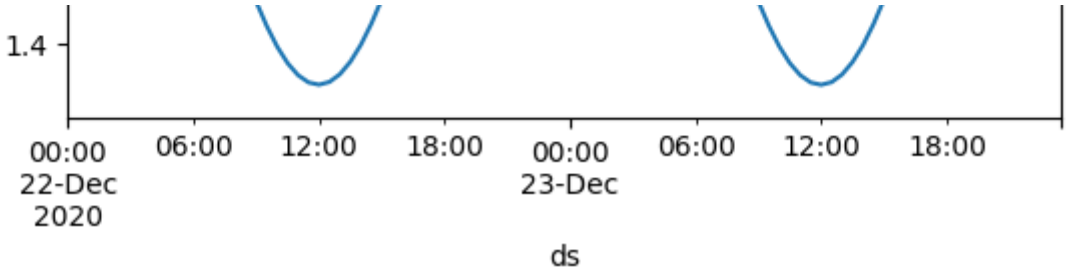
```
  merge_cols = ['month', 'day', 'hour', 'minute']
  df = df.merge(data, on=merge_cols)
  df.drop(merge_cols, inplace=True, axis=1)
  df.set_index('ds', drop=False, inplace=True)
  df = df[~df.index.duplicated(keep='first')]
  df = df.asfreq(freq='30min')

  if solar_calc == 'declination':
    df['declination_diff_1'] = df['declination'].diff(1)
  elif solar_calc == 'irradiance':
    df['irradiance_diff_1'] = df['irradiance'].diff(1)
  elif solar_calc == 'zenith':
    df['za_diff_1'] = df['za'].diff(1)
    df['za_rad_diff_1'] = df['za_rad'].diff(1)
  elif solar_calc == 'azimuth':
    df['azimuth_diff_1'] = df['azimuth'].diff(1)
    df['azimuth_rad_diff_1'] = df['azimuth_rad'].diff(1)
    df['azimuth_cos_diff_1'] = df['azimuth_cos'].diff(1)
    df['azimuth_sin_diff_1'] = df['azimuth_sin'].diff(1)

  return df


def plot_solar(solar, title, ylab):
  solar.plot()
  plt.ylabel(ylab)
  plt.title(title)
  plt.show()


def plot_solar_annual_and_solstice(solar, var, title_var, ylab):
  title = 'Annual ' + title_var + ' - Cambridge UK'
  plot_solar(solar.loc['2020-01-01':'2020-12-31', var], title, ylab)

  title = 'Winter solstice ' + title_var + ' - Cambridge UK'
  plot_solar(solar.loc['2020-12-22':'2020-12-23', var], title, ylab)

  title = 'Summer solstice ' + title_var + ' - Cambridge UK'
  plot_solar(solar.loc['2020-06-21':'2020-06-22', var], title, ylab)


df_before_solar = df.copy()

za_lab = 'Zenith angle - radians'
za_title = 'zenith angle'
df = calc_solar_data(df, 'zenith')
plot_solar_annual_and_solstice(df, 'za_rad', za_title, za_lab)

irr_ylab = 'irradiance - W / m^2'
irr_title = 'irradiance'
df = calc_solar_data(df, irr_title)
plot_solar_annual_and_solstice(df, irr_title, irr_title, irr_ylab)
```

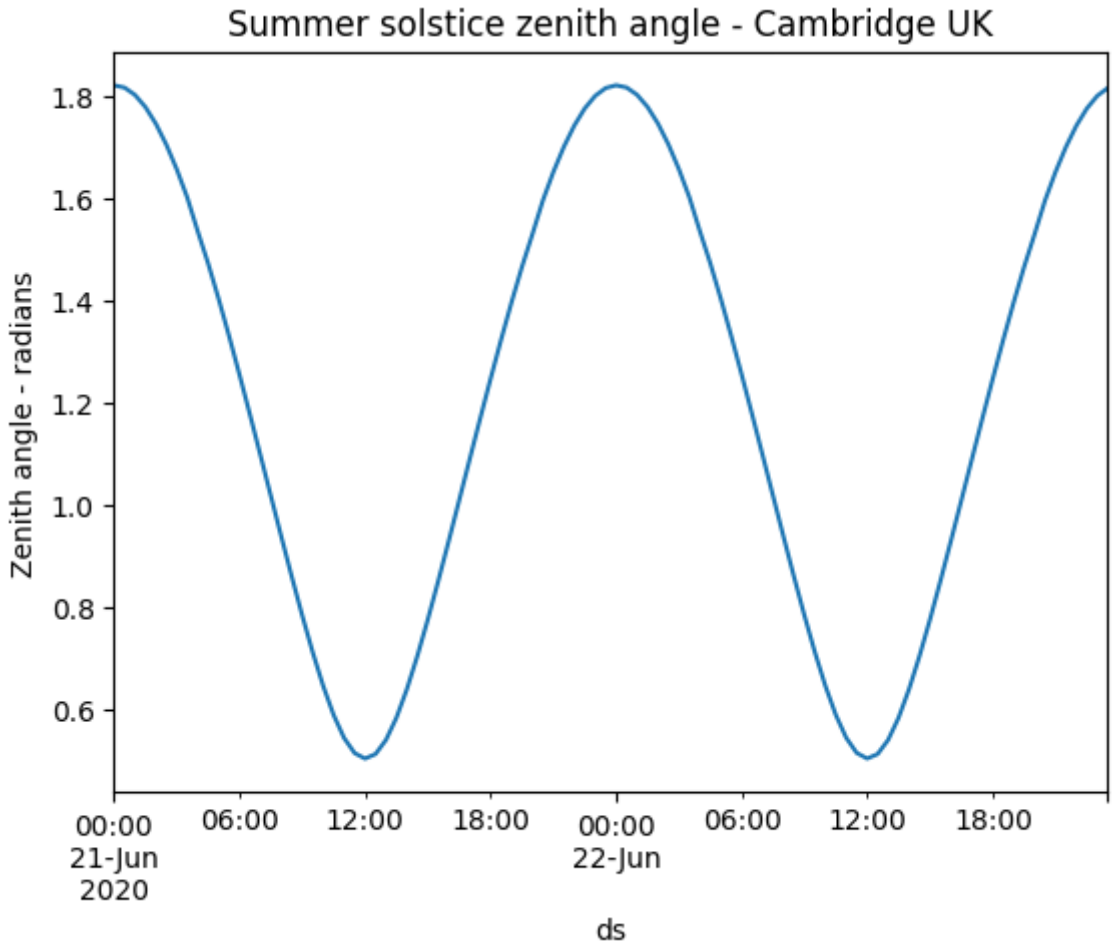
```
az_lab = 'Azimuth - radians'
az_title = 'azimuth'
df = calc_solar_data(df, az_title)
plot_solar_annual_and_solstice(df, 'azimuth_cos', az_title, 'cos(azimuth)')
plot_solar_annual_and_solstice(df, 'azimuth_sin', az_title, 'sin(azimuth)')
plot_solar_annual_and_solstice(df, 'azimuth_rad', az_title, az_lab)

dec_ylab = 'Declination - radians'
dec_title = 'declination'
df = calc_solar_data(df, dec_title)
plot_solar_annual_and_solstice(df, dec_title, dec_title, dec_ylab)

df = df.fillna(method='bfill')  # Small number of NAs in first row
df_solar_sanity = sanity_check_before_after_dfs(df_before_solar, df, 'df')
check_high_low_thresholds(df)
```

```
Installing {'solarpy', 'pysolar'} ... Done
Zenith
Calculating zenith: 100%|██████████| 17568/17568 [00:16<00:00, 1088.42it/s]
count    17568.000000
mean        89.672425
std         29.857274
min         28.767886
25%         67.803213
50%         89.235373
75%        111.449144
max        151.223633
Name: za, dtype: float64
```

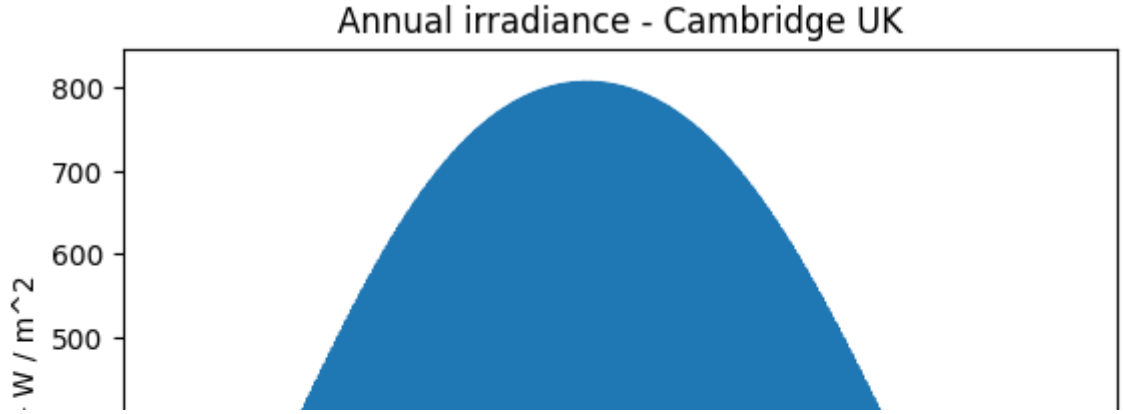


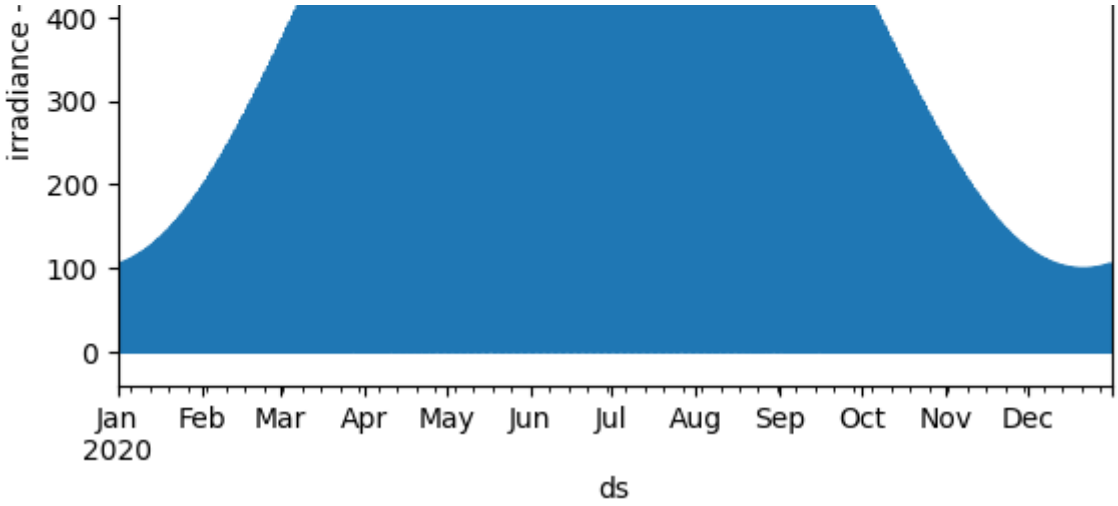


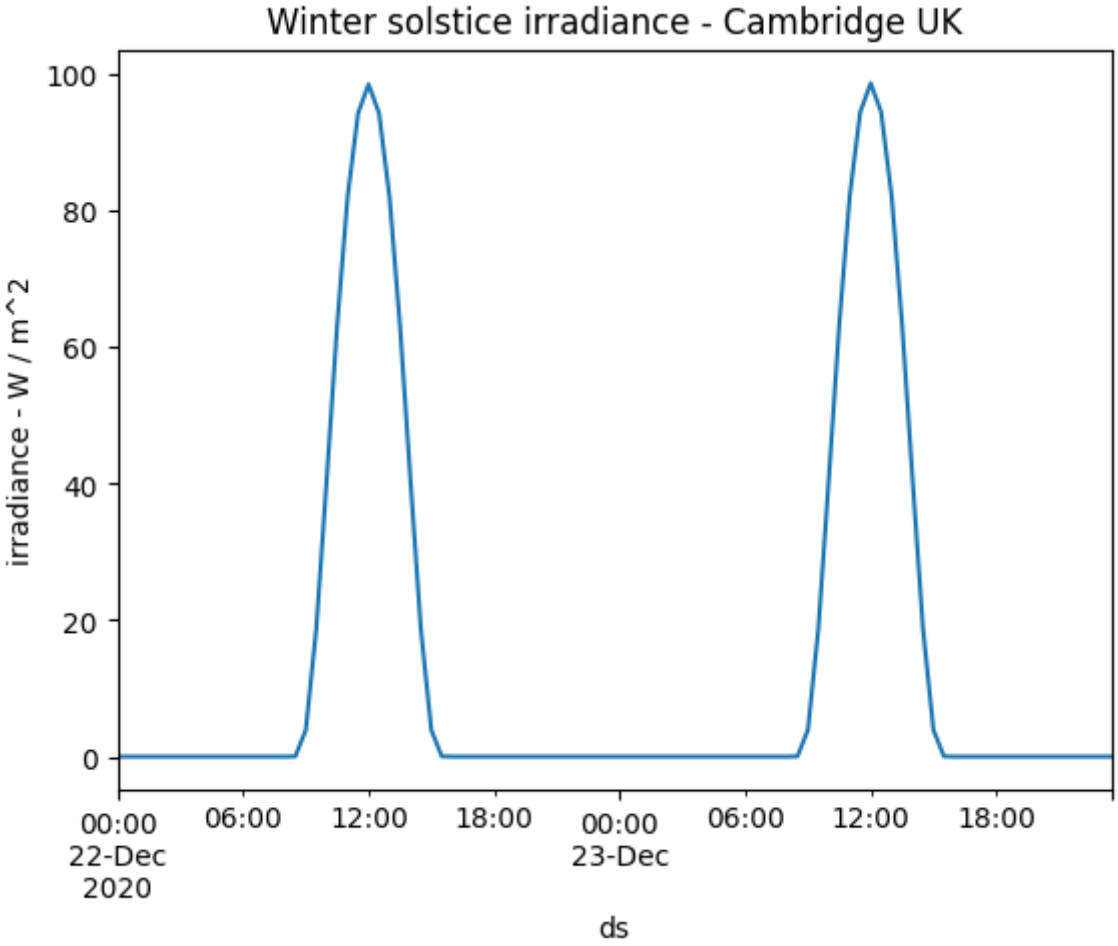


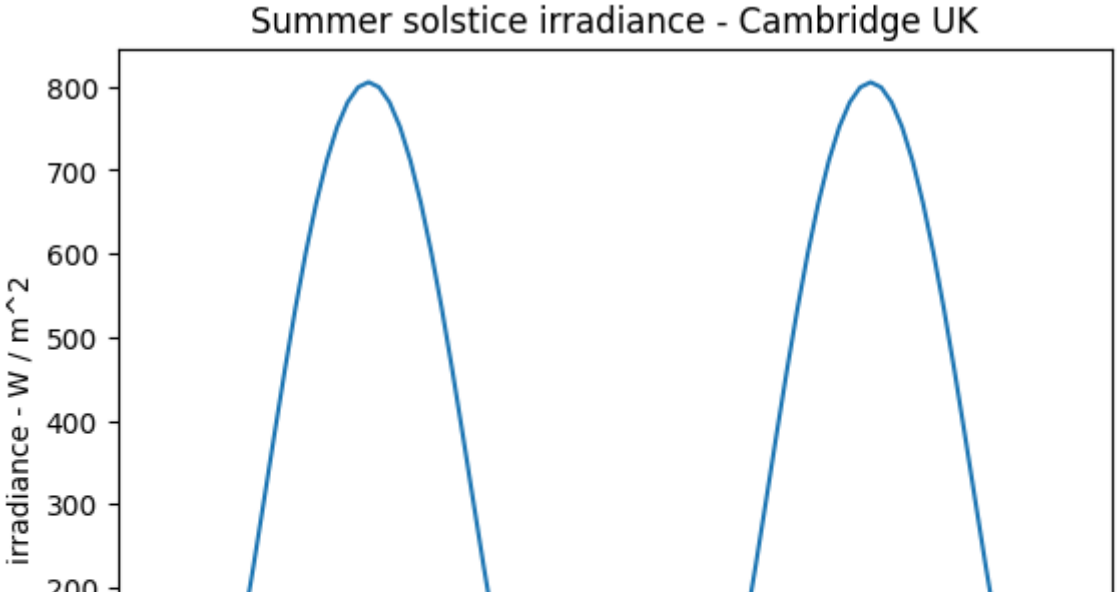


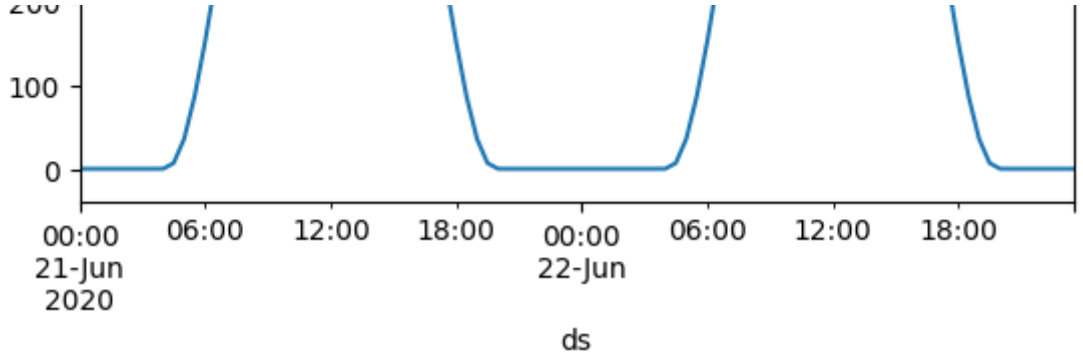

```
Irradiance
Calculating irradiance: 100%|██████████| 17568/17568 [00:03<00:00, 4962.53it/s
count    17568.000000
mean       142.940765
std        226.586194
min          0.000000
25%          0.000000
50%          0.000137
75%        220.386913
max        805.382014
Name: irradiance, dtype: float64
```



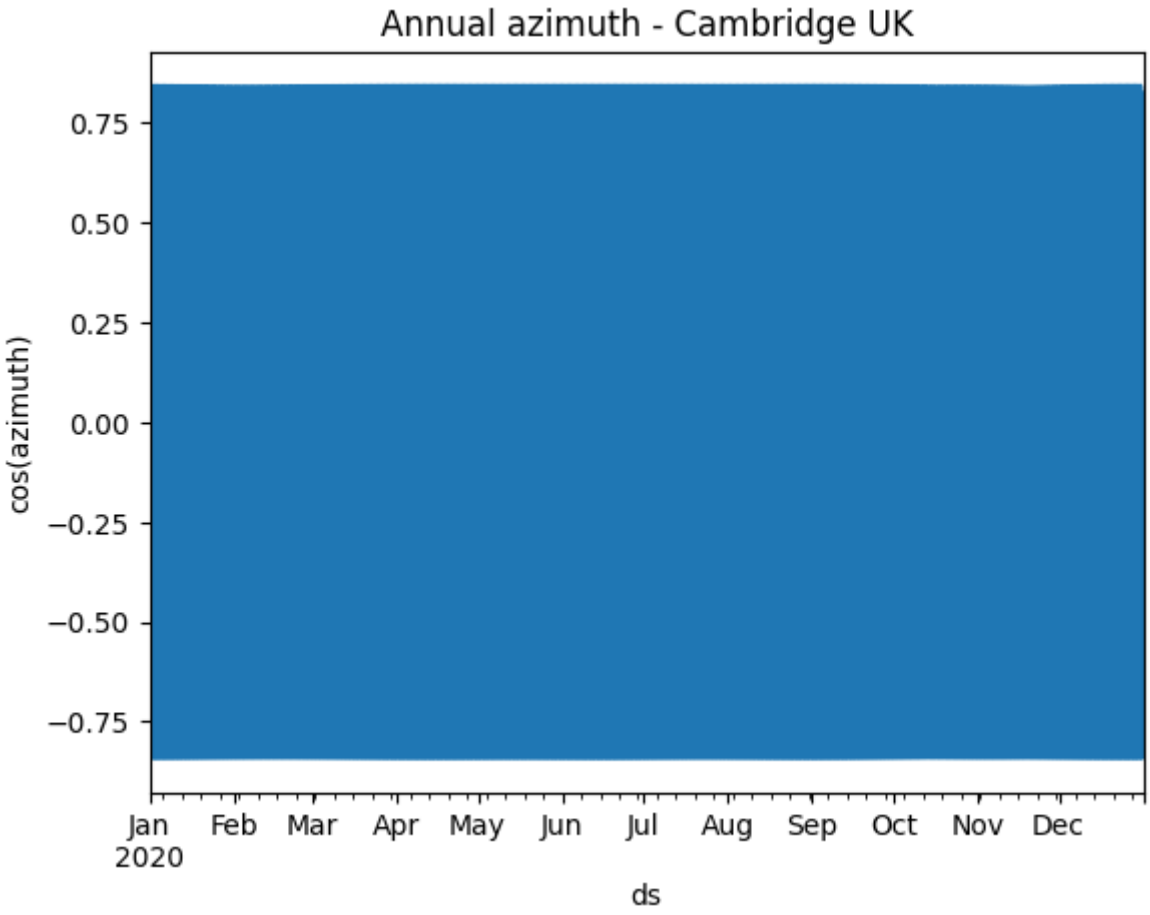

irradiance -
400
300
200
100
0
Jan
2020
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
ds

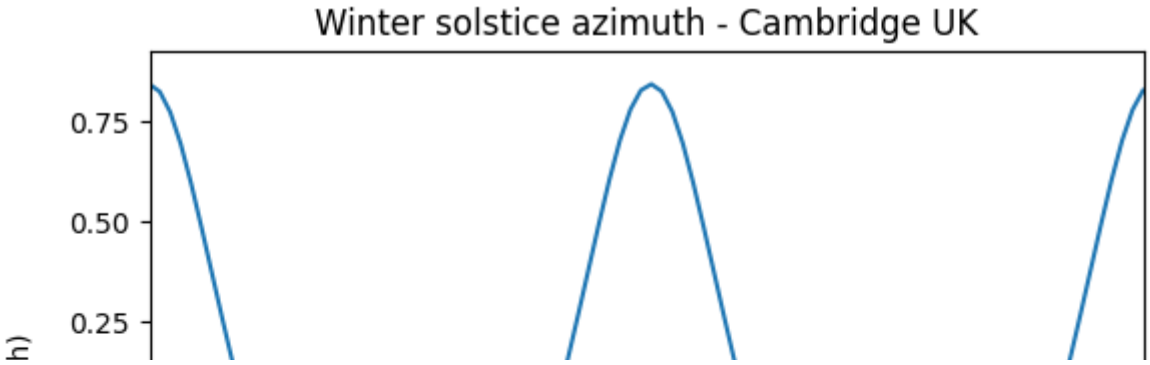

Winter solstice irradiance - Cambridge UK
irradiance - W / m^2
100
80
60
40
20
0
00:00
22-Dec
2020
06:00
12:00
18:00
00:00
23-Dec
06:00
12:00
18:00
ds

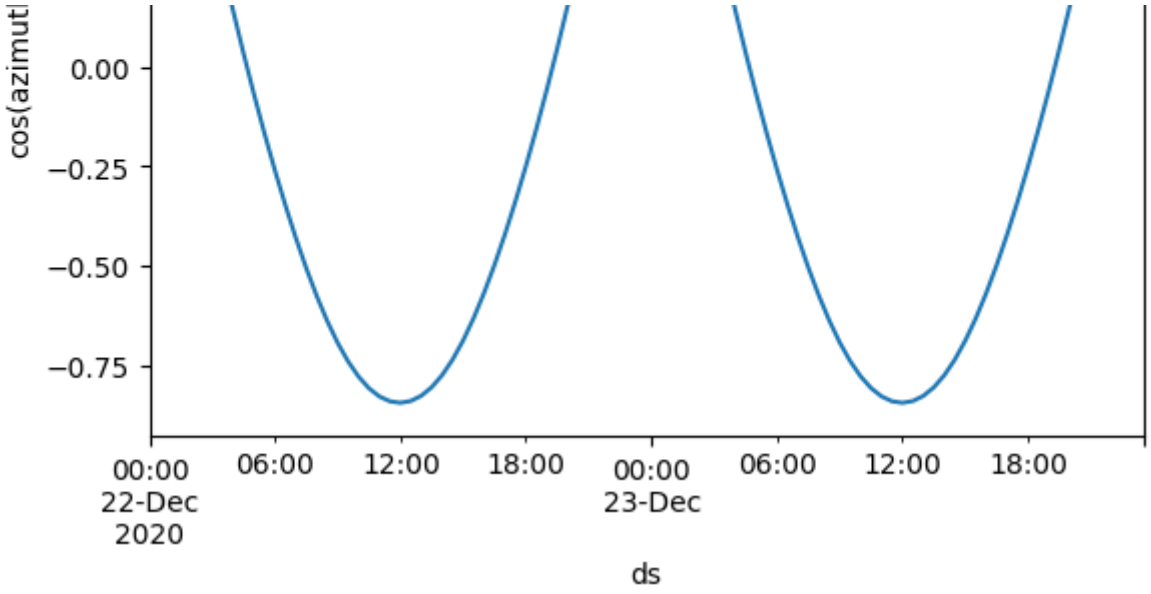

Summer solstice irradiance - Cambridge UK
irradiance - W / m^2
800
700
600
500
400
300
200

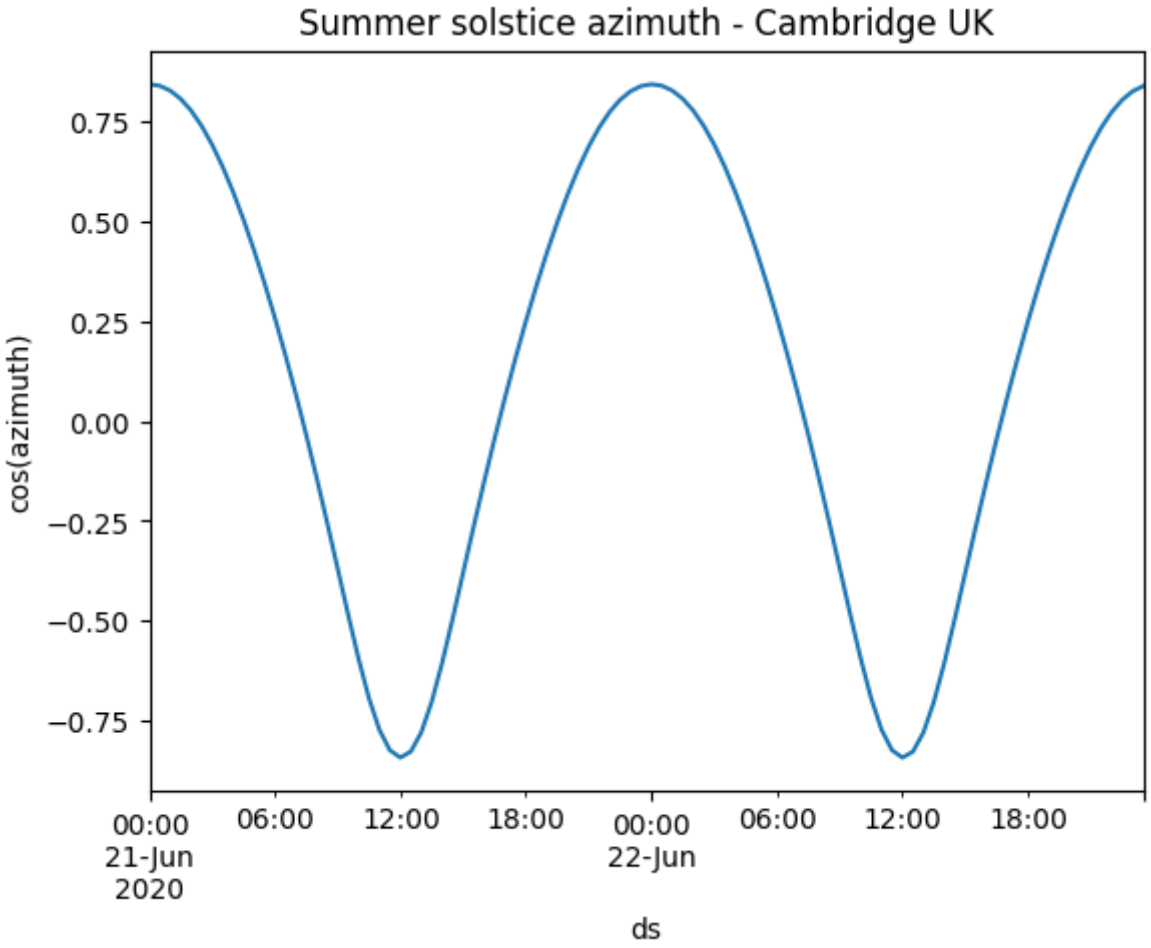


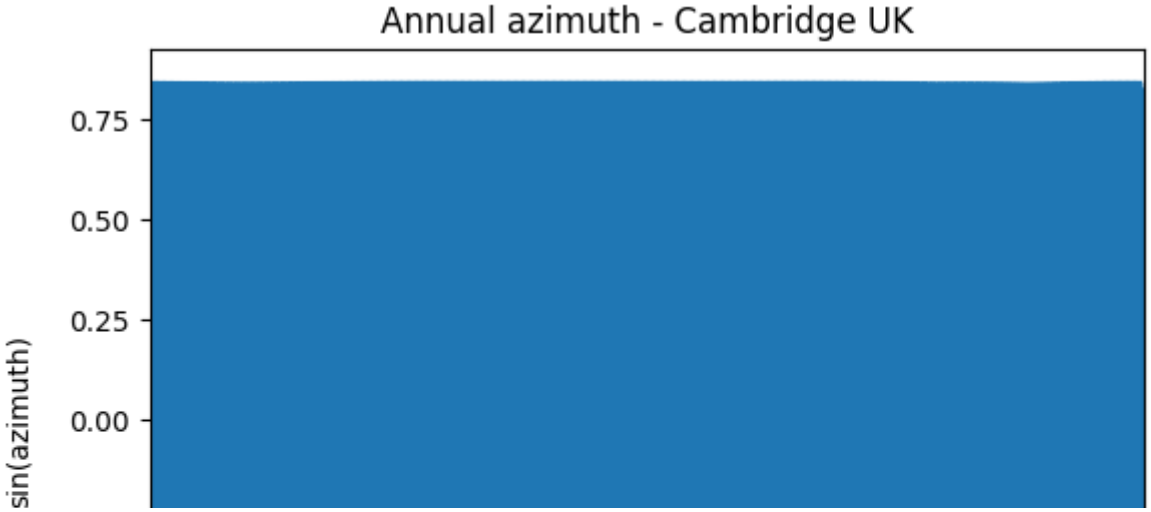

```
Azimuth
Calculating azimuth: 100%|██████████| 17568/17568 [00:15<00:00, 1115.88it/s]
count    17568.000000
mean         0.001957
std          0.588445
min         -0.841471
25%         -0.579791
50%          0.006409
75%          0.582247
max          0.841471
Name: azimuth, dtype: float64
```

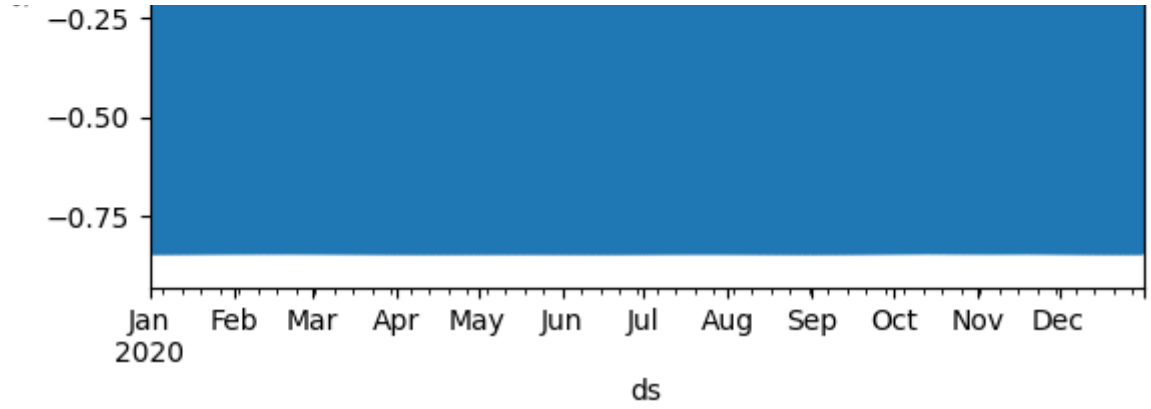


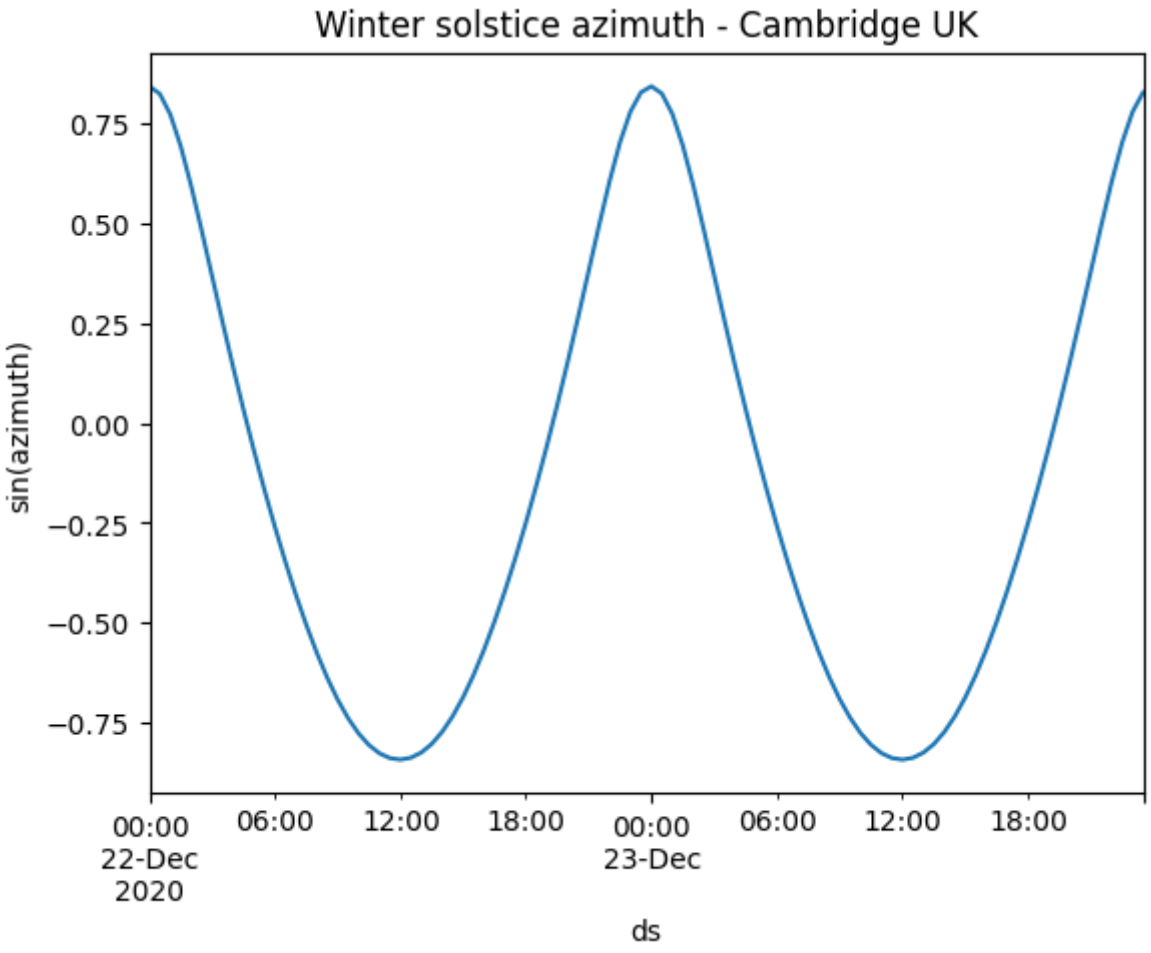


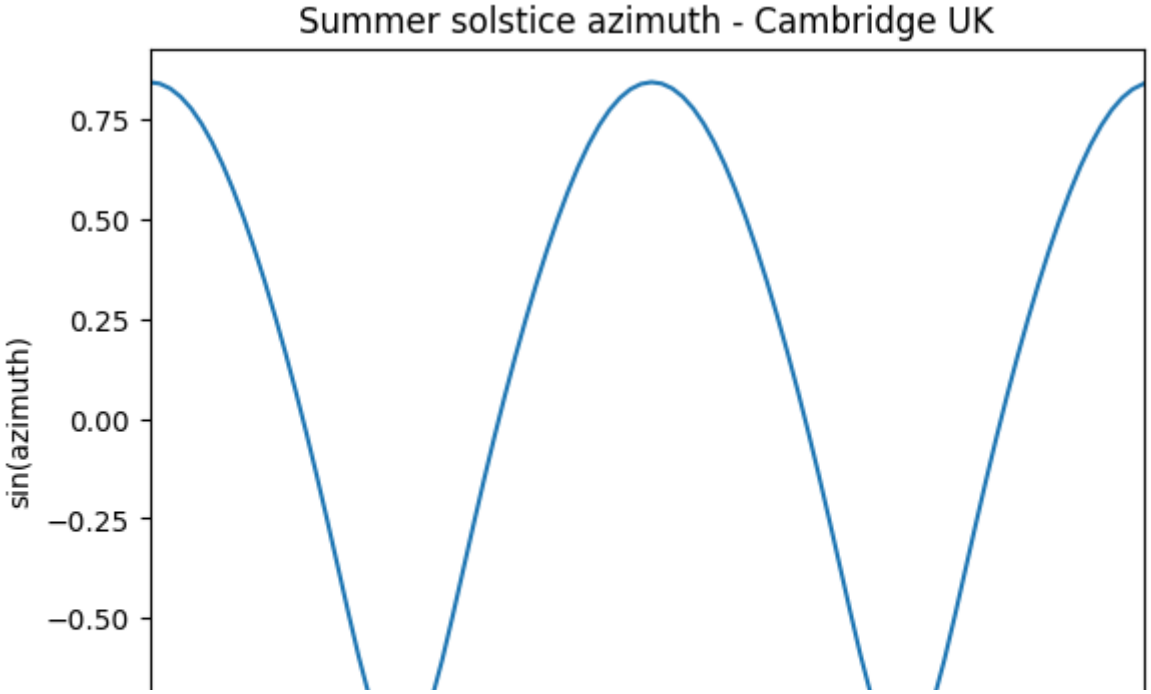

cos(azimuth)
0.00
−0.25
−0.50
−0.75
00:00
22-Dec
2020
06:00
12:00
18:00
00:00
23-Dec
06:00
12:00
18:00
ds

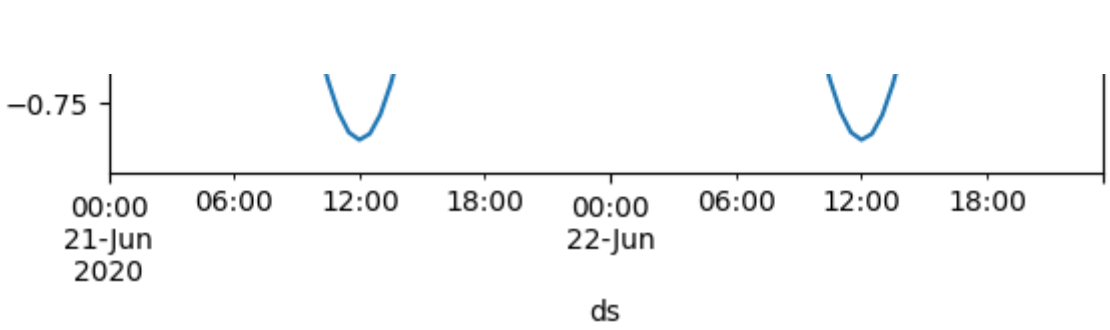

Summer solstice azimuth - Cambridge UK
cos(azimuth)
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
00:00
21-Jun
2020
06:00
12:00
18:00
00:00
22-Jun
06:00
12:00
18:00
ds

