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, 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

```
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_co
      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_mic
  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")
```

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_d
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_ST


year_splines = periodic_spline_transformer(YEAR, n_splines=12).fit_transform(mont
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, 60)

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 60 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  day.sin.1                       252768 non-null  float64
 40  day.cos.1                       252768 non-null  float64
 41  year.sin.1                      252768 non-null  float64
 42  year.cos.1                      252768 non-null  float64
 43  missing                         252768 non-null  int64
```

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

```
      01        08-01   19.1   85.572379     1.397409   1009.902342          6.917609
```

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

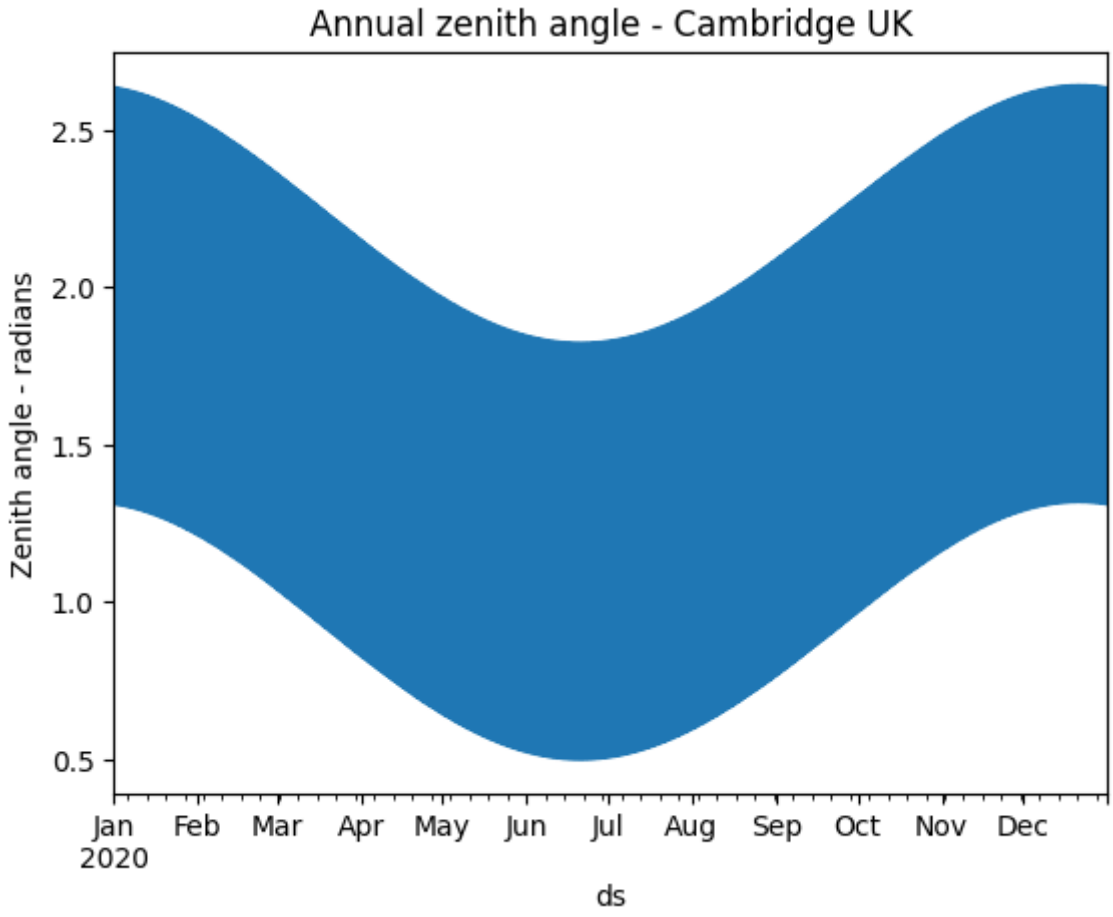
```
if missing == required:
    print('Installing', missing, '...', sep=' ', end='')
    python = sys.executable
    subprocess.check_call([python, '-m', 'pip', 'install', *missing])  #, stdout=s
    print(' Done')


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)
```

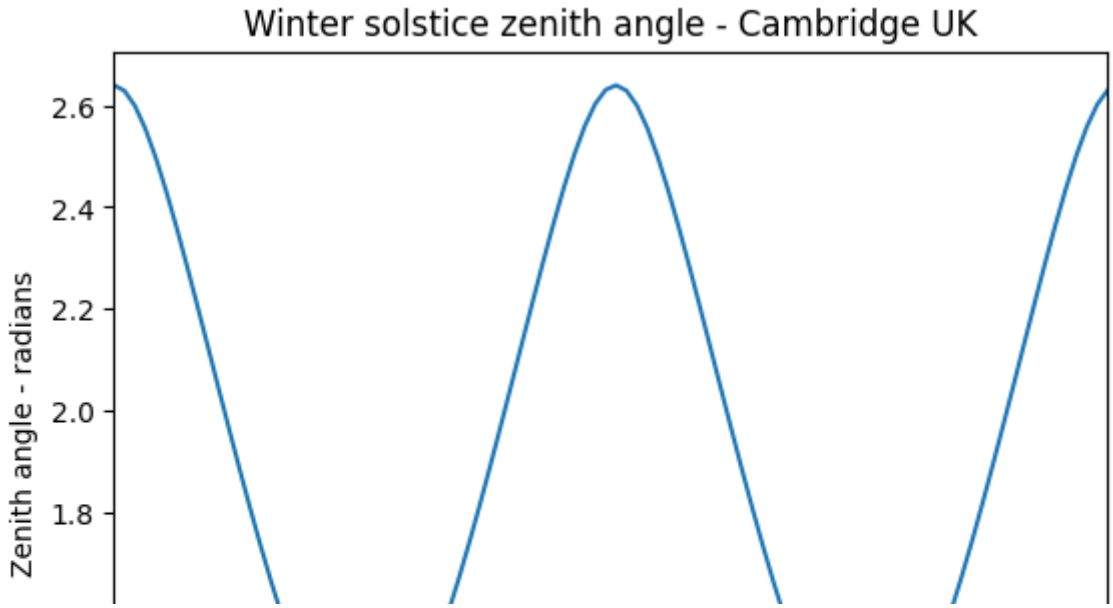
```
    za_rad[i] = np.radians(za[i])
    i += 1

  # 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 {'pysolar', 'solarpy'} ... Done
Zenith
Calculating zenith: 100%|██████████| 17568/17568 [00:16<00:00, 1041.51it/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
```





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

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.

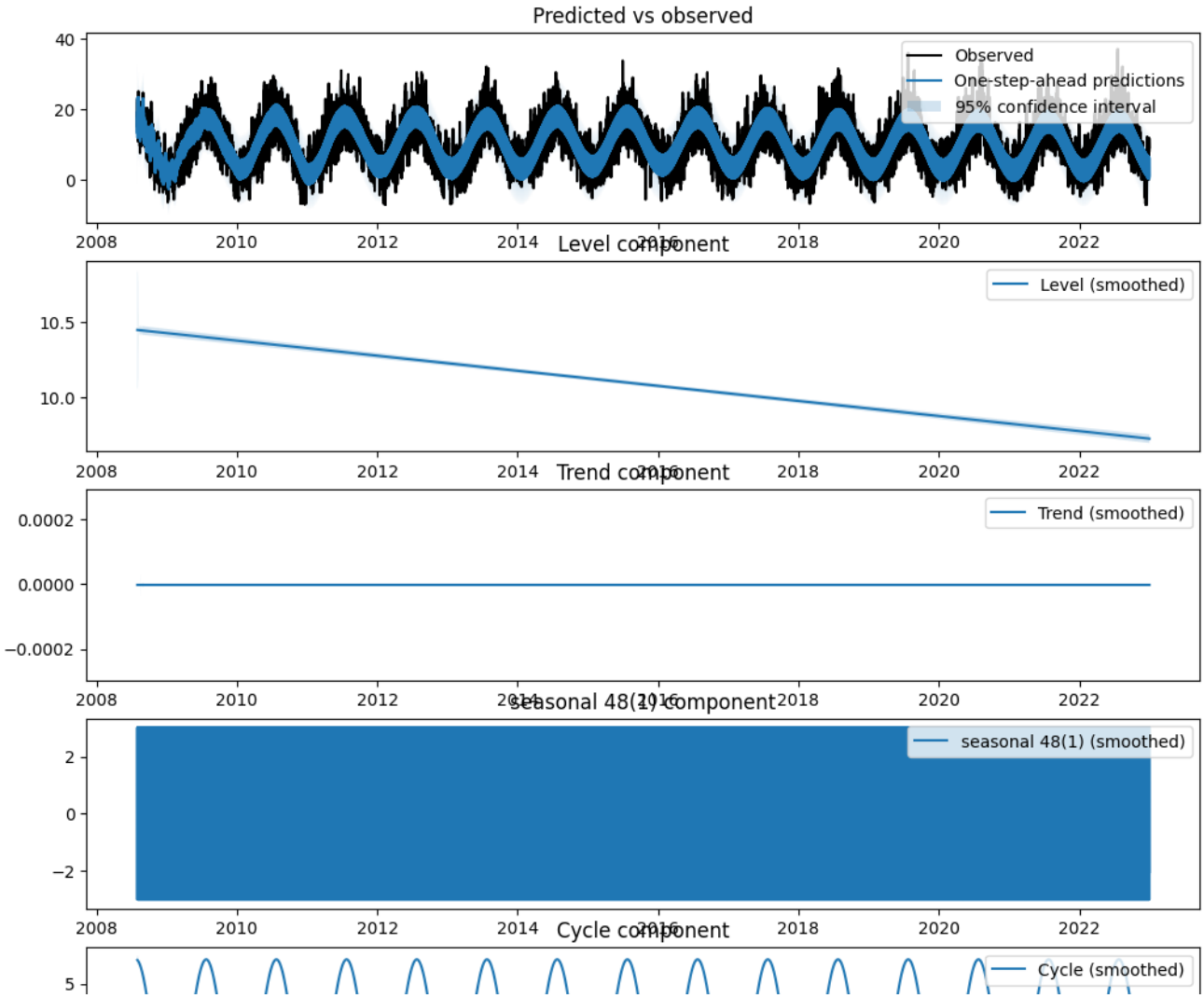
```
def get_uc_model(data, params, y_col='y'):
  ucm = sm.tsa.UnobservedComponents(data[y_col], #.dropna().values,
                                    **params)
  res_f = ucm.fit(method='powell', disp=False)