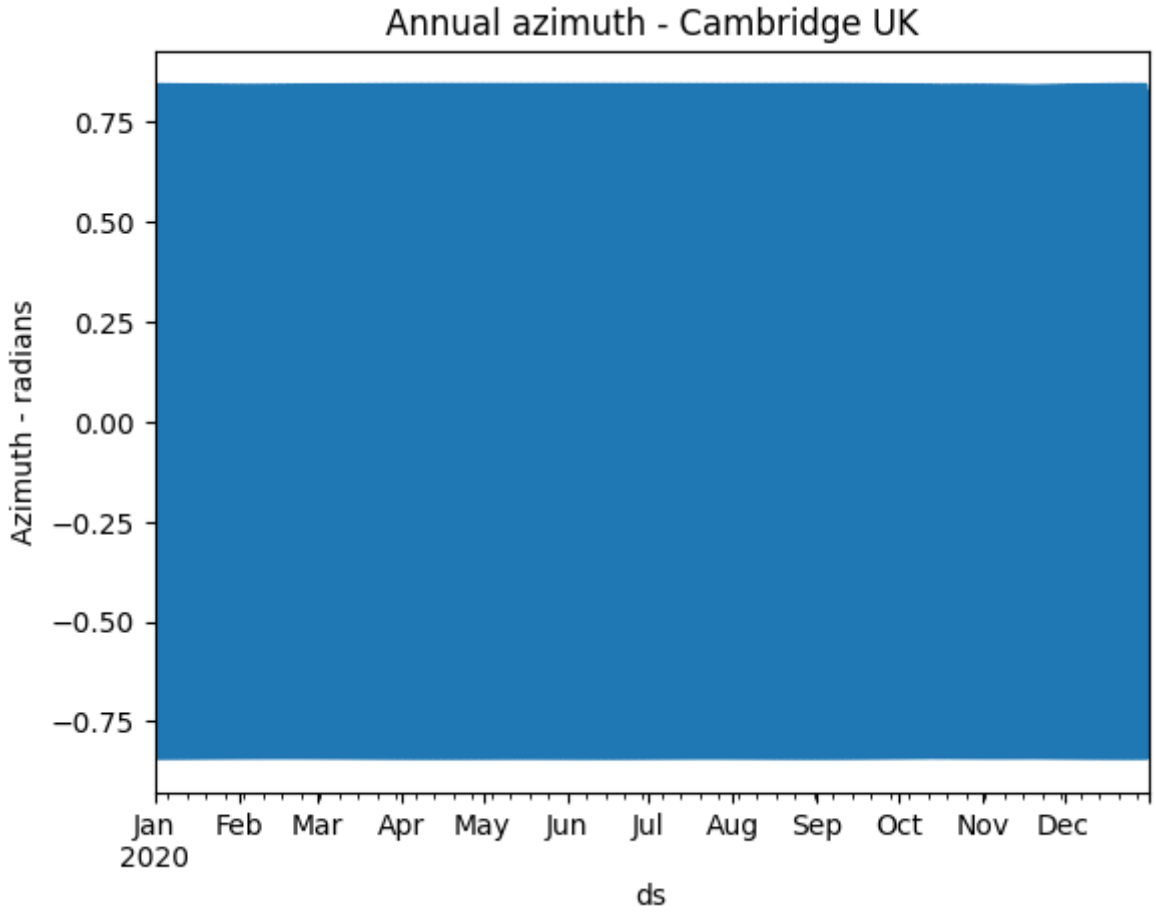

Annual azimuth - Cambridge UK
sin(azimuth)
0.75
0.50
0.25
0.00

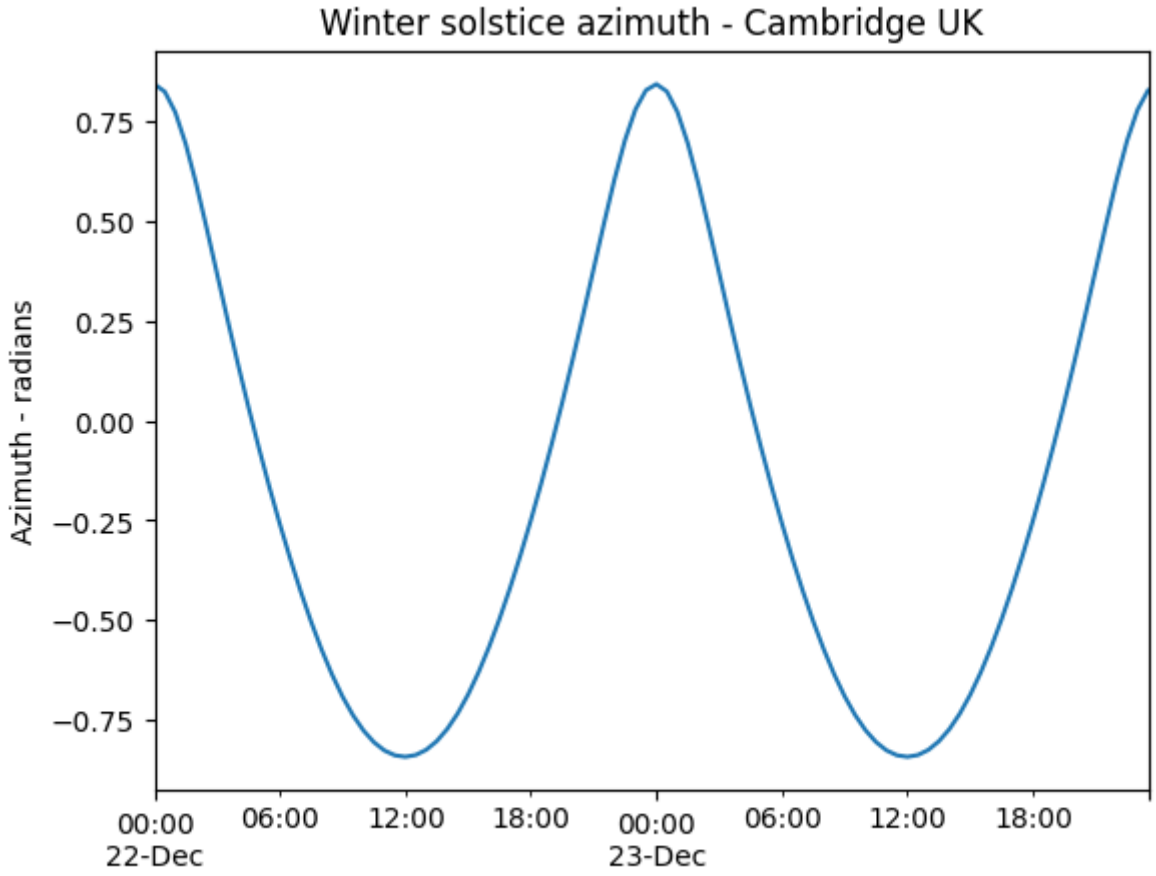

−0.25
−0.50
−0.75
Jan
2020
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
ds


Winter solstice azimuth - Cambridge UK
sin(azimuth)
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
00:00
22-Dec
2020
06:00
12:00
18:00
00:00
23-Dec
06:00
12:00
18:00
ds

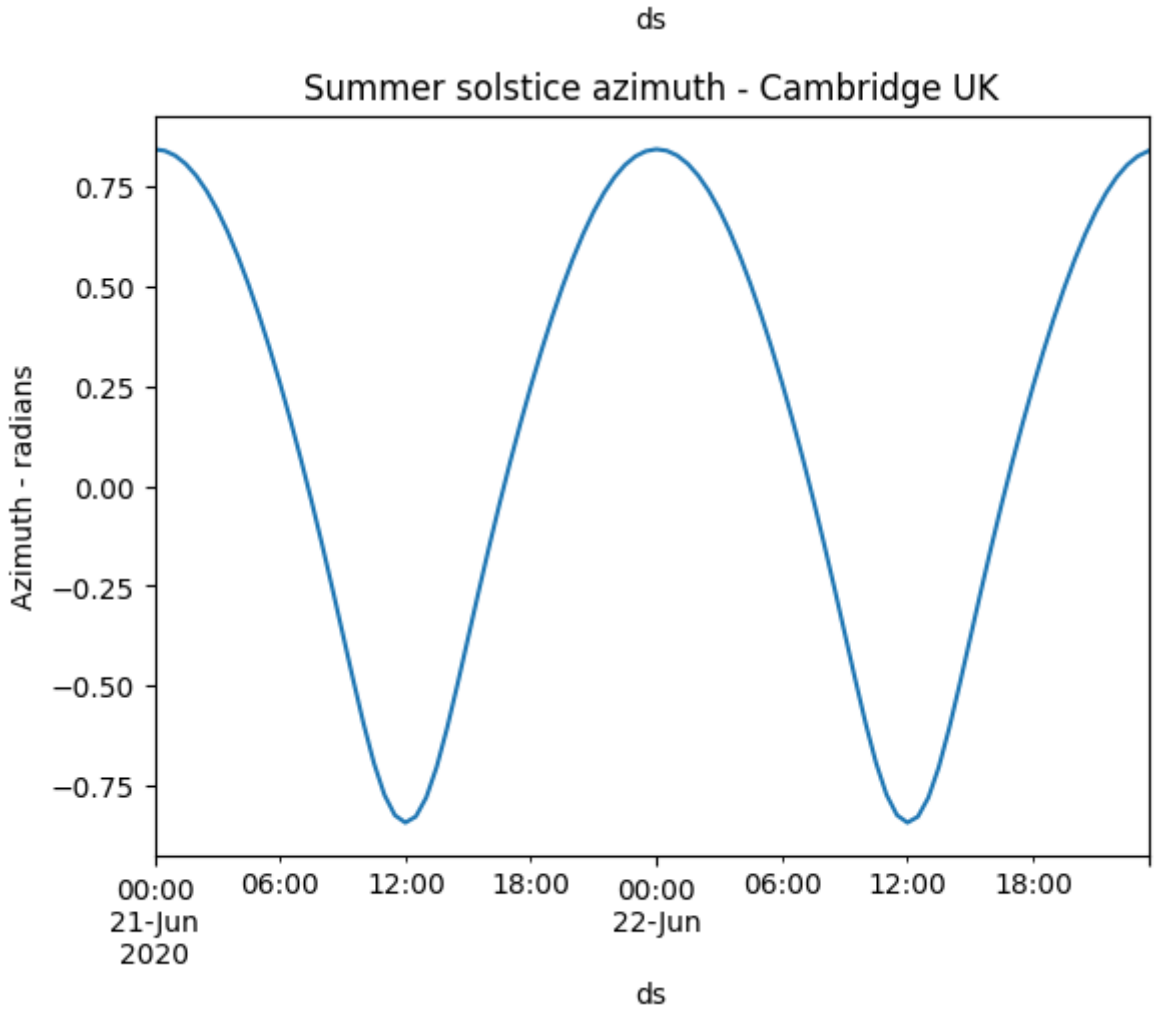

Summer solstice azimuth - Cambridge UK
sin(azimuth)
0.75
0.50
0.25
0.00
−0.25
−0.50


−0.75
00:00
21-Jun
2020
06:00
12:00
18:00
00:00
22-Jun
06:00
12:00
18:00
ds

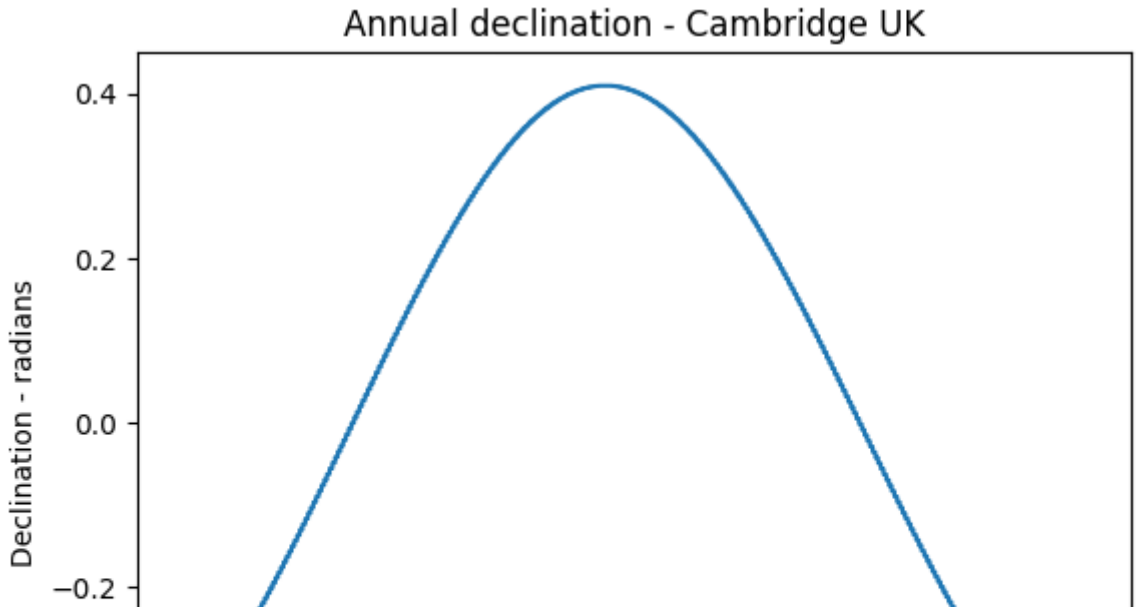

Annual azimuth - Cambridge UK
Azimuth - radians
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Jan
2020
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
ds

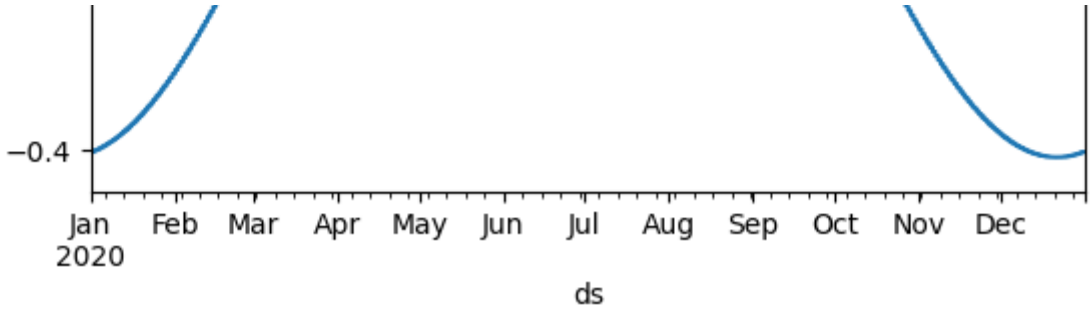

Winter solstice azimuth - Cambridge UK
Azimuth - radians
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
00:00
22-Dec
06:00
12:00
18:00
00:00
23-Dec
06:00
12:00
18:00

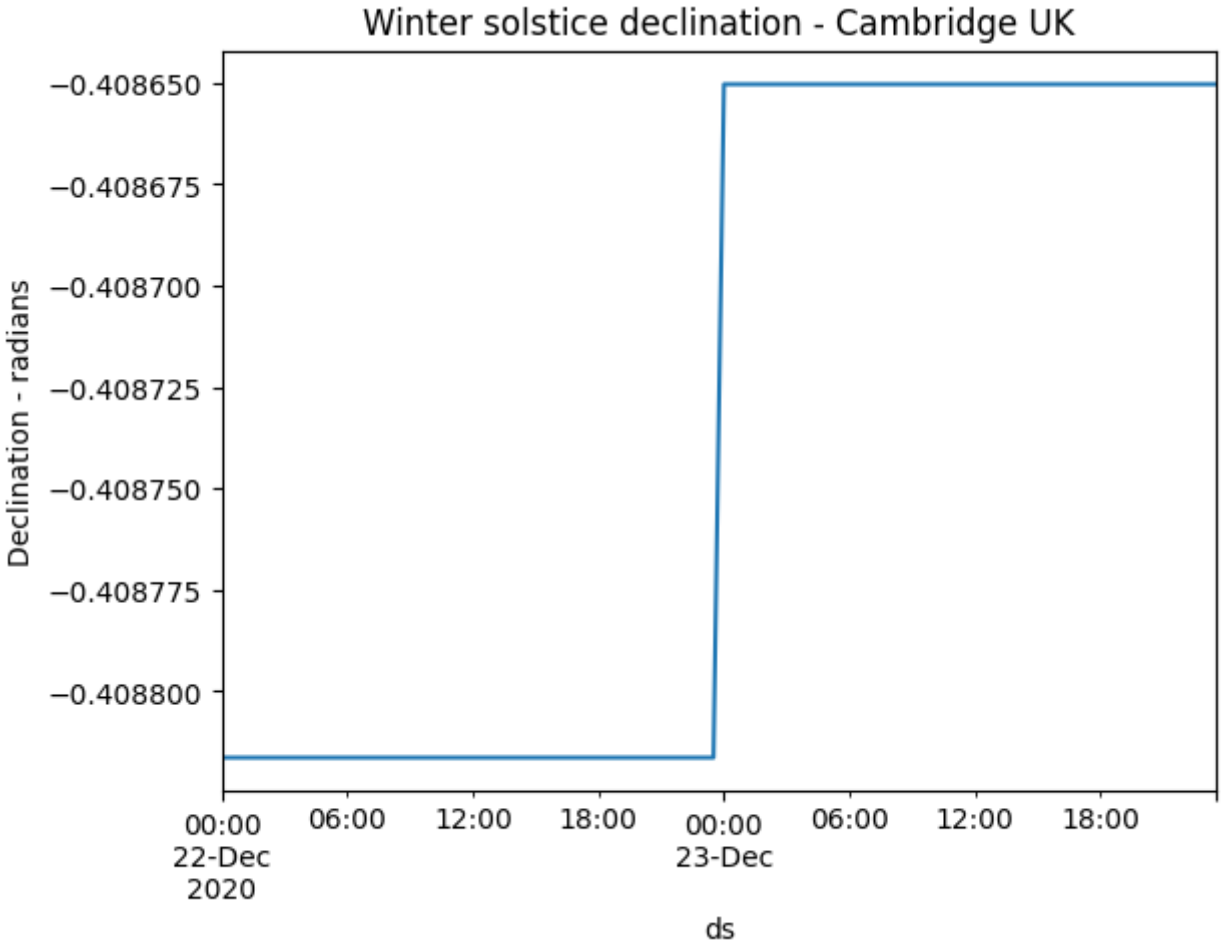
2020

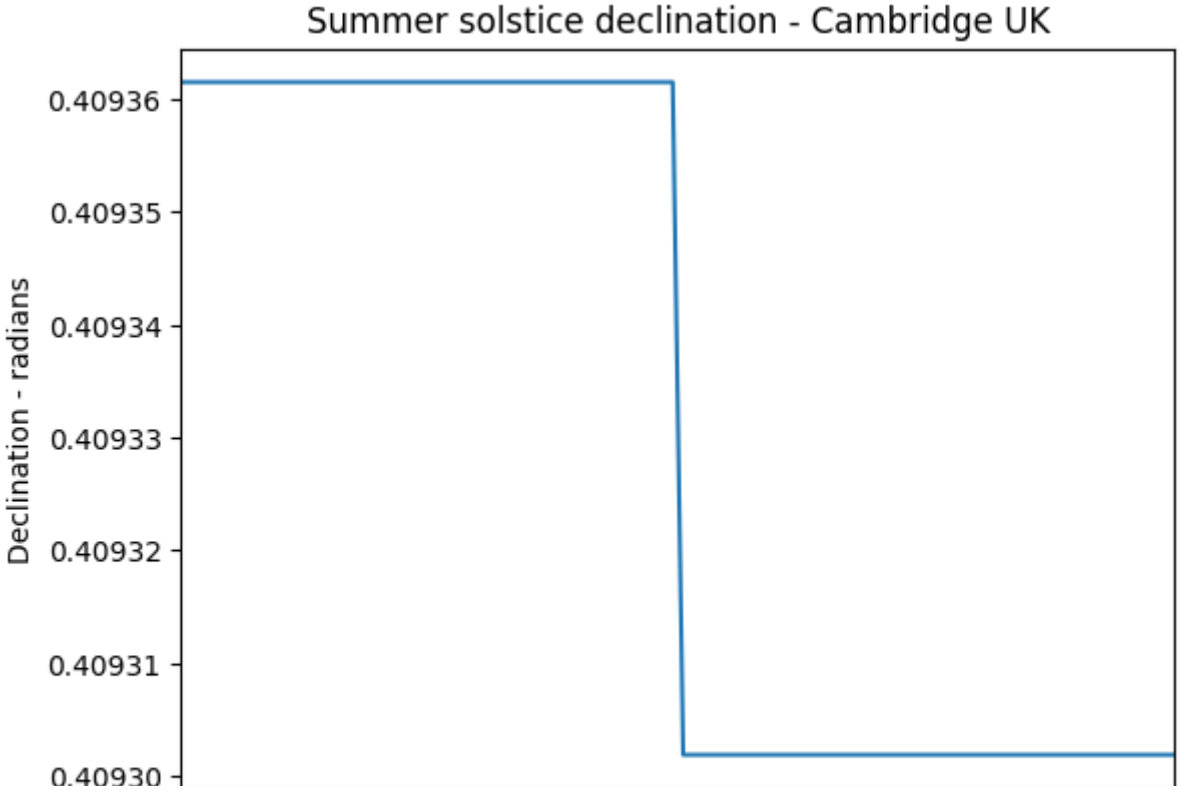


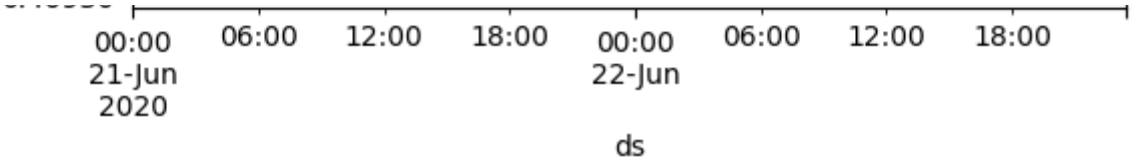

```
Declination
Calculating declination: 100%|██████████| 17568/17568 [00:00<00:00, 97131.66it
count    17568.000000
mean         0.005800
std          0.287570
min         -0.408846
25%         -0.280364
50%          0.011877
75%          0.290560
max          0.409361
Name: declination, dtype: float64
```




−0.4
Jan
2020
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
ds


Winter solstice declination - Cambridge UK
−0.408650
−0.408675
−0.408700
−0.408725
−0.408750
−0.408775
−0.408800
Declination - radians
00:00
22-Dec
2020
06:00
12:00
18:00
00:00
23-Dec
06:00
12:00
18:00
ds


Summer solstice declination - Cambridge UK
0.40936
0.40935
0.40934
0.40933
0.40932
0.40931
0.40930
Declination - radians




```
df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): True
redundancy before > after: False
mean before feature redundancy: 40.862
mean after feature redundancy:  60.045
```

	before	after	diff
rows	252768	252768	0
cols	64	81	17
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	0	0	0
low_var_rows	0	0	0
low_var_cols	6	7	1
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

There is zero solar irradiance between sunset and sunrise. Hence the near zero median value for irradiance.

Periodogram plots

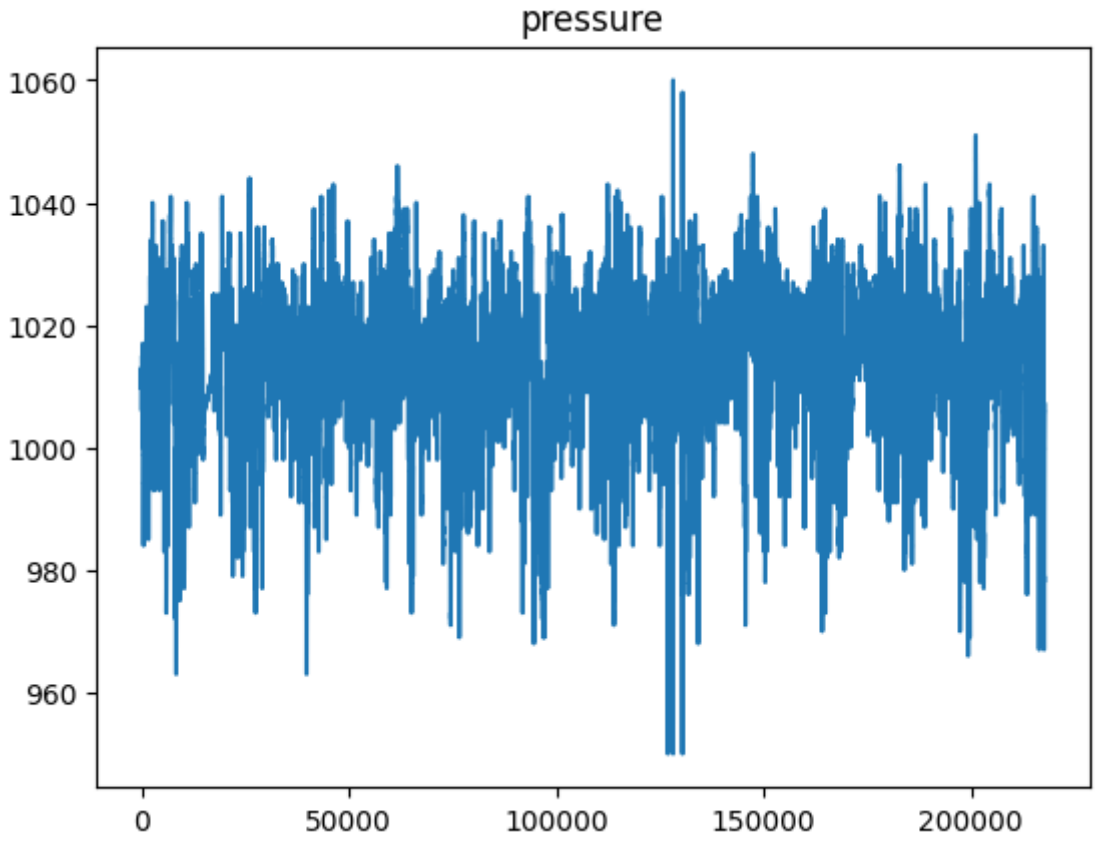
Some quick exploratory periodogram plots.

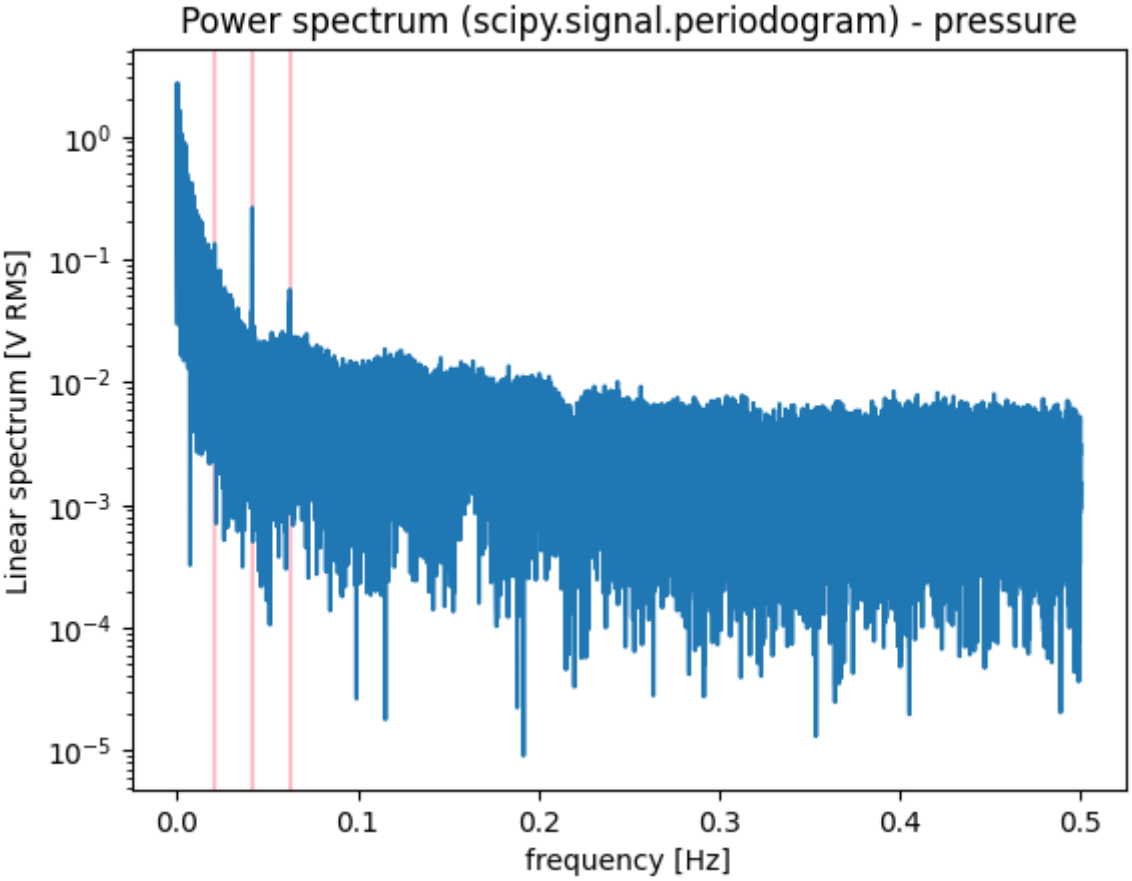
Check periodicities:

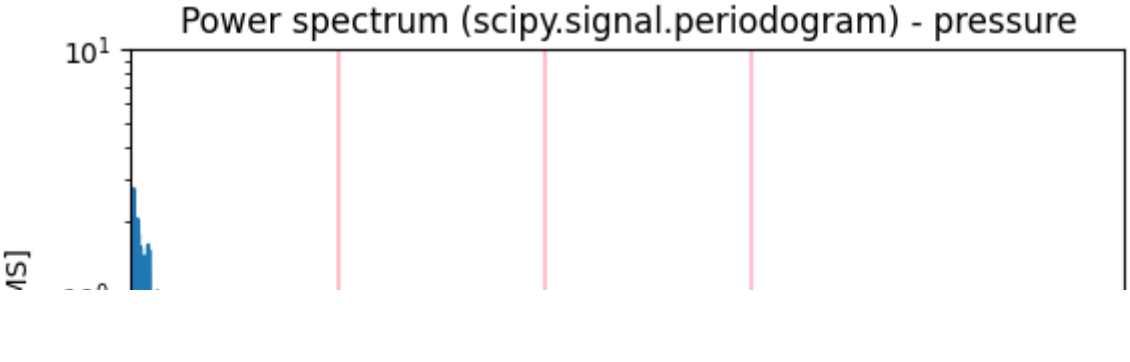
- pressure
- dew.point

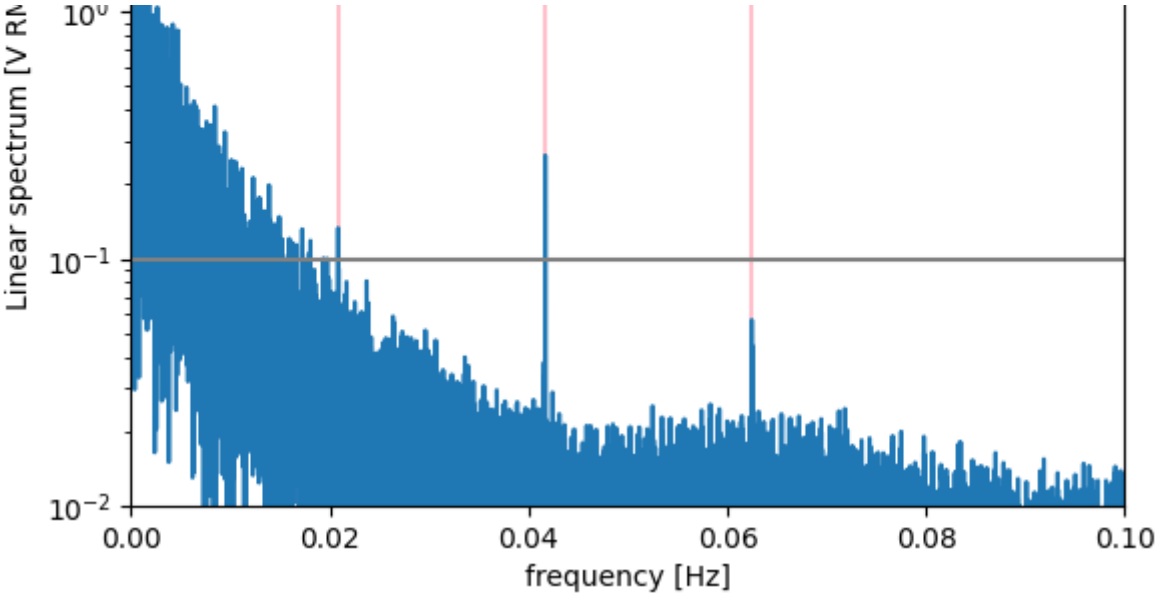
- temperature

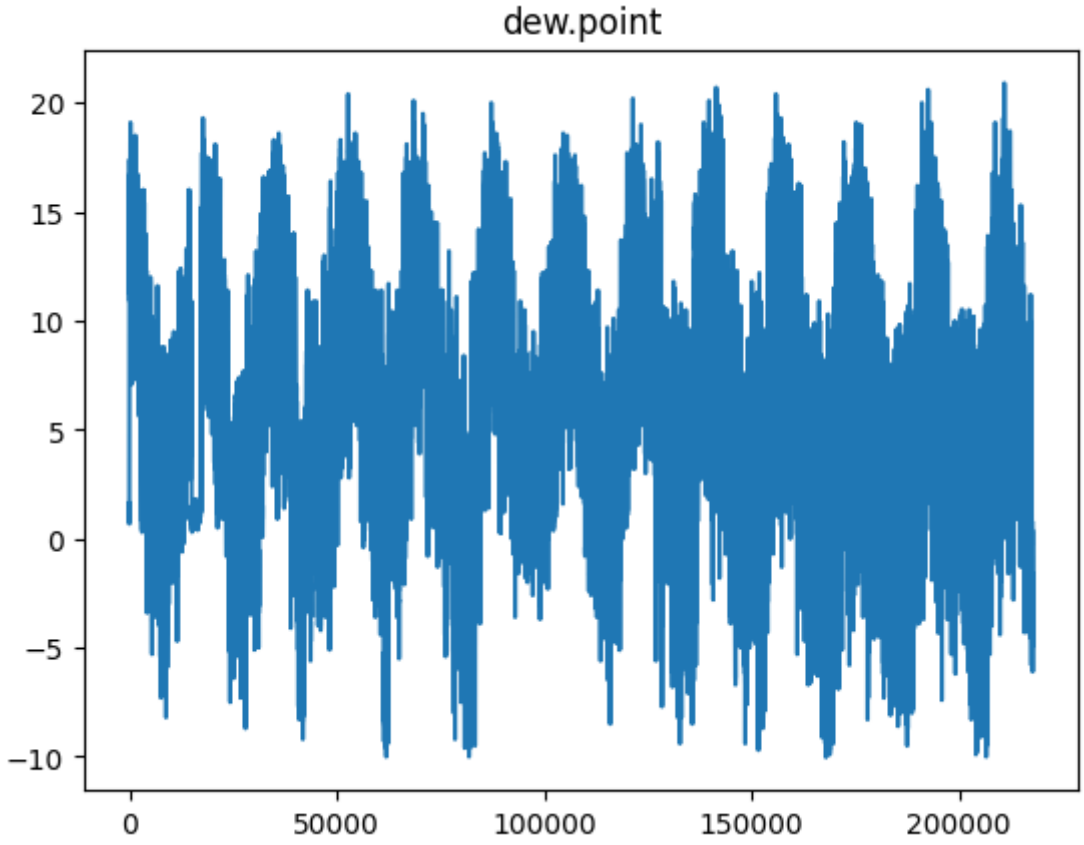
```
train_df = df.loc[df['year'] < min(VALID_YEAR, TEST_YEAR)]


plot_periodograms(train_df, 'pressure', [1,2,3])
plot_periodograms(train_df, 'dew.point', [1,2,3,4,5])
plot_periodograms(train_df, 'y', [1,2,3,4,5,6,7])
```

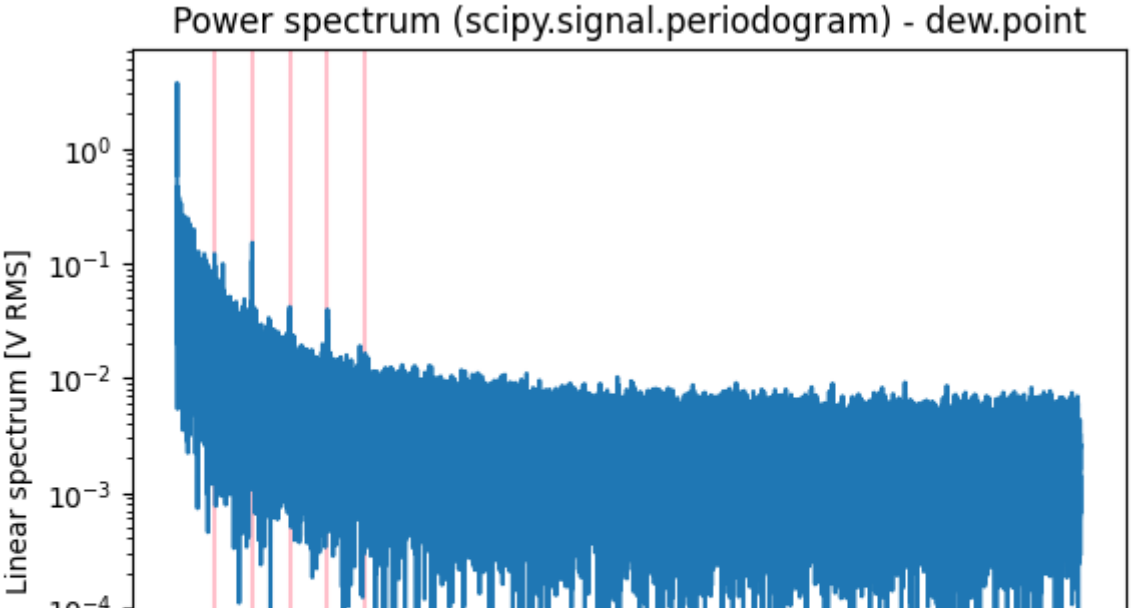

pressure
1060
1040
1020
1000
980
960
0
50000
100000
150000
200000

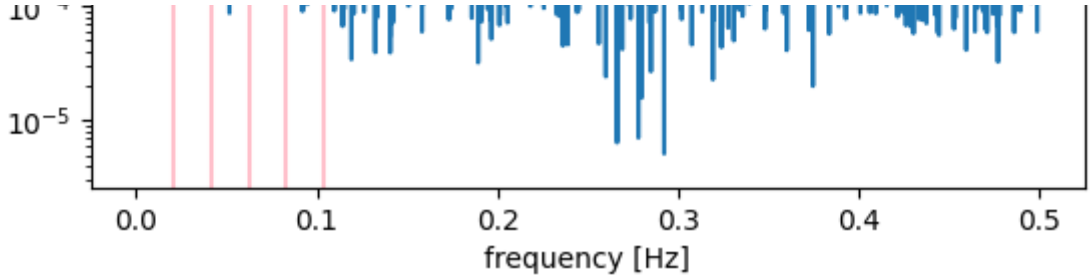

Power spectrum (scipy.signal.periodogram) - pressure
Linear spectrum [V RMS]
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0.0
0.1
0.2
0.3
0.4
0.5
frequency [Hz]