  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 = 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:                     Wed, 07 Feb 2024   HQIC                      1371905
Time:                             09:00:49
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])
```



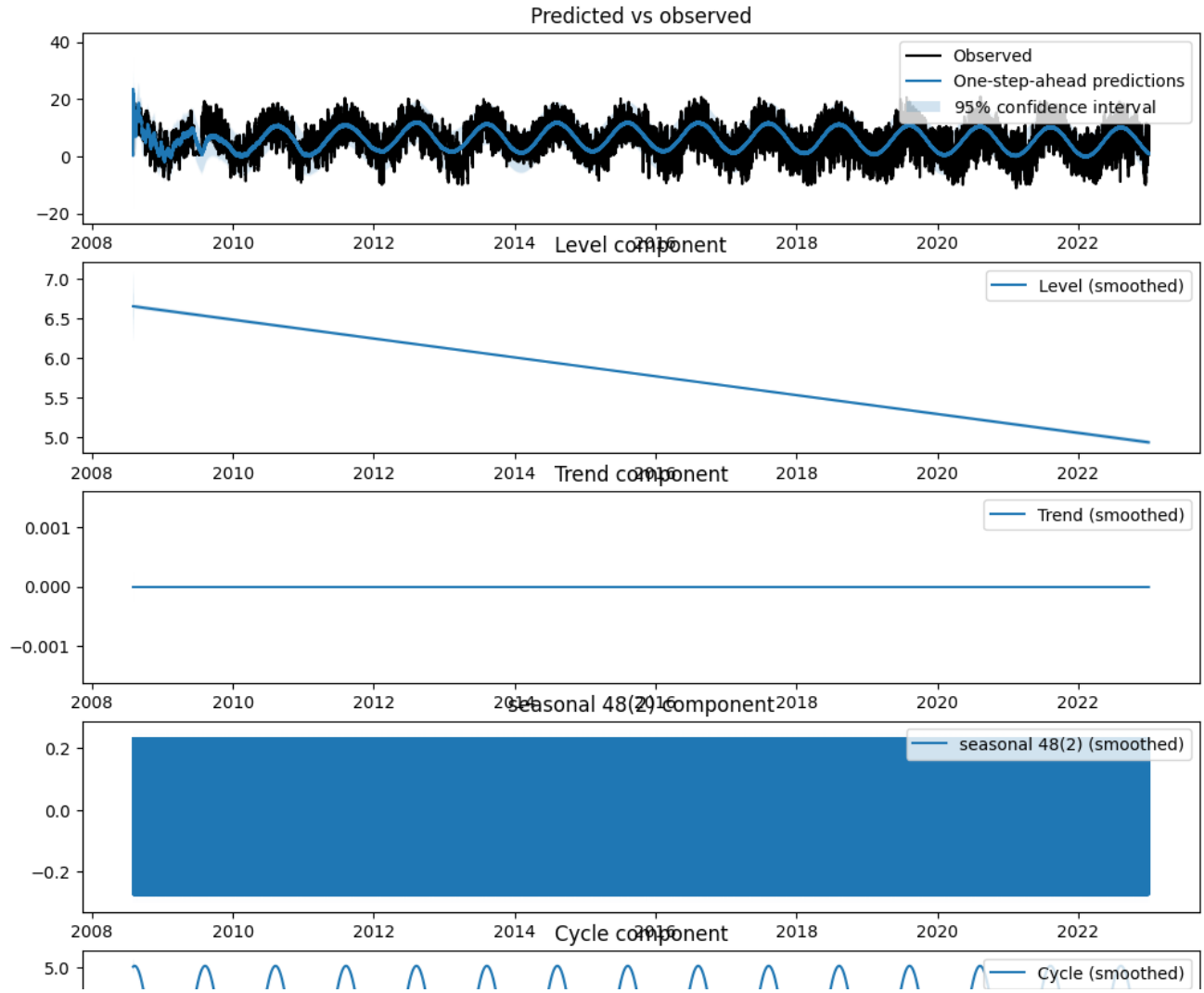
The temperature decomposition seemed acceptible.

Initial few months of decomposition show high variance.

`dew.point` second:

```
df_before_ucm_dp = 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       Log Likelihood           -685958
                    + freq_seasonal(48(2))      AIC                      1371921
                                  + cycle       BIC                      1371941
Date:                     Wed, 07 Feb 2024      HQIC                     1371927
Time:                             09:02:40
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])
```



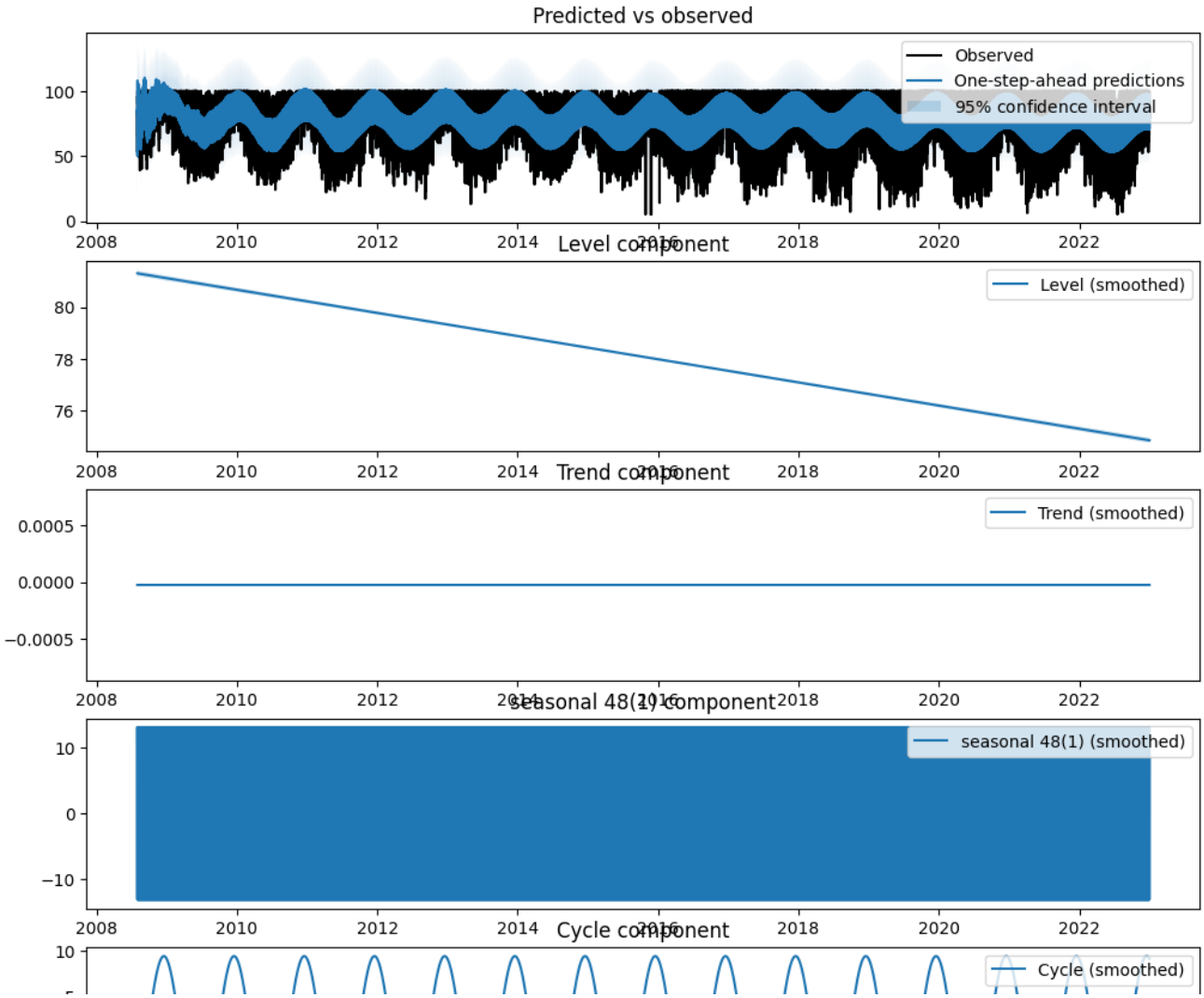
The due point decomposition seemed acceptible. Note the daily bimodal peaks.

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(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:                     Wed, 07 Feb 2024   HQIC                      2022055
Time:                             09:05:03
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])
```



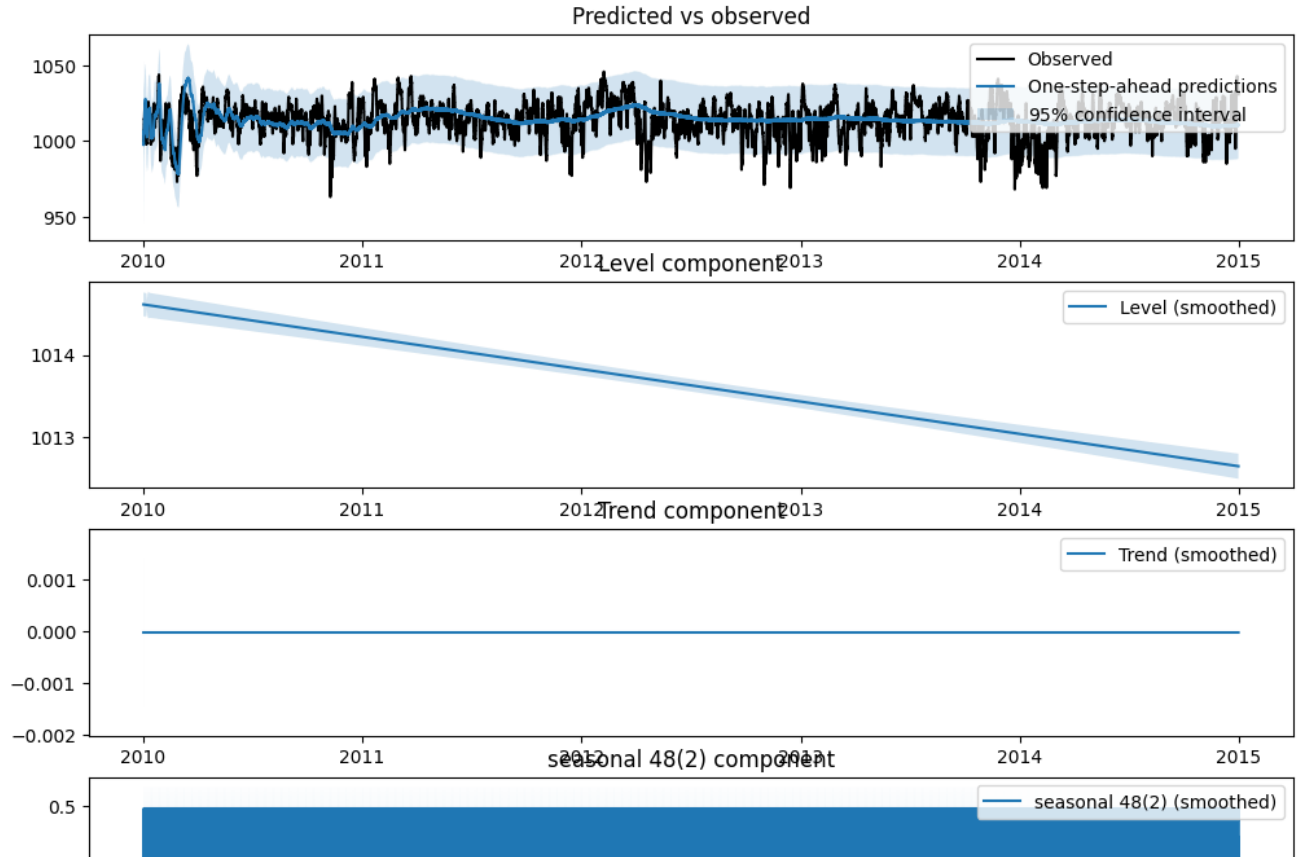
The `humidity` decomposition is problematic.

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:                     Wed, 07 Feb 2024   HQIC                        676468
Time:                             09:05:58
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])
```



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

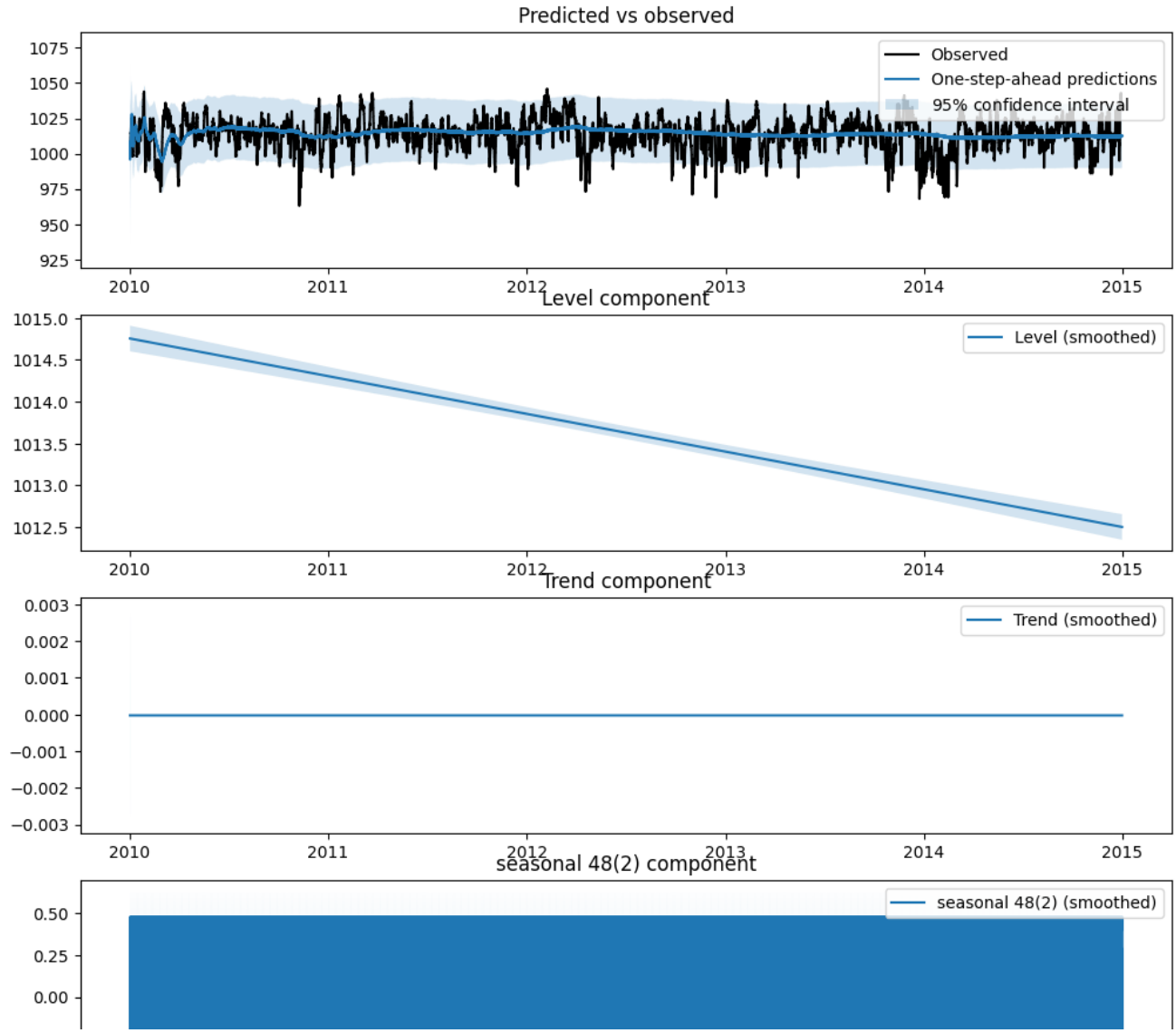
Dep. Variable:	pressure	No. Observations:	8
Model:	deterministic trend + freq_seasonal(48(2))	Log Likelihood	-338895
		AIC	677793
Date:	Wed, 07 Feb 2024	BIC	677802
Time:	09:06:21	HQIC	677796
Sample:	01-01-2010 - 12-31-2014		
Covariance Type:	opg		

	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])
```



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

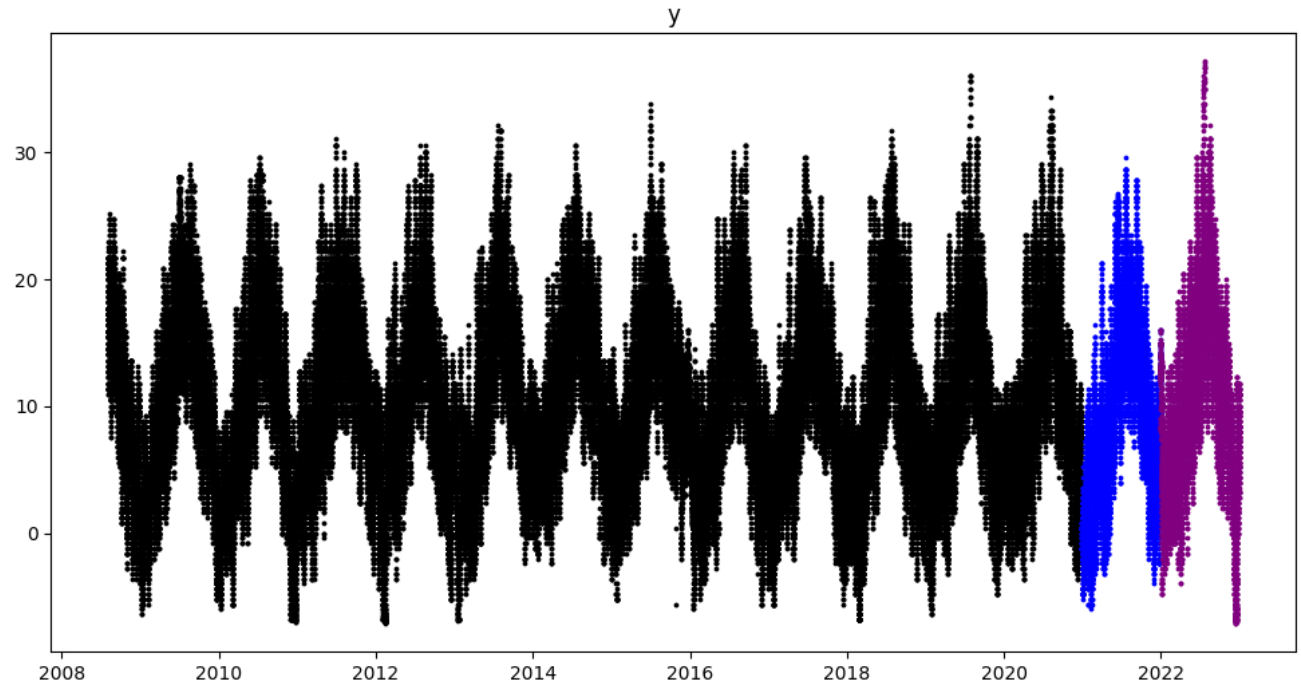
Initial few months of decomposition show high variance.