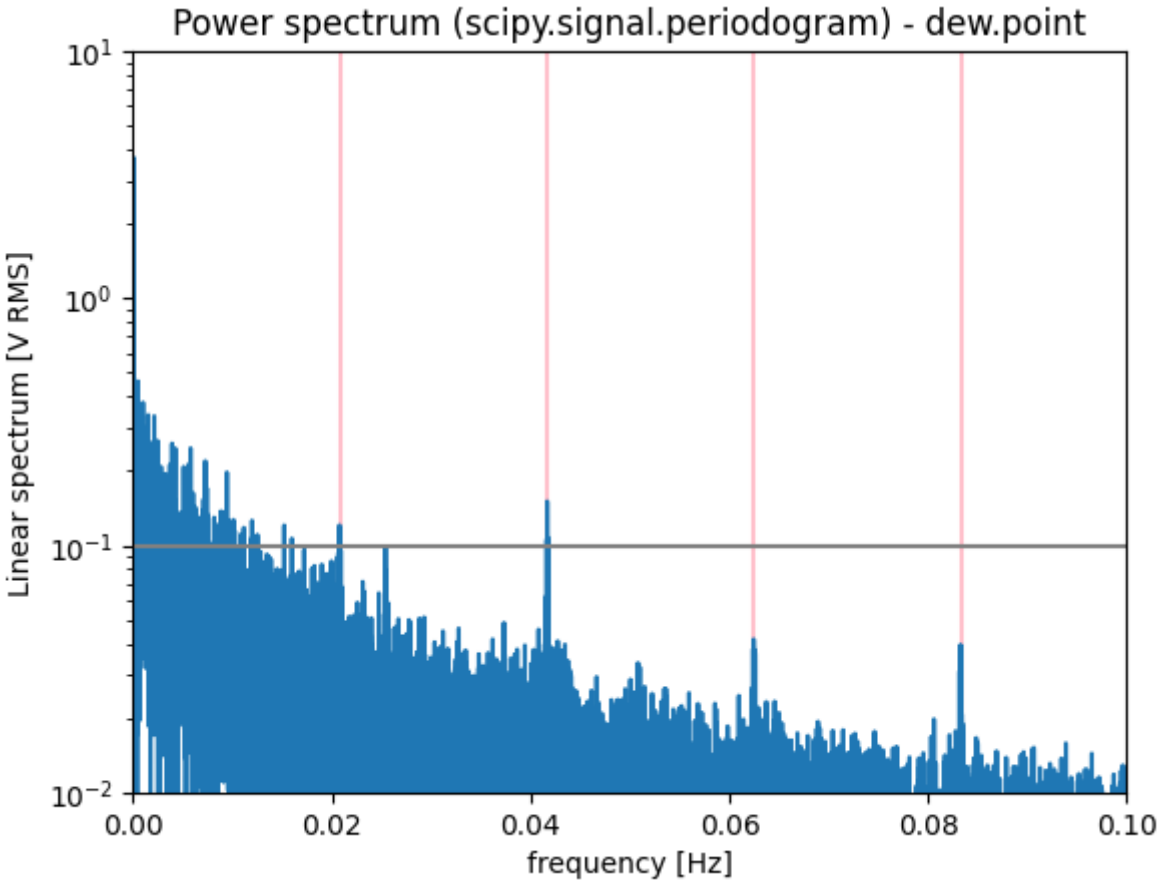

Power spectrum (scipy.signal.periodogram) - pressure
10^{1}

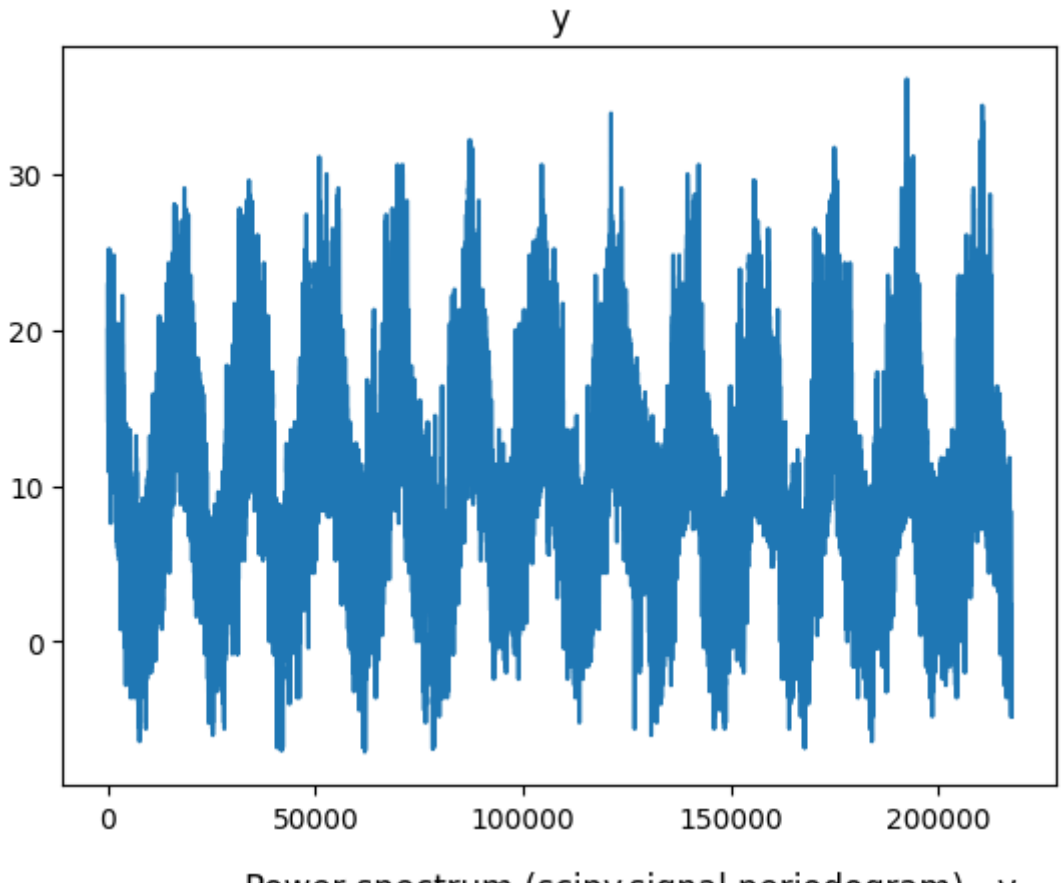

Linear spectrum [V RM
10^{0}
10^{-1}
10^{-2}
0.00
0.02
0.04
0.06
0.08
0.10
frequency [Hz]

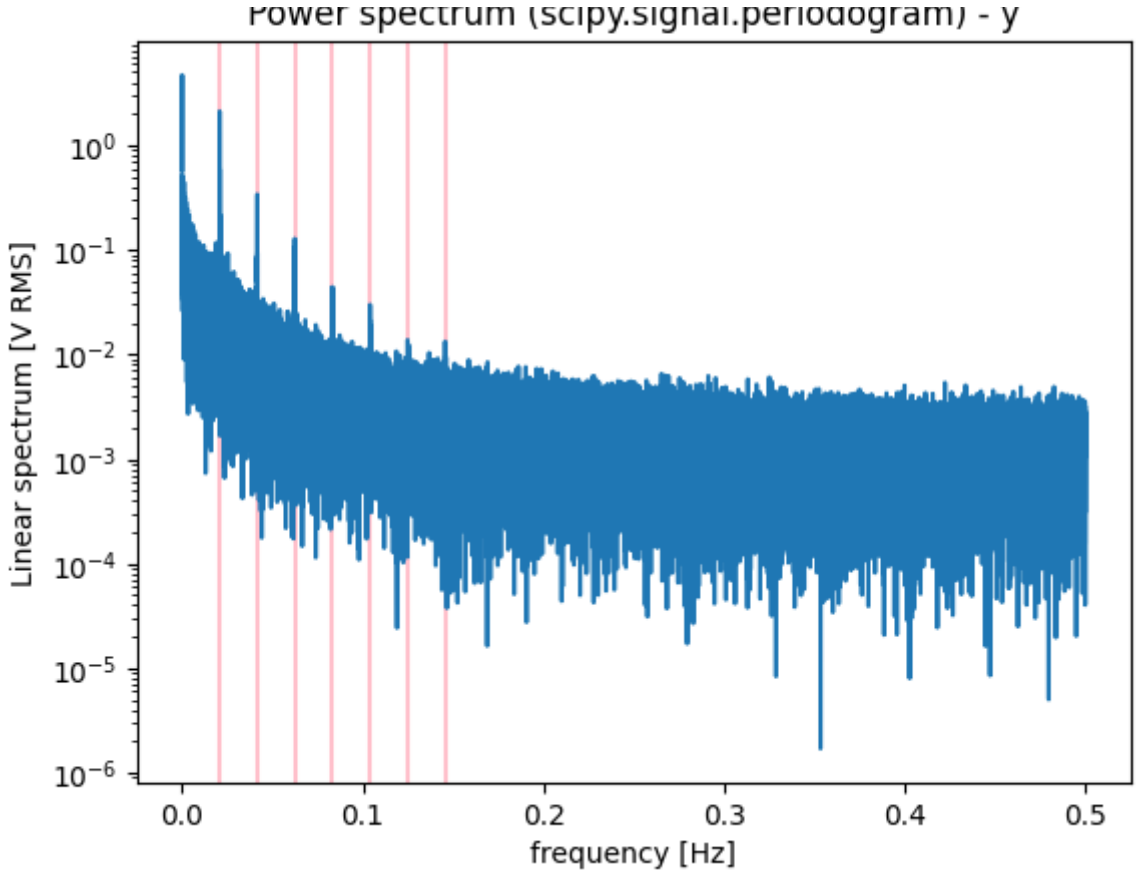

dew.point
20
15
10
5
0
−5
−10
0
50000
100000
150000
200000

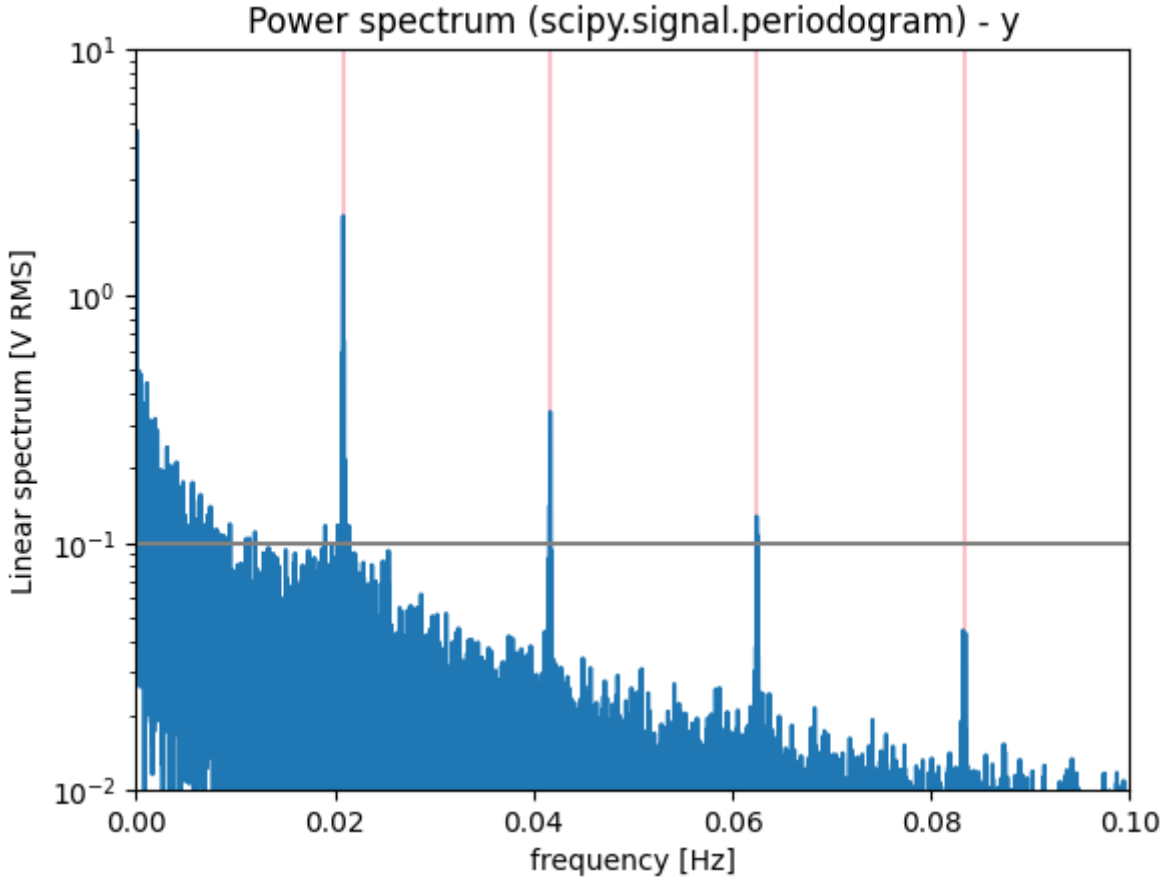

Power spectrum (scipy.signal.periodogram) - dew.point
Linear spectrum [V RMS]
10^{0}
10^{-1}
10^{-2}
10^{-3}


10^{-5}
0.0
0.1
0.2
0.3
0.4
0.5
frequency [Hz]


Power spectrum (scipy.signal.periodogram) - dew.point
Linear spectrum [V RMS]
10^{1}
10^{0}
10^{-1}
10^{-2}
0.00
0.02
0.04
0.06
0.08
0.10
frequency [Hz]


y
30
20
10
0
0
50000
100000
150000
200000





For `pressure`, `dew.point` and `y` the zoomed in periodograms show clear peaks corresponding to seasonalities at 24 hours and 12 hours (approximately 0.02 and 0.04 Hertz respectively). For both `pressure` and `dew.point` the peak at 12 hours is higher than the peak at 24 hours. For `y`

the 24 hours peak is higher than the 12 hours peak. There are also a few 'alias harmonics' at higher frequencies.

For `pressure`, `dew.point` and `y`, there is no indication of seaonality at 29.53 days which corresponds to the lunar cycle.

Unobserved components model

The unobserved components model (UCM) is an alternative to prophet decomposition. It produces trend, cyclic, seasonal and residual terms which can be stochastic or deterministic. The cyclic component of the unobserved components model can model the relatively long annual seasonality (48 * 365.2425 = 17,532 steps) using harmonic terms.

- statsmodels - unobserved components
- statsmodels - Detrending, Stylized Facts and the Business Cycle
- Daniel Toth - Multi seasonal time series analysis: decomposition and forecasting with Python

Here, I run a few experiments using UCM for decomposition. There is some data leakage here but these features are not used in later modeling notebooks. It may be worthwhile considering UCM models with exogenous data as an advanced baseline.

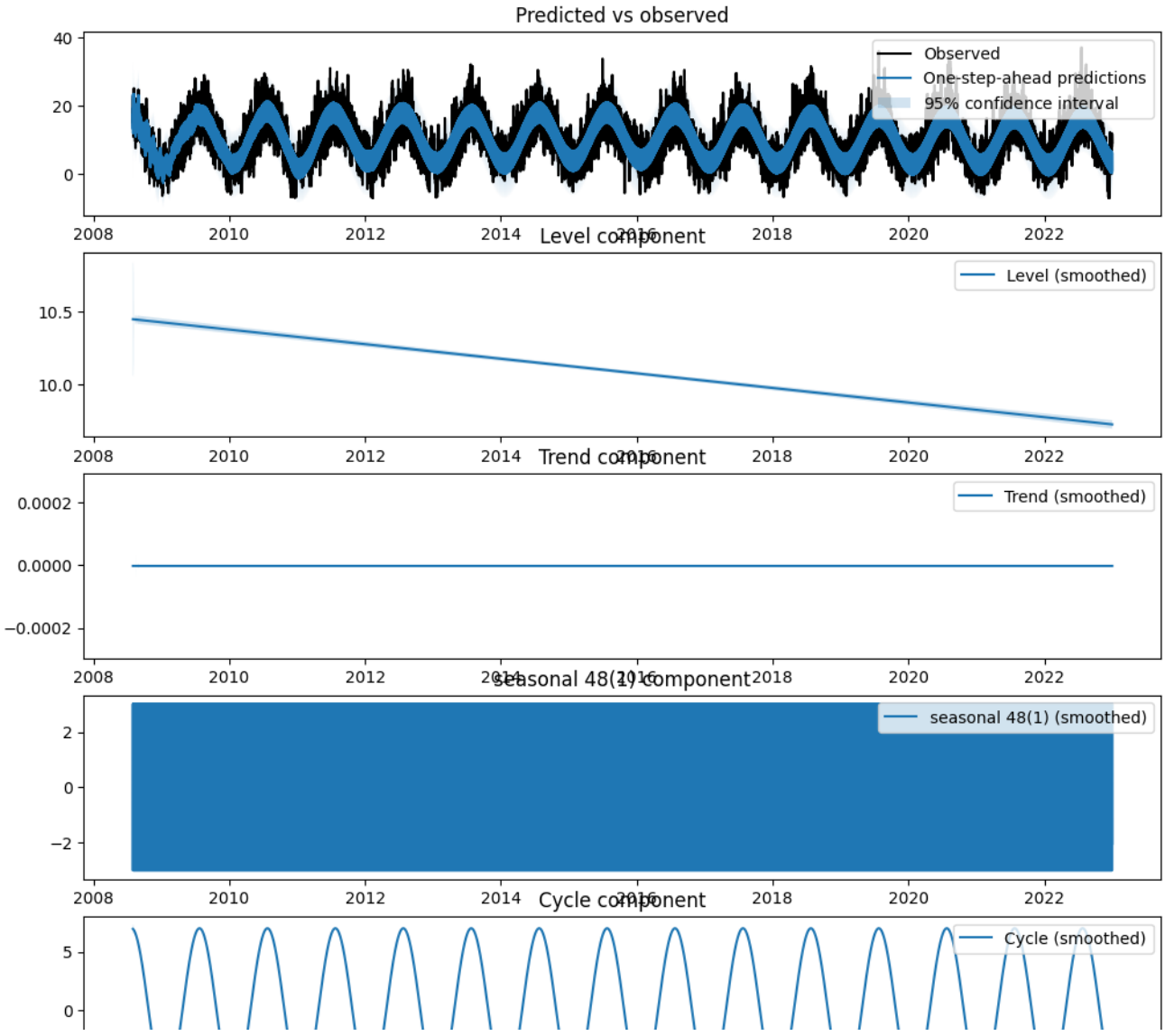
```
def get_uc_model(data, params, y_col='y'):
  ucm = sm.tsa.UnobservedComponents(data[y_col], #.dropna().values,
                                    **params)
  res_f = ucm.fit(method='powell', disp=False)

  return res_f


def check_uc_model(ucmodel, valid_data = None):
  print(ucmodel.summary())

  ucmodel.plot_components(figsize=(12, 12))
  plt.show()

  ucmodel.plot_diagnostics(figsize=(12, 12), lags=96)
  plt.show()

  if valid_data is not None:
    # model forecast
    steps_ = len(valid_data) #int(48 * 365.2425)
    test_df = valid_data[['y']].interpolate(method='linear')  #.head(steps_)
    forecast_ucm = ucmodel.forecast(steps=steps_, exog=test_df)

    # calculating mean absolute error and root mean squared error for out-of-sampl
    RMSE_ucm = np.sqrt(np.mean([(test_df.iloc[x, 0] - forecast_ucm[x]) ** 2 for x
    MAE_ucm  = np.mean([np.abs(test_df.iloc[x, 0] - forecast_ucm[x]) for x in ran

    print(f"\nOut-of-sample mean absolute error (MAE): {'%.2f' % MAE_ucm}")
```

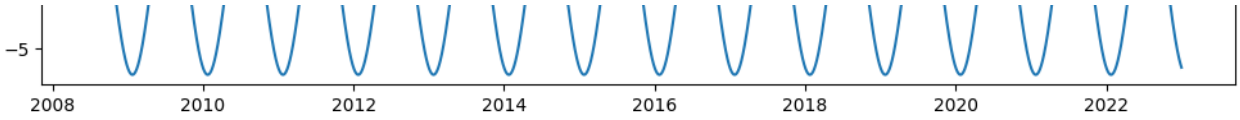
```
    print(f"Out-of-sample root mean squared error (RMSE): {'%.2f' % RMSE_ucm}")


def add_ucm_decomposition(data_, ucm_mod, feat = 'y'):
  data = data_.copy(deep=True)

  level_name  = feat + '_ucm_' + 'level'
  yearly_name = feat + '_ucm_' + 'yearly'
  daily_name  = feat + '_ucm_' + 'daily'
  res_name    = feat + '_ucm_' + 'res'

  data.loc[:, level_name]  = getattr(ucm_mod, 'level')['smoothed']
  data.loc[:, yearly_name] = getattr(ucm_mod, 'cycle')['smoothed']
  data.loc[:, daily_name]  = getattr(ucm_mod, 'freq_seasonal')[0]['smoothed']

  data.loc[:, res_name] = data[feat] - data[level_name] - data[yearly_name] - data

  return data
```

Temperature first:

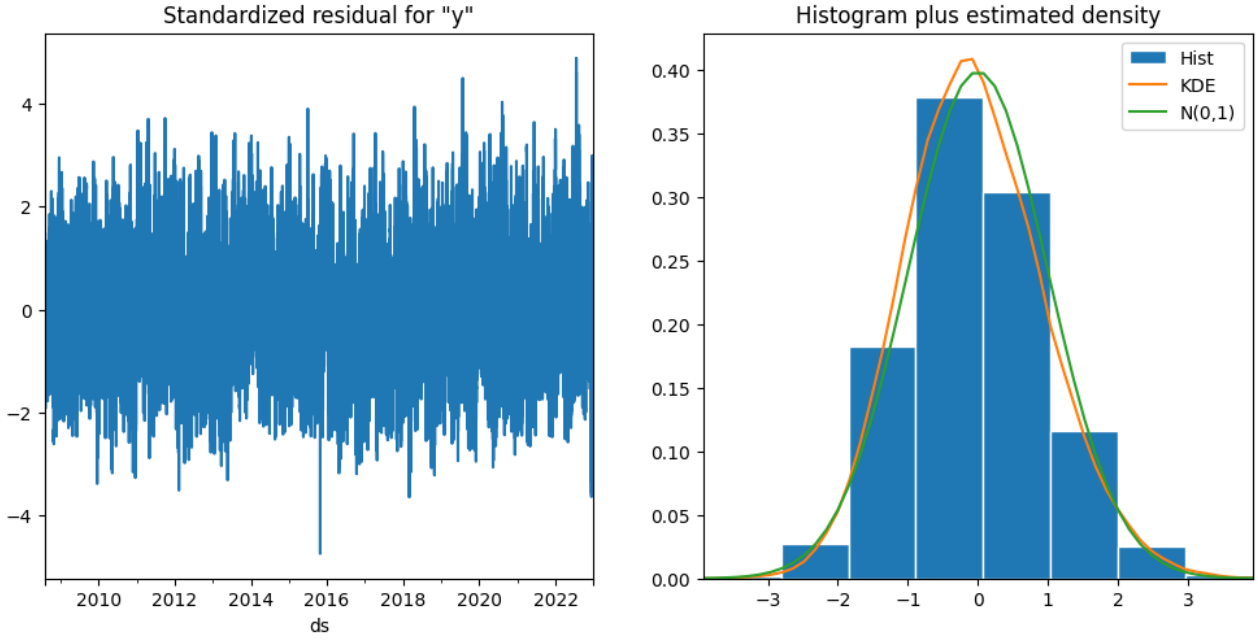
```
df_before_ucm_y = train_df.copy()

uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             #'exog': train_df[['humidity','dew.point','irradiance']].interpolate(
             #exog = data[['dew.point']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             'cycle': True,
             # irregular = True,
             #'autoregressive': 1,
             'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  1}]}

res_y = get_uc_model(df, uc_params)
check_uc_model(res_y)

# WARN: Proably data leakage here
df = add_ucm_decomposition(df, res_y)

df_sanity_ucm_y = sanity_check_before_after_dfs(df_before_ucm_y, df, 'df')
check_high_low_thresholds(df)
```

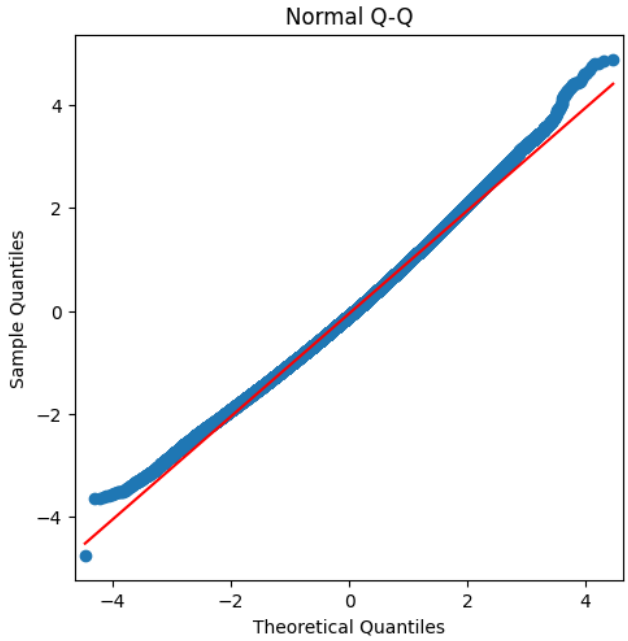
```
                        Unobserved Components Results
==============================================================================
Dep. Variable:                          y   No. Observations:                25
Model:                deterministic trend   Log Likelihood             -685947
                    + freq_seasonal(48(1))   AIC                       1371899
                                 + cycle     BIC                       1371920
Date:                     Mon, 15 Apr 2024   HQIC                      1371905
Time:                             18:17:21
Sample:                         08-01-2008
                              - 12-31-2022
Covariance Type:                      opg
==============================================================================
                     coef    std err          z      P>|z|      [0.025
------------------------------------------------------------------------------
sigma2.irregular  13.3197      0.037    359.480      0.000       13.247
frequency.cycle    0.0004   1.94e-08   1.84e+04      0.000        0.000
==============================================================================
Ljung-Box (L1) (Q):                 246946.86   Jarque-Bera (JB):                25
Prob(Q):                                 0.00   Prob(JB):
Heteroskedasticity (H):                  1.04   Skew:
Prob(H) (two-sided):                     0.00   Kurtosis:
==============================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

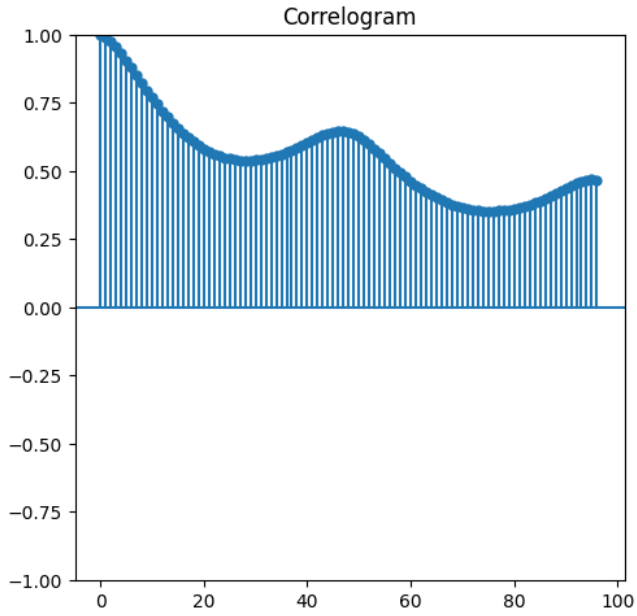




Note: The first 6 observations are not shown, due to approximate diffuse initialization.








```
df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): True
redundancy before > after: True
mean before feature redundancy: 60.045
mean after feature redundancy:  57.338
```

	before	after	diff
rows	252768	252768	0

cols	81	85	4
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	0	0	0
low_var_rows	0	0	0
low_var_cols	7	7	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

The temperature decomposition seemed acceptible.

Initial few months of decomposition show high variance.

`dew.point` second:

```
df_before_ucm_dp = train_df.copy()

uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             # exog = train_df[['humidity','dew.point']].interpolate(method='sline
             # 'exog': train_df[['dew.point_des']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             'cycle': True,
             # irregular = True,
             # 'autoregressive': 1,
             'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  2}]}

res_dp = get_uc_model(df, uc_params, 'dew.point')
check_uc_model(res_dp)

# WARN: Proably data leakage here
df = add_ucm_decomposition(df, res_dp, feat='dew.point')

plt.plot(res_dp.freq_seasonal[0]['smoothed'][:144])
plt.show()

df_sanity_ucm_dp = sanity_check_before_after_dfs(df_before_ucm_dp, df, 'df')
check_high_low_thresholds(df)
```

Unobserved Components Results

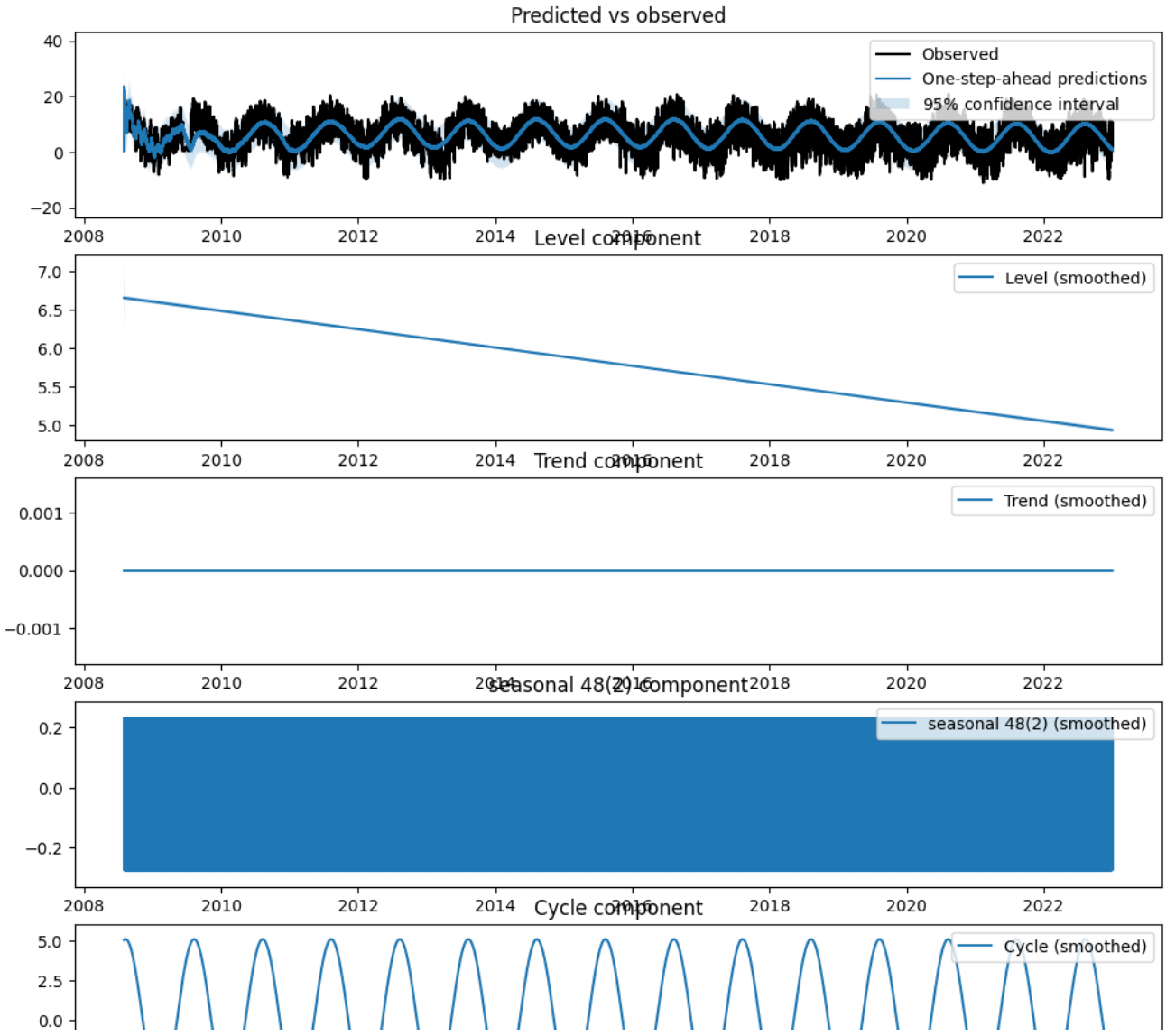
Dep. Variable:	dew.point	No. Observations:	25
Model:	deterministic trend + freq_seasonal(48(2)) + cycle	Log Likelihood	-685958
		AIC	1371921
		BIC	1371941
Date:	Mon, 15 Apr 2024	HQIC	1371927
Time:	18:18:23		
Sample:	08-01-2008 - 12-31-2022		
Covariance Type:	opg		

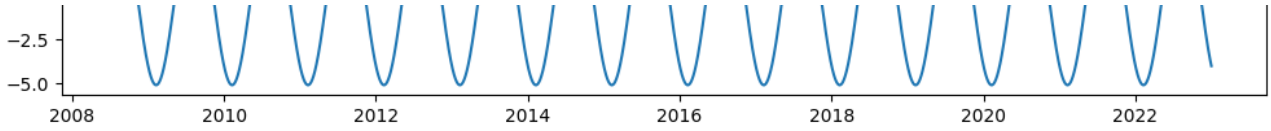
	coef	std err	z	P>|z|	[0.025
sigma2.irregular	13.3196	0.037	360.240	0.000	13.247
frequency.cycle	0.0004	2.76e-08	1.3e+04	0.000	0.000

Ljung-Box (L1) (Q):	229465.85	Jarque-Bera (JB):	3
Prob(Q):	0.00	Prob(JB):	
Heteroskedasticity (H):	0.92	Skew:	
Prob(H) (two-sided):	0.00	Kurtosis:	

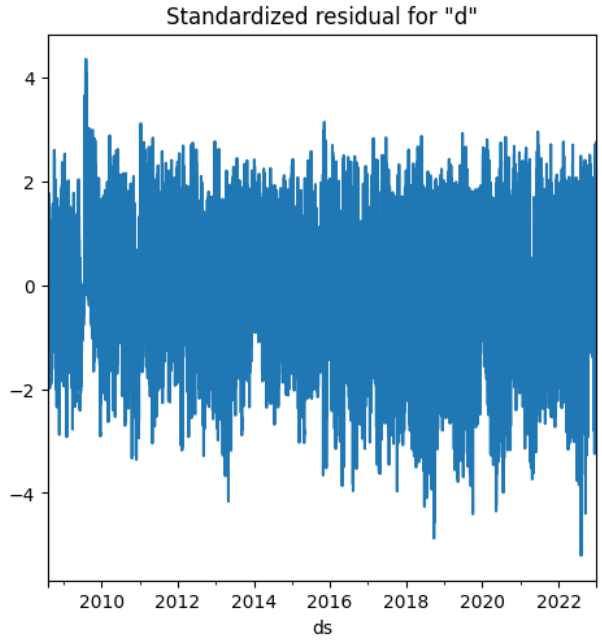
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex

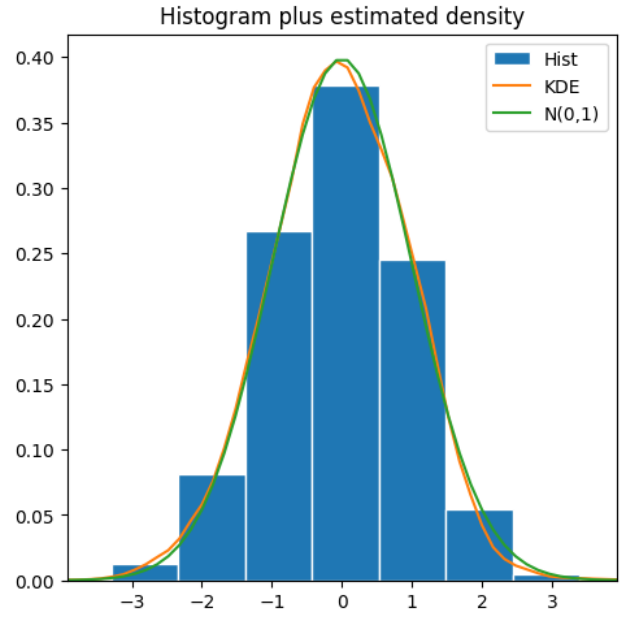
```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

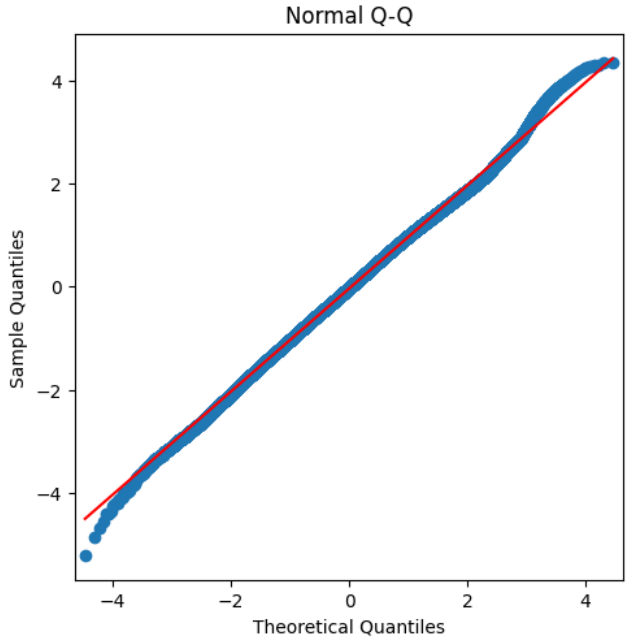


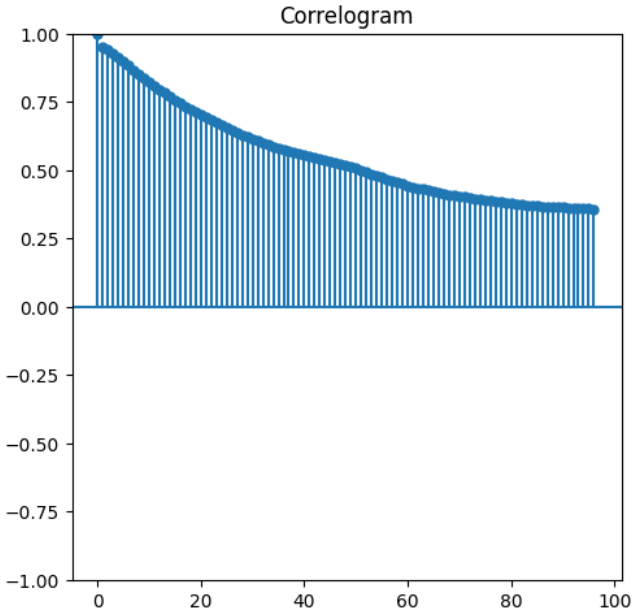


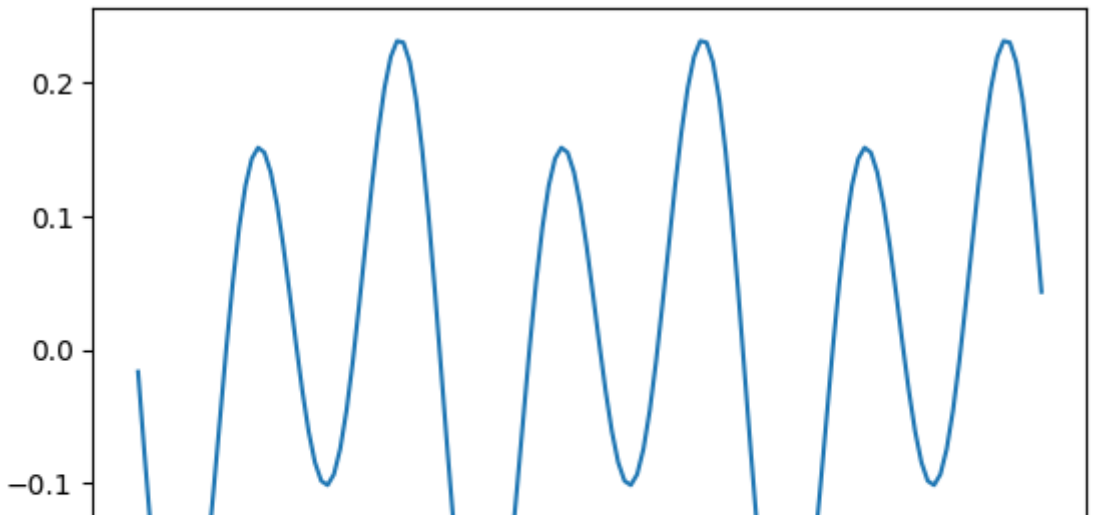
Note: The first 8 observations are not shown, due to approximate diffuse initialization.

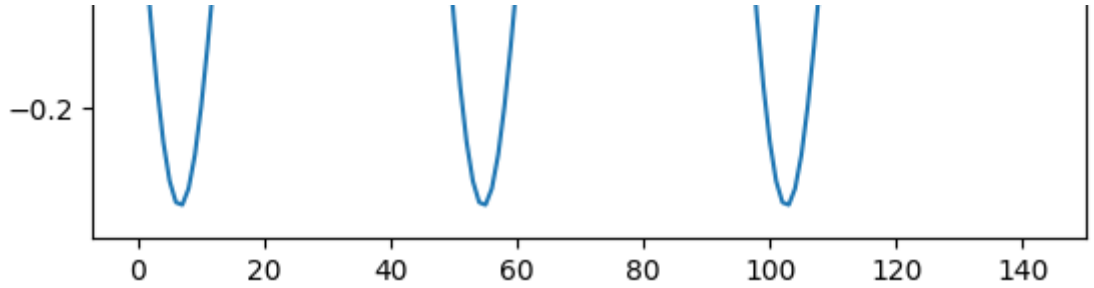













```
df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): True
redundancy before > after: True
mean before feature redundancy: 57.338
mean after feature redundancy:  54.827
```

	before	after	diff
rows	252768	252768	0
cols	85	89	4
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	0	0	0
low_var_rows	0	0	0
low_var_cols	7	7	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

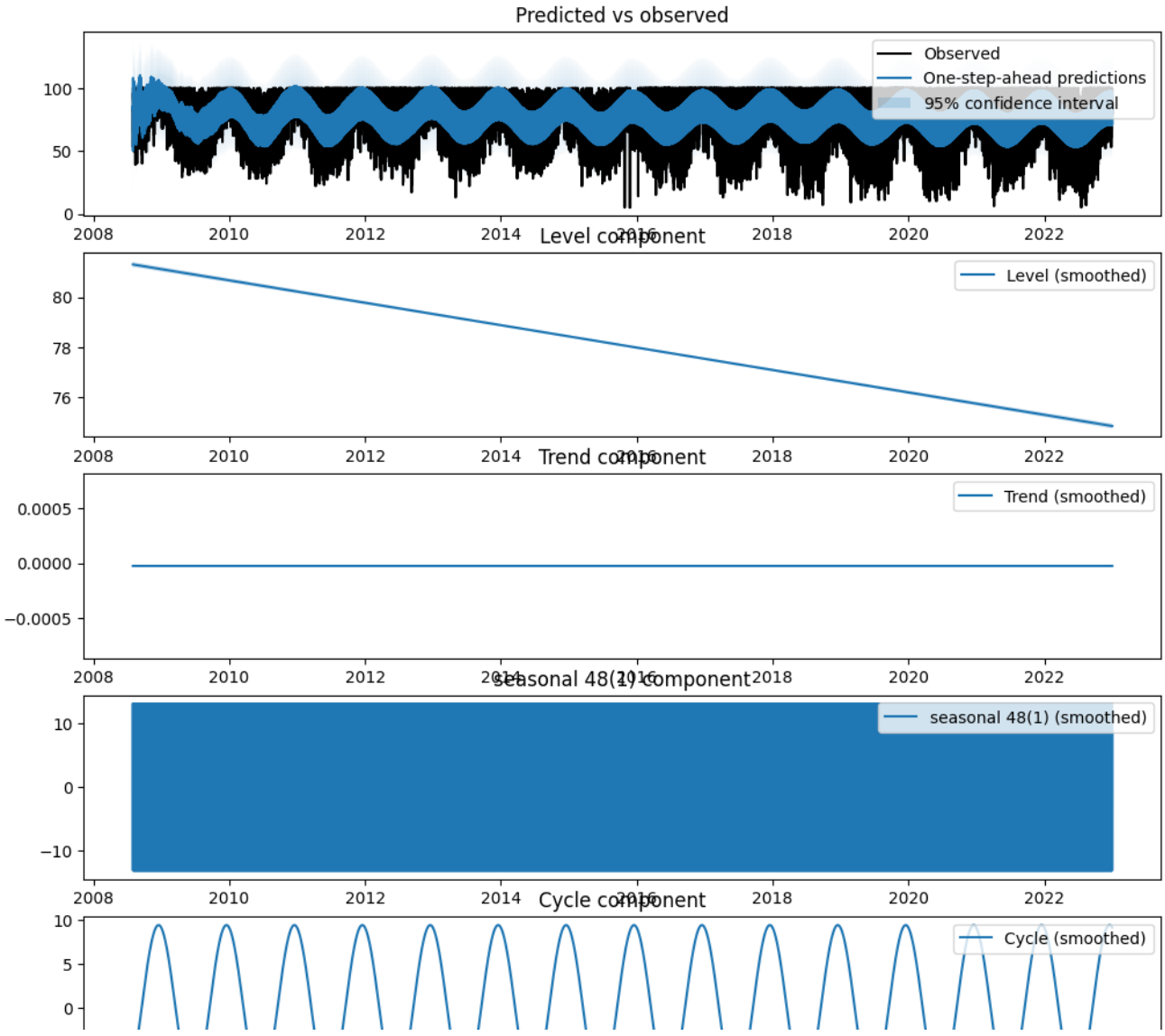
The due point decomposition seemed acceptible. Note the daily bimodal peaks.

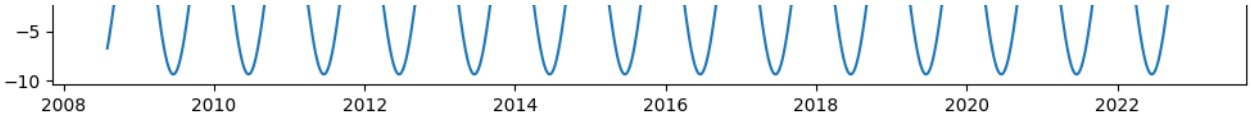
Initial few months of decomposition show high variance.

`humidity` third:

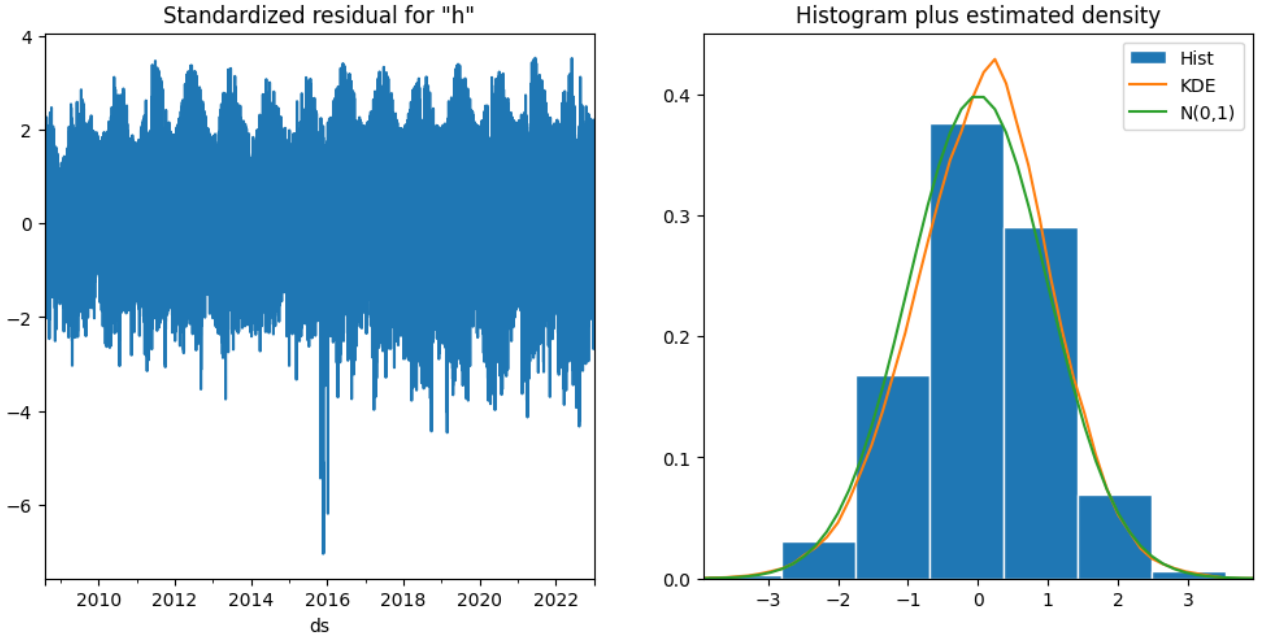
```
uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             # exog = train_df[['humidity','dew.point']].interpolate(method='sline
             # 'exog': train_df[['dew.point_des']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             'cycle': True,
             # irregular = True,
             # 'autoregressive': 1,
             'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  1}]}

res_hum = get_uc_model(train_df, uc_params, 'humidity')
check_uc_model(res_hum)

plt.plot(res_hum.freq_seasonal[0]['smoothed'][:144])
plt.show()
```

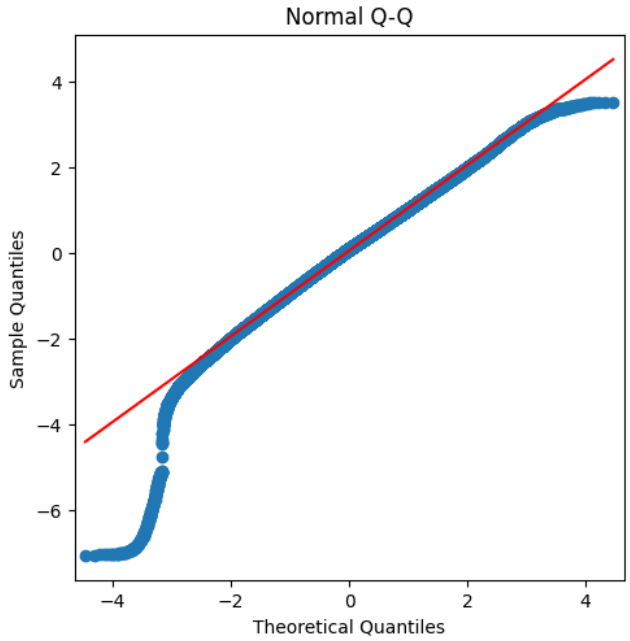
```
                        Unobserved Components Results
==============================================================================
Dep. Variable:                    humidity   No. Observations:                25
Model:                deterministic trend   Log Likelihood           -1011022
                    + freq_seasonal(48(1))   AIC                       2022049
                                 + cycle     BIC                       2022070
Date:                     Mon, 15 Apr 2024   HQIC                      2022055
Time:                             18:19:43
Sample:                         08-01-2008
                              - 12-31-2022
Covariance Type:                      opg
==============================================================================
                      coef    std err          z      P>|z|      [0.025
------------------------------------------------------------------------------
sigma2.irregular   174.4179      0.397    439.783      0.000     173.641      1
frequency.cycle      0.0004   5.25e-08   6832.922      0.000       0.000
==============================================================================
Ljung-Box (L1) (Q):                   214117.41   Jarque-Bera (JB):               170
Prob(Q):                                  0.00   Prob(JB):
Heteroskedasticity (H):                   1.31   Skew:
Prob(H) (two-sided):                      0.00   Kurtosis:
==============================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

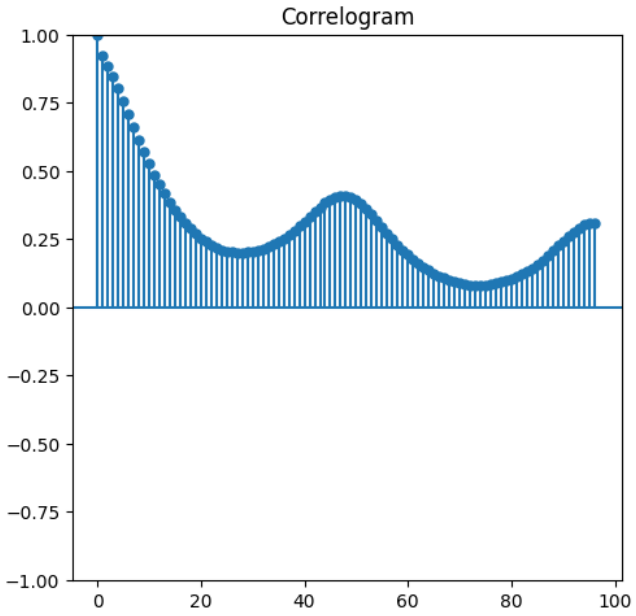


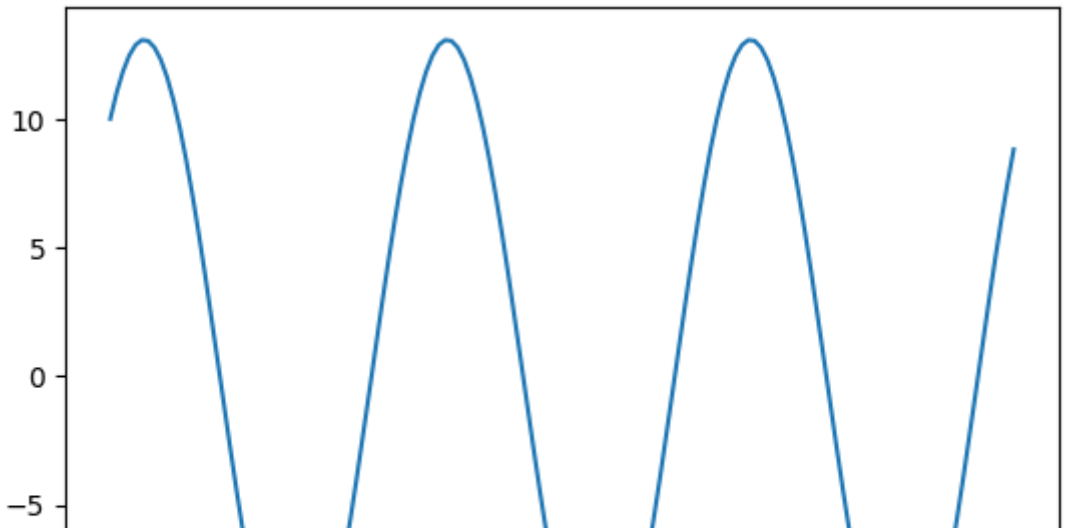


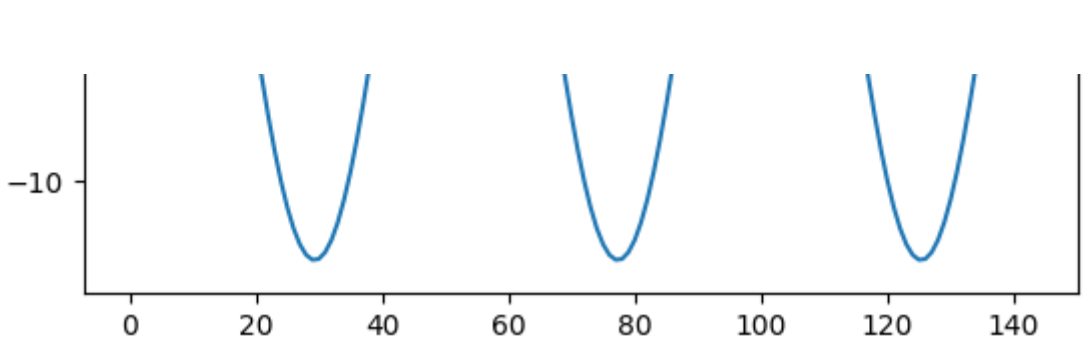
Note: The first 6 observations are not shown, due to approximate diffuse initialization.











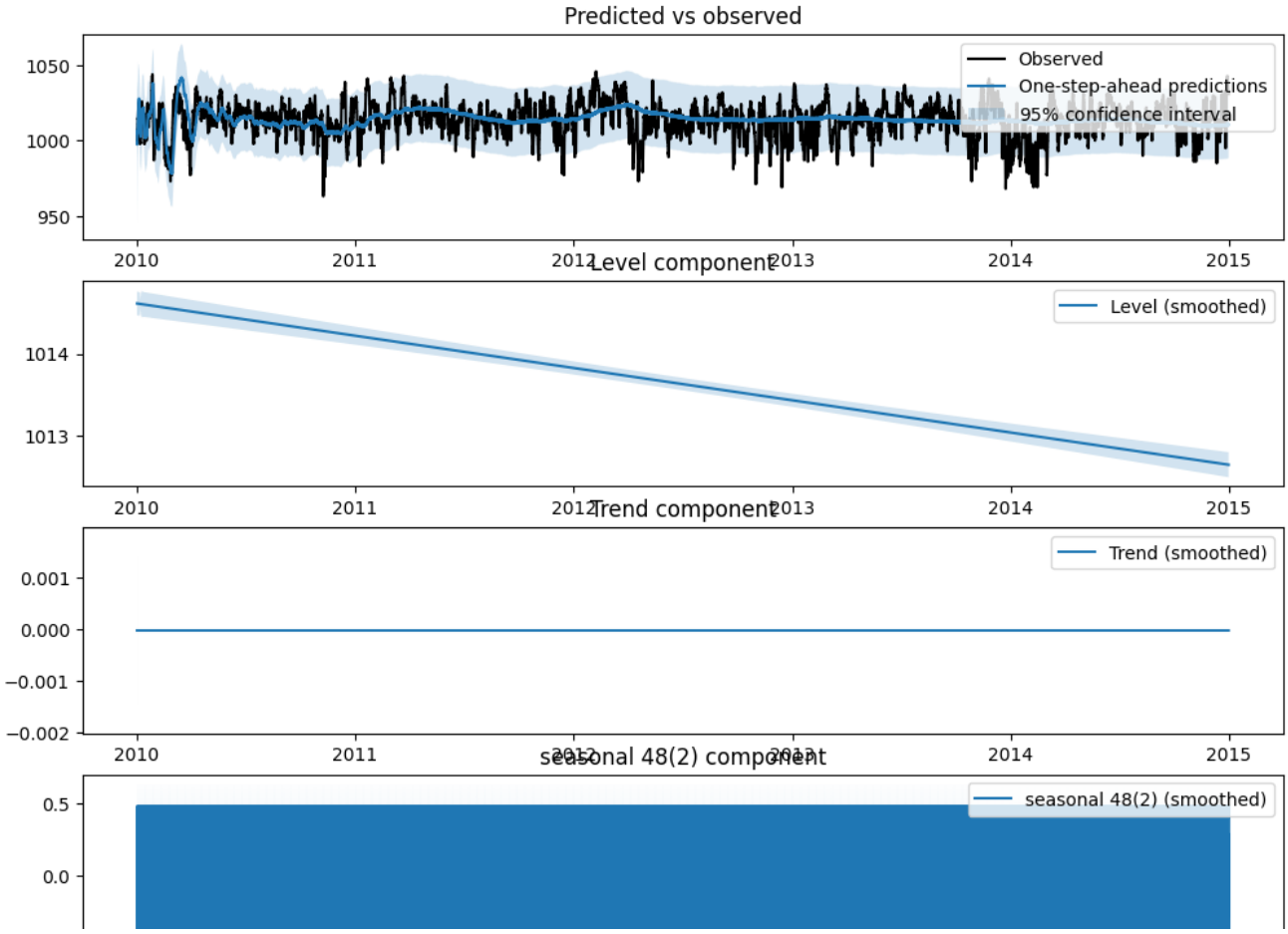
The `humidity` decomposition is problematic.

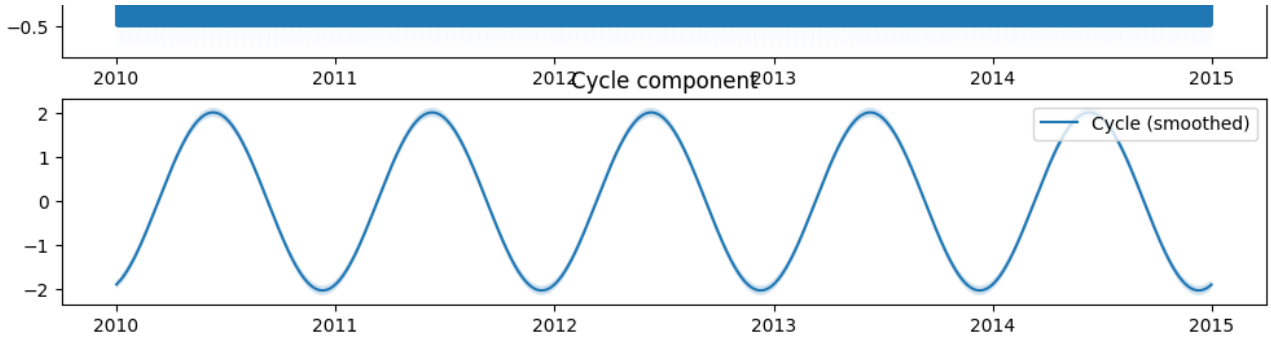
Initial few months of decomposition show high variance.

`pressure` fourth:

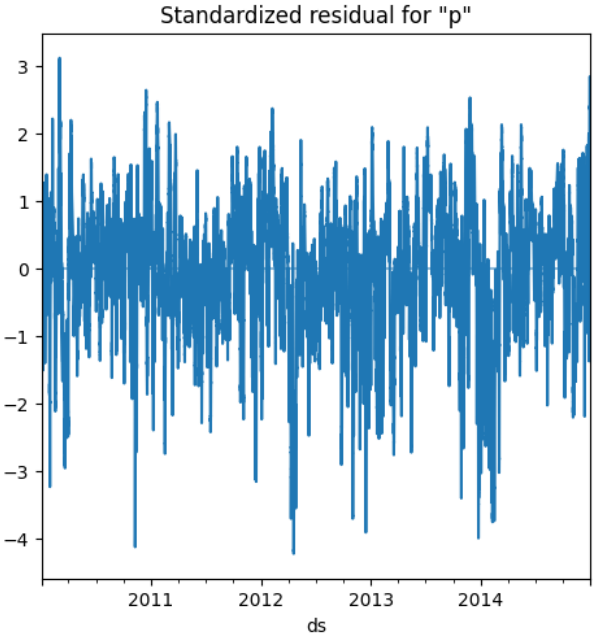
```
uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             # exog = train_df[['humidity','dew.point']].interpolate(method='sline
             # 'exog': train_df[['dew.point_des']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             #'level': 'smooth trend',
             #'level': 'local linear trend',
             'cycle': True,
             # irregular = True,
             # 'autoregressive': 1,
             'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  2}]}

press_df = df[(df.index.year >= 2010) & (df.index.year < 2015)]
press_df['log_pressure'] = np.log(press_df['pressure'])
res_press = get_uc_model(press_df, uc_params, 'pressure')
check_uc_model(res_press)

plt.plot(res_press.freq_seasonal[0]['smoothed'][:144])
plt.show()
```

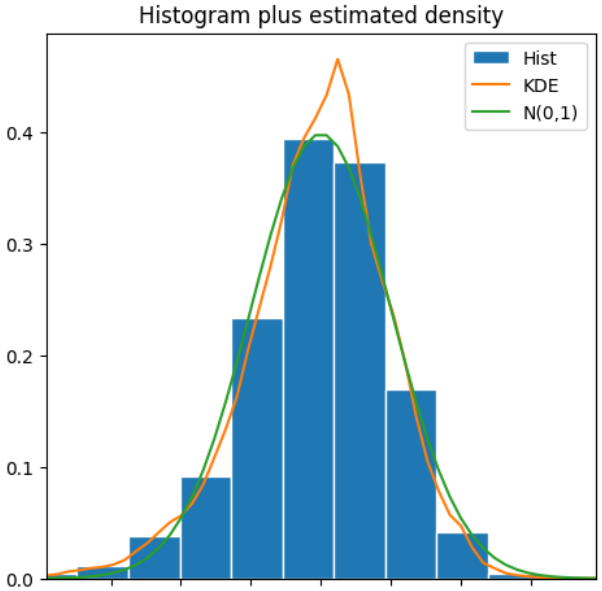
```
<ipython-input-10-73d5c880cf37>:17: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  press_df['log_pressure'] = np.log(press_df['pressure'])
                           Unobserved Components Results
==============================================================================
Dep. Variable:                    pressure   No. Observations:                8
Model:                deterministic trend   Log Likelihood             -338229
                    + freq_seasonal(48(2))   AIC                         676463
                                 + cycle     BIC                         676481
Date:                     Mon, 15 Apr 2024   HQIC                        676468
Time:                             18:20:19
Sample:                         01-01-2010
                              - 12-31-2014
Covariance Type:                      opg
==============================================================================
                      coef    std err          z      P>|z|      [0.025      1
------------------------------------------------------------------------------
sigma2.irregular   131.5175      0.559    235.361      0.000     130.422      1
frequency.cycle      0.0004   9.82e-07    364.800      0.000       0.000
==============================================================================
Ljung-Box (L1) (Q):                87458.26   Jarque-Bera (JB):                42
Prob(Q):                               0.00   Prob(JB):
Heteroskedasticity (H):                1.23   Skew:
Prob(H) (two-sided):                   0.00   Kurtosis:
==============================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

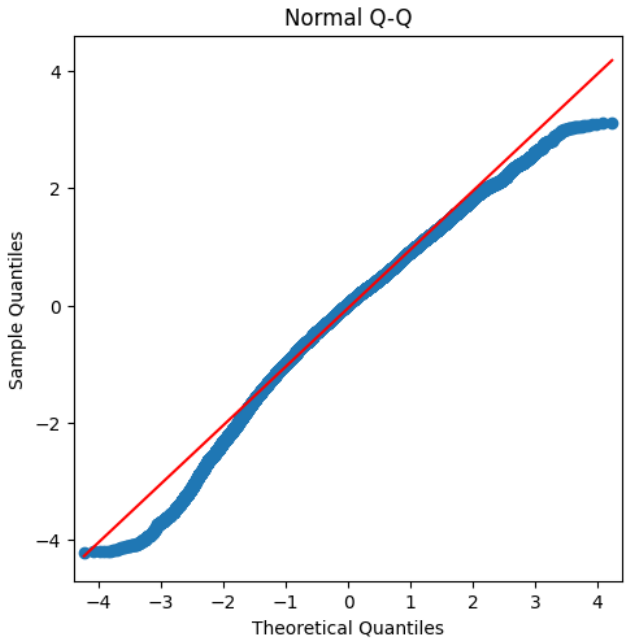


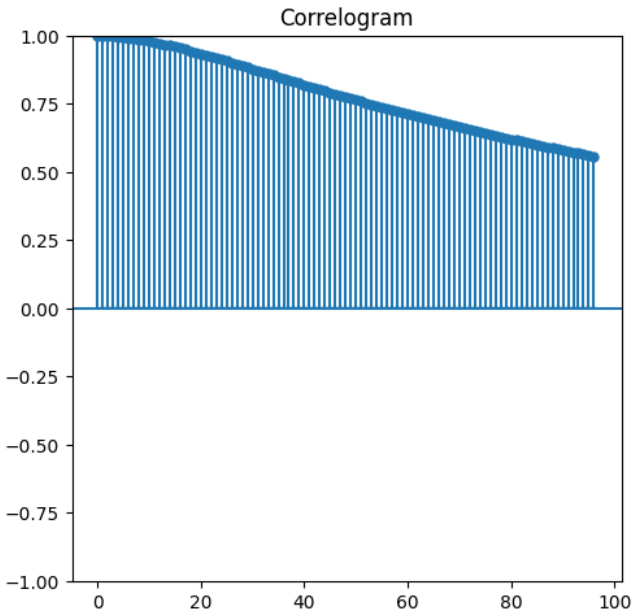


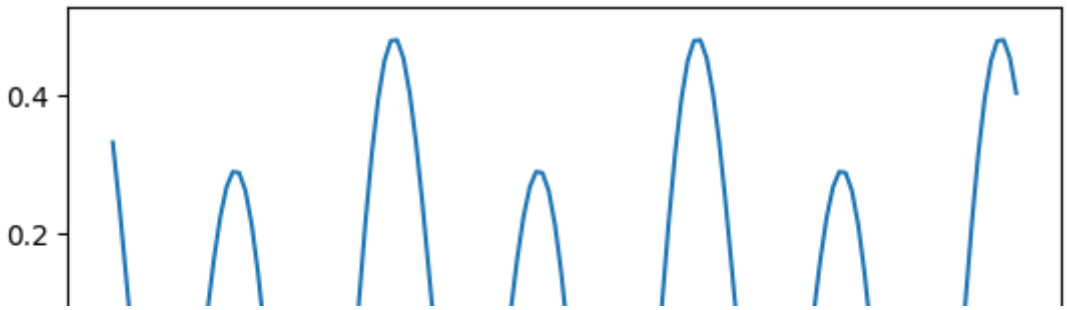
Note: The first 8 observations are not shown, due to approximate diffuse initialization.

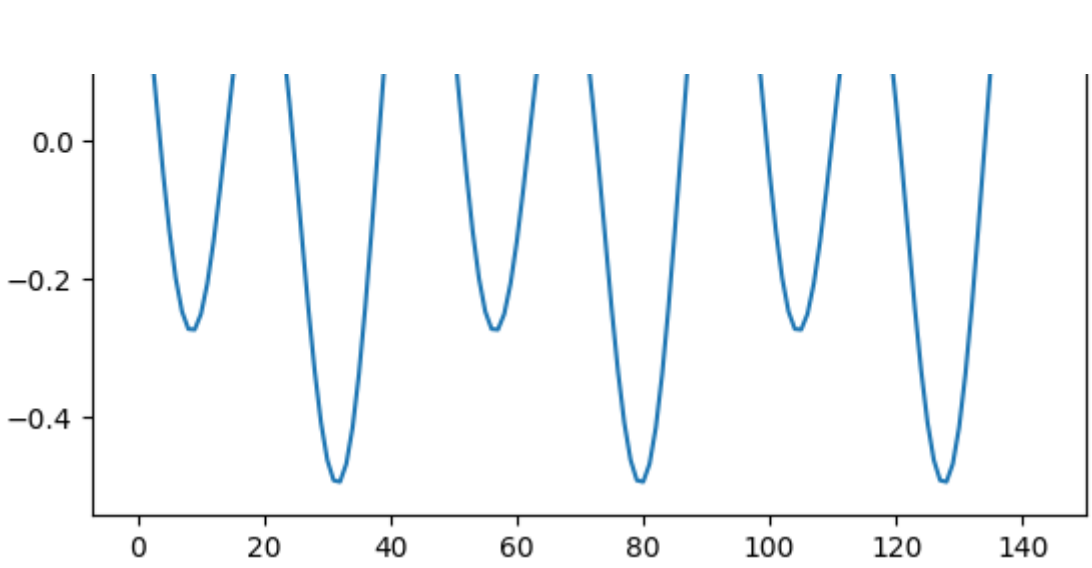











0.0
−0.2
−0.4
0
20
40
60
80
100
120
140

The `pressure` decomposition is problematic.

Initial few months of decomposition show high variance.

Try building UCM for `pressure` without annual seasonal cycle:

```
uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             # exog = train_df[['humidity','dew.point']].interpolate(method='sline
             # 'exog': train_df[['dew.point_des']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             #'level': 'smooth trend',
             #'level': 'local linear trend',
             'cycle': False,
             # irregular = True,
             # 'autoregressive': 1,
             # 'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  2}]}

press_df = df[(df.index.year >= 2010) & (df.index.year < 2015)]
# press_df['log_pressure'] = np.log(press_df['pressure'])
res_press = get_uc_model(press_df, uc_params, 'pressure')
check_uc_model(res_press)

plt.plot(res_press.freq_seasonal[0]['smoothed'][:144])
plt.show()
```

Unobserved Components Results

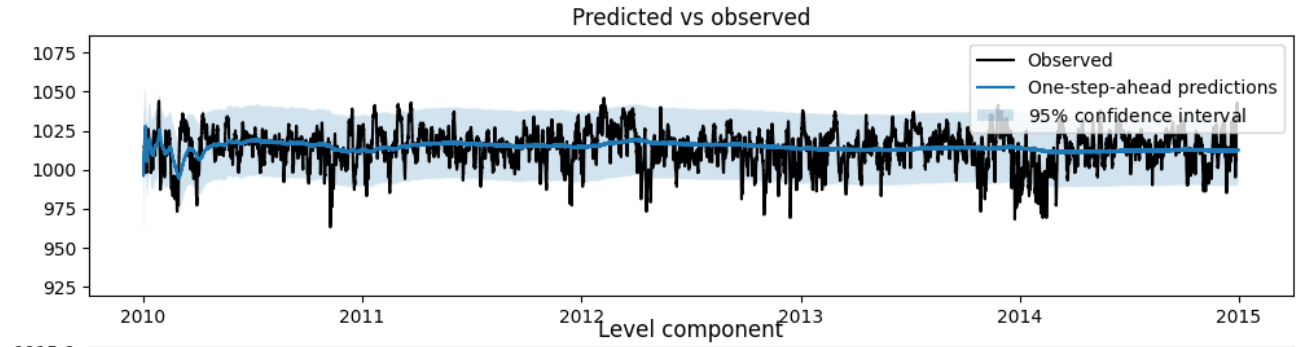
Dep. Variable:	pressure	No. Observations:	8
Model:	deterministic trend + freq_seasonal(48(2))	Log Likelihood	-338895
		AIC	677793
Date:	Mon, 15 Apr 2024	BIC	677802
Time:	18:20:33	HQIC	677796
Sample:	01-01-2010 - 12-31-2014		
Covariance Type:	opg		

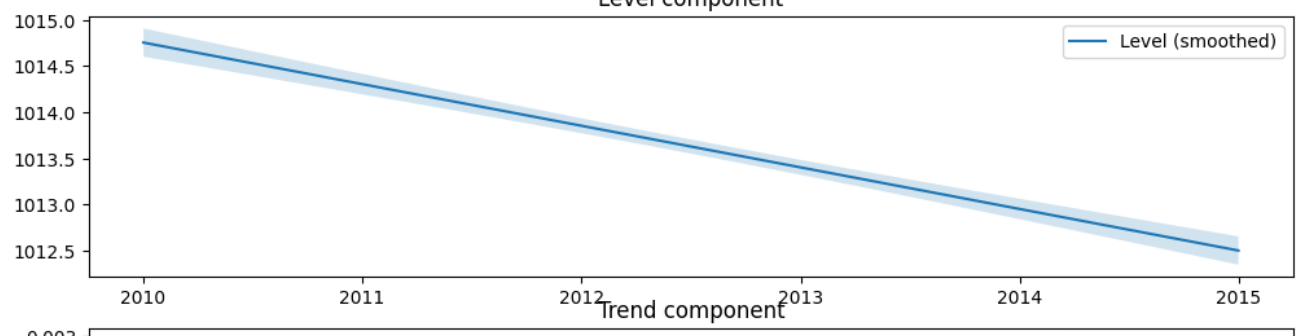
	coef	std err	z	P>|z|	[0.025	
sigma2.irregular	133.5580	0.556	240.276	0.000	132.469	1

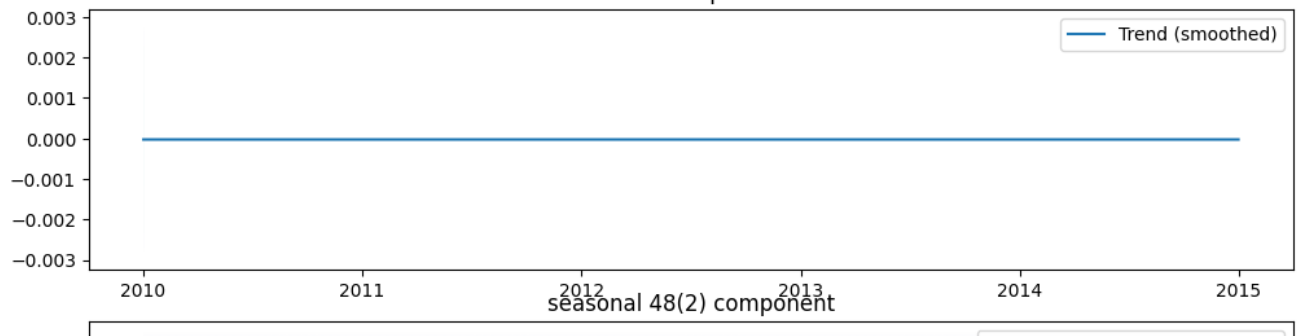
Ljung-Box (L1) (Q):	87463.45	Jarque-Bera (JB):	29
Prob(Q):	0.00	Prob(JB):	
Heteroskedasticity (H):	1.20	Skew:	
Prob(H) (two-sided):	0.00	Kurtosis:	

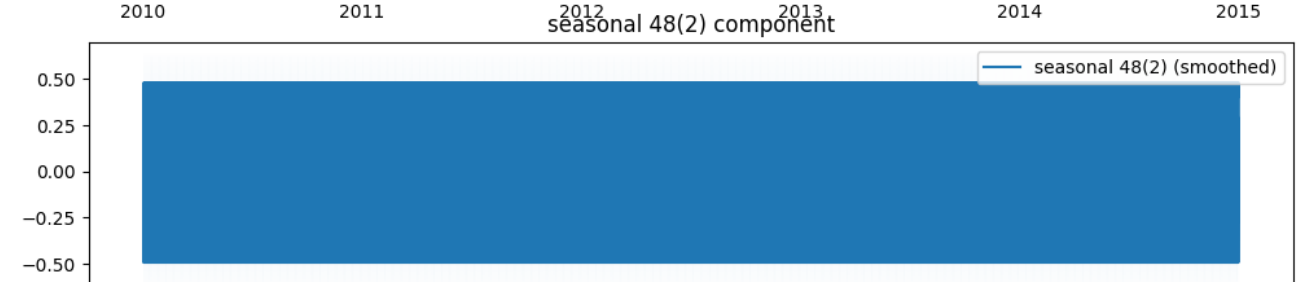
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

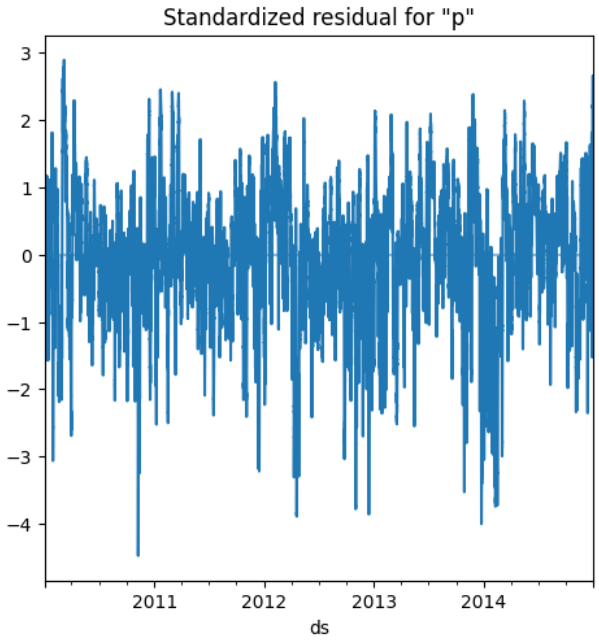


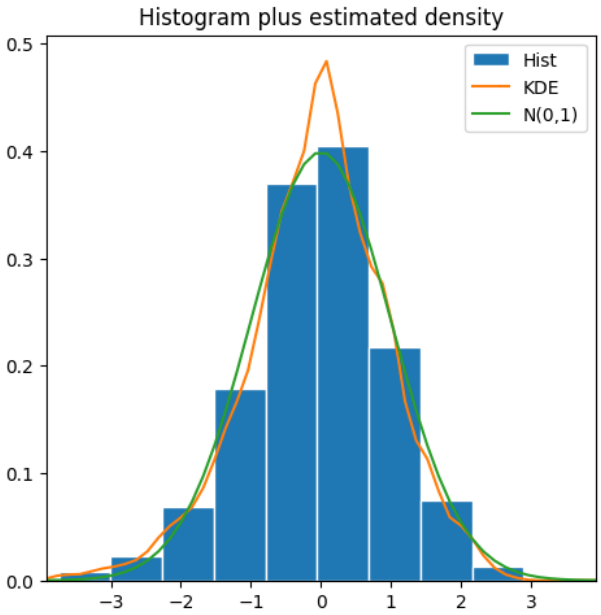


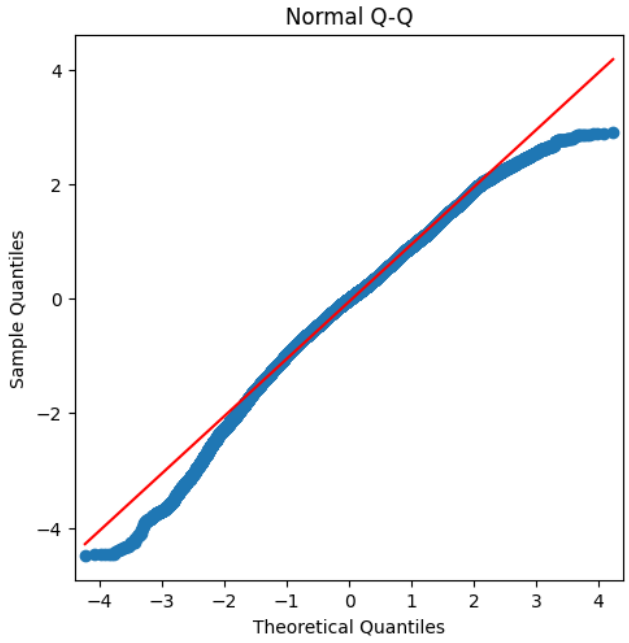


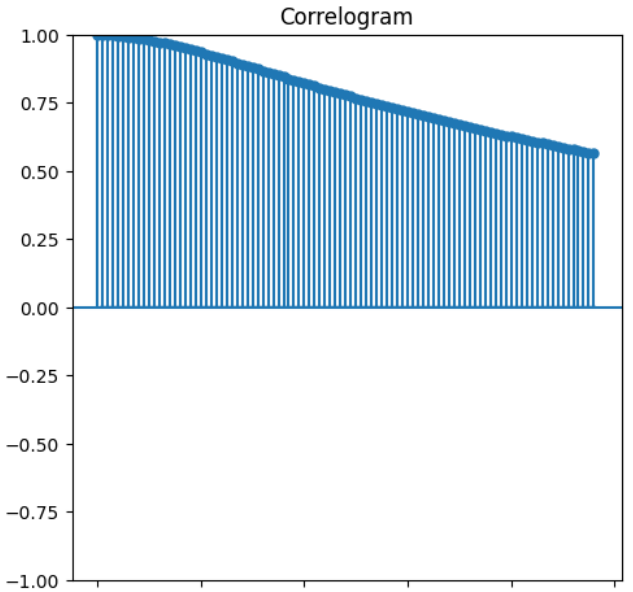


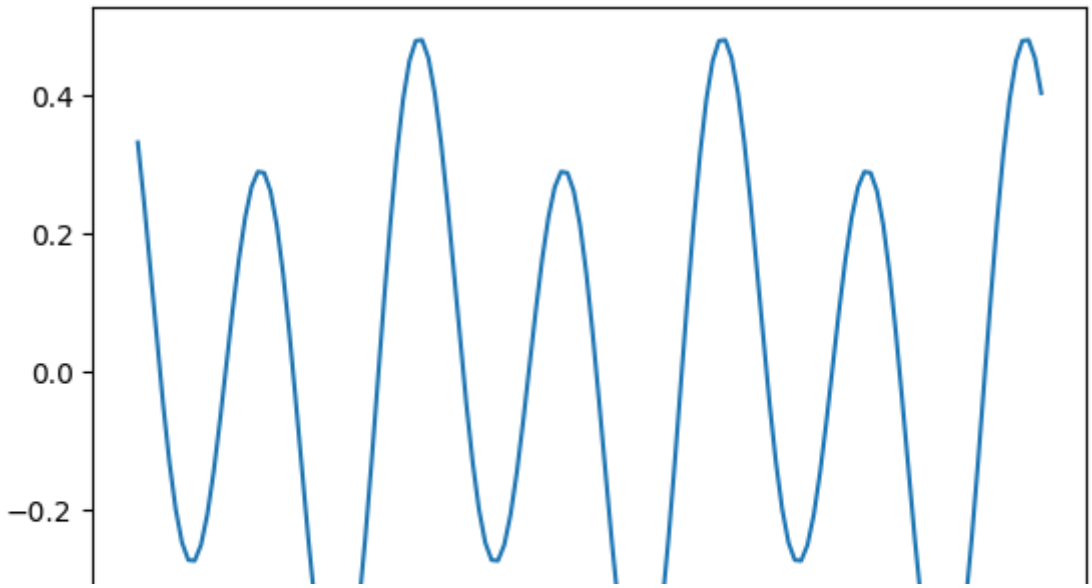
Note: The first 6 observations are not shown, due to approximate diffuse initialization.