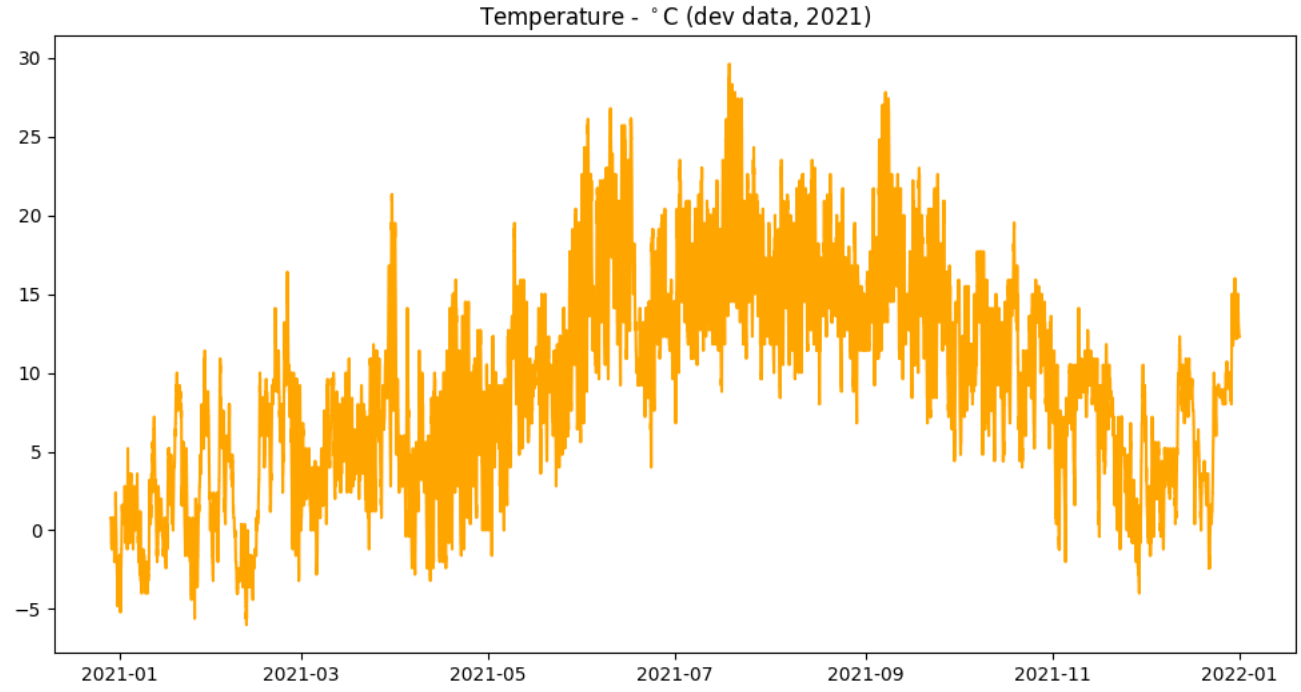
Split Data

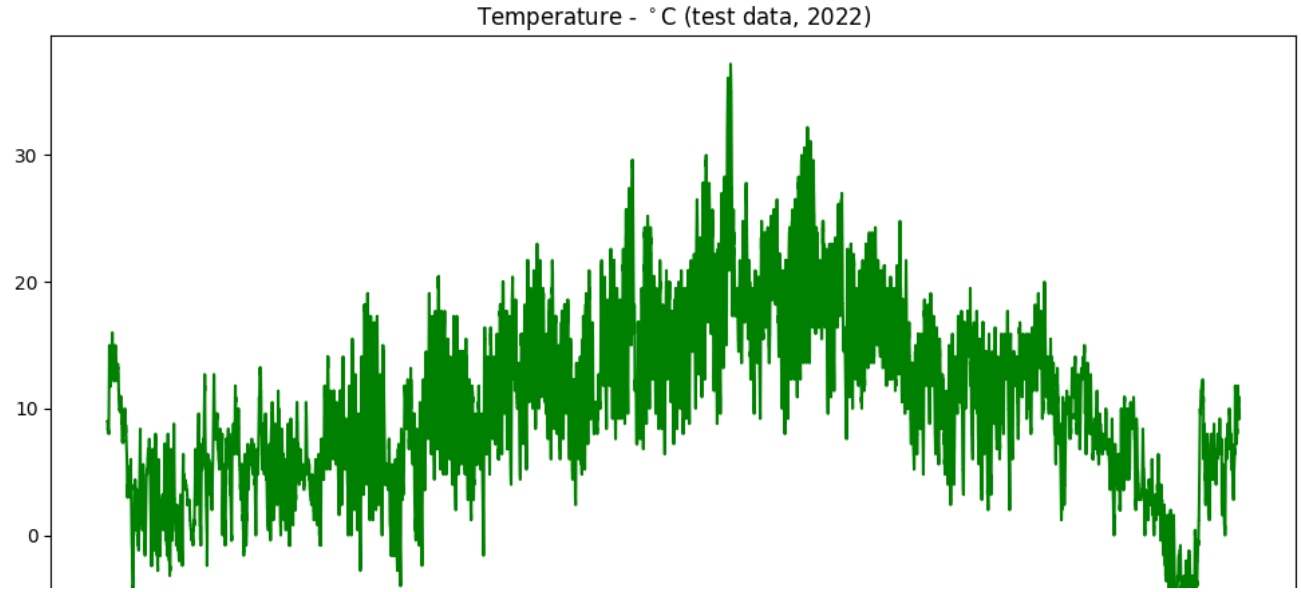
I use data from 2021 for validation, 2022 for testing and data before 2021 for training. These are entirely arbitrary choices. This results in an approximate 88%, 6%, 6% split for the training, validation, and test sets respectively.

```
# 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

Normalise Data

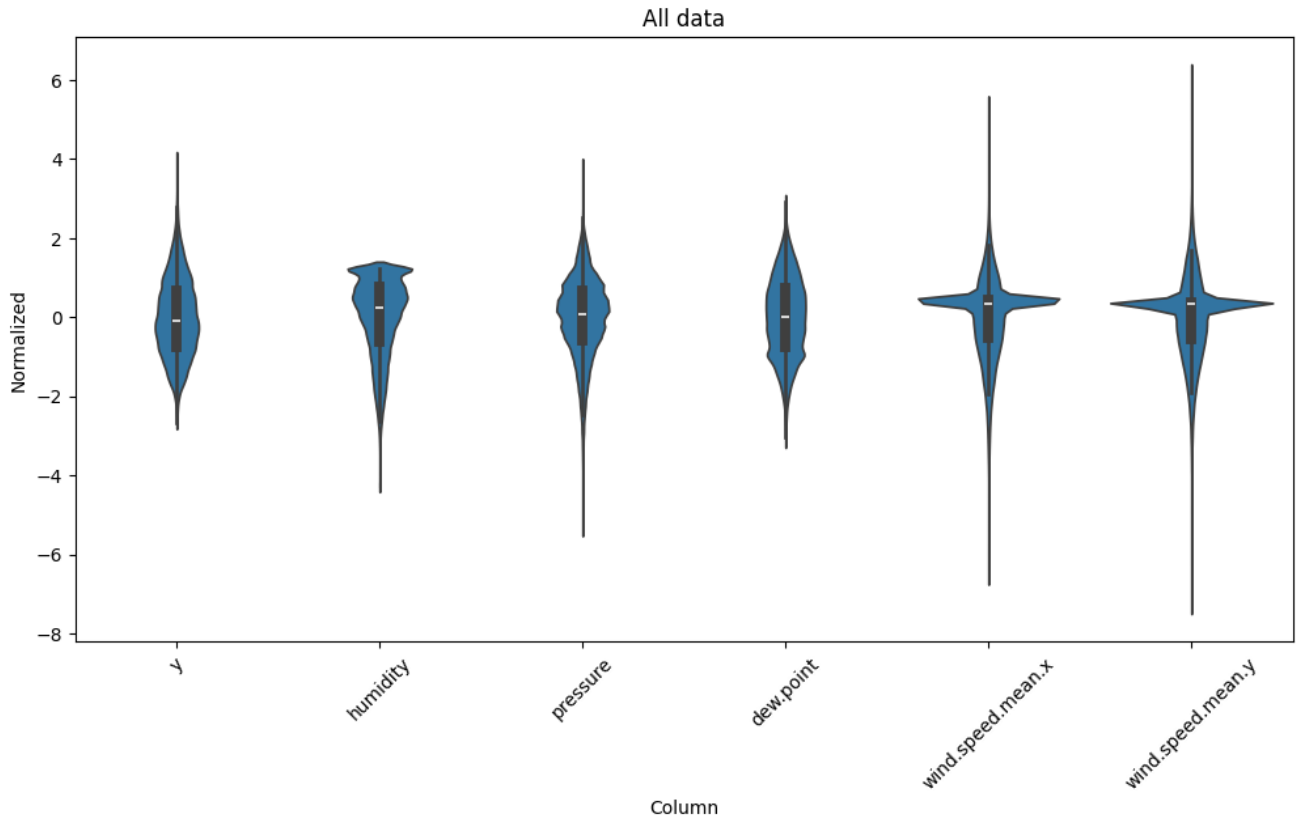
Features do not need to be scaled for gradient boosting methods. Nonetheless, it can often be a useful sanity check to plot these values.

The violin plot shows the distribution of features.