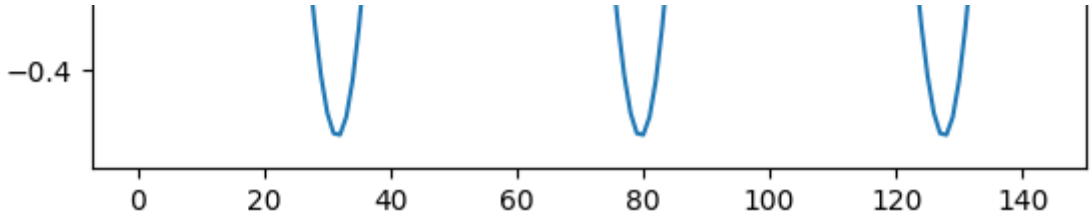












This `pressure` decomposition without annual seasonal cycle is still problematic. The daily seasonality does not capture much of the variance.

Initial few months of decomposition show high variance.

Try building UCM for `pressure` without annual seasonal cycle and separate 12 and 24 hour seasonalities:

```
uc_params = {# exog  = df[['humidity','dew.point','pressure','wind.speed.mean.x',
             # exog = train_df[['humidity','dew.point']].interpolate(method='sline
             # 'exog': train_df[['dew.point_des']].interpolate(method='linear'),
             # level = 'deterministic constant',
             'level': 'deterministic trend',
             #'level': 'smooth trend',
             #'level': 'local linear trend',
             'cycle': False,
             # irregular = True,
             # 'autoregressive': 1,
             # 'cycle_period_bounds': (17531, 17533),
             'stochastic_freq_seasonal': [False, False],
             'freq_seasonal': [{'period':    48,
                                'harmonics':  1},
                               {'period':    24,
                                'harmonics':  1}]}

press_df = df[(df.index.year >= 2010) & (df.index.year < 2015)]
# press_df['log_pressure'] = np.log(press_df['pressure'])
res_press = get_uc_model(press_df, uc_params, 'pressure')
check_uc_model(res_press)

plt.plot(res_press.freq_seasonal[0]['smoothed'][:144])
plt.title('24 hour seaonality')
plt.show()
plt.plot(res_press.freq_seasonal[1]['smoothed'][:144])
plt.title('12 hour seaonality')
plt.show()
```

Unobserved Components Results

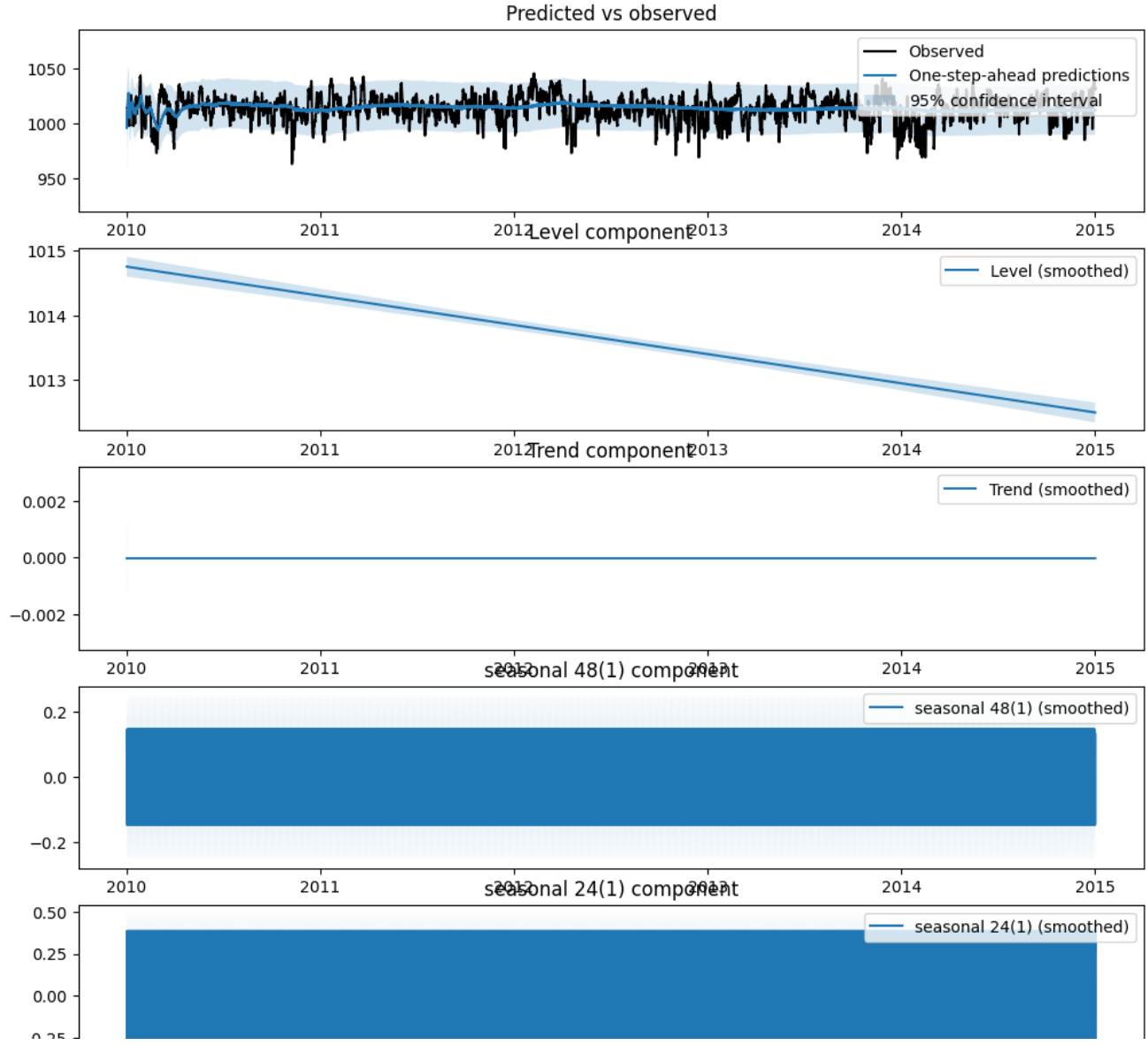
Dep. Variable:	pressure	No. Observations:	8
Model:	deterministic trend + freq_seasonal(48(1)) + freq_seasonal(24(1))	Log Likelihood	-338895
		AIC	677793
		BIC	677802
Date:	Mon, 15 Apr 2024	HQIC	677796
Time:	18:24:42		
Sample:	01-01-2010 - 12-31-2014		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	
sigma2.irregular	133.5580	0.556	240.276	0.000	132.469	1

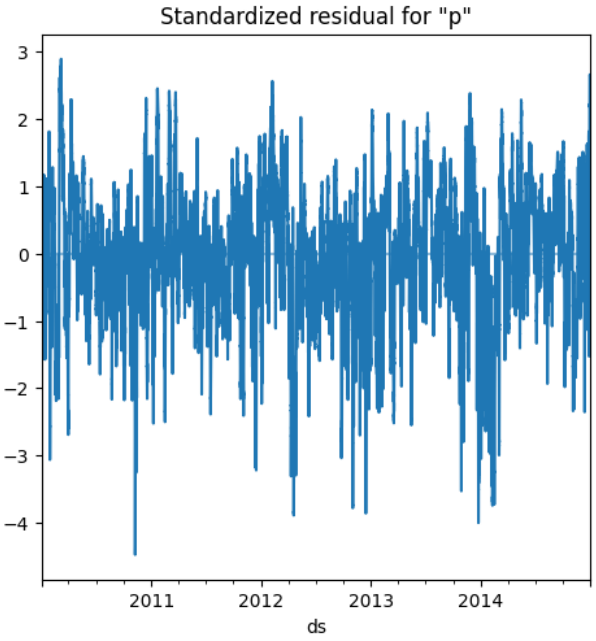
Ljung-Box (L1) (Q):	87463.45	Jarque-Bera (JB):	29
Prob(Q):	0.00	Prob(JB):	
Heteroskedasticity (H):	1.20	Skew:	
Prob(H) (two-sided):	0.00	Kurtosis:	

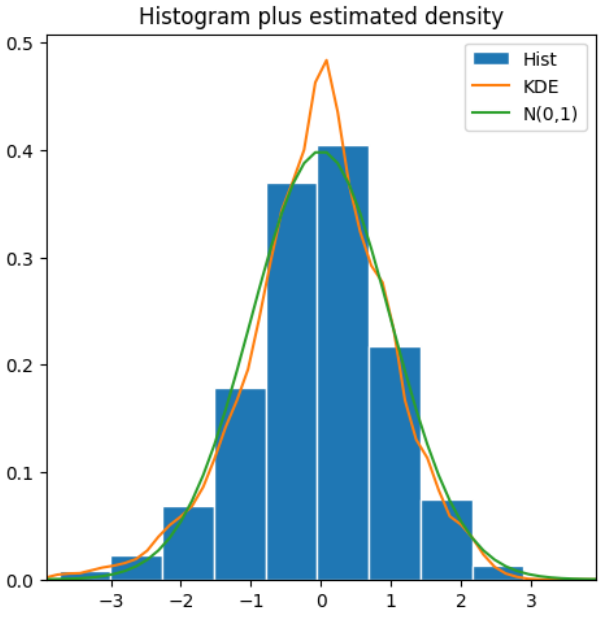
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex

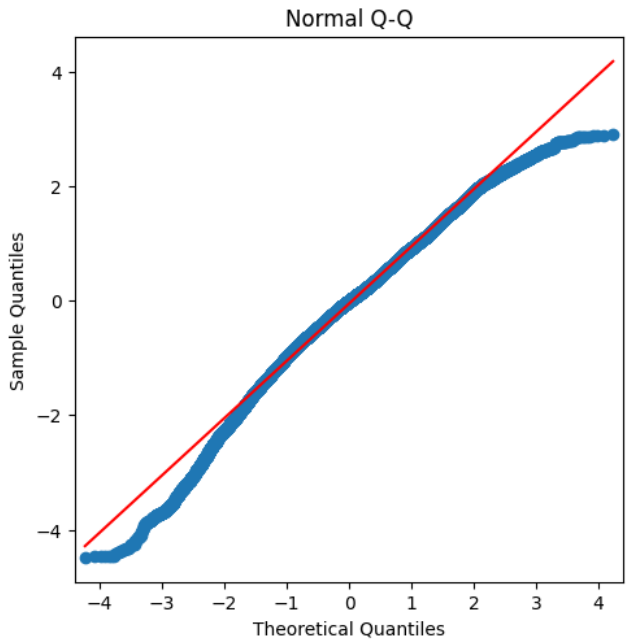
```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/structural.
  std_errors = np.sqrt(component_bunch['%s_cov' % which])
```

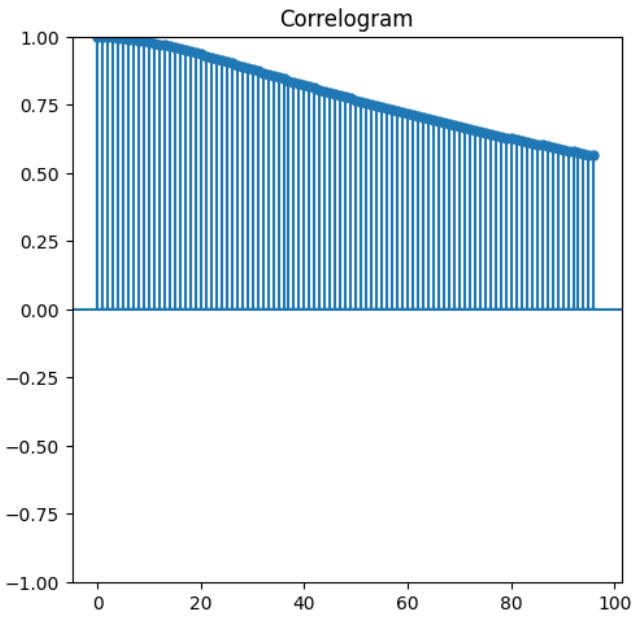


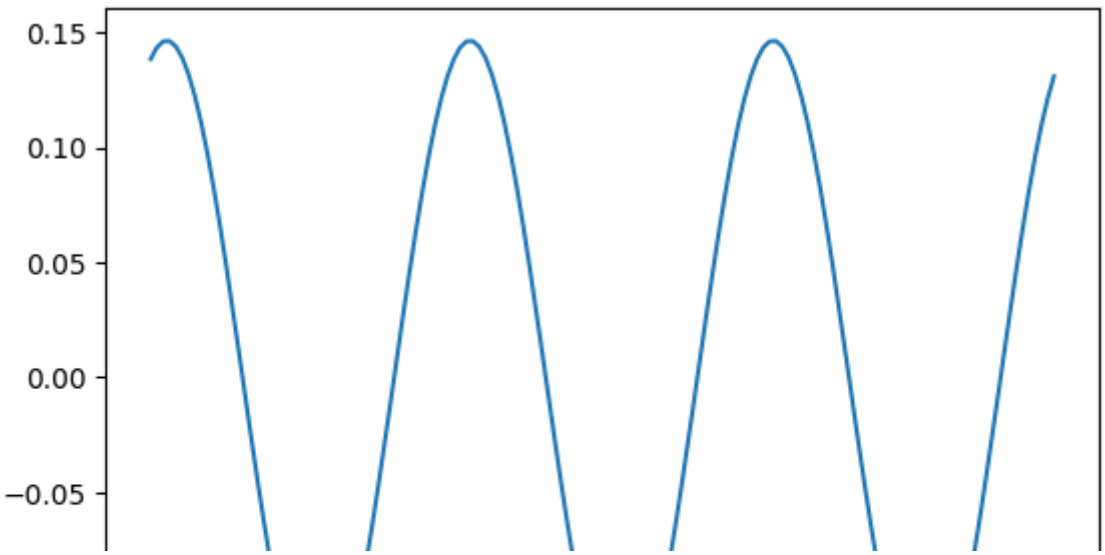
Note: The first 6 observations are not shown, due to approximate diffuse initialization.

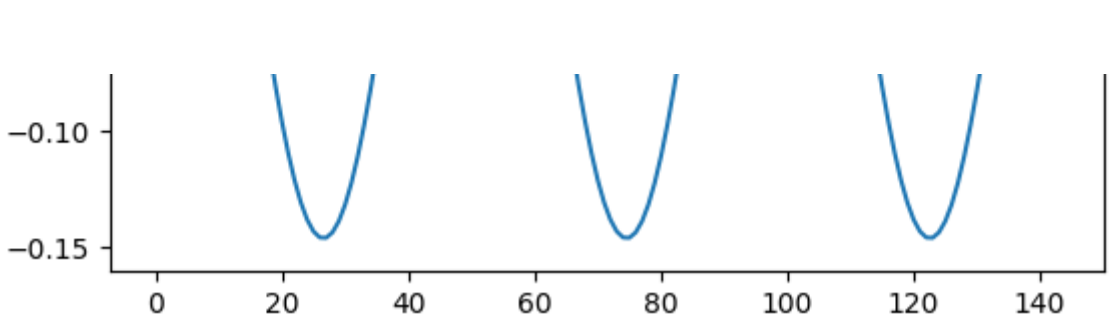


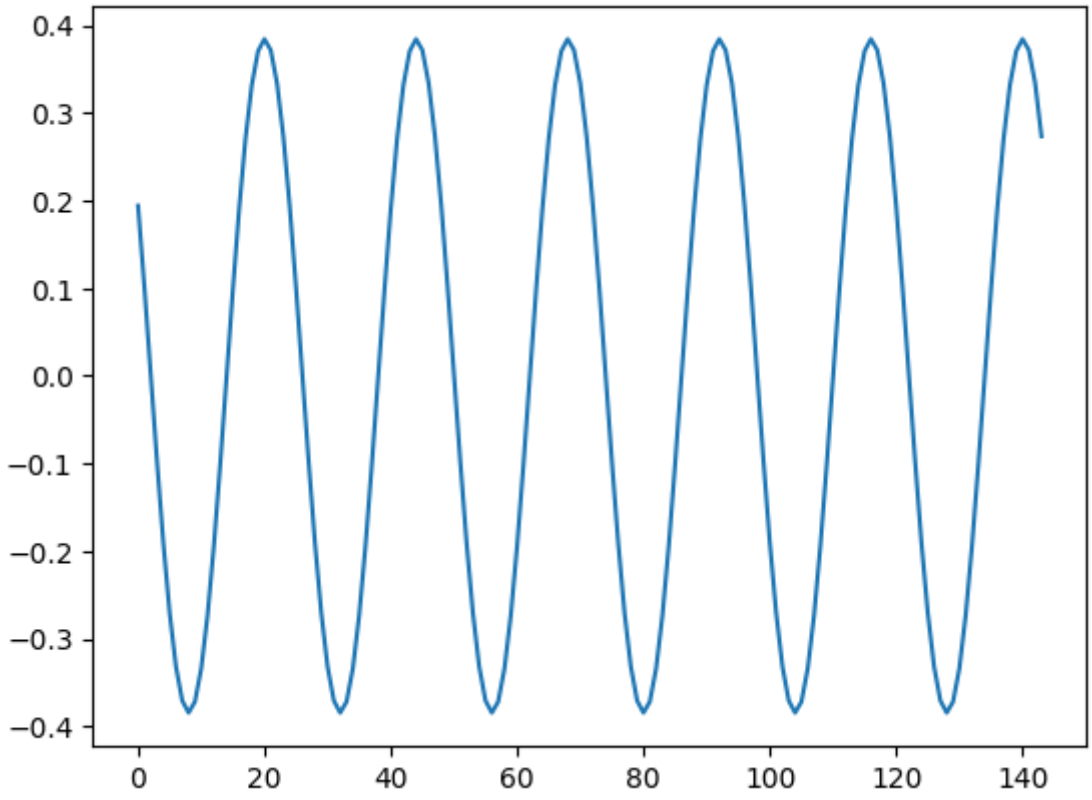









−0.10
−0.15
0
20
40
60
80
100
120
140


0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0
20
40
60
80
100
120
140

The 12 hour seasonality has a higher amplitude than the 24 hour seaonality for `pressure`. It may be better to use separate 12 and 24 hour seaonalities with single harmonics instead of a single 24 hour seaonality with 2 harmonics.

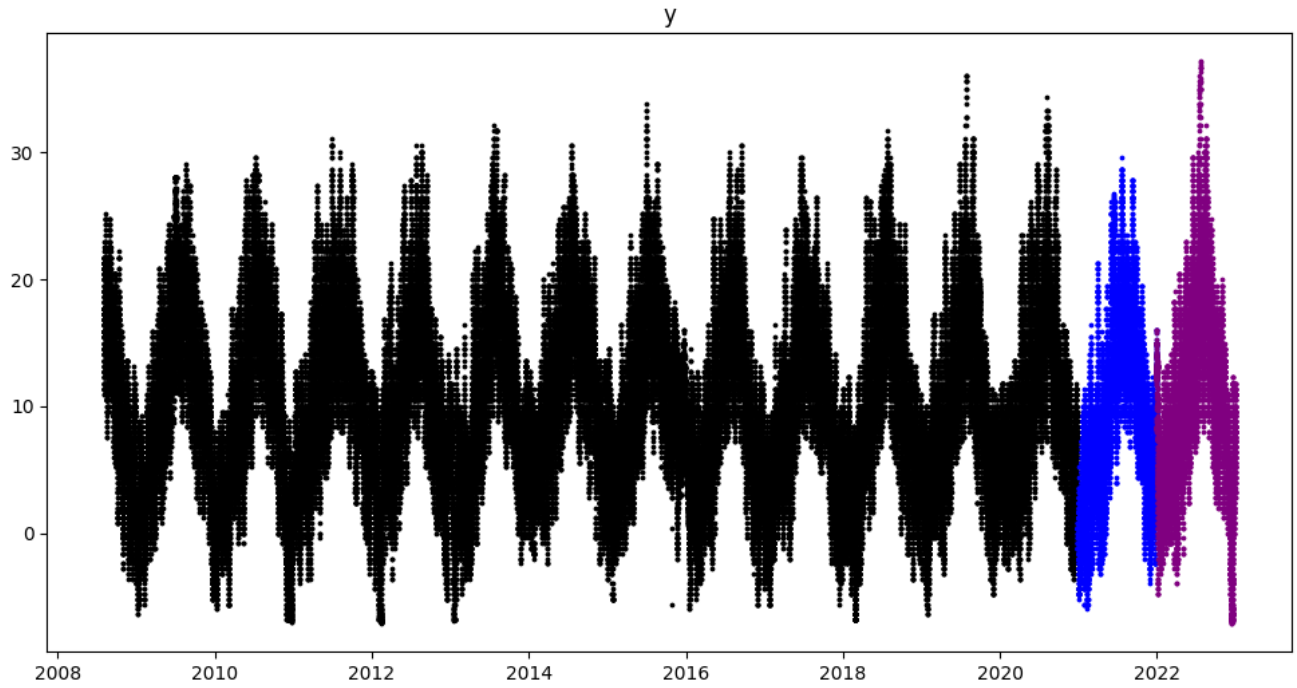
Initial few months of decomposition show high variance.

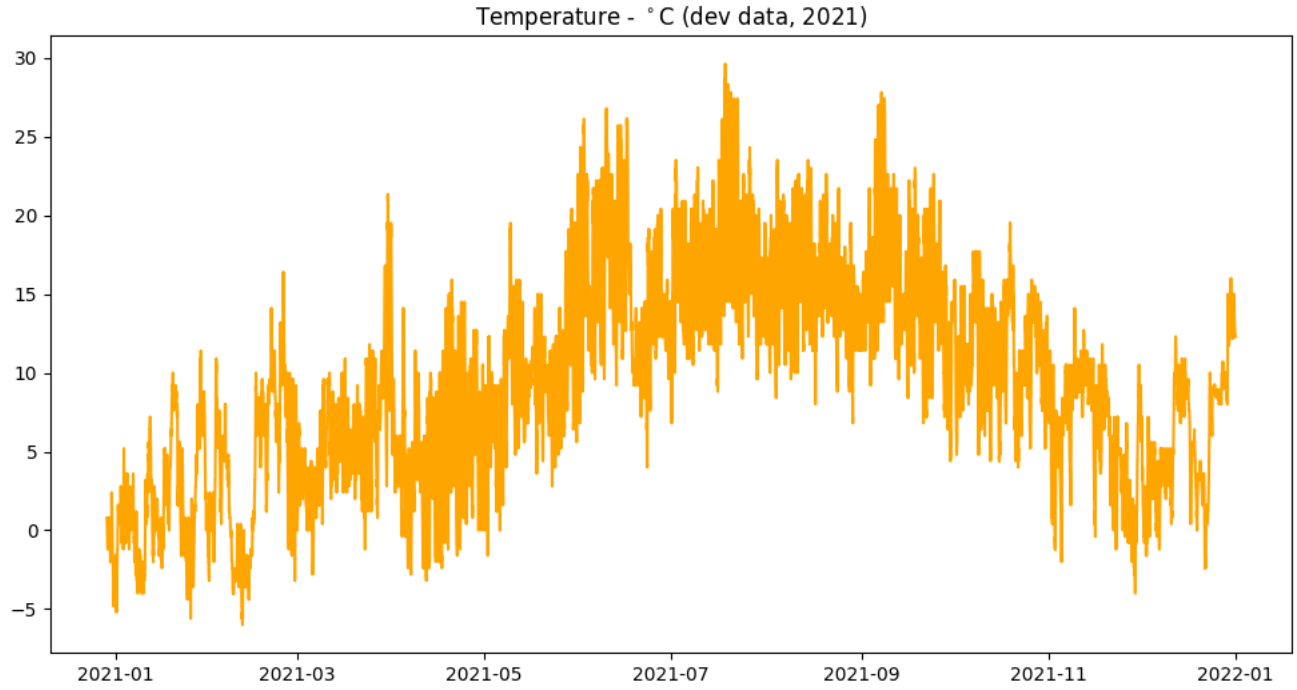
Split Data

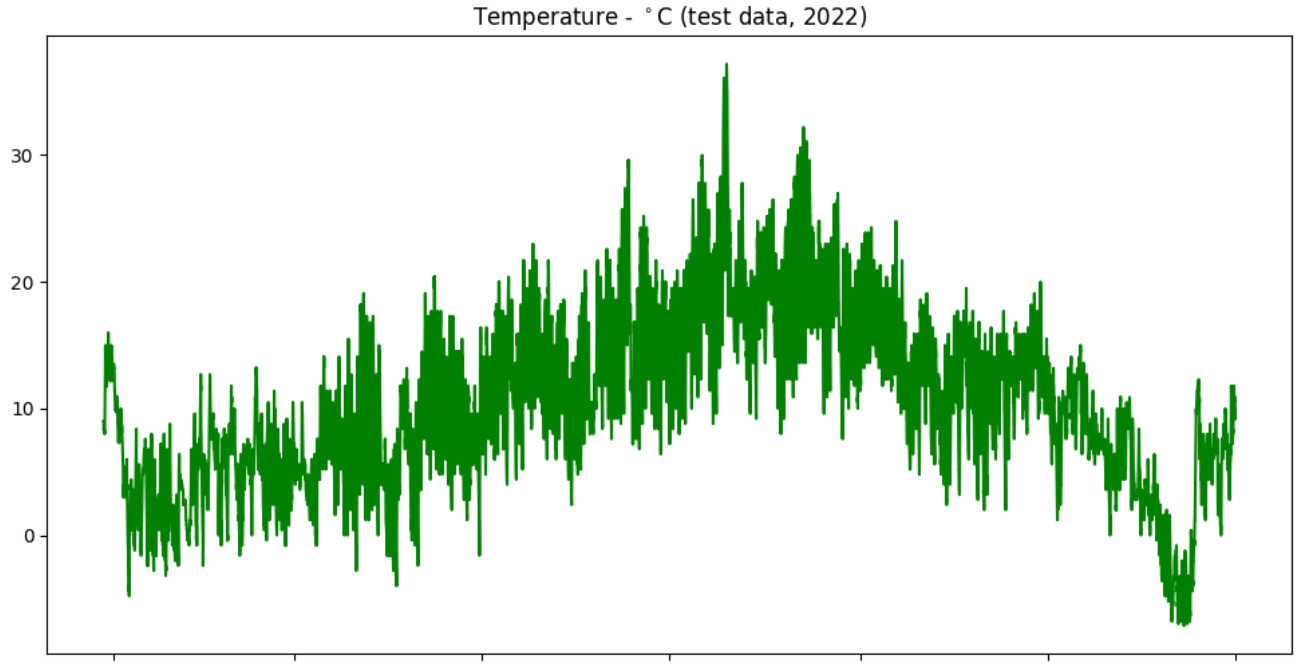
I use data from 2021 for validation, 2022 for testing and data before 2021 for training. These are entirely arbitrary choices. This results in an approximate 88%, 6%, 6% split for the training, validation, and test sets respectively.

```
# df['year'] = df['ds'].dt.year
# WARN: Crucially important valid & test data are both after train data to
#       avoid data leakage
# train_df = df.loc[(df['year'] != VALID_YEAR) & (df['year'] != TEST_YEAR)]
train_df = df.loc[df['year'] < min(VALID_YEAR, TEST_YEAR)]
valid_df = df.loc[(df['ds'] >= '2020-12-29') & (df['ds'] < '2022-01-01')]
test_df  = df.loc[(df['ds'] >= '2021-12-29') & (df['ds'] < '2023-01-01')]
# valid_df = df.loc[df['year'] == VALID_YEAR]
# test_df  = df.loc[df['year'] == TEST_YEAR]

#train_df.loc[train_df['ds'].dt.year >= 2018, 'ds'] = train_df.loc[train_df['ds'].
#train_df.set_index('ds', drop=False, inplace=True)
#train_df = train_df[~train_df.index.duplicated(keep='first')]


plot_feature_history(train_df, valid_df, test_df, 'y')

plt.figure(figsize = (12, 6))
plt.plot(valid_df.ds, valid_df.y, color='orange')
plt_title = 'Temperature - $^\circ$C (dev data, ' + str(VALID_YEAR) + ')'
plt.title(plt_title)
plt.show()

plt.figure(figsize = (12, 6))
```

```
plt.plot(test_df.ds, test_df.y, color='green')
plt_title = 'Temperature - $^\circ$C (test data, ' + str(TEST_YEAR) + ')'
plt.title(plt_title)
plt.show()


del_cols = ['year']
train_df = train_df.drop(del_cols, axis = 1)
valid_df = valid_df.drop(del_cols, axis = 1)
test_df  = test_df.drop(del_cols,  axis = 1)
df = df.drop(['year'], axis = 1)

# DO NOT USE ffill with train_df - ffills 2018 & 2019!
# train_df = train_df.asfreq(freq='30min', method='ffill')
train_df = train_df.asfreq(freq='30min', fill_value=np.nan)

# DO USE ffill with valid_df, test_df
# avoids missing value errors when calculating metrics etc
valid_df = valid_df.asfreq(freq='30min', fill_value=np.nan)
test_df  =  test_df.asfreq(freq='30min', fill_value=np.nan)
# valid_df = valid_df.asfreq(freq='30min', method='ffill')
# test_df  =  test_df.asfreq(freq='30min', method='ffill')

print_train_valid_test_shapes(df, train_df, valid_df, test_df)
sanity_check_train_valid_test(train_df, valid_df, test_df)
print_null_columns(train_df, 'train_df')
print_null_columns(valid_df, 'valid_df')
print_null_columns(test_df,  'test_df')
```


y
30
20
10
0
2008
2010
2012
2014
2016
2018
2020
2022


Temperature - °C (dev data, 2021)
30
25
20
15
10
5
0
−5
2021-01
2021-03
2021-05
2021-07
2021-09
2021-11
2022-01


Temperature - °C (test data, 2022)
30
20
10
0

```
df shape:  (252768, 88)
train shape:    (217728, 88)
valid shape:    (17664, 88)
test shape:     (17664, 88)
ERROR: Overlap between train_df, valid_df indices!
max(train_df.index): 2020-12-31 23:30:00
min(valid_df.index): 2021-12-31 23:30:00
ERROR: Overlap between valid_df, test_df indices!
valid_df.index: 2021-12-31 23:30:00 - 2021-12-31 23:30:00
test_df.index: 2022-12-31 23:30:00 - 2022-12-31 23:30:00
ERROR: valid_df should be 1 year long [ 17520 , 17568 ]!
valid_df observations: 17664
ERROR: test_df should be 1 year long [ 17520 , 17568 ]!
test_df observations: 17664

 train_df null columns:
Series([], dtype: float64)

 valid_df null columns:
Series([], dtype: float64)

 test_df null columns:
Series([], dtype: float64)
```

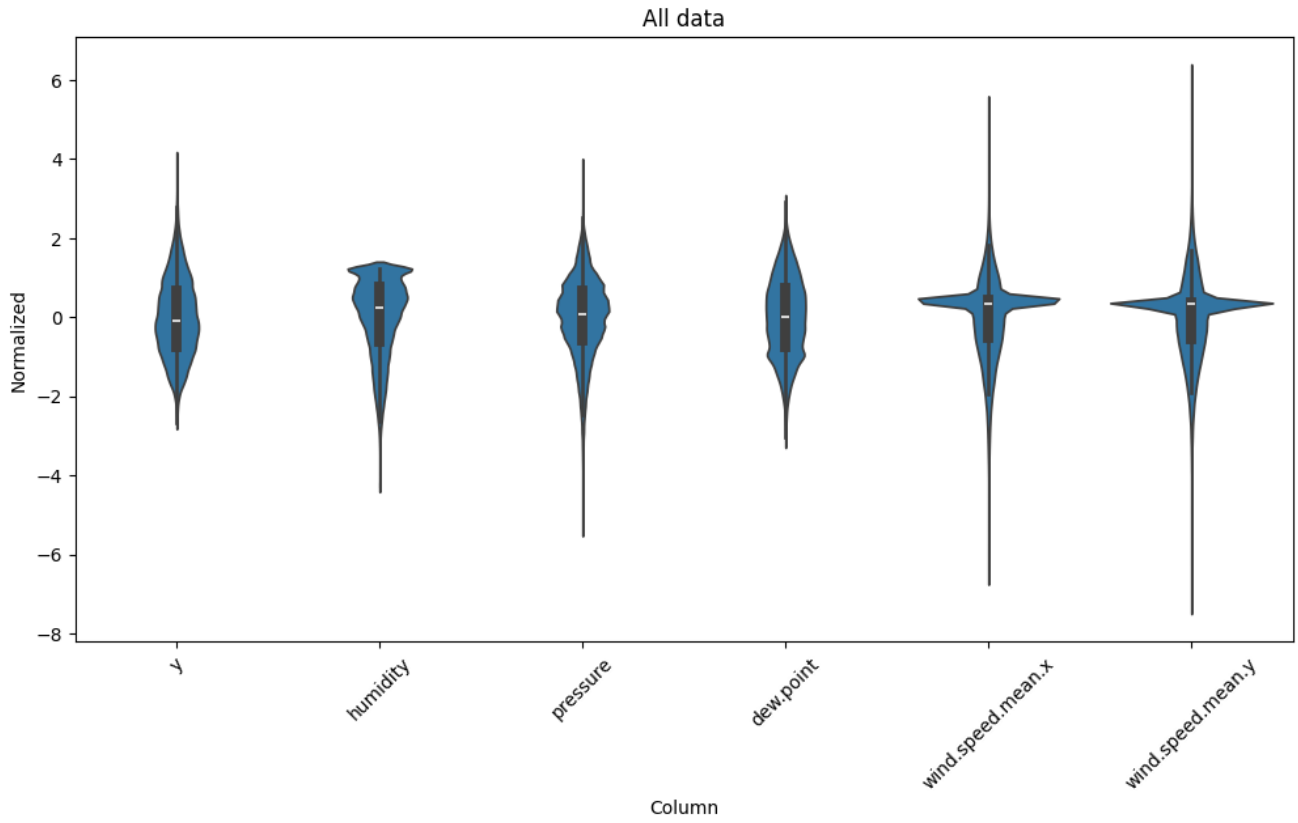
Normalise Data

Features do not need to be scaled for gradient boosting methods. Nonetheless, it can often be a useful sanity check to plot these values.

The violin plot shows the distribution of features.

```
def inv_transform(scaler, data, colName, colNames):
    """An inverse scaler for use in model validation section

    For later use in plot_forecasts, plot_horizon_metrics and check_residuals

    See https://stackoverflow.com/a/62170887/100129"""

    dummy = pd.DataFrame(np.zeros((len(data), len(colNames))), columns=colNames)
    dummy[colName] = data
    dummy = pd.DataFrame(scaler.inverse_transform(dummy), columns=colNames)

    return dummy[colName].values


cols = ['y', 'humidity', 'pressure', 'dew.point', 'wind.speed.mean.x',
        'wind.speed.mean.y']
train_df_norm = train_df.loc[:, cols]
scaler = StandardScaler()
scaler.fit(train_df_norm)

train_df_norm = scaler.transform(train_df_norm)
# valid_df[valid_df.columns] = scaler.transform(valid_df[valid_df.columns] )
# test_df[test_df.columns]   = scaler.transform(test_df[test_df.columns] )

#df_std = scaler.transform(df)
df_std = pd.DataFrame(train_df_norm)
df_std = df_std.melt(var_name = 'Column', value_name = 'Normalized')

plt.figure(figsize=(12, 6))
ax = sns.violinplot(x = 'Column', y = 'Normalized', data = df_std)
ax.set_xticklabels(cols, rotation = 45)
ax.set_title('All data');
```

```
<ipython-input-13-6bbf30968e11>:31: UserWarning: FixedFormatter should only be
  ax.set_xticklabels(cols, rotation = 45)
```



Some features have long tails.

Seasonal Decomposition

Time series decomposition with prophet.

Decomposition transforms time series into trend, seasonal and noise (residual) components. These components are combined either additively or multiplicatively. It is important to understand the time series components to better model and forecast future values. In another notebook, I forecast the residual temperature components using lightgbm and add the trend and seasonal components to these forecasts to make the final predictions.

Prophet is a python and R package for forecasting. It is based on a model where non-linear trends are fit with yearly and daily seasonality but is quite flexible. It works best with time series

that have strong seasonal effects and several seasons of historical data. In my experience it does not produce the best forecasts but I like its API and it can produce both additive and multiplicative multi-seasonal harmonic-based decompositions. Like loess-based decompositions, a little parameter tuning is beneficial. The details of how Prophet calculates multiplicative seasonality is outlined here and here.