```
    ERROR: Overlap between valid_df, test_df indices!
```

```
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()
```

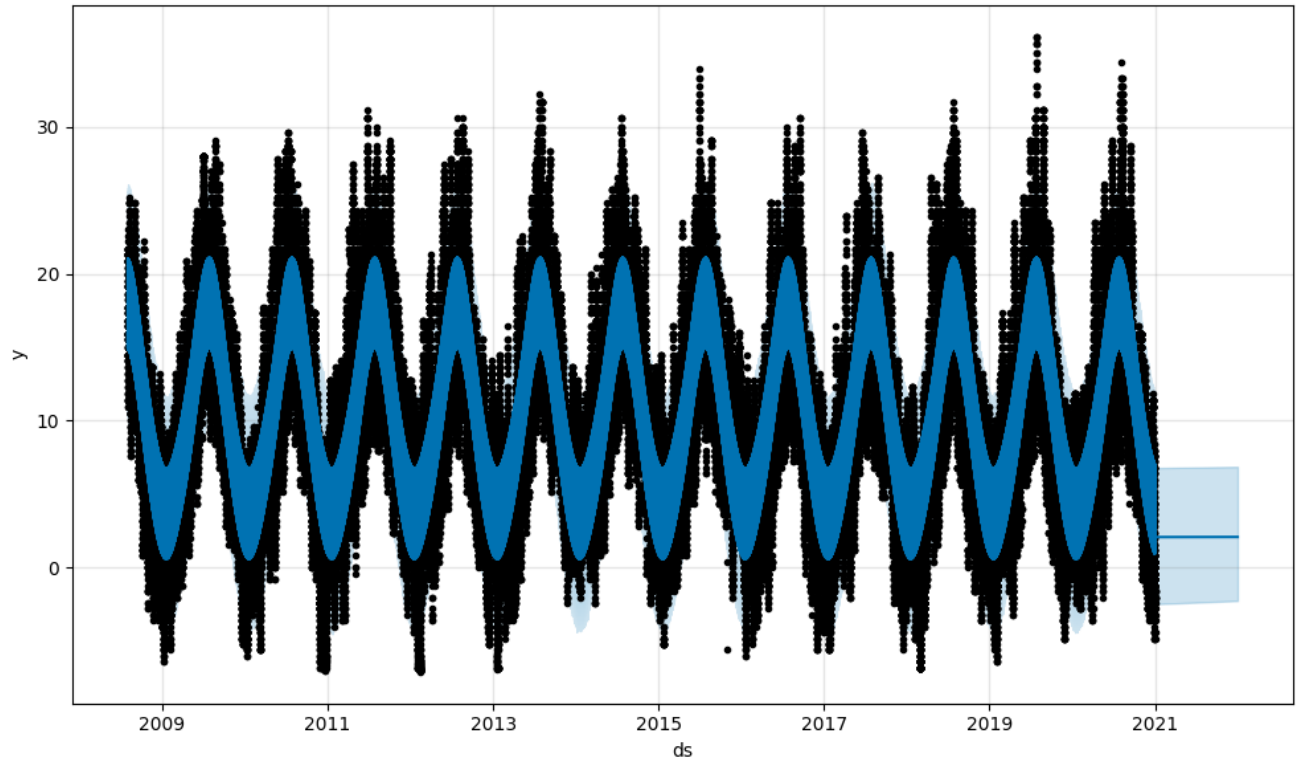
```
  # plt.axvline(x=pd.to_datetime('2018-01-01 00:00:00'), color='black', lw=1)  # v
  # plt.axvline(x=pd.to_datetime('00:00 02-Jan-2018'), color='black', lw=1)
  # plt.axvline(x=pd.Timestamp('00:00 02-Jan-2018'), color='black', lw=1)
  # plt.axvline(x=df.index.values[48], color='black', lw=1)
  # 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)
```

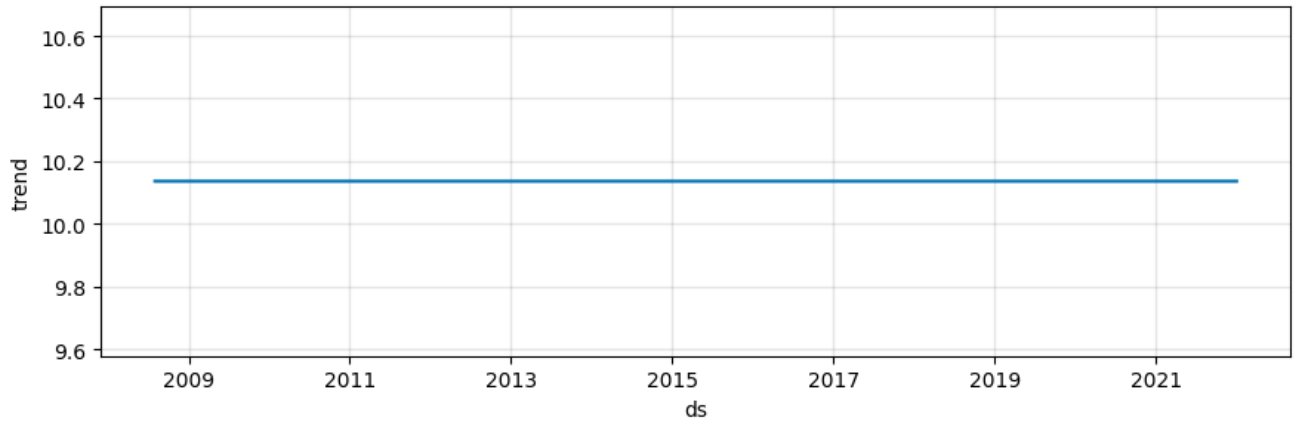
```
col_name = 'dew.point'
m2 = Prophet(yearly_seasonality = 3,
             daily_seasonality  = 3,
             weekly_seasonality = False,
             growth = 'flat')
f2 = add_prophet_components(train_df.loc['2016-01-12':,], m2, col_name)
display(train_df)

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)
```

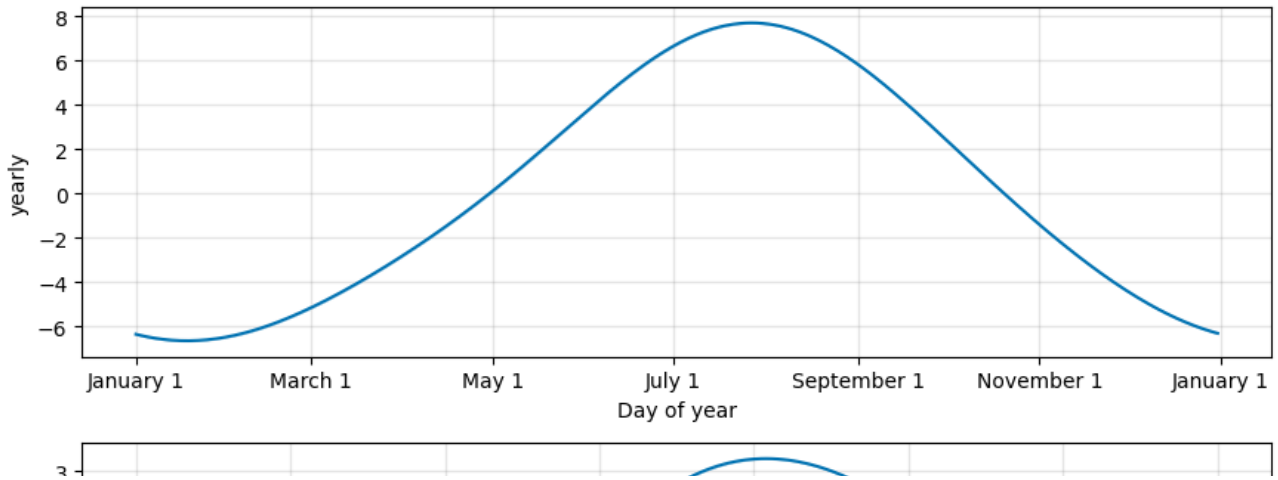
```
f_all = pd.merge(f1, f2, left_on=f1.index, right_on=f2.index)
f_all.set_index('key_0', inplace=True)
f_all = pd.merge(f_all, f3, left_on=f_all.index, right_on=f3.index)
f_all.set_index('key_0', inplace=True)
f_all = pd.merge(f_all, f4, left_on=f_all.index, right_on=f4.index)
f_all.set_index('key_0',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







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

22:00:00 22.00.00

```
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-	2021-						

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

```
y window 12 skew 3
```

```
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	

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.

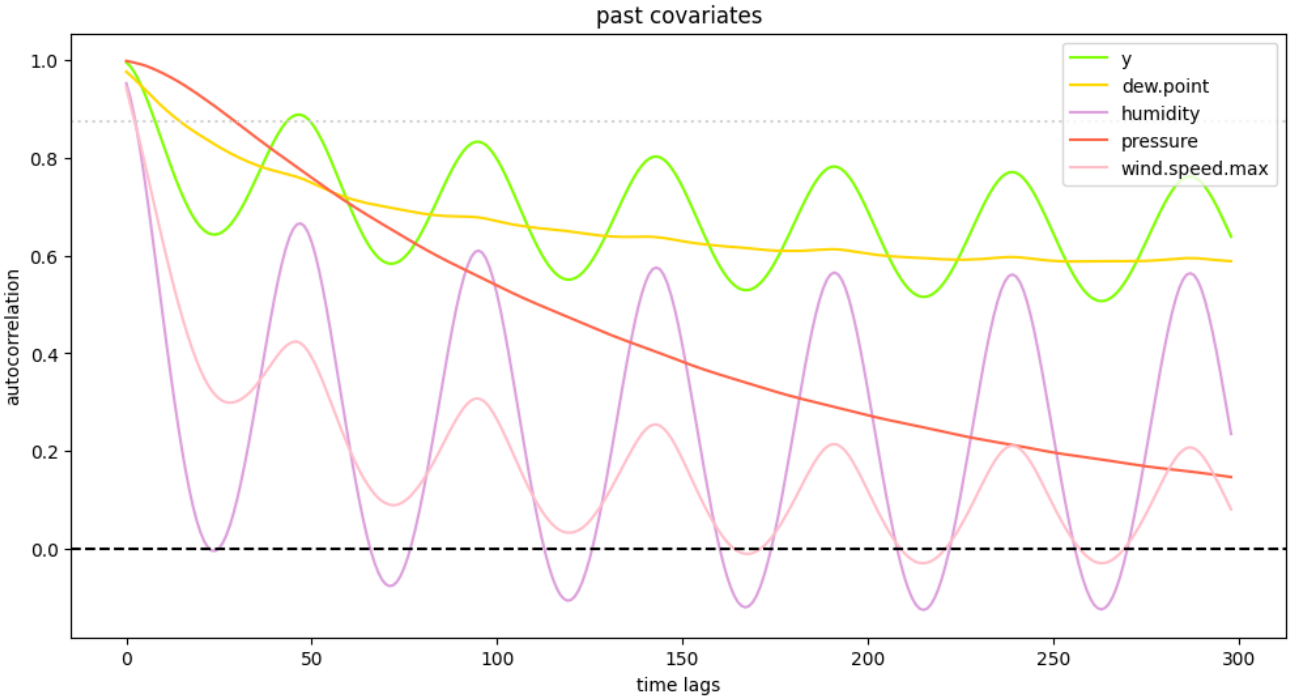
Compare main features

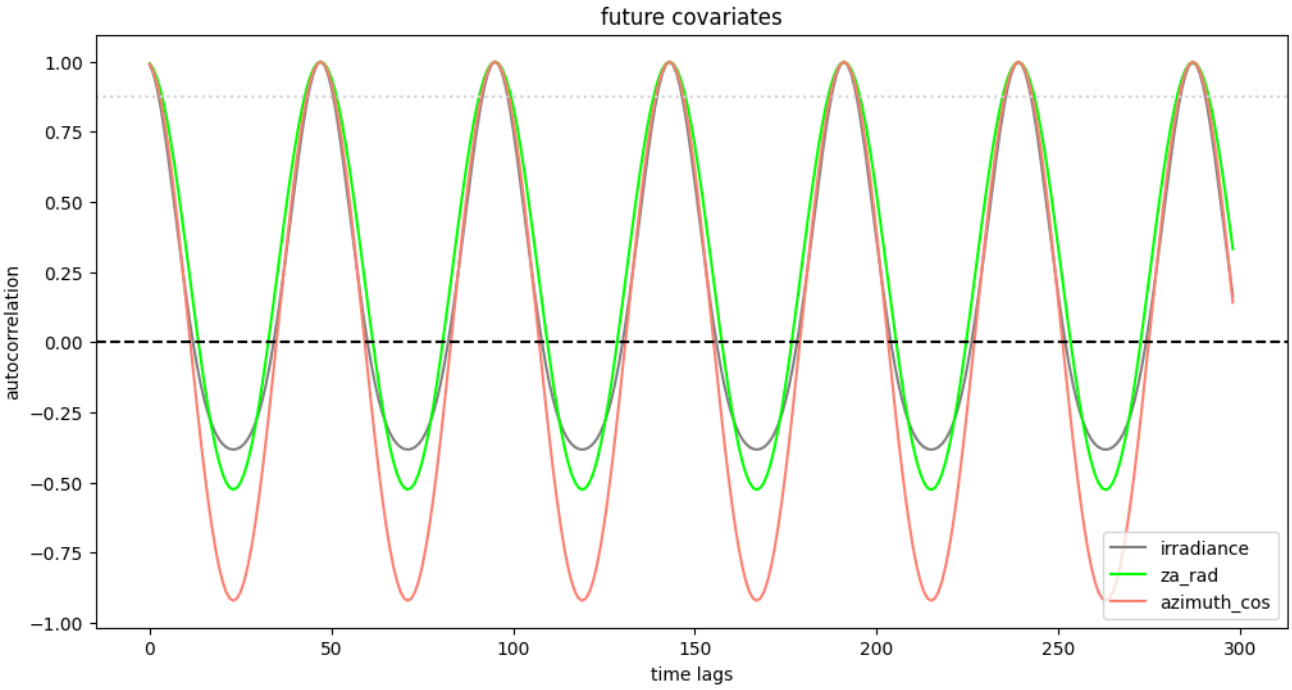
- past covariates
- future covariates

Short term autocorrelations first.

17664 rows × 139 columns

```
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


future covariates
irradiance
za_rad
azimuth_cos
autocorrelation
time lags

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

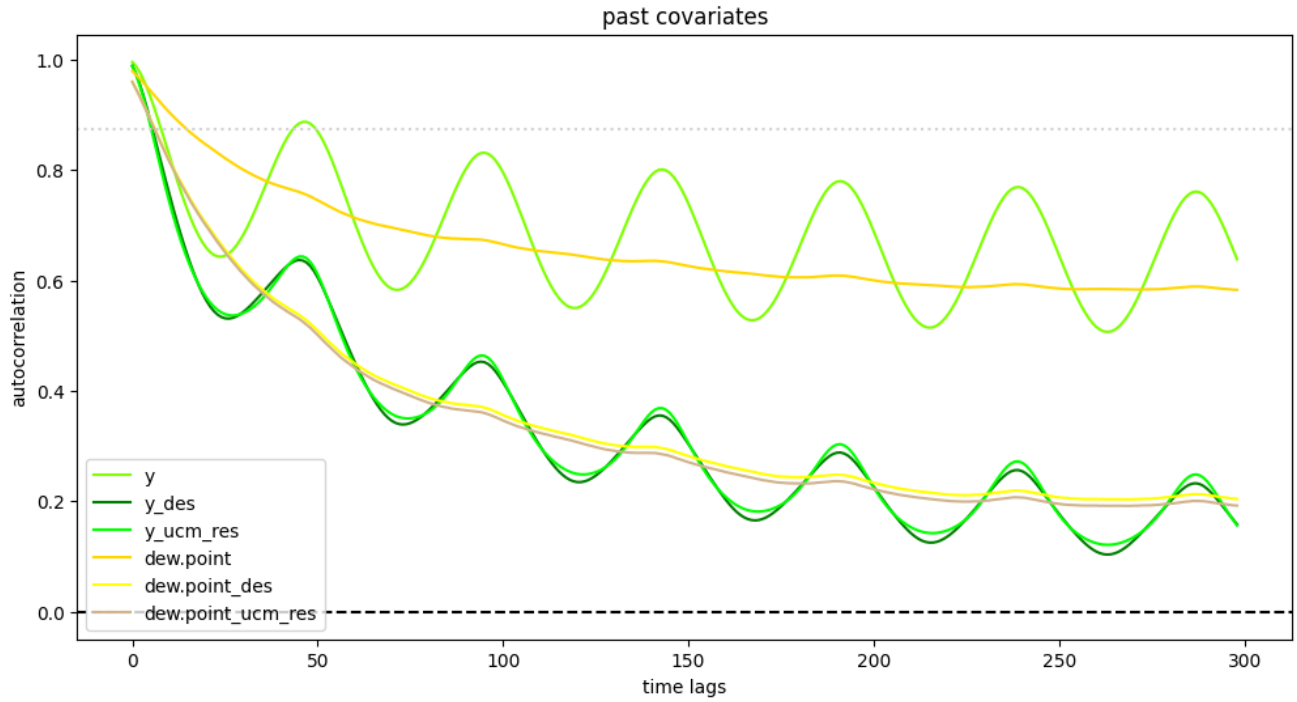
Autocorrelation of `pressure` and `dew.point` are notably different.

`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

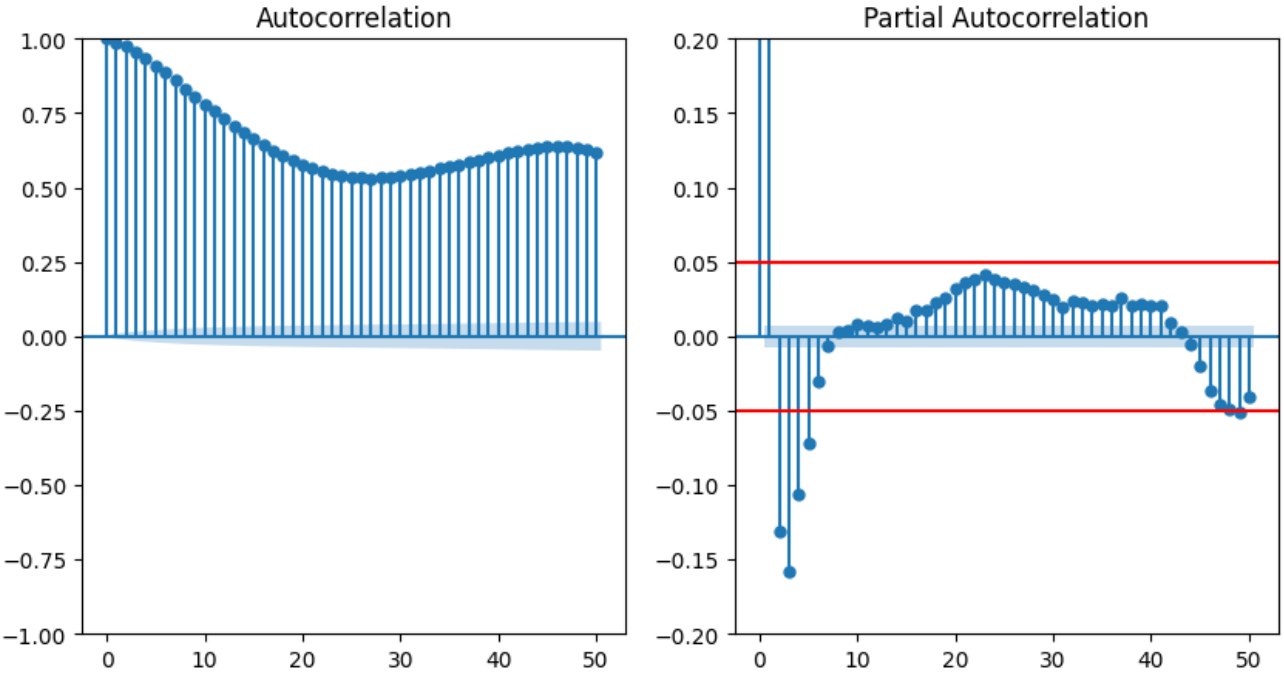
The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.

Here I plot both the autocorrelation and partial autocorrelations for:

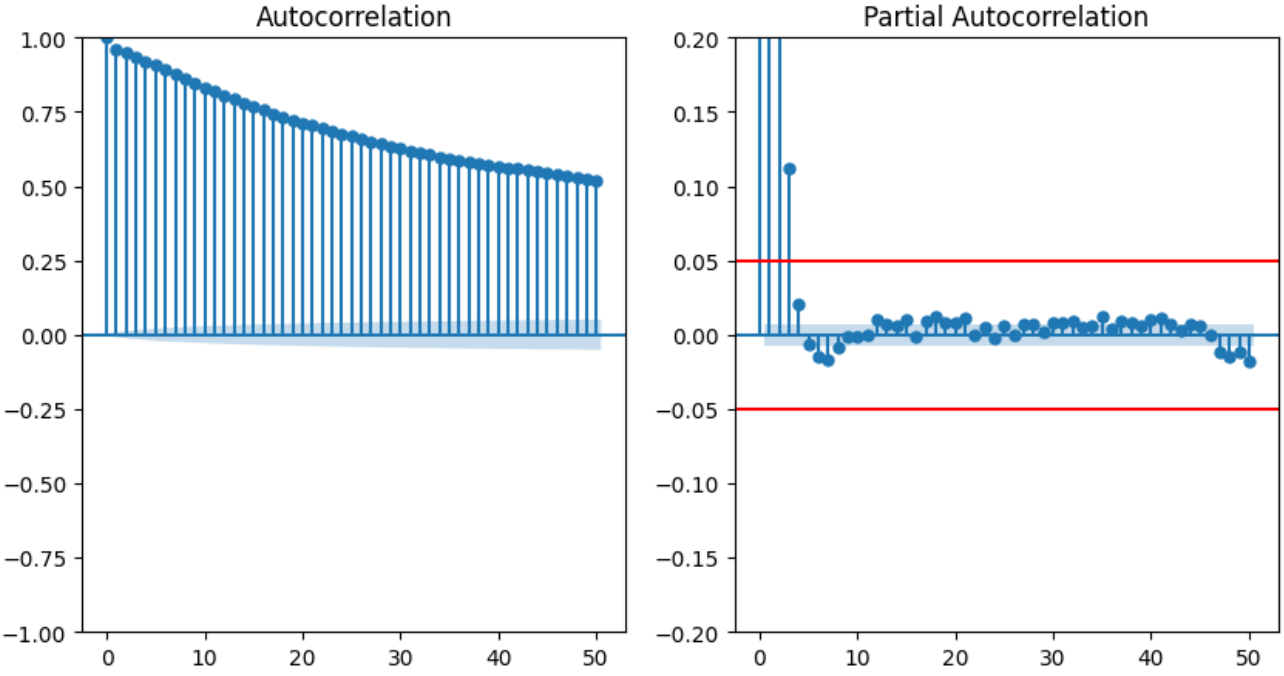
- core features
- future covariates