The Season-Trend decomposition using LOESS for multiple seasonalities from statsmodels is *currently* additive only. Similarly, the unobserved components model (UCM) is additive only.

```
def merge_decompositions(df, decomps):
  '''NOTE: changes made here may also need to be made in add_prophet_components fu

  decomp_cols  = ['y', 'dew.point', 'humidity', 'pressure']#, 'wind.speed.mean']
  decomp_terms = ['daily', 'yearly', 'des']

  # df_drop_cols = [j + '_' + i for i in decomp_terms for j in decomp_cols]
  # df.drop(df_drop_cols, axis=1, inplace=True)

  # decomps_drop_cols = [j + '_des' for j in decomp_cols]
  # decomps.drop(decomps_drop_cols, axis=1, inplace=True)

  df = df[~df.index.duplicated(keep='first')]
  decomps = decomps[~decomps.index.duplicated(keep='first')]

  # Merge on month, day, time

  df['month']  = df.index.month
  df['day']    = df.index.day
  df['hour']   = df.index.hour
  df['minute'] = df.index.minute
  decomps['month']  = decomps.index.month
  decomps['day']    = decomps.index.day
  decomps['hour']   = decomps.index.hour
  decomps['minute'] = decomps.index.minute

  merge_cols = ['month','day','hour','minute']
  df = df.merge(decomps, on=merge_cols)
  # df = pd.merge(df, decomps, left_on=df.index, right_on=decomps.index)
  df.set_index('ds', drop=False, inplace=True)
  df = df[~df.index.duplicated(keep='first')]
  # df.drop('key_0', axis=1, inplace=True)
  df.drop(merge_cols, axis=1, inplace=True)

  for j in decomp_cols:
    if j in ['y', 'dew.point']:
      df[j + '_des'] = df[j] - df[j + '_yhat']
    elif j in ['humidity', 'pressure']:
      df[j + '_des'] = df[j] / df[j + '_yhat']
      # df[j + '_des'] = df[j] - df[j + '_yhat']
      # df[j + '_des'] = np.log(df[j]) / df[j + '_yhat']
    #elif j in ['wind.speed.mean']:
    #  df[j + '_des'] = np.sqrt(df[j]) / df[j + '_yhat']
```

```
    #
    #  for var in ['_daily', '_yearly', '_des', '_yhat']:
    #    df = convert_wind_to_xy(df, 'wind.speed.mean', 'wind.bearing.mean', var)
    df[j + '_des_diff_1'] = df[j + '_des'].diff(1)

  # df = df.dropna()
  df = df.asfreq(freq='30min', fill_value=np.nan)

  return df


def add_prophet_components(df, m, col_name):  #, l_frac = 3000):
  '''NOTE: changes made here may also need to be made in merge_decompositions func

  # lmbda = 0.0

  if col_name == 'y':
    df['y'] = df['y_orig']
  elif col_name == 'dew.point':
    df['y'] = df[col_name]
  elif col_name == 'pressure':
    df['y'] = df[col_name]
    # x = np.log(df[col_name])
    # df['y'] = x
  elif col_name ==  'humidity':
    df['y'] = df[col_name]
    # x = np.log(df[col_name])
    # df['y'] = x
    # x, lmbda = stats.boxcox(df[col_name])
    # df['y'] = x
    # print('lambda:', lmbda)
  elif col_name == 'wind.speed.mean':
    x = np.sqrt(df[col_name])
    df['y'] = x

  m.fit(df)
  future = m.make_future_dataframe(periods=1, freq='Y').dropna()
  forecast = m.predict(future)
  m.plot(forecast)
  m.plot_components(forecast)
  plt.show()

  # year_ex = 2018
  # n = YEARLY_OBS
  # df['year'] = df['ds'].dt.year
  # y_l = lowess(df.loc[df.year == year_ex, col_name + '_yearly'], df.loc[df.year
  # df.loc[df.year == year_ex, col_name + '_yearly'].head(n).plot()
  # plt.plot(df.loc[df.year == year_ex, 'ds'].head(n), y_l[:n, 1], 'blue', label=
  # plt.title(col_name + ' yearly')
  # plt.show()

  # n = DAILY_OBS * 2 + 1
  # n = DAILY_OBS + 1
  # df.loc[df.year == year_ex, col_name + '_daily'].head(n).plot()
```

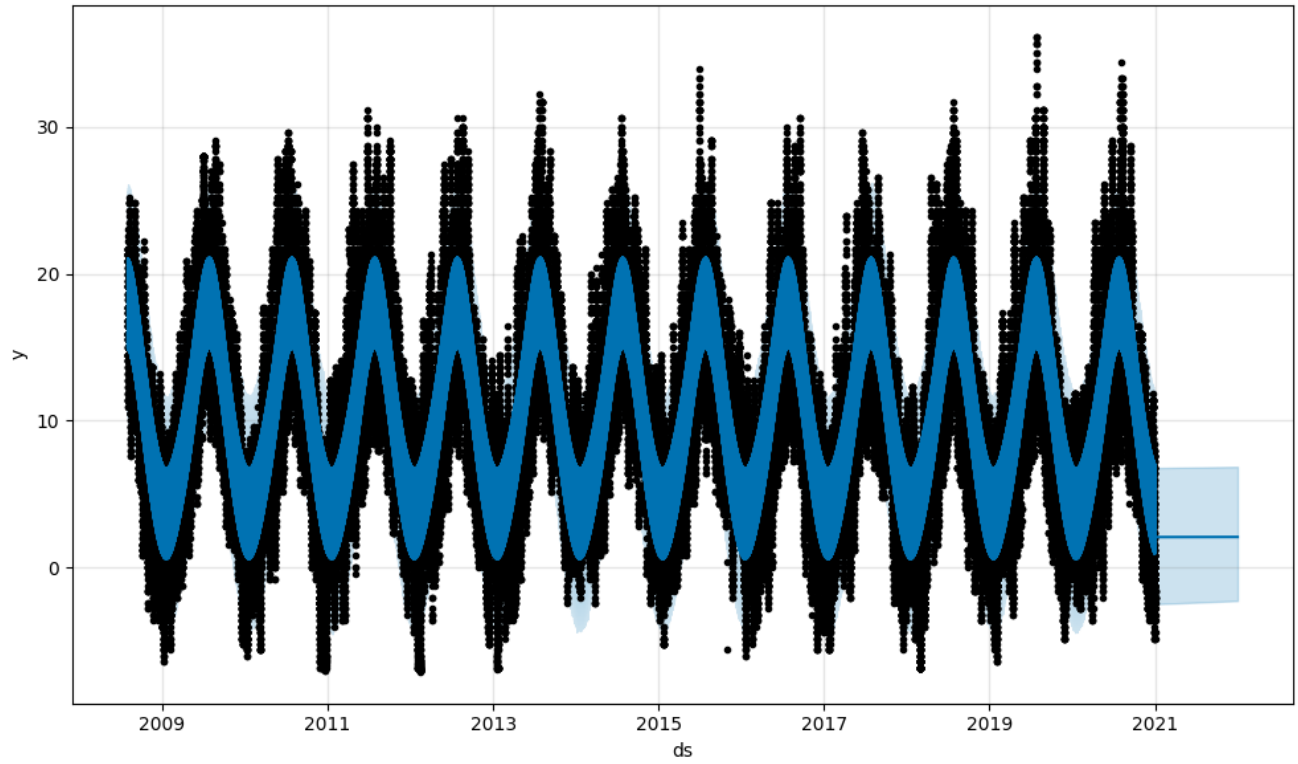
```
  # plt.axvline(x=pd.to_datetime('2018-01-01 00:00:00'), color='black', lw=1)  # v
  # plt.axvline(x=pd.to_datetime('00:00 02-Jan-2018'), color='black', lw=1)
  # plt.axvline(x=pd.Timestamp('00:00 02-Jan-2018'), color='black', lw=1)
  # plt.axvline(x=df.index.values[48], color='black', lw=1)
  # somewhere someone is watching the world burn
  # plt.title(col_name + ' daily')
  # plt.show()

  forecast.set_index('ds', drop=False, inplace=True)
  f_cols_old = ['trend', 'daily', 'yearly', 'yhat']
  forecast.rename(columns={f_col: col_name + '_' + f_col for f_col in f_cols_old},
                  inplace=True)

  if col_name in ['y', 'dew.point']:
    forecast[col_name + '_des'] = df['y'] - forecast[col_name + '_yhat']
  elif col_name in ['humidity', 'pressure', 'wind.speed.mean']:
    # forecast[col_name + '_des'] = df['y'] - forecast[col_name + '_yhat']
    forecast[col_name + '_des'] = df['y'] / forecast[col_name + '_yhat']
  #if col_name in ['y', 'dew.point', 'wind.speed.mean']:
  #  forecast[col_name + '_des'] = df['y'] - forecast['yhat']
  #else:
  #  forecast[col_name + '_des'] = df['y'] - special.inv_boxcox(forecast['yhat'],

  f_cols_new = [col_name + '_' + f_col for f_col in f_cols_old]
  f_cols_new.append(col_name + '_des')

  forecast = forecast[f_cols_new]
  # print('df:', df.shape)
  # print('forecast:', forecast.shape)

  ## drop_cols = [col_name + '_' + i for i in ['daily', 'yearly', 'des']]
  ## df.drop(drop_cols, axis=1, inplace=True)
  # df = pd.merge(df, forecast, left_on=df.index, right_on=forecast.index)
  # df.set_index('ds', drop=False, inplace=True)
  # df.drop(['key_0'], axis=1, inplace=True)
  ## df = df.dropna()
  # df = df.asfreq(freq='30min', fill_value=np.nan)
  # return df, forecast

  return forecast


train_df_orig = train_df.copy()
valid_df_orig = valid_df.copy()
test_df_orig  = test_df.copy()

train_df['y_orig'] = train_df['y']
col_name = 'y'
m1 = Prophet(yearly_seasonality = 3,
             daily_seasonality  = 3,
             weekly_seasonality = False,
             growth = 'flat')
f1 = add_prophet_components(train_df, m1, col_name)
display(train_df)
```

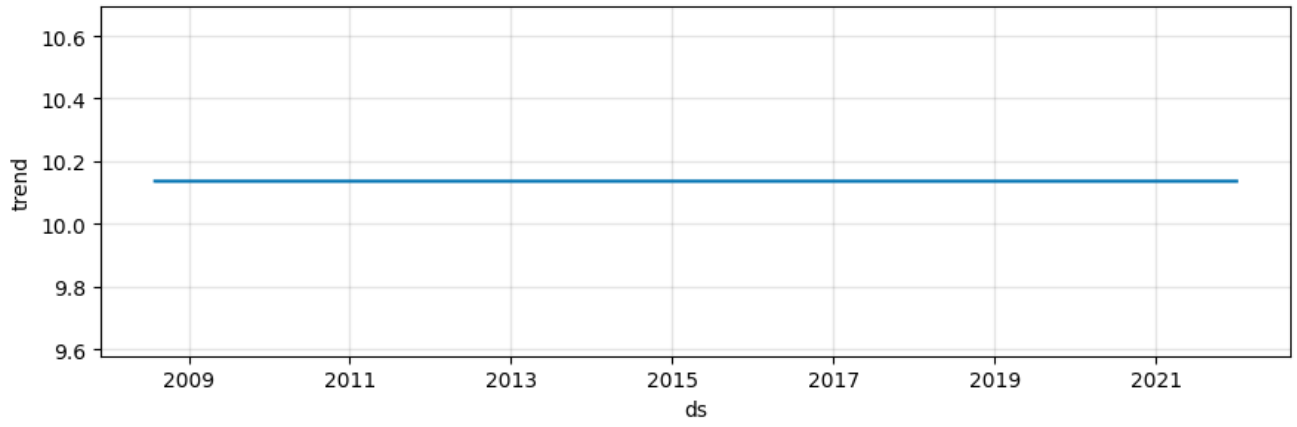
```
col_name = 'dew.point'
m2 = Prophet(yearly_seasonality = 3,
             daily_seasonality  = 3,
             weekly_seasonality = False,
             growth = 'flat')
f2 = add_prophet_components(train_df.loc['2016-01-12':,], m2, col_name)
display(train_df)

col_name = 'humidity'
m3 = Prophet(yearly_seasonality = 3,
             weekly_seasonality = False,
             seasonality_mode   = 'multiplicative',
             seasonality_prior_scale = 100,
             growth = 'flat')
f3 = add_prophet_components(train_df.loc['2016-01-12':,], m3, col_name)
display(train_df)

col_name = 'pressure'
m4 = Prophet(yearly_seasonality = 2,
             daily_seasonality  = 2,
             weekly_seasonality = False,
             seasonality_mode   = 'multiplicative',
             seasonality_prior_scale = 100,
             growth = 'flat')
f4 = add_prophet_components(train_df, m4, col_name)

#col_name = 'wind.speed.mean'
#m5 = Prophet(yearly_seasonality = 2,
#             daily_seasonality  = True,
#             weekly_seasonality = False,
#             seasonality_mode   = 'multiplicative',
#             growth = 'flat')
#train_df, f5 = add_prophet_components(train_df, m5, col_name, 6000)


# Convert wind direction and speed to x and y vectors, so the model can more easil
#wd_rad = train_df['wind.bearing.mean'] * np.pi / 180  # Convert to radians
#
#for var in ['_daily', '_yearly', '_des', '_yhat']:
#  wv = train_df['wind.speed.mean' + var]
#
#  # Calculate the wind x and y components
#  train_df['wind.speed.mean.x' + var] = wv * np.cos(wd_rad)
#  train_df['wind.speed.mean.y' + var] = wv * np.sin(wd_rad)

#for var in ['_daily', '_yearly', '_des', '_yhat']:
#  train_df = convert_wind_to_xy(train_df, 'wind.speed.mean', 'wind.bearing.mean',

train_df['y'] = train_df['y_orig']
train_df.drop('y_orig', inplace=True, axis=1)
# display(train_df)
```

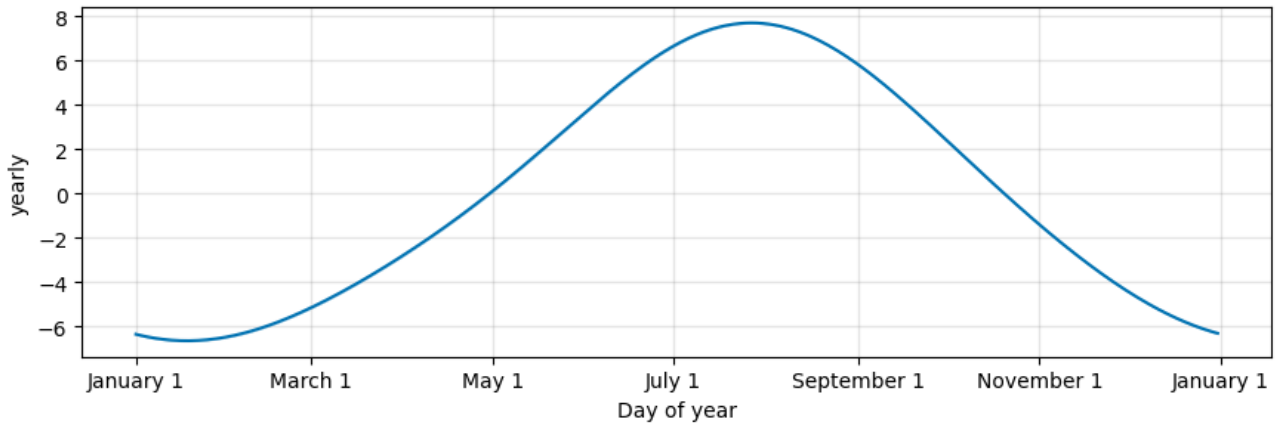
```
f_all = pd.merge(f1, f2, left_on=f1.index, right_on=f2.index)
f_all.set_index('key_0', inplace=True)
f_all = pd.merge(f_all, f3, left_on=f_all.index, right_on=f3.index)
f_all.set_index('key_0', inplace=True)
f_all = pd.merge(f_all, f4, left_on=f_all.index, right_on=f4.index)
f_all.set_index('key_0', inplace=True)
#f_all = pd.merge(f_all, f5, left_on=f_all.index, right_on=f5.index)
#f_all.set_index('key_0', inplace=True)
print('f_all:')
display(f_all)


decomps_drop_cols = [j + '_des' for j in ['y', 'dew.point', 'humidity', 'pressure
f_all.drop(decomps_drop_cols, axis=1, inplace=True)

valid_df = merge_decompositions(valid_df, f_all)
print('valid_df:')
display(valid_df)
valid_df = valid_df.fillna(method='bfill')  # Small number of NAs in 1st row

test_df  = merge_decompositions(test_df,  f_all)
print('test_df:')
display(test_df)
test_df = test_df.fillna(method='bfill')  # Small number of NAs in 1st row

train_df  = merge_decompositions(train_df,  f_all)
print('train_df:')
display(train_df)
train_df = train_df.fillna(method='bfill')  # Small number of NAs in 1st row

for col in ['y_des', 'dew.point_des']:
  train_df[col+'_grad'] = np.gradient(train_df[col])
  valid_df[col+'_grad'] = np.gradient(valid_df[col])
  test_df[col+'_grad']  = np.gradient(test_df[col])

# Add Boruta-style "shadow" variables for feature selection
train_df['y_des_shadow'] = np.random.permutation(train_df['y_des'])
valid_df['y_des_shadow'] = np.random.permutation(valid_df['y_des'])
test_df['y_des_shadow']  = np.random.permutation( test_df['y_des'])


#merge_cols = ['month', 'day', 'hour', 'minute']
#train_df.drop(merge_cols, inplace=True, axis=1)

#del_cols = ['ds']
#train_df = train_df.drop(del_cols, axis = 1)
#valid_df = valid_df.drop(del_cols, axis = 1)
#test_df  = test_df.drop(del_cols,  axis = 1)
print_train_valid_test_shapes(df, train_df, valid_df, test_df)
sanity_check_train_valid_test(train_df, valid_df, test_df)

train_des_sanity = sanity_check_before_after_dfs(train_df_orig, train_df, 'train_d
valid_des_sanity = sanity_check_before_after_dfs(valid_df_orig, valid_df, 'valid_d
test_des_sanity  = sanity_check_before_after_dfs(test_df_orig,  test_df,  'test_df
```

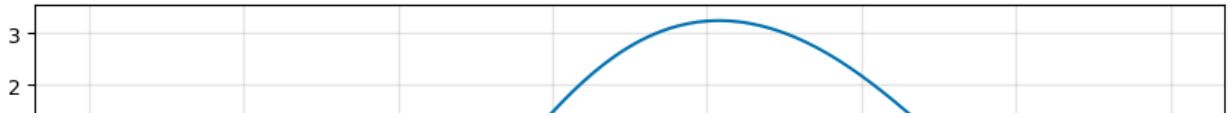
```
compare_train_valid_test_sanity_dfs(train_des_sanity, valid_des_sanity, test_des_s
check_high_low_thresholds(train_df)
check_high_low_thresholds(valid_df)
check_high_low_thresholds(test_df)
```

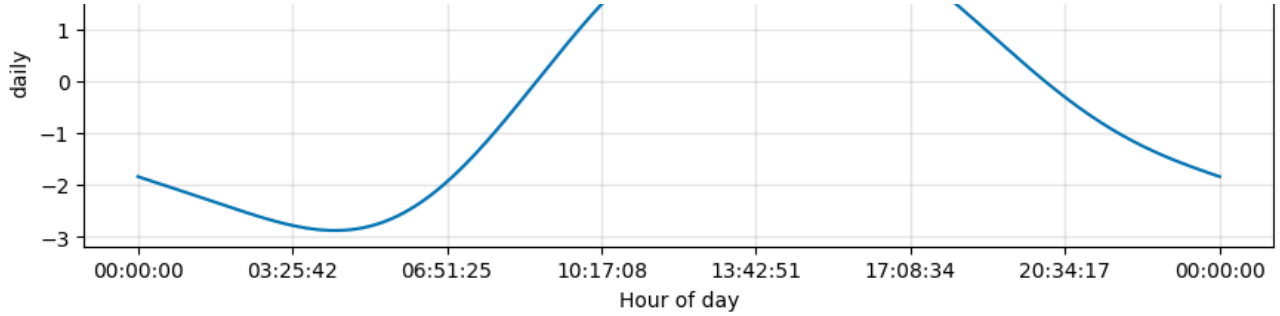
```
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/np1_ud_c.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/r9rh5z5k.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophe
09:06:51 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
09:06:54 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```









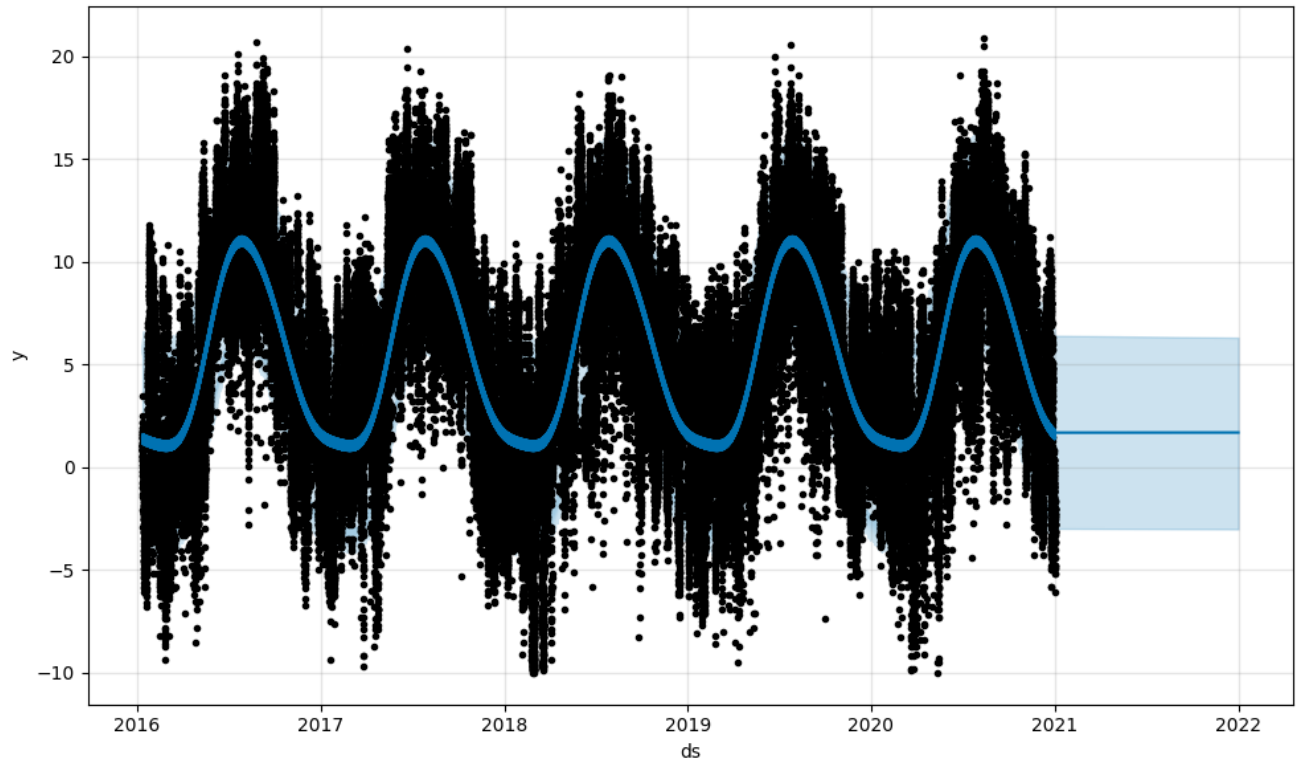


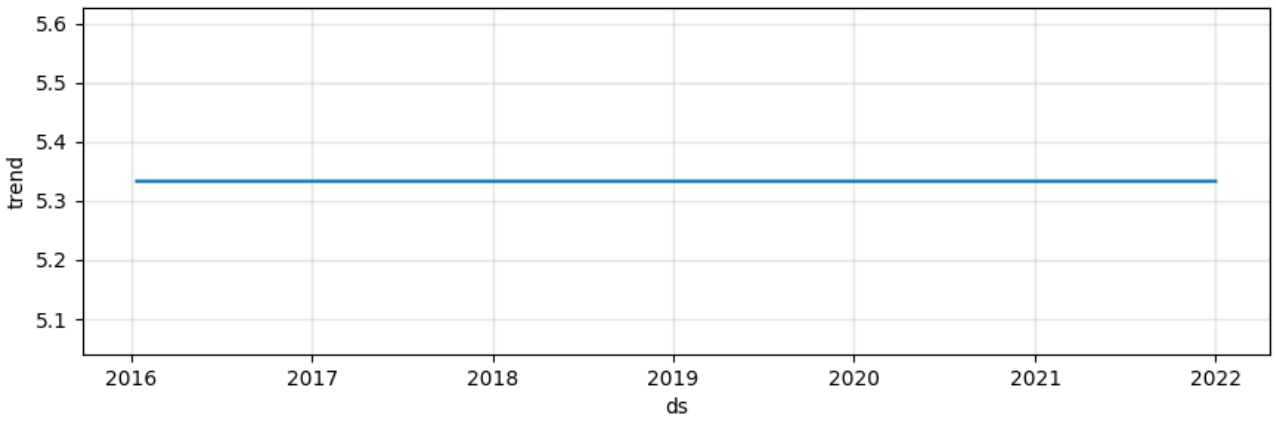
ds	ds	y	humidity	dew.point	pressure	pressure.log	y_window_
2008-08-01 00:00:00	2008-08-01 00:00:00	20.0	89.221997	1.610001	1009.926805	6.917633	
2008-08-01 00:30:00	2008-08-01 00:30:00	19.5	88.290807	1.657847	1009.931495	6.917638	
2008-08-01 01:00:00	2008-08-01 01:00:00	19.1	85.755790	1.432873	1010.092861	6.917798	
2008-08-01 01:30:00	2008-08-01 01:30:00	19.1	85.572379	1.397409	1009.902342	6.917609	
2008-08-01 02:00:00	2008-08-01 02:00:00	19.1	84.345642	1.278219	1010.133476	6.917838	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.8	96.000000	-3.300000	1006.000000	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.2	100.000000	-3.200000	1007.000000	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.6	100.000000	-3.600000	1007.000000	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.4	97.000000	-4.800000	1007.000000	6.914731	
2020-12-31 23:30:00	2020-12-31 23:30:00	-4.8	99.000000	-4.900000	1007.000000	6.914731	

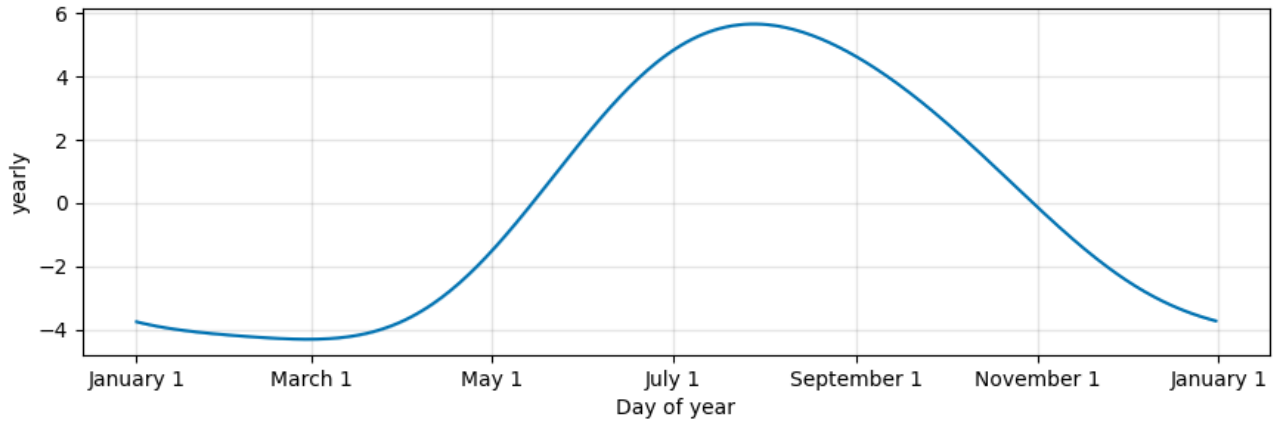
217728 rows × 85 columns

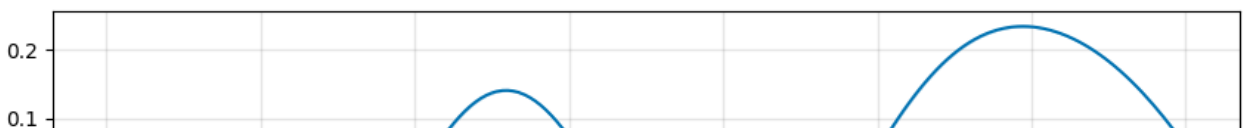
```
<ipython-input-14-186207976926>:63: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  df['y'] = df[col_name]
```

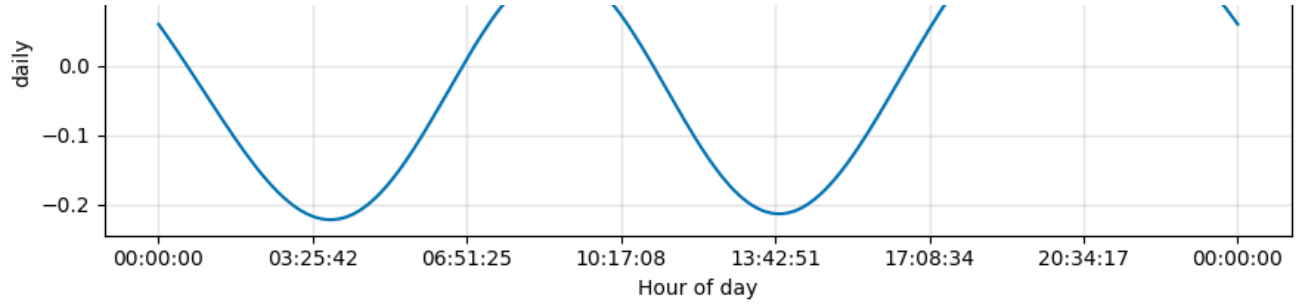
```
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/b00m8x1t.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/mrbhzway.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophe
09:08:04 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
09:08:05 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```









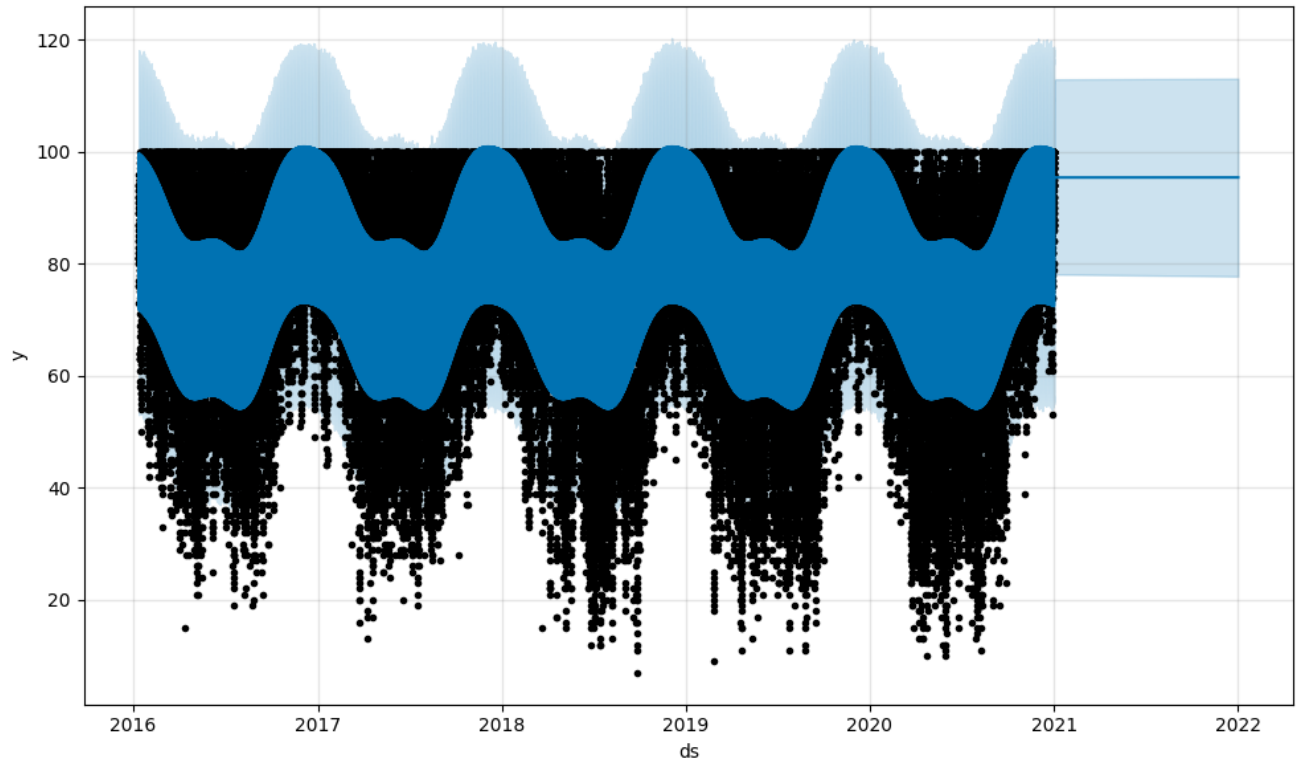


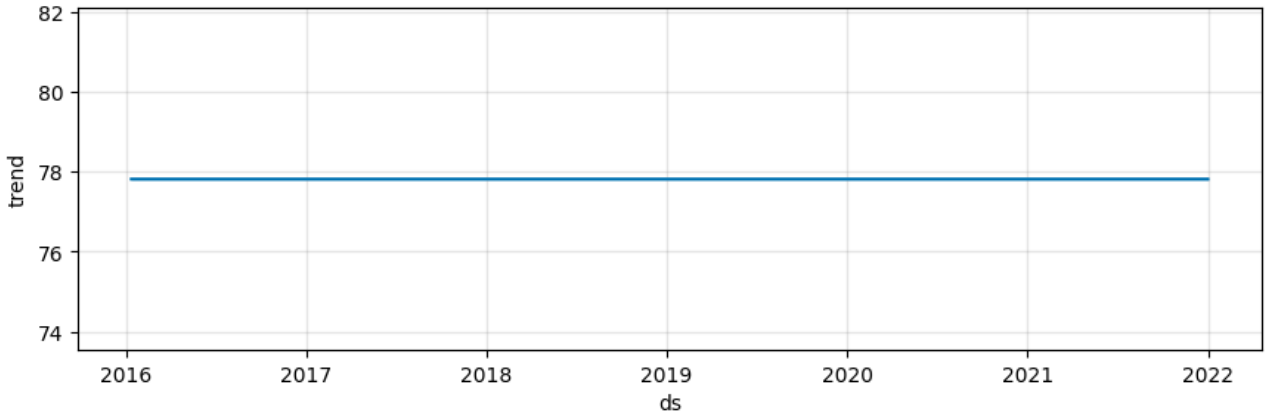
	ds	y	humidity	dew.point	pressure	pressure.log	y_window_
ds							
2008-08-01 00:00:00	2008-08-01 00:00:00	20.0	89.221997	1.610001	1009.926805	6.917633	
2008-08-01 00:30:00	2008-08-01 00:30:00	19.5	88.290807	1.657847	1009.931495	6.917638	
2008-08-01 01:00:00	2008-08-01 01:00:00	19.1	85.755790	1.432873	1010.092861	6.917798	
2008-08-01 01:30:00	2008-08-01 01:30:00	19.1	85.572379	1.397409	1009.902342	6.917609	
2008-08-01 02:00:00	2008-08-01 02:00:00	19.1	84.345642	1.278219	1010.133476	6.917838	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.8	96.000000	-3.300000	1006.000000	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.2	100.000000	-3.200000	1007.000000	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.6	100.000000	-3.600000	1007.000000	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.4	97.000000	-4.800000	1007.000000	6.914731	
2020-12-31 23:30:00	2020-12-31 23:30:00	-4.8	99.000000	-4.900000	1007.000000	6.914731	

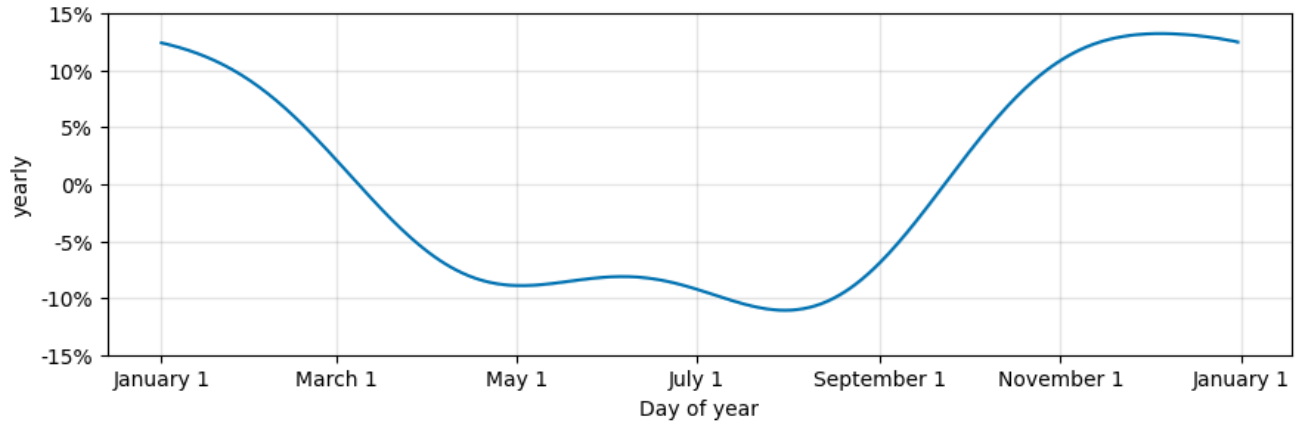
217728 rows × 85 columns

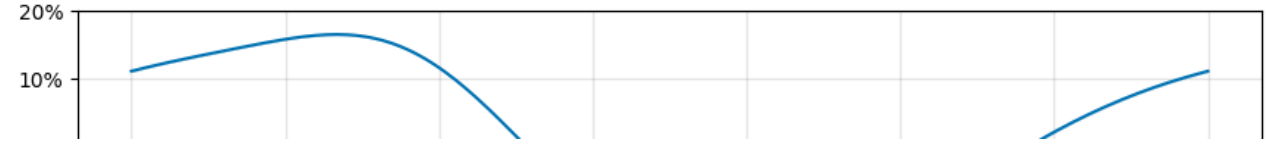
```
<ipython-input-14-186207976926>:69: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  df['y'] = df[col_name]
```

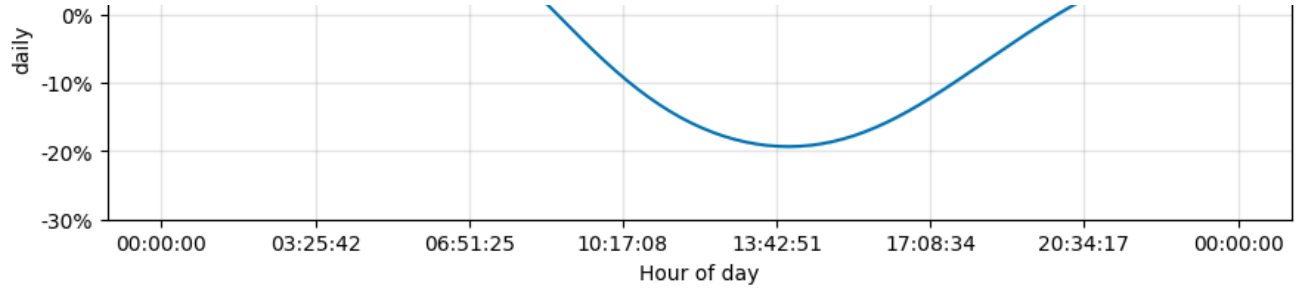
```
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/g2r13yfw.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/zhko8b8f.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophe
09:08:32 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
09:08:33 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```









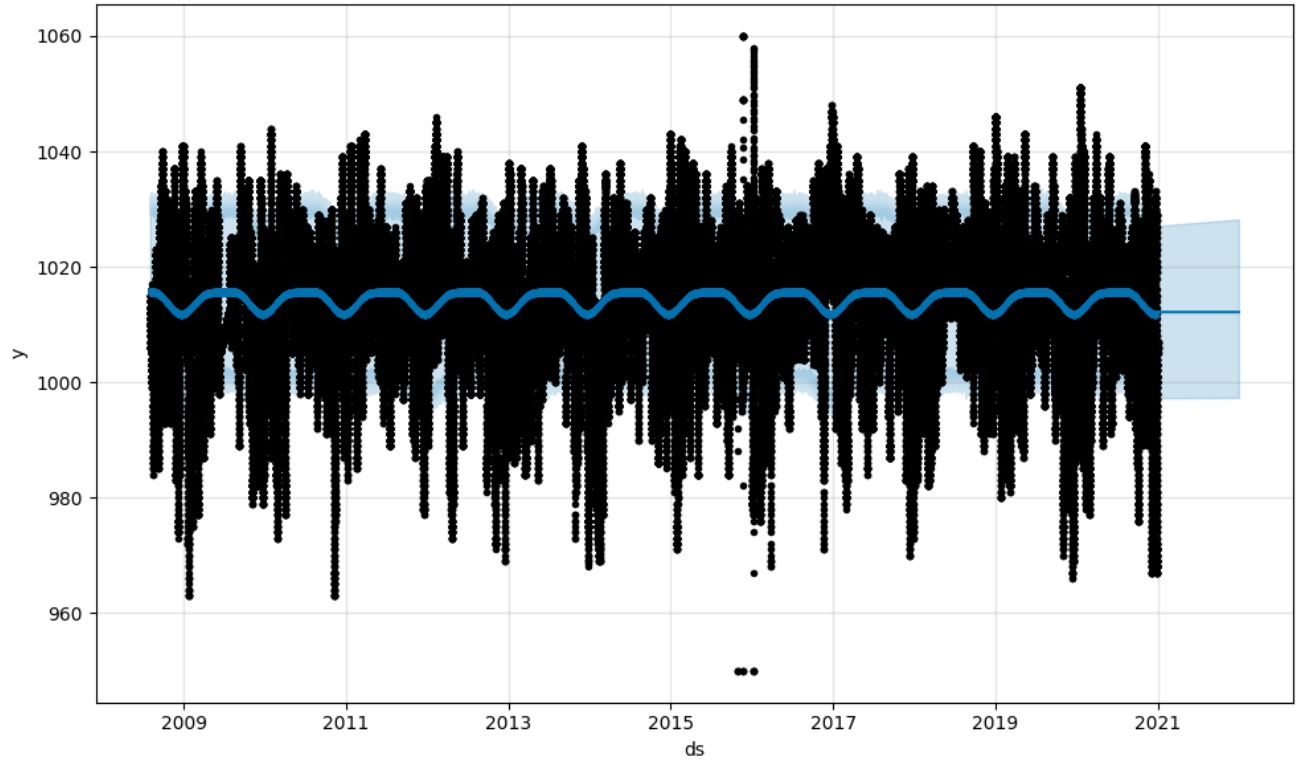


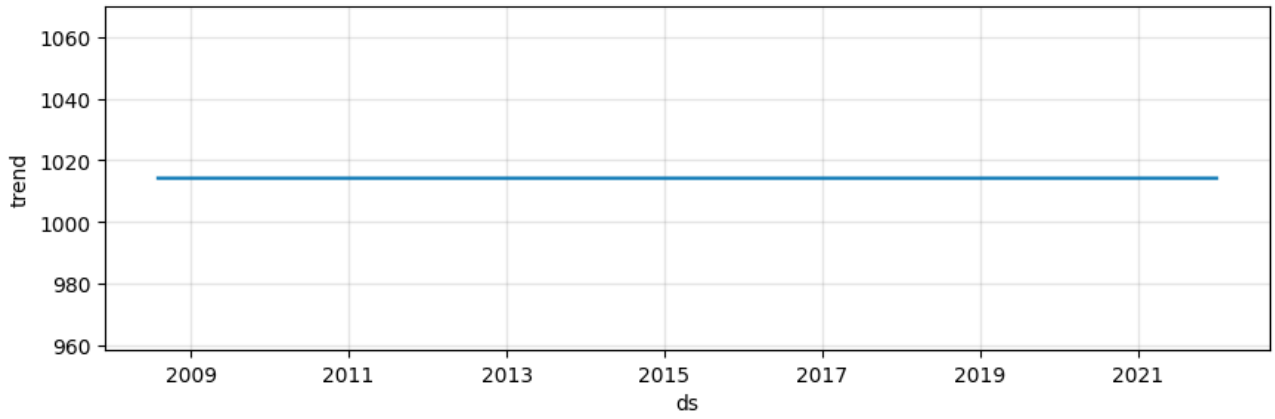
	ds	y	humidity	dew.point	pressure	pressure.log	y_window_
ds							
2008-08-01 00:00:00	2008-08-01 00:00:00	20.0	89.221997	1.610001	1009.926805	6.917633	
2008-08-01 00:30:00	2008-08-01 00:30:00	19.5	88.290807	1.657847	1009.931495	6.917638	
2008-08-01 01:00:00	2008-08-01 01:00:00	19.1	85.755790	1.432873	1010.092861	6.917798	
2008-08-01 01:30:00	2008-08-01 01:30:00	19.1	85.572379	1.397409	1009.902342	6.917609	
2008-08-01 02:00:00	2008-08-01 02:00:00	19.1	84.345642	1.278219	1010.133476	6.917838	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.8	96.000000	-3.300000	1006.000000	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.2	100.000000	-3.200000	1007.000000	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.6	100.000000	-3.600000	1007.000000	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.4	97.000000	-4.800000	1007.000000	6.914731	
2020-12-31 23:30:00	2020-12-31 23:30:00	-4.8	99.000000	-4.900000	1007.000000	6.914731	

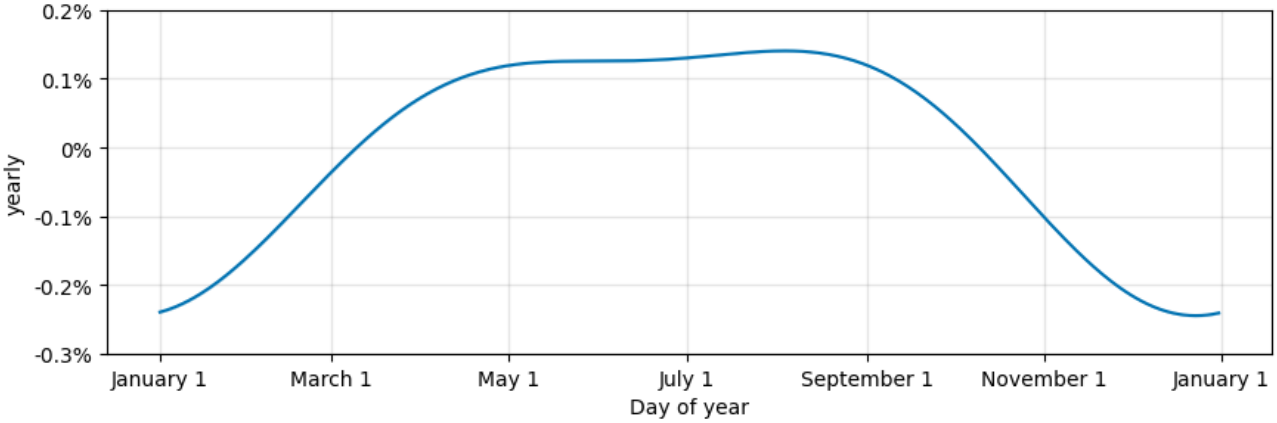
217728 rows × 85 columns

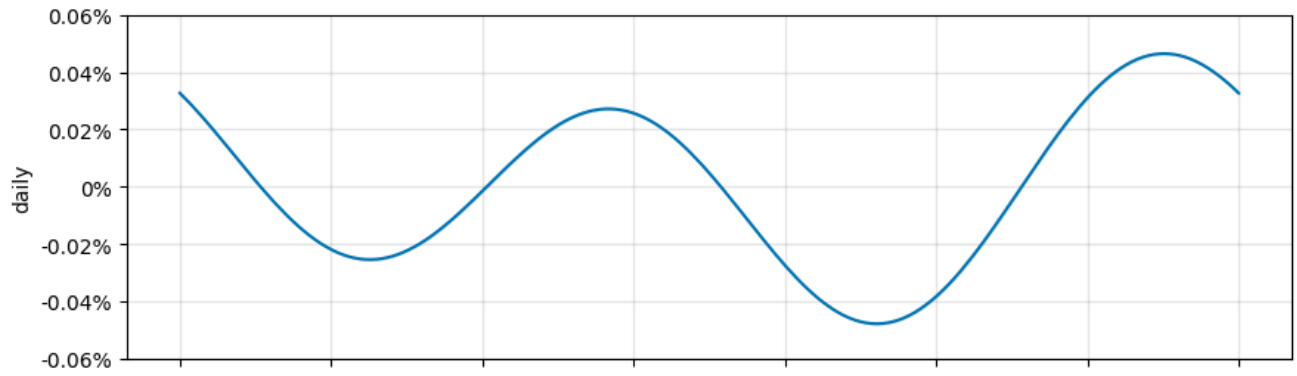
```
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/f0t8v_oi.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmp2gyl89uc/l9e6fjba.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophe
09:09:02 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
```

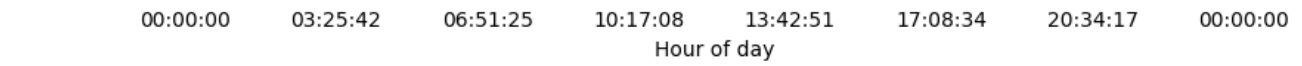
```
09:09:04 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```