```
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()
```

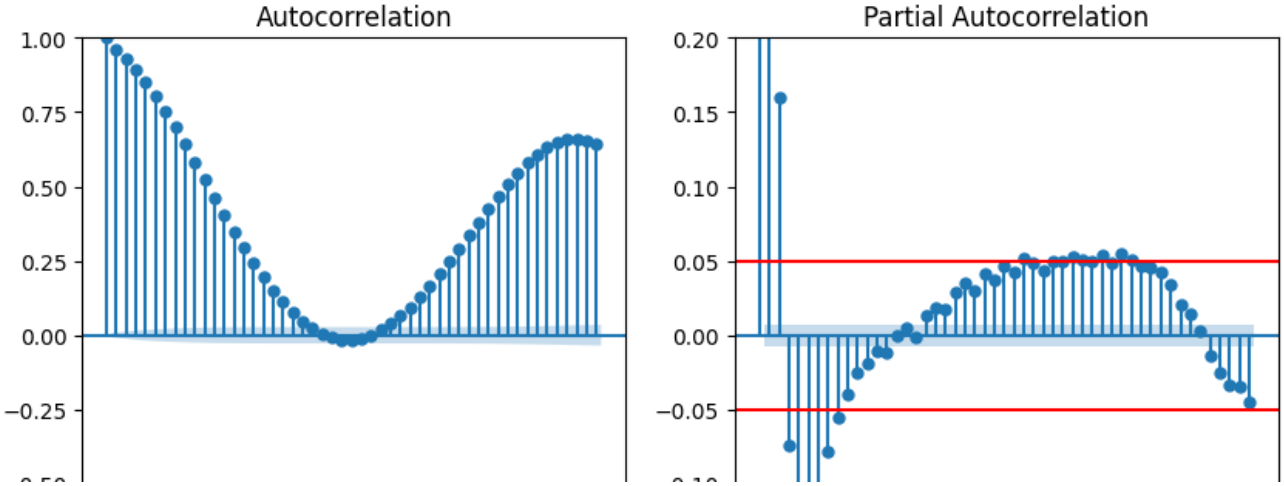
y_des



dew.point_des



humidity



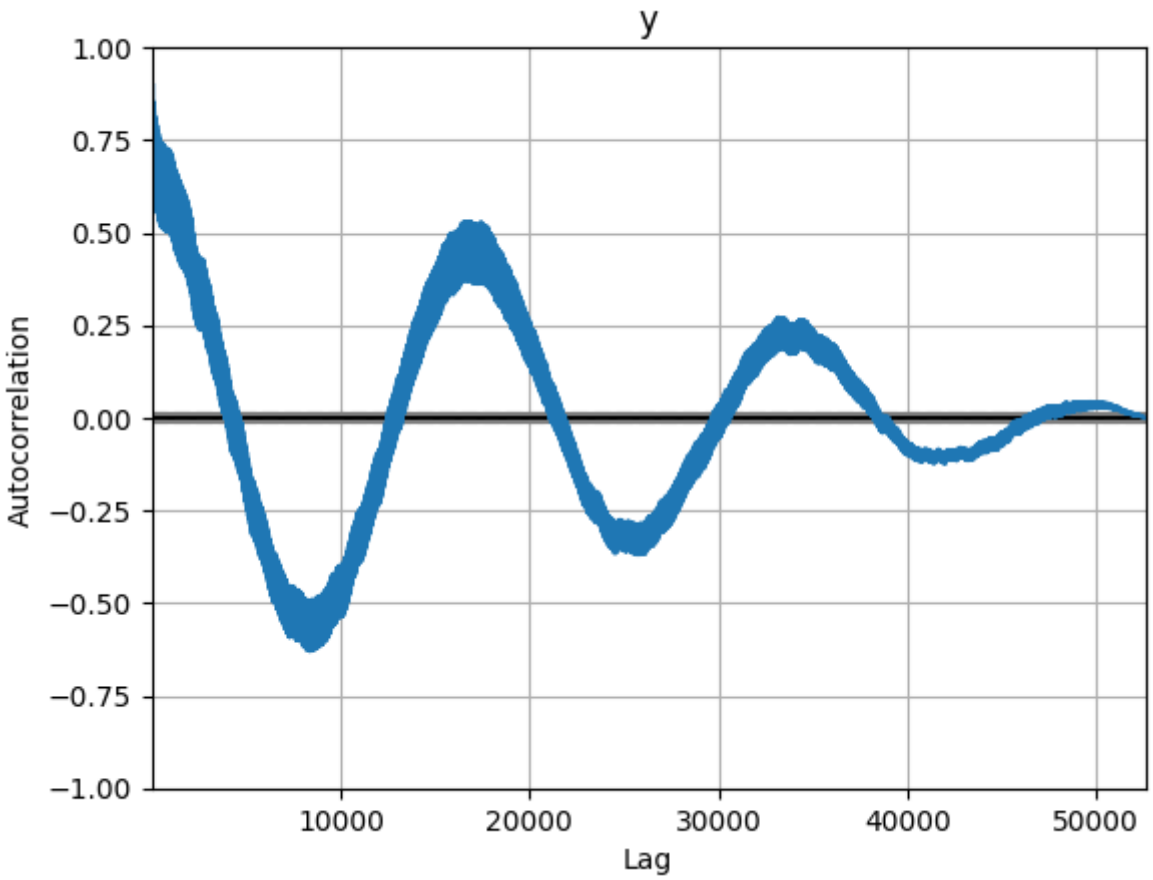
In general, the future covariates (`irradiance, za_rad, azimuth_cos`) have similar autocorrelation and partial autocorrelations functions.

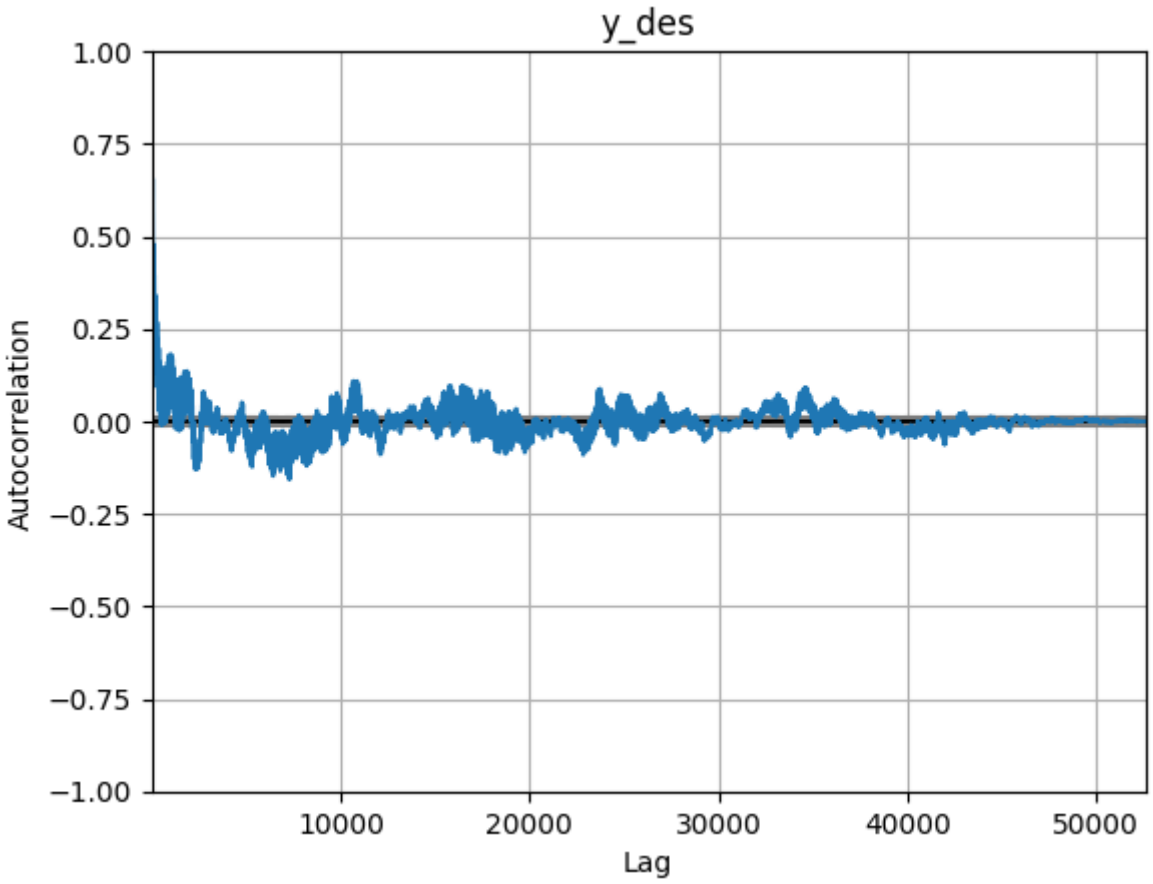
Plot longer term autocorrelations

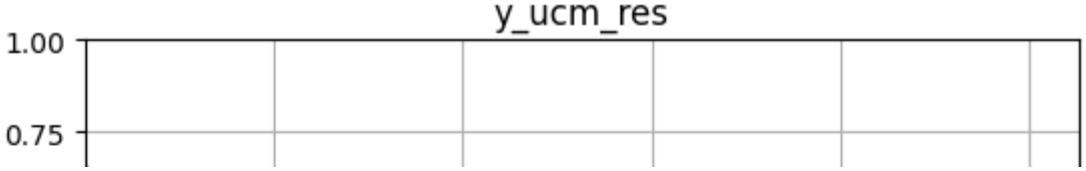
Compare base and deseasonalised features over 3 year time spans.

There are 17532 = 48 * 365.25 half hour steps or lags in one year.

```
plot_long_term_acf(train_df, 'y')
plot_long_term_acf(train_df, 'y_des')
plot_long_term_acf(train_df, 'y_ucm_res')
plot_long_term_acf(train_df, 'dew.point')
plot_long_term_acf(train_df, 'dew.point_des')
plot_long_term_acf(train_df, 'dew.point_ucm_res')
plot_long_term_acf(train_df, 'humidity')
plot_long_term_acf(train_df, 'pressure')
plot_long_term_acf(train_df, 'wind.speed.max')
```


y
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Autocorrelation
10000
20000
30000
40000
50000
Lag


y_des
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Autocorrelation
10000
20000
30000
40000
50000
Lag


y_ucm_res
1.00
0.75

There is clear long term seasonality present for `y`, `dew.point` and `humidity`. This long term seasonality is removed in the deseasonalised features: `y_des`, `y_ucm_res`, `dew.point_des` and `dew.point_ucm_res`.

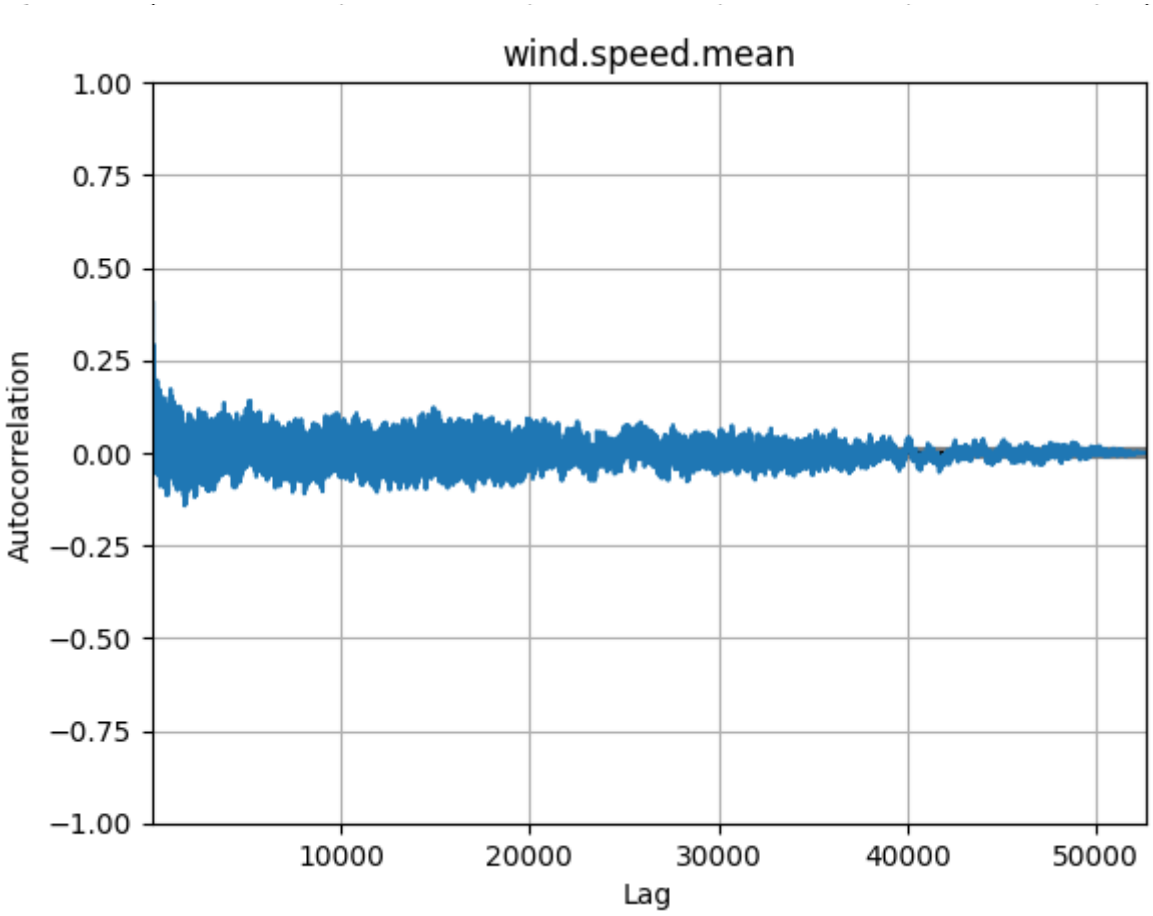
There is remarkably low daily seasonality present for `dew.point` and remarkably high daily seasonality for `humidity`. The daily seasonality for `humidity` also is not smoothly decreasing like daily seasonality of `y` and `dew.point`. This may be related to the right censoring at 100 %.

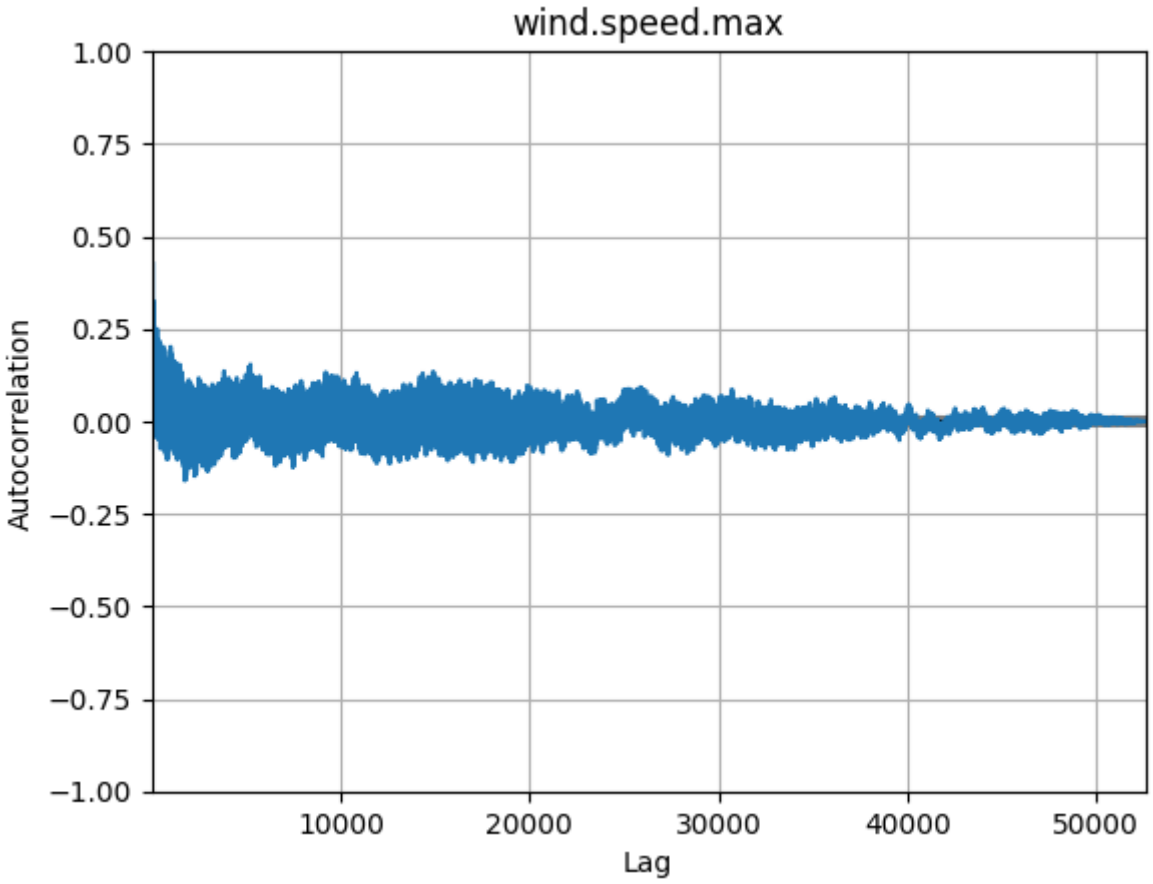
There is little obvious long term autocorrelation for `pressure` or `wind.speed.max`, meaning fitting annual seasonality may not be justified.

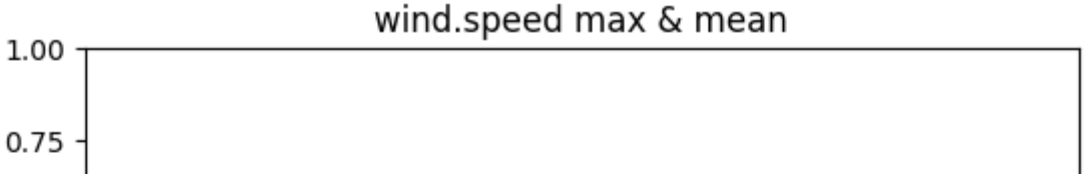
Compare long term autocorrelations for wind speed features.

```
                        10000       20000       30000       40000       50000
```

```
plot_long_term_acf(df, 'wind.speed.mean')
plot_long_term_acf(df, 'wind.speed.max')

# overlay 'wind.speed.main' on 'wind.speed.max'
pd.plotting.autocorrelation_plot(df['wind.speed.max'])
pd.plotting.autocorrelation_plot(df['wind.speed.mean