`f_all:`

key_0	y_trend	y_daily	y_yearly	y_yhat	y_des	dew.point_trend	dew.pc
2016-01-12 00:00:00	10.136808	-1.843996	-6.607021	1.685791	-0.085791	5.333283	
2016-01-12 00:30:00	10.136808	-1.987274	-6.607307	1.542227	0.457773	5.333283	
2016-01-12 01:00:00	10.136808	-2.129682	-6.607593	1.399533	1.400467	5.333283	
2016-01-12 01:30:00	10.136808	-2.274232	-6.607878	1.254698	0.745302	5.333283	
2016-01-12 02:00:00	10.136808	-2.420156	-6.608162	1.108490	1.291510	5.333283	
...	...	...	...	...	...	...	
2020-12-31 22:00:00	10.136808	-1.125864	-6.358769	2.652175	-5.852175	5.333283	
2020-12-31 22:30:00	10.136808	-1.342326	-6.359480	2.435002	-6.035002	5.333283	
2020-12-31 23:00:00	10.136808	-1.529351	-6.360190	2.247266	-6.647266	5.333283	
2020-12-31 23:30:00	10.136808	-1.693909	-6.360900	2.081999	-6.881999	5.333283	
2021-12-31 23:30:00	10.136808	-1.693909	-6.352330	2.090570	NaN	5.333283	

87169 rows × 20 columns

`valid_df:`

ds	ds	y	humidity	dew.point	pressure	pressure.log	y_wind
2020-12-29 00:00:00	2020-12-29 00:00:00	0.800000	90.000000	-0.700000	979.000000	6.886532	
2020-12-29 00:30:00	2020-12-29 00:30:00	0.800000	94.000000	-0.100000	978.000000	6.885510	

	ds	y	humidity	dew.point	pressure	pressure.log
2020-12-29 01:00:00	2020-12-29 01:00:00	0.400000	89.000000	-1.200000	979.000000	6.886532
2020-12-29 01:30:00	2020-12-29 01:30:00	0.400000	99.000000	0.300000	979.000000	6.886532
2020-12-29 02:00:00	2020-12-29 02:00:00	0.000000	89.000000	-1.600000	979.000000	6.886532
...	...	...	...	...	...	...
2021-12-31 21:30:00	2021-12-31 21:30:00	12.438743	76.824022	1.414153	1010.049436	6.917755
2021-12-31 22:00:00	2021-12-31 22:00:00	12.473437	76.215531	1.363846	1010.064737	6.917770
2021-12-31 22:30:00	2021-12-31 22:30:00	12.440330	76.236801	1.361175	1010.082097	6.917787
2021-12-31 23:00:00	2021-12-31 23:00:00	12.263176	77.519451	1.132940	1010.095954	6.917801
2021-12-31 23:30:00	2021-12-31 23:30:00	12.269498	77.802763	1.113068	1010.110135	6.917815

17664 rows × 108 columns

`test_df:`

	ds	y	humidity	dew.point	pressure	pressure.log	y_wind
ds							
2021-12-29 00:00:00	2021-12-29 00:00:00	9.018013	80.712206	1.424063	1007.854619	6.915579	
2021-12-29 00:30:00	2021-12-29 00:30:00	9.024635	80.785103	1.398200	1007.871259	6.915596	
2021-12-29 01:00:00	2021-12-29 01:00:00	8.791503	83.094214	1.432567	1007.888116	6.915612	
2021-12-29 01:30:00	2021-12-29 01:30:00	8.271475	82.180770	1.665441	1007.904126	6.915628	
2021-12-29 02:00:00	2021-12-29 02:00:00	8.271705	82.324921	1.633612	1007.919349	6.915643	
...	...	...	...	...	...	...	
2022-12-	2022-						

2022-12-31 21:30:00	2022-12-31 21:30:00	9.600000	80.000000	6.300000	998.000000	6.905753
2022-12-31 22:00:00	2022-12-31 22:00:00	10.000000	88.000000	8.100000	998.000000	6.905753
2022-12-31 22:30:00	2022-12-31 22:30:00	10.000000	80.000000	6.700000	999.000000	6.906755
2022-12-31 23:00:00	2022-12-31 23:00:00	9.600000	79.000000	6.100000	1000.000000	6.907755
2022-12-31 23:30:00	2022-12-31 23:30:00	9.200000	82.000000	6.300000	1000.000000	6.907755

17664 rows × 108 columns

`train_df:`

	ds	y	humidity	dew.point	pressure	pressure.log	y_window_
ds							
2008-08-01 00:00:00	2008-08-01 00:00:00	20.0	89.221997	1.610001	1009.926805	6.917633	
2008-08-01 00:30:00	2008-08-01 00:30:00	19.5	88.290807	1.657847	1009.931495	6.917638	
2008-08-01 01:00:00	2008-08-01 01:00:00	19.1	85.755790	1.432873	1010.092861	6.917798	
2008-08-01 01:30:00	2008-08-01 01:30:00	19.1	85.572379	1.397409	1009.902342	6.917609	
2008-08-01 02:00:00	2008-08-01 02:00:00	19.1	84.345642	1.278219	1010.133476	6.917838	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.8	96.000000	-3.300000	1006.000000	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.2	100.000000	-3.200000	1007.000000	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.6	100.000000	-3.600000	1007.000000	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.4	97.000000	-4.800000	1007.000000	6.914731	

2020-12-31 23:30:00	2020-12-31 23:30:00	-4.8	99.000000	-4.900000	1007.000000	6.914731

217728 rows × 108 columns

```
df shape:  (252768, 84)
train shape:    (217728, 109)
valid shape:    (17664, 109)
test shape:     (17664, 109)
ERROR: Overlap between train_df, valid_df indices!
max(train_df.index): 2020-12-31 23:30:00
min(valid_df.index): 2021-12-31 23:30:00
ERROR: Overlap between valid_df, test_df indices!
valid_df.index: 2021-12-31 23:30:00 - 2021-12-31 23:30:00
test_df.index: 2022-12-31 23:30:00 - 2022-12-31 23:30:00
ERROR: valid_df should be 1 year long [ 17520 , 17568 ]!
valid_df observations: 17664
ERROR: test_df should be 1 year long [ 17520 , 17568 ]!
test_df observations: 17664

train_df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): True

before[common_cols].equals(after[common_cols]): True
redundancy before > after: False
mean before feature redundancy: 34.237
mean after feature redundancy:  46.731
```

	before	after	diff
rows	217728	217728	0
cols	84	109	25
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	0	4	4
low_var_rows	0	0	0
low_var_cols	8	16	8
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
valid_df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
```

```
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): True

before[common_cols].equals(after[common_cols]): True
redundancy before > after: False
mean before feature redundancy: 33.227
mean after feature redundancy:  45.924
```

	before	after	diff
rows	17664	17664	0
cols	84	109	25
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	0	4	4
low_var_rows	0	0	0
low_var_cols	10	18	8
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
test_df
before.index.equals(after.index): True
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): True

before[common_cols].equals(after[common_cols]): True
redundancy before > after: False
mean before feature redundancy: 33.222
mean after feature redundancy:  45.942
```

	before	after	diff
rows	17664	17664	0
cols	84	109	25
missing_rows	0	0	0
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0

single_value_cols	1	5	4
low_var_rows	0	0	0
low_var_cols	9	17	8
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
WARN: train_sanity != valid_sanity
```

	before	after	diff
low_var_cols	8	16	8
low_var_cols	10	18	8

```
WARN: train_sanity != test_sanity
```

	before	after	diff
single_value_cols	0	4	4
low_var_cols	8	16	8
single_value_cols	1	5	4
low_var_cols	9	17	8

```
WARN: test_sanity != valid_sanity
```

	before	after	diff
single_value_cols	1	5	4
low_var_cols	9	17	8
single_value_cols	0	4	4
low_var_cols	10	18	8

The prophet seasonal decompositions take quite a while to run. Similar to the UCM decomposition, temperature and dew point seem to be acceptbible but humidity and pressure are questionable and the `wind.speed.mean` decomposition is unusable. I've removed the `wind.speed.mean` decomposition.

Calculate simple rolling statistics

Add simple and fast to calculate rolling statistics.

Features:

- `y`
- `dew.point`
- `y_des`
- `dew.point_des`
- `pressure`
- `humidity`

Functions:

- minimum
- maximum
- mean
- standard deviation
- skew
- kurtosis
- auto-correlation

Windows:

- 12, 24, 48, 96

```
def get_rolling_stat_features(data, feat_cols, windows, aggs, verbose=False, shift

  aggs_l = []  # Prevents highly fragmented dataframes

  for feat_col in feat_cols:
    agg = pd.DataFrame(index=data.index)  # Prevents highly fragmented dataframes
    for window in windows:
      rol = data[feat_col].shift(shift_).rolling(window=window, min_periods=1,)
      new_col_name = feat_col + '_window_' + str(window) + '_'

      agg[new_col_name + 'min']  = rol.min()
      agg[new_col_name + 'max']  = rol.max()
      agg[new_col_name + 'std']  = rol.std()
      agg[new_col_name + 'mean'] = rol.mean()
      agg[new_col_name + 'skew'] = rol.skew()
      agg[new_col_name + 'kurt'] = rol.kurt()
      agg[new_col_name + 'quantile_25'] = rol.quantile(.25)
      agg[new_col_name + 'quantile_75'] = rol.quantile(.75)

      # faster than np.p2p
      agg[new_col_name + 'peak2peak'] = rol.agg(['min', 'max']).diff(axis=1)['max

      # Slow  Doubt if its worth it :-(
      # TODO https://stackoverflow.com/questions/46470743/how-to-efficiently-compu
      # agg[new_col_name + 'nunique'] = rol.apply(lambda x: np.unique(x).shape[0])

    # From https://stackoverflow.com/a/43748653/100129
    #data[new_col_name + 'autocorr'] = data[feat_col].rolling(window=window, min_p
    # data[new_col_name + 'autocorr'] = rol.corr(data[feat_col].shift(1)) # NOPE
    #if feat_col in ['humidity', 'pressure']:
    #  # inf values with window = 48 - probably long sequences of equal values
    #  window = 96
    #else:
    #  window = 48

    agg[new_col_name + 'autocorr'] = data[feat_col].rolling(window=window, min_per
    # Doubt if it's worth it
    # agg[new_col_name + 'autocorr_48'] = data[feat_col].rolling(window=window, mi

    aggs_l.append(agg)  # Prevents highly fragmented dataframes

  aggs = pd.concat(aggs_l, axis=1)  # Prevents highly fragmented dataframes
  aggs = drop_problem_cols(aggs, window)
  aggs = drop_cols_correlated_with_feat_cols(aggs, data[feat_cols])

  return aggs


windows = [12, 24, 48, 96]
roll_cols = ['y', 'dew.point', 'y_des', 'dew.point_des', 'humidity', 'pressure']
stat_aggs = ['min', 'max', 'mean', 'std', 'skew', 'kurt', 'quantile', 'corr']
params_stat = {'windows':    windows,
```

```
                'feat_cols':  roll_cols,
                'aggs':       stat_aggs,
                'agg_func':   get_rolling_stat_features,
                'verbose':    True,
                'dataset':    'valid',
                'regenerate': True,
                'feat_name':  'roll_stats_',
                'date_str':   '.2022.09.20',
                'save_and_download': False,
                'save_to_gdrive': False,
               }

train_df_stat, valid_df_stat, test_df_stat = get_features(train_df,
                                                          valid_df,
                                                          test_df,
                                                          params_stat)


##############################################################################
# approx. 1 min - mostly dataframe checks
#           for dataframe statistics functions on valid_df, test_df & train_df
# 111 features added - 0 NAs
#
# windows = [12, 24, 48, 96]
# roll_cols = ['y', 'dew.point', 'y_des', 'dew.point_des', 'humidity', 'pressure']
# stat_aggs = ['min', 'max', 'mean', 'std', 'skew', 'kurt', 'corr']
# params_stat = {'windows':    windows,
#                'feat_cols':  roll_cols,
#                'aggs':       stat_aggs,
#                'agg_func':   get_rolling_stat_features,
#                'verbose':    True,
#                'dataset':    'valid',
#                'regenerate': True,
#                'feat_name':  'df_stats_',
#                'date_str':   '.2022.09.20',
#               }
```

```
dataset: valid
columns with null values:
y_window_12_min                1
y_window_12_max                1
y_window_12_std                2
y_window_12_skew               3
y_window_12_kurt               4
                              ..
pressure_window_96_max         1
pressure_window_96_std         2
pressure_window_96_skew        3
pressure_window_96_kurt        4
pressure_window_96_autocorr    2
Length: 120, dtype: int64
before: (17664, 109)
after:  (17664, 227)
```

	ds	y	humidity	dew.point	pressure	pressure.log	y_winc
ds							
2020-12-29 00:00:00	2020-12-29 00:00:00	0.800000	90.000000	-0.700000	979.000000	6.886532	
2020-12-29 00:30:00	2020-12-29 00:30:00	0.800000	94.000000	-0.100000	978.000000	6.885510	
2020-12-29 01:00:00	2020-12-29 01:00:00	0.400000	89.000000	-1.200000	979.000000	6.886532	
2020-12-29 01:30:00	2020-12-29 01:30:00	0.400000	99.000000	0.300000	979.000000	6.886532	
2020-12-29 02:00:00	2020-12-29 02:00:00	0.000000	89.000000	-1.600000	979.000000	6.886532	
...	...	...	...	...	...	...	...
2021-12-31 21:30:00	2021-12-31 21:30:00	12.438743	76.824022	1.414153	1010.049436	6.917755	
2021-12-31 22:00:00	2021-12-31 22:00:00	12.473437	76.215531	1.363846	1010.064737	6.917770	
2021-12-31 22:30:00	2021-12-31 22:30:00	12.440330	76.236801	1.361175	1010.082097	6.917787	
2021-12-31 23:00:00	2021-12-31 23:00:00	12.263176	77.519451	1.132940	1010.095954	6.917801	
2021-12-31 23:30:00	2021-12-31 23:30:00	12.269498	77.802763	1.113068	1010.110135	6.917815	

17664 rows × 227 columns

	y	humidity	dew.point	pressure	pressure.log	y_windo
count	17664.000000	17664.000000	17664.000000	17664.000000	17664.000000	
mean	9.196032	77.371524	4.790745	1015.718777	6.923282	
std	6.546011	17.357854	5.708252	12.011594	0.011864	
min	-6.000000	6.000000	-11.164754	968.000000	6.875232	
25%	4.400000	67.000000	0.400000	1008.000000	6.915723	
50%	9.200000	81.000000	5.000000	1017.000000	6.924612	
75%	13.600000	90.000000	9.525000	1024.000000	6.931472	
max	29.600000	100.000000	19.100000	1044.000000	6.950815	

8 rows × 226 columns

```
max NA seq len:
Series([], dtype: int64)

dataset: test
columns with null values:
y_window_12_min            1
y_window_12_max            1
y_window_12_std            2
y_window_12_skew           3
y_window_12_kurt           4
                         ..
pressure_window_96_max     1
pressure_window_96_std     2
pressure_window_96_mean    1
pressure_window_96_skew    3
pressure_window_96_kurt    4
Length: 123, dtype: int64
before: (17664, 109)
after:  (17664, 231)
```

	ds	y	humidity	dew.point	pressure	pressure.log	y_win
ds							
2021-12-29 00:00:00	2021-12-29 00:00:00	9.018013	80.712206	1.424063	1007.854619	6.915579	
2021-12-29 00:30:00	2021-12-29 00:30:00	9.024635	80.785103	1.398200	1007.871259	6.915596	
2021-12-29 01:00:00	2021-12-29 01:00:00	8.791503	83.094214	1.432567	1007.888116	6.915612	
2021-12-29 01:30:00	2021-12-29 01:30:00	8.271475	82.180770	1.665441	1007.904126	6.915628	
2021-12-29	2021-12-29	8.271705	82.324921	1.633612	1007.919349	6.915643	

02:00:00	02:00:00					
...	...	...	...	...	...	...
2022-12-31 21:30:00	2022-12-31 21:30:00	9.600000	80.000000	6.300000	998.000000	6.905753
2022-12-31 22:00:00	2022-12-31 22:00:00	10.000000	88.000000	8.100000	998.000000	6.905753
2022-12-31 22:30:00	2022-12-31 22:30:00	10.000000	80.000000	6.700000	999.000000	6.906755
2022-12-31 23:00:00	2022-12-31 23:00:00	9.600000	79.000000	6.100000	1000.000000	6.907755
2022-12-31 23:30:00	2022-12-31 23:30:00	9.200000	82.000000	6.300000	1000.000000	6.907755

17664 rows × 231 columns

	y	humidity	dew.point	pressure	pressure.log	y_windo
count	17664.000000	17664.000000	17664.000000	17664.000000	17664.000000	
mean	10.370738	73.094140	4.926538	1016.288722	6.923853	
std	6.887136	19.453441	4.919329	11.053516	0.010903	
min	-7.144690	5.000000	-10.000000	972.000000	6.879356	
25%	5.600000	60.000000	1.500000	1009.000000	6.916715	
50%	10.000000	77.000000	5.000000	1017.000000	6.924612	
75%	15.000000	88.000000	8.500000	1024.000000	6.931472	
max	37.200000	100.000000	19.100000	1047.000000	6.953684	

8 rows × 230 columns

```
max NA seq len:
Series([], dtype: int64)

dataset: train
columns with null values:
y_window_12_min            1
y_window_12_max            1
y_window_12_std            2
y_window_12_skew           3
y_window_12_kurt           4
                         ..
pressure_window_96_max     1
pressure_window_96_std     2
pressure_window_96_mean    1
pressure_window_96_skew    3
pressure_window_96_kurt    4
Length: 123, dtype: int64
before: (87696, 109)
```

after: (87696, 231)

	ds	y	humidity	dew.point	pressure	pressure.log	y_windo
ds							
2016-01-01 00:00:00	2016-01-01 00:00:00	10.087433	57.0	3.2	1019.0	6.926577	
2016-01-01 00:30:00	2016-01-01 00:30:00	10.088860	72.0	6.5	1020.0	6.927558	
2016-01-01 01:00:00	2016-01-01 01:00:00	10.084001	83.0	8.6	1021.0	6.928538	
2016-01-01 01:30:00	2016-01-01 01:30:00	10.004534	88.0	9.5	1021.0	6.928538	
2016-01-01 02:00:00	2016-01-01 02:00:00	10.008839	91.0	10.0	1021.0	6.928538	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.800000	96.0	-3.3	1006.0	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.200000	100.0	-3.2	1007.0	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.600000	100.0	-3.6	1007.0	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.400000	97.0	-4.8	1007.0	6.914731	
2020-12-31 23:30:00	2020-12-31 23:30:00	-4.800000	99.0	-4.9	1007.0	6.914731	

87696 rows × 231 columns

	y	humidity	dew.point	pressure	pressure.log	y_window_
count	87696.000000	87696.000000	87696.00000	87696.000000	87696.000000	
mean	9.635242	77.750276	5.34814	1014.747182	6.922322	
std	6.489911	18.112267	5.09842	12.233392	0.012099	
min	-6.800000	7.000000	-10.00000	950.000000	6.856462	
25%	4.800000	67.000000	1.50000	1008.000000	6.915723	
50%	8.800000	82.000000	5.30000	1016.000000	6.923629	
75%	14.100000	92.000000	9.10000	1023.000000	6.930495	

max	36.100000	100.000000	20.90000	1058.000000	6.964136

8 rows × 230 columns

```
max NA seq len:
Series([], dtype: int64)
```

	r_test	f_test	mi
y_des_window_12_max	0.042134	0.133039	0.042570
y_des_window_12_min	0.041644	0.118076	0.042128
y_des_window_24_max	0.038643	0.066194	0.034513
y_des_window_24_mean	0.038580	0.065519	0.024267
y_des_window_24_min	0.037396	0.054505	0.032091
y_des_window_48_mean	0.036730	0.049467	0.022064
y_des_window_48_max	0.036007	0.044722	0.028959
y_des_window_48_min	0.035093	0.039582	0.027378
dew.point_des_window_12_max	0.034653	0.037390	0.021900
dew.point_des_window_24_max	0.033575	0.032657	0.021935
dew.point_des_window_12_quantile_25	0.032396	0.028317	0.017220
dew.point_des	0.032145	0.027489	0.015490
y_des_window_96_mean	0.031800	0.026396	0.016065
dew.point_des_window_48_mean	0.031717	0.026141	0.015616
dew.point_des_window_48_max	0.031391	0.025167	0.020721
y_des_window_96_min	0.030141	0.021802	0.023115
y_des_window_96_max	0.029859	0.021117	0.023832
dew.point_des_window_12_min	0.028342	0.017807	0.017216
dew.point_des_window_96_mean	0.028053	0.017244	0.013561
dew.point_des_window_24_min	0.027481	0.016179	0.017461
dew.point_des_window_96_max	0.026930	0.015218	0.018976
dew.point_des_window_48_min	0.024628	0.011770	0.016662
y_window_12_min	0.021573	0.008288	0.009977
dew.point_window_12_max	0.020503	0.007296	0.013471
dew.point_des_window_96_min	0.020307	0.007125	0.016481
y_window_12_max	0.020193	0.007027	0.009628
y_window_24_min	0.019480	0.006436	0.009952
dew.point_window_12_min	0.018343	0.005574	0.012213
dew.point_window_48_max	0.017465	0.004969	0.015230
dew.point_window_48_quantile_25	0.017078	0.004907	0.011838

dew.point_window_48_quantile_25	0.017372	0.004907	0.011333
y_window_48_min	0.016822	0.004556	0.009366
y_window_48_max	0.016238	0.004203	0.010552
dew.point_window_48_min	0.015908	0.004012	0.014963
dew.point_window_96_max	0.013702	0.002879	0.015721
dew.point_window_96_quantile_25	0.013382	0.002734	0.010809
y_window_96_min	0.012251	0.002258	0.008927
dew.point_window_96_min	0.012092	0.002196	0.015608
humidity	-0.011791	0.002081	0.002765
y_window_96_std	0.009464	0.001309	0.006208
pressure_window_96_skew	-0.008396	0.001021	0.002742

```
df shape:  (252768, 84)
train shape:    (87696, 109)
valid shape:    (17664, 109)
test shape:     (17664, 109)
df shape:  (252768, 84)
train shape:    (87168, 232)
valid shape:    (17520, 227)
test shape:     (17520, 231)
df shape:  (252768, 84)
train shape:    (87168, 220)
valid shape:    (17520, 220)
test shape:     (17520, 220)

train_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 46.191
mean after feature redundancy:  42.4
```

	before	after	diff
rows	87696	87168	-528
cols	109	220	111
missing_rows	0	528	528
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	4	4	0
low_var_rows	0	0	0

low_var_rows	0	0	0
low_var_cols	17	17	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
valid_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 45.924
mean after feature redundancy:  42.201
```

	before	after	diff
rows	17664	17520	-144
cols	109	220	111
missing_rows	0	144	144
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	4	4	0
low_var_rows	0	0	0
low_var_cols	18	18	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
test_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 45.942
mean after feature redundancy:  42.045
```

	before	after	diff
rows	17664	17520	-144

cols	109	220	111
missing_rows	0	144	144
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	5	5	0
low_var_rows	0	0	0
low_var_cols	17	17	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
WARN: train_sanity != valid_sanity
```

	before	after	diff
missing_rows	0	528	528
low_var_cols	17	17	0
missing_rows	0	144	144
low_var_cols	18	18	0

```
WARN: train_sanity != test_sanity
```

	before	after	diff
missing_rows	0	528	528
single_value_cols	4	4	0
missing_rows	0	144	144
single_value_cols	5	5	0

```
WARN: test_sanity != valid_sanity
```

	before	after	diff
single_value_cols	5	5	0
low_var_cols	17	17	0
single_value_cols	4	4	0
low_var_cols	18	18	0

111 features added (0 NAs) in approximately 1 min.

TODO add `histogram_mode` and `np.std` of `*_grad` features

- both are fast to calculate

This is a huge number of features. All the feature files combined are too large for the git LFS free-usage tier. These files are stored on my google drive which is not publicly shared.

Calculate cross-correlation statistics

Calculate bivariate correlation between core features:

bivariate features:

- `y, dew.point`
- `y, pressure`
- `y_des, dew.point_des`
- `humidity, pressure`
- `pressure, dew.point`

windows:

- 48
- 96

```
def get_rolling_cross_corr_stats(data, feat_cols, windows, aggs, verbose=False, sh
  '''Cross-correlation between features'''

  aggs = pd.DataFrame(index=data.index)

  inf_val_cols = ['humidity', 'pressure']

  for feat1, feat2 in feat_cols:
    # print('feats:', feat1, feat2)
    #if feat1 in inf_val_cols or feat2 in inf_val_cols:
    #  # inf values with window = 48 - probably long sequences of equal values
    #  window = 96
    #else:
    #  window = 48
    # print('window:', window)
    for window in windows:

      # rol = data[feat1].shift(shift_).rolling(window=window, min_periods=1,)
      new_col_name_12 = feat1 + '_' + feat2 + '_window_' + str(window) + '_ccorr'
      new_col_name_21 = feat2 + '_' + feat1 + '_window_' + str(window) + '_ccorr'

      # From https://stackoverflow.com/a/43748653/100129
      # data[new_col_name + 'autocorr'] = data[feat_col].rolling(window=window, mi
      # data[new_col_name + 'autocorr'] = rol.corr(data[feat_col].shift(1)) # NOPE
      aggs[new_col_name_12] = data[feat1].rolling(window=window, min_periods=windo
      aggs[new_col_name_21] = data[feat2].rolling(window=window, min_periods=windo
```

```
  aggs = drop_problem_cols(aggs, window)
  keep_cols = ['y_des', 'dew.point_des', 'pressure', 'humidity']
  aggs = drop_cols_correlated_with_feat_cols(aggs, data[keep_cols])

  return aggs


windows = [48, 96, 144]
ccor_cols = [['y', 'dew.point'],
             ['y', 'pressure'],
             ['y', 'humidity'],  # inf values
             ['y', 'irradiance'],
             ['y_des', 'dew.point_des'],
             ['y_des', 'humidity'],  # inf values
             ['y_des', 'pressure'],
             ['y_des', 'irradiance'],
             ['dew.point_des', 'humidity'],
             ['dew.point_des', 'pressure'],
             ['humidity', 'dew.point'],  # inf values
             ['humidity', 'pressure'],
             ['pressure', 'dew.point'],
            ]
ccor_aggs = ['corr']
params_ccor = {'windows':    windows,
               'feat_cols':  ccor_cols,
               'aggs':       ccor_aggs,
               'agg_func':   get_rolling_cross_corr_stats,
               'verbose':    True,
               'dataset':    'valid',
               'regenerate': True,
               'feat_name':  'cross_corr_',
               'date_str':   '.2022.09.20',
               'save_and_download': False,
               'save_to_gdrive': False,
              }

train_df_ccor, valid_df_ccor, test_df_ccor = get_features(train_df,
                                                          valid_df,
                                                          test_df,
                                                          params_ccor)


#############################################################################
# 30 secs - mostly dataframe checks
#           for dataframe statistics functions on valid_df, test_df & train_df
# 29 features added - 0 NAs
#
# windows = [48, 96, 144]
# ccor_cols = [['y', 'dew.point'],
#              ['y', 'pressure'],
#              ['y', 'humidity'],  # inf values
#              ['y', 'irradiance'],
#              ['y_des', 'dew.point_des'],
```

```
#              ['y_des', 'humidity'],  # inf values
#              ['y_des', 'pressure'],
#              ['y_des', 'irradiance'],
#              ['dew.point_des', 'humidity'],
#              ['dew.point_des', 'pressure'],
#              ['humidity', 'dew.point'],  # inf values
#              ['humidity', 'pressure'],
#              ['pressure', 'dew.point'],]
# ccor_aggs = ['corr']
# params_stat = {'windows':    windows,
#                'feat_cols':  ccor_cols,
#                'aggs':       ccor_aggs,
#                'agg_func':   get_rolling_cross_corr_stats,
#                'verbose':    True,
#                'dataset':    'valid',
#                'regenerate': True,
#                'feat_name':  'cross_corr_',
#                'date_str':   '.2022.09.20', }
```

```
dataset: valid
columns with null values:
y_dew.point_window_48_ccorr                 48
y_dew.point_window_96_ccorr                 96
y_dew.point_window_144_ccorr               144
y_pressure_window_96_ccorr                  96
y_pressure_window_144_ccorr                144
y_humidity_window_48_ccorr                  48
y_humidity_window_96_ccorr                  96
y_humidity_window_144_ccorr                144
y_irradiance_window_48_ccorr                48
y_irradiance_window_96_ccorr                96
y_irradiance_window_144_ccorr              144
y_des_dew.point_des_window_48_ccorr         48
y_des_dew.point_des_window_96_ccorr         96
y_des_dew.point_des_window_144_ccorr       144
y_des_humidity_window_48_ccorr              48
y_des_humidity_window_96_ccorr              96
y_des_humidity_window_144_ccorr            144
y_des_pressure_window_96_ccorr              96
y_des_pressure_window_144_ccorr            144
y_des_irradiance_window_48_ccorr            48
y_des_irradiance_window_96_ccorr            96
y_des_irradiance_window_144_ccorr          144
dew.point_des_humidity_window_48_ccorr      48
dew.point_des_humidity_window_96_ccorr      96
dew.point_des_humidity_window_144_ccorr    144
dew.point_des_pressure_window_96_ccorr      96
dew.point_des_pressure_window_144_ccorr    144
humidity_pressure_window_48_ccorr           57
humidity_pressure_window_96_ccorr           96
humidity_pressure_window_144_ccorr         144
dtype: int64
before: (17664, 109)
after:  (17664, 139)
```

ds	y	humidity	dew.point	pressure	pressure.log	y_winc
ds						
2020-12-29 00:00:00	2020-12-29 00:00:00	0.800000	90.000000	-0.700000	979.000000	6.886532
2020-12-29 00:30:00	2020-12-29 00:30:00	0.800000	94.000000	-0.100000	978.000000	6.885510
2020-12-29 01:00:00	2020-12-29 01:00:00	0.400000	89.000000	-1.200000	979.000000	6.886532
2020-12-29 01:30:00	2020-12-29 01:30:00	0.400000	99.000000	0.300000	979.000000	6.886532
2020-12-29 02:00:00	2020-12-29 02:00:00	0.000000	89.000000	-1.600000	979.000000	6.886532
...	...	...	...	...	...	...

2021-12-31 21:30:00	2021-12-31 21:30:00	12.438743	76.824022	1.414153	1010.049436	6.917755
2021-12-31 22:00:00	2021-12-31 22:00:00	12.473437	76.215531	1.363846	1010.064737	6.917770
2021-12-31 22:30:00	2021-12-31 22:30:00	12.440330	76.236801	1.361175	1010.082097	6.917787
2021-12-31 23:00:00	2021-12-31 23:00:00	12.263176	77.519451	1.132940	1010.095954	6.917801
2021-12-31 23:30:00	2021-12-31 23:30:00	12.269498	77.802763	1.113068	1010.110135	6.917815

17664 rows × 139 columns

	y	humidity	dew.point	pressure	pressure.log	y_windo
count	17664.000000	17664.000000	17664.000000	17664.000000	17664.000000	
mean	9.196032	77.371524	4.790745	1015.718777	6.923282	
std	6.546011	17.357854	5.708252	12.011594	0.011864	
min	-6.000000	6.000000	-11.164754	968.000000	6.875232	
25%	4.400000	67.000000	0.400000	1008.000000	6.915723	
50%	9.200000	81.000000	5.000000	1017.000000	6.924612	
75%	13.600000	90.000000	9.525000	1024.000000	6.931472	
max	29.600000	100.000000	19.100000	1044.000000	6.950815	

8 rows × 138 columns

```
max NA seq len:
humidity_pressure_window_48_ccorr    9
dtype: int64
interpolate: humidity_pressure_window_48_ccorr

dataset: test
columns with null values:
y_dew.point_window_48_ccorr                 48
y_dew.point_window_96_ccorr                 96
y_dew.point_window_144_ccorr               144
y_pressure_window_48_ccorr                  48
y_pressure_window_96_ccorr                  96
y_pressure_window_144_ccorr                144
y_humidity_window_48_ccorr                  48
y_humidity_window_96_ccorr                  96
y_humidity_window_144_ccorr                144
y_irradiance_window_48_ccorr                48
y_irradiance_window_96_ccorr                96
y_irradiance_window_144_ccorr              144
y_des_dew.point_des_window_48_ccorr         48
y_des_dew.point_des_window_96_ccorr         96
```

```
y_des_dew.point_des_window_144_ccorr       144
y_des_humidity_window_48_ccorr              48
y_des_humidity_window_96_ccorr              96
y_des_humidity_window_144_ccorr            144
y_des_pressure_window_48_ccorr              48
y_des_pressure_window_96_ccorr              96
y_des_pressure_window_144_ccorr            144
y_des_irradiance_window_48_ccorr            48
y_des_irradiance_window_96_ccorr            96
y_des_irradiance_window_144_ccorr          144
dew.point_des_humidity_window_48_ccorr      48
dew.point_des_humidity_window_96_ccorr      96
dew.point_des_humidity_window_144_ccorr    144
dew.point_des_pressure_window_48_ccorr      48
dew.point_des_pressure_window_96_ccorr      96
dew.point_des_pressure_window_144_ccorr    144
humidity_pressure_window_48_ccorr           48
humidity_pressure_window_96_ccorr           96
humidity_pressure_window_144_ccorr         144
dtype: int64
before: (17664, 109)
after:  (17664, 142)
```

	ds	y	humidity	dew.point	pressure	pressure.log	y_winc
ds							
2021-12-29 00:00:00	2021-12-29 00:00:00	9.018013	80.712206	1.424063	1007.854619	6.915579	
2021-12-29 00:30:00	2021-12-29 00:30:00	9.024635	80.785103	1.398200	1007.871259	6.915596	
2021-12-29 01:00:00	2021-12-29 01:00:00	8.791503	83.094214	1.432567	1007.888116	6.915612	
2021-12-29 01:30:00	2021-12-29 01:30:00	8.271475	82.180770	1.665441	1007.904126	6.915628	
2021-12-29 02:00:00	2021-12-29 02:00:00	8.271705	82.324921	1.633612	1007.919349	6.915643	
...	...	...	...	...	...	...	...
2022-12-31 21:30:00	2022-12-31 21:30:00	9.600000	80.000000	6.300000	998.000000	6.905753	
2022-12-31 22:00:00	2022-12-31 22:00:00	10.000000	88.000000	8.100000	998.000000	6.905753	
2022-12-31 22:30:00	2022-12-31 22:30:00	10.000000	80.000000	6.700000	999.000000	6.906755	
2022-12-31	2022-12-31	9.600000	79.000000	6.100000	1000.000000	6.907755	

23:00:00	23:00:00						
2022-12-31 23:30:00	2022-12-31 23:30:00	9.200000	82.000000	6.300000	1000.000000	6.907755	

17664 rows × 142 columns

	y	humidity	dew.point	pressure	pressure.log	y_windo
count	17664.000000	17664.000000	17664.000000	17664.000000	17664.000000	
mean	10.370738	73.094140	4.926538	1016.288722	6.923853	
std	6.887136	19.453441	4.919329	11.053516	0.010903	
min	-7.144690	5.000000	-10.000000	972.000000	6.879356	
25%	5.600000	60.000000	1.500000	1009.000000	6.916715	
50%	10.000000	77.000000	5.000000	1017.000000	6.924612	
75%	15.000000	88.000000	8.500000	1024.000000	6.931472	
max	37.200000	100.000000	19.100000	1047.000000	6.953684	

8 rows × 141 columns

```
max NA seq len:
Series([], dtype: int64)

dataset: train
columns with null values:
y_dew.point_window_48_ccorr                48
y_dew.point_window_96_ccorr                96
y_dew.point_window_144_ccorr              144
y_pressure_window_96_ccorr                 96
y_pressure_window_144_ccorr               144
y_humidity_window_48_ccorr                 48
y_humidity_window_96_ccorr                 96
y_humidity_window_144_ccorr               144
y_irradiance_window_48_ccorr               48
y_irradiance_window_96_ccorr               96
y_irradiance_window_144_ccorr             144
y_des_dew.point_des_window_48_ccorr        48
y_des_dew.point_des_window_96_ccorr        96
y_des_dew.point_des_window_144_ccorr      144
y_des_humidity_window_48_ccorr             48
y_des_humidity_window_96_ccorr             96
y_des_humidity_window_144_ccorr           144
y_des_pressure_window_96_ccorr             96
y_des_pressure_window_144_ccorr           144
y_des_irradiance_window_48_ccorr           48
y_des_irradiance_window_96_ccorr           96
y_des_irradiance_window_144_ccorr         144
dew.point_des_humidity_window_48_ccorr     48
dew.point_des_humidity_window_96_ccorr     96
dew.point_des_humidity_window_144_ccorr   144
dew.point_des_pressure_window_96_ccorr     96
dew.point_des_pressure_window_144_ccorr   144
humidity_pressure_window_96_ccorr          96
humidity_pressure_window_144_ccorr        144
dtype: int64
before: (87696, 109)
```

```
after:  (87696, 138)
```

	ds	y	humidity	dew.point	pressure	pressure.log	y_windo
ds							
2016-01-01 00:00:00	2016-01-01 00:00:00	10.087433	57.0	3.2	1019.0	6.926577	
2016-01-01 00:30:00	2016-01-01 00:30:00	10.088860	72.0	6.5	1020.0	6.927558	
2016-01-01 01:00:00	2016-01-01 01:00:00	10.084001	83.0	8.6	1021.0	6.928538	
2016-01-01 01:30:00	2016-01-01 01:30:00	10.004534	88.0	9.5	1021.0	6.928538	
2016-01-01 02:00:00	2016-01-01 02:00:00	10.008839	91.0	10.0	1021.0	6.928538	
...	...	...	...	...	...	...	
2020-12-31 21:30:00	2020-12-31 21:30:00	-2.800000	96.0	-3.3	1006.0	6.913737	
2020-12-31 22:00:00	2020-12-31 22:00:00	-3.200000	100.0	-3.2	1007.0	6.914731	
2020-12-31 22:30:00	2020-12-31 22:30:00	-3.600000	100.0	-3.6	1007.0	6.914731	
2020-12-31 23:00:00	2020-12-31 23:00:00	-4.400000	97.0	-4.8	1007.0	6.914731	
2020-12-31 23:30:00	2020-12-31 23:30:00	-4.800000	99.0	-4.9	1007.0	6.914731	

87696 rows × 138 columns

	y	humidity	dew.point	pressure	pressure.log	y_window_
count	87696.000000	87696.000000	87696.00000	87696.000000	87696.000000	
mean	9.635242	77.750276	5.34814	1014.747182	6.922322	
std	6.489911	18.112267	5.09842	12.233392	0.012099	
min	-6.800000	7.000000	-10.00000	950.000000	6.856462	
25%	4.800000	67.000000	1.50000	1008.000000	6.915723	
50%	8.800000	82.000000	5.30000	1016.000000	6.923629	
75%	14.100000	92.000000	9.10000	1023.000000	6.930495	

max	36.100000	100.000000	20.90000	1058.000000	6.964136

8 rows × 137 columns

```
max NA seq len:
Series([], dtype: int64)
```

	r_test	f_test	mi
dew.point_des	0.722249	0.715602	0.105804
humidity	-0.264918	0.054168	0.018893
humidity_pressure_window_144_ccorr	0.204525	0.031423	0.025915
y_des_irradiance_window_144_ccorr	0.180396	0.024234	0.036872
y_des_irradiance_window_96_ccorr	0.172805	0.022181	0.030719
humidity_pressure_window_96_ccorr	0.171657	0.021879	0.022719
y_window_48_min_max_diff	0.150924	0.016809	0.063681
dew.point_des_humidity_window_144_ccorr	-0.125474	0.011543	0.028030
y_des_irradiance_window_48_ccorr	0.125238	0.011499	0.027746
y_des_humidity_window_96_ccorr	-0.117426	0.010091	0.029304
irradiance	0.107402	0.008424	0.006899
y_des_humidity_window_144_ccorr	-0.100703	0.007397	0.035064
y_des_humidity_window_48_ccorr	-0.099326	0.007194	0.024381
dew.point_des_pressure_window_144_ccorr	0.096525	0.006791	0.031852
dew.point_des_pressure_window_96_ccorr	0.090555	0.005971	0.022965
dew.point_des_humidity_window_96_ccorr	-0.090224	0.005927	0.022541
y_des_dew.point_des_window_144_ccorr	0.084118	0.005147	0.035063
y_pressure_window_96_ccorr	-0.080124	0.004667	0.020544
y_humidity_window_96_ccorr	-0.079489	0.004593	0.025828
y_pressure_window_144_ccorr	-0.070568	0.003615	0.027591
y_des_pressure_window_96_ccorr	-0.064438	0.003012	0.016665
y_des_dew.point_des_window_96_ccorr	0.062188	0.002805	0.027116
y_humidity_window_48_ccorr	-0.061906	0.002779	0.018470
y_humidity_window_144_ccorr	-0.056944	0.002350	0.038919
y_des_dew.point_des_window_48_ccorr	0.056067	0.002278	0.012426
y_des_pressure_window_144_ccorr	-0.051861	0.001948	0.024941
y_irradiance_window_144_ccorr	0.048143	0.001679	0.034182
y_irradiance_window_96_ccorr	0.035859	0.000930	0.027531
y_dew.point_window_48_ccorr	-0.034105	0.000841	0.015410

za_rad	-0.032450	0.000762	0.009355
dew.point_des_humidity_window_48_ccorr	-0.027833	0.000560	0.021017
y_dew.point_window_144_ccorr	0.023532	0.000400	0.033279
azimuth	0.019298	0.000269	0.005675
pressure	0.013164	0.000125	0.024380
y_dew.point_window_96_ccorr	-0.011320	0.000093	0.024433
y_irradiance_window_48_ccorr	0.004464	0.000014	0.023790

```
df shape:  (252768, 84)
train shape:    (87696, 109)
valid shape:    (17664, 109)
test shape:     (17664, 109)
df shape:  (252768, 84)
train shape:    (87168, 139)
valid shape:    (17520, 139)
test shape:     (17520, 142)
df shape:  (252768, 84)
train shape:    (87168, 138)
valid shape:    (17520, 138)
test shape:     (17520, 138)

train_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 46.191
mean after feature redundancy:  36.504
```

	before	after	diff
rows	87696	87168	-528
cols	109	138	29
missing_rows	0	528	528
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	4	4	0
low_var_rows	0	0	0
low_var_cols	17	17	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0

duplicate_col_labels	0	0	0

```
valid_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 45.924
mean after feature redundancy:  36.315
```

	before	after	diff
rows	17664	17520	-144
cols	109	138	29
missing_rows	0	144	144
missing_cols	0	0	0
total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	4	4	0
low_var_rows	0	0	0
low_var_cols	18	18	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
test_df
before.index.equals(after.index): False
before.index.freq == after.index.freq: True
before[common_cols].dtypes == after[common_cols].dtypes: True
before[common_cols].describe() == after[common_cols].describe(): False

before[common_cols].equals(after[common_cols]): False
redundancy before > after: True
mean before feature redundancy: 45.942
mean after feature redundancy:  36.275
```

	before	after	diff
rows	17664	17520	-144
cols	109	138	29
missing_rows	0	144	144
missing_cols	0	0	0

total_nas	0	0	0
rows_with_nas	0	0	0
cols_with_nas	0	0	0
single_value_cols	5	5	0
low_var_rows	0	0	0
low_var_cols	17	17	0
duplicate_rows	0	0	0
duplicate_index_labels	0	0	0
duplicate_col_labels	0	0	0

```
WARN: train_sanity != valid_sanity
```

	before	after	diff
missing_rows	0	528	528
low_var_cols	17	17	0
missing_rows	0	144	144
low_var_cols	18	18	0

```
WARN: train_sanity != test_sanity
```

	before	after	diff
missing_rows	0	528	528
single_value_cols	4	4	0
missing_rows	0	144	144
single_value_cols	5	5	0

```
WARN: test_sanity != valid_sanity
```

	before	after	diff
single_value_cols	5	5	0
low_var_cols	17	17	0
single_value_cols	4	4	0
low_var_cols	18	18	0

29 features added (0 NAs) in approximately 1 min.

Files are saved in Apache parquet format and are available on github:

- https://github.com/makeyourownmaker/CambridgeTemperatureNotebooks/tree/main/data/features

---

Plot short term autocorrelations

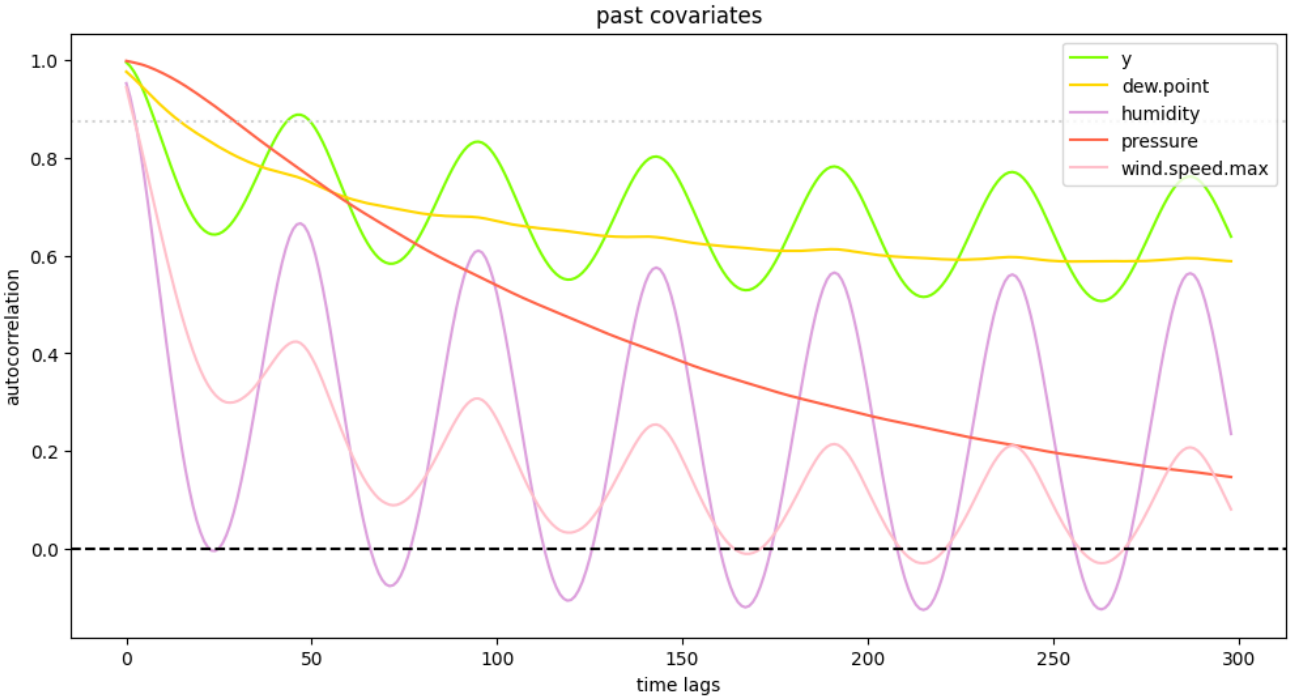
Correlation measures the linear relationship between two variables. Autocorrelation measures the linear relationship between lagged values of a time series.

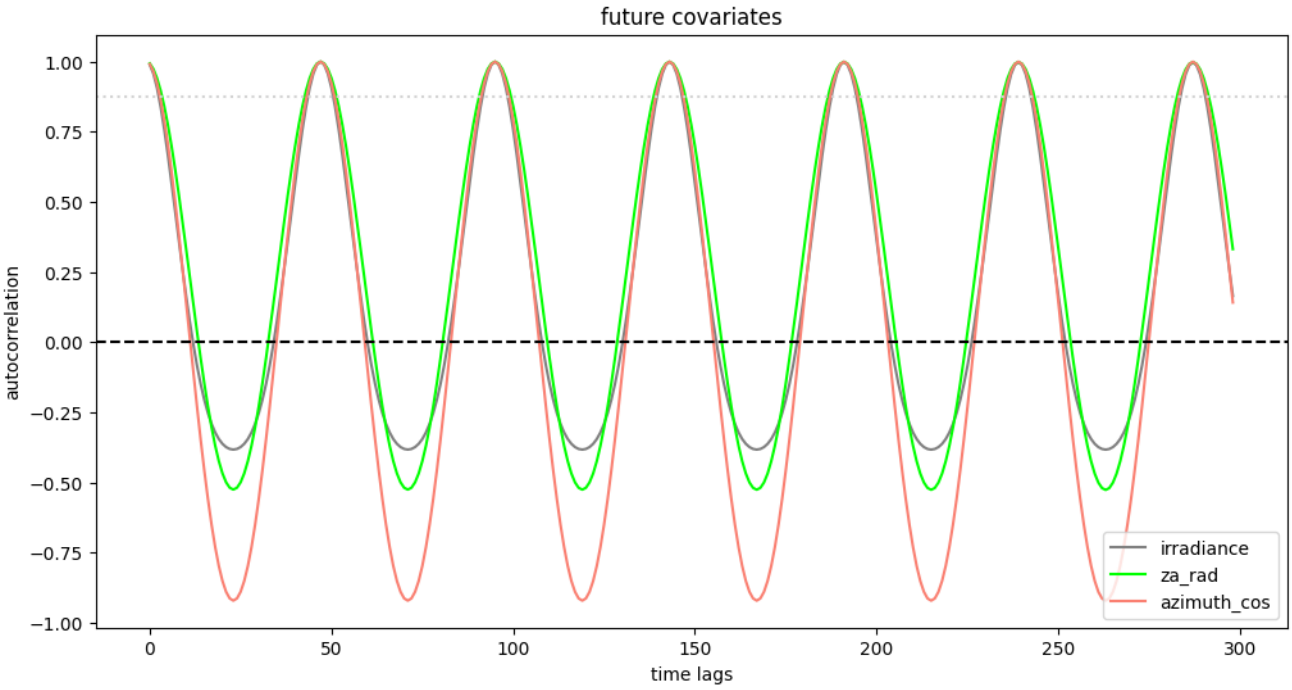
Compare main features

- past covariates
- future covariates

Short term autocorrelations first.

```
pc_feats = ['y', 'dew.point', 'humidity', 'pressure', 'wind.speed.max']
pc_cols  = ['chartreuse', 'gold', 'plum', 'tomato', 'pink']
plot_short_term_acf(df, pc_feats, pc_cols, 'past covariates')

fc_feats = ['irradiance', 'za_rad', 'azimuth_cos']
fc_cols  = ['grey', 'lime', 'salmon']
plot_short_term_acf(df, fc_feats, fc_cols, 'future covariates')
```


past covariates
y
dew.point
humidity
pressure
wind.speed.max
autocorrelation
time lags
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300


future covariates
irradiance
za_rad
azimuth_cos
autocorrelation
time lags
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
50
100
150
200
250
300

`y`, `humidity`, `wind.speed.max`, `irradiance`, `za_rad` (zenith angle) and `azimuth_cos` have strong daily seasonality.

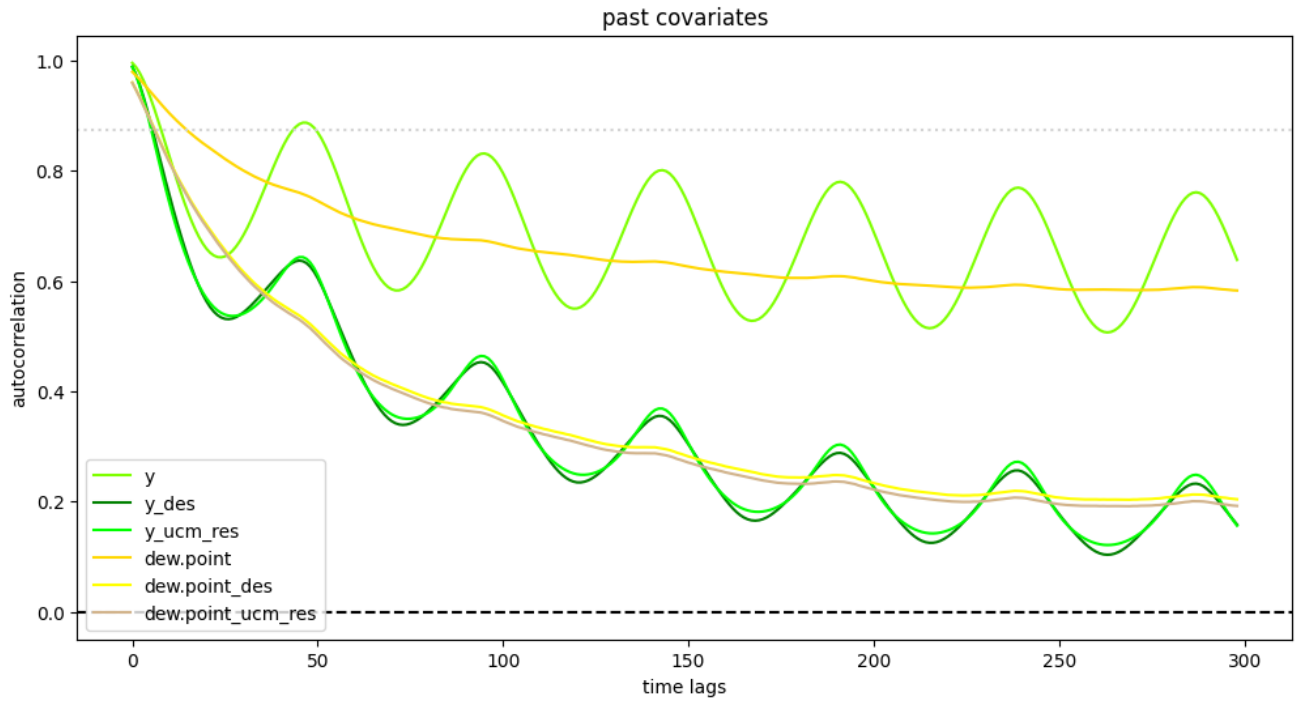
Autocorrelation of `pressure` and `dew.point` are notably different.

`pressure` has surprisingly high initial autocorrelation.

Compare core and deseasonalised features

Compare UCM and prophet decompositions.

```
pc_feats = ['y', 'y_des', 'y_ucm_res', 'dew.point', 'dew.point_des',
            'dew.point_ucm_res']
pc_cols  = ['chartreuse', 'green', 'lime', 'gold', 'yellow', 'tan']
plot_short_term_acf(train_df, pc_feats, pc_cols, 'past covariates')
```



The prophet and UCM decompositions are remarkably similar. Both decomposition approaches removed a substantial amount of seasonality.

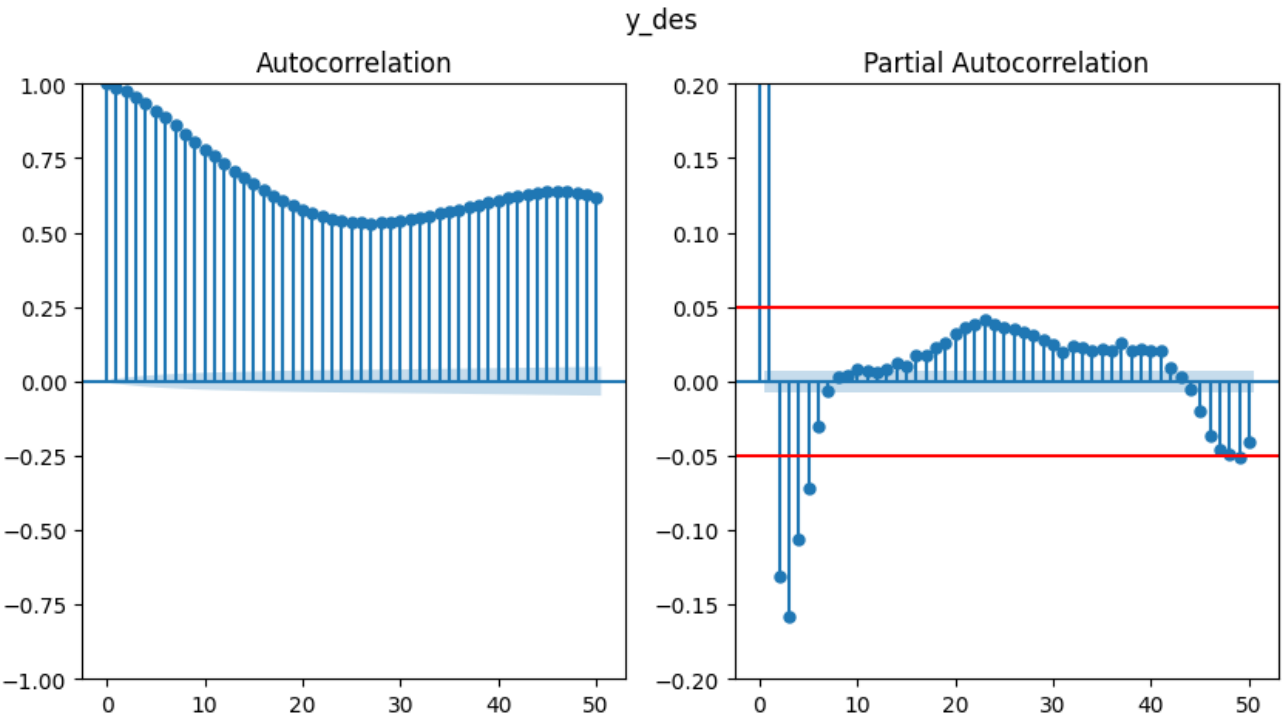
Plot partial autocorrelations

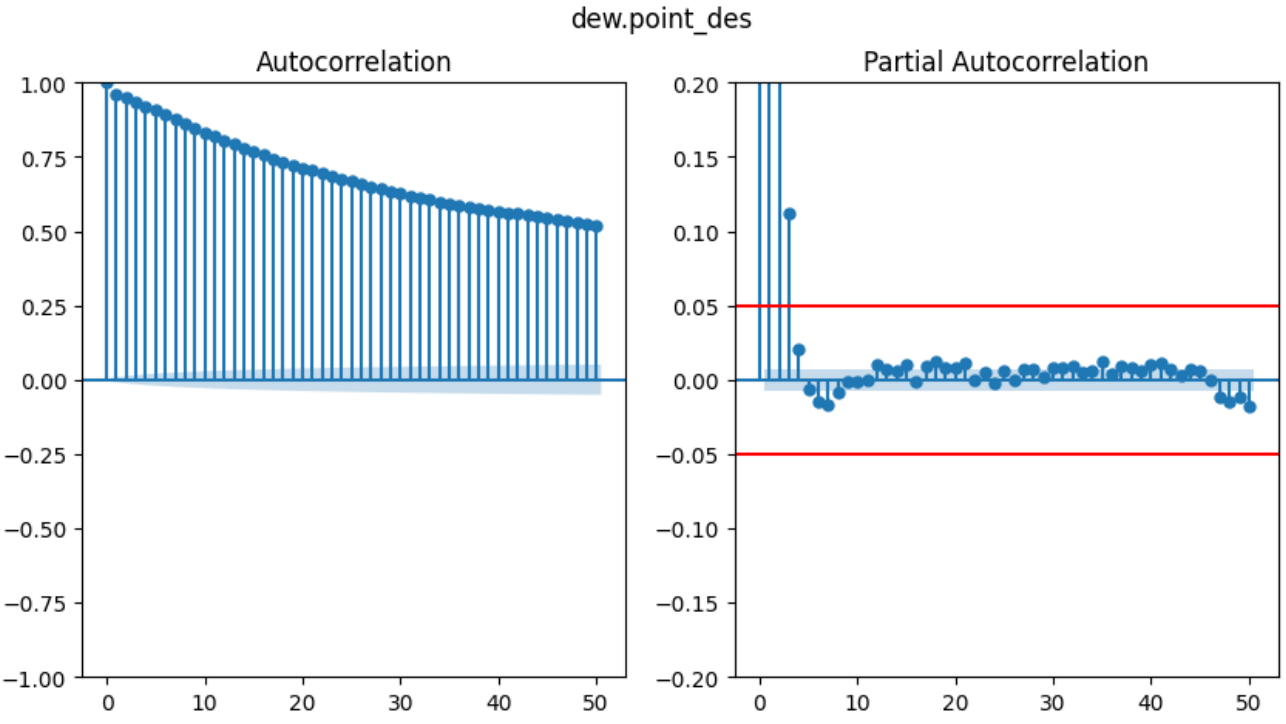
The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.

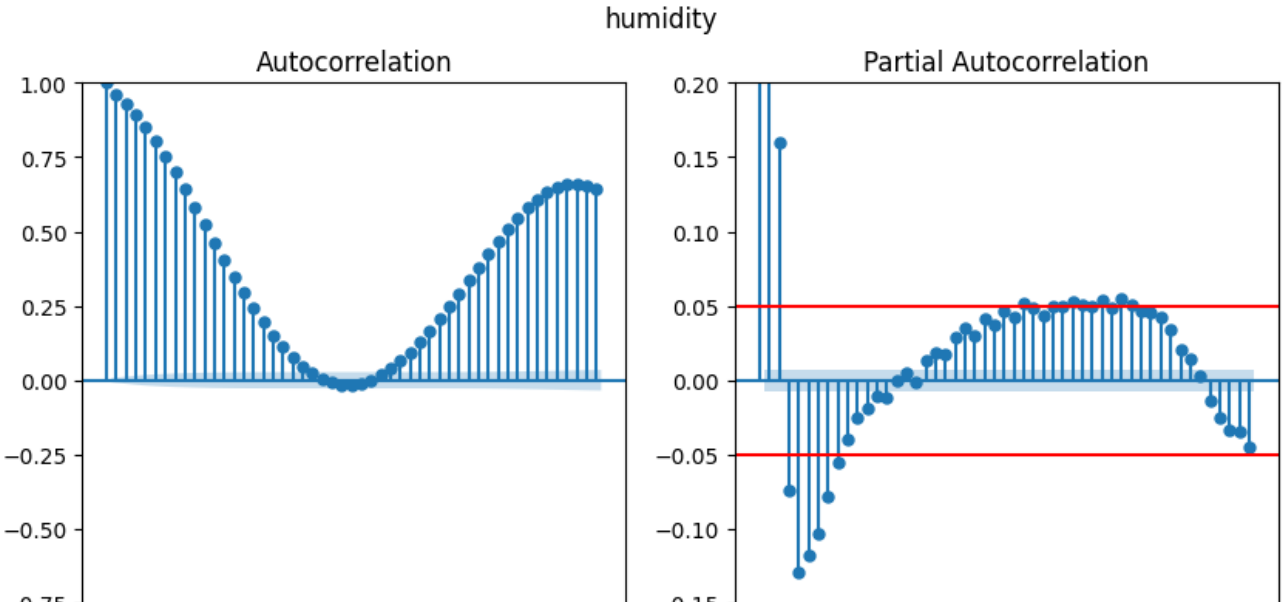
Here I plot both the autocorrelation and partial autocorrelations for:

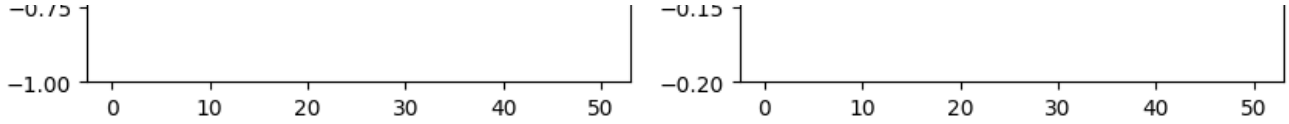
- core features
- future covariates

```
n = int(365.2425 * 48) * 12
max_lag = 50
pc_feats = ['y_des', 'dew.point_des', 'humidity', 'pressure',
            'irradiance', 'za_rad', 'azimuth_cos']
pacf_lim = 0.05
alpha = 0.001

for pc_feat in pc_feats:
  fig, ax = plt.subplots(1, 2, figsize=(10, 5))
  sm.graphics.tsa.plot_acf(train_df[pc_feat].head(n),  alpha=alpha, lags=max_lag,
  sm.graphics.tsa.plot_pacf(train_df[pc_feat].head(n), alpha=alpha, lags=max_lag,
  plt.axhline(y=pacf_lim,  color='r', linestyle='-')
  plt.axhline(y=-pacf_lim, color='r', linestyle='-')
  ax[1].set_ylim((-0.2, 0.2))
  plt.suptitle(pc_feat)
  plt.show()
```

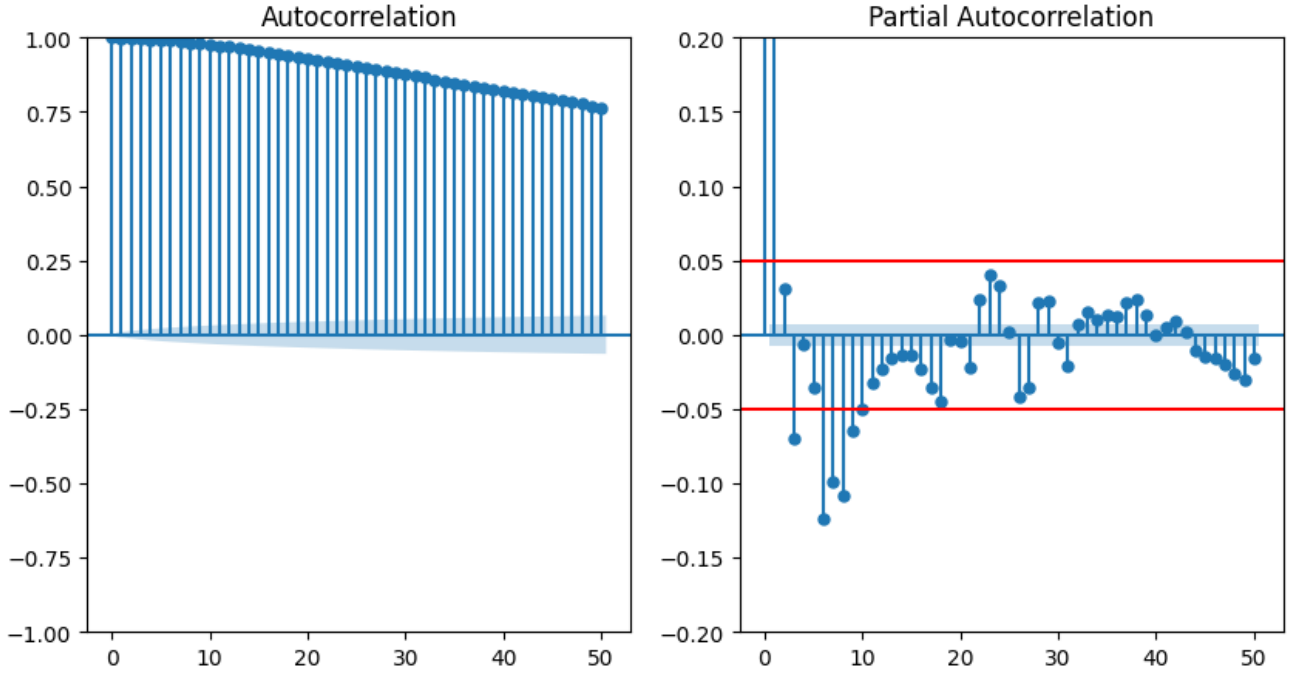

y_des
Autocorrelation
Partial Autocorrelation


dew.point_des
Autocorrelation
Partial Autocorrelation

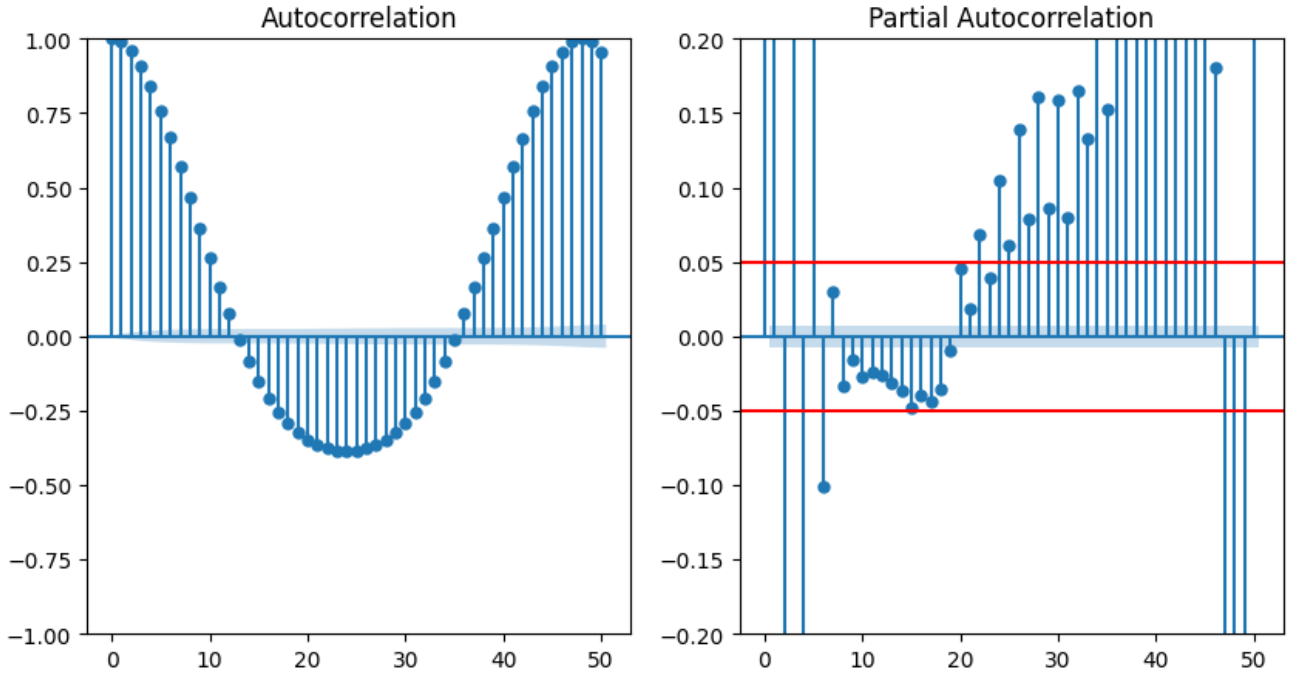

humidity
Autocorrelation
Partial Autocorrelation



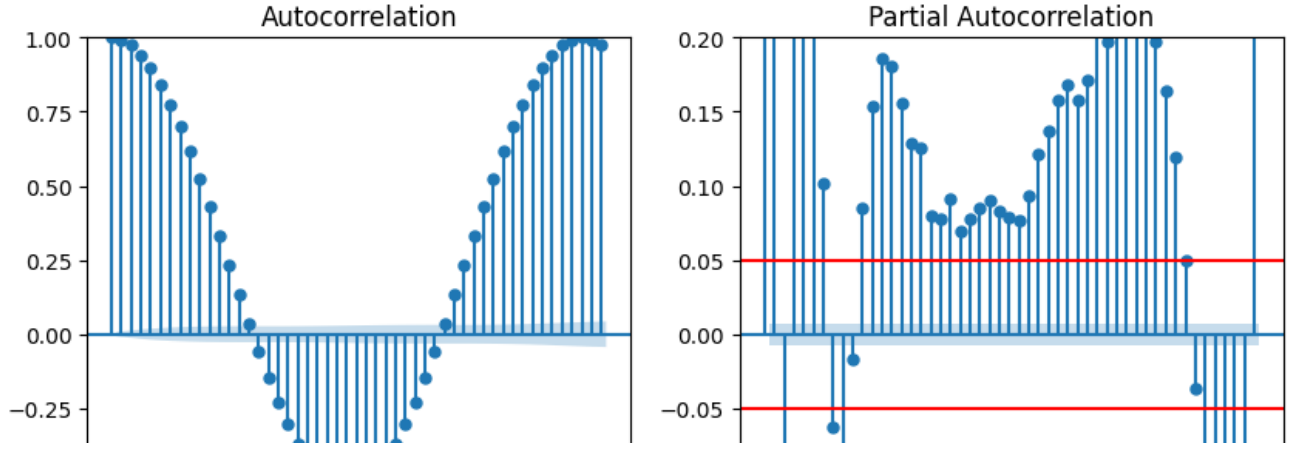
pressure

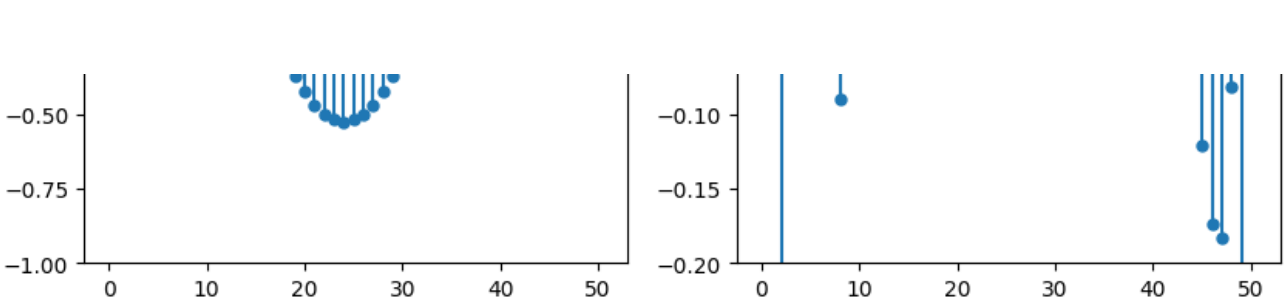


irradiance

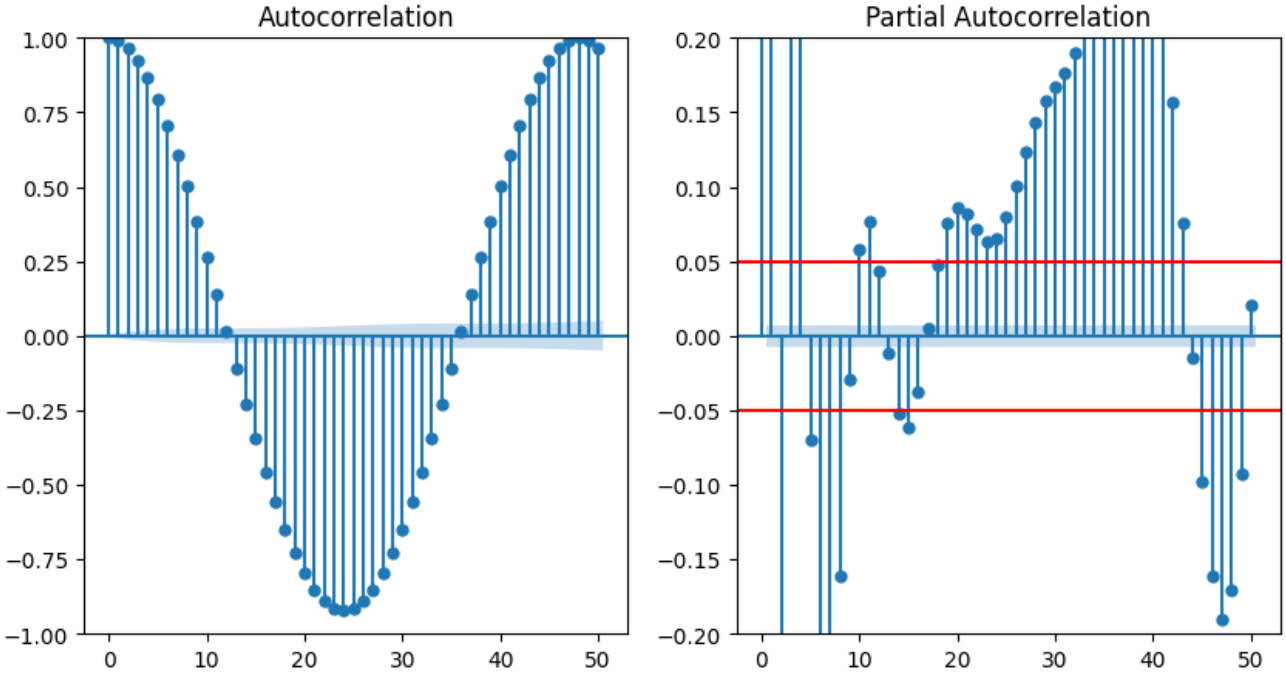


za_rad




−0.50
−0.75
−1.00
0
10
20
30
40
50
−0.10
−0.15
−0.20

azimuth_cos


Autocorrelation
Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0
10
20
30
40
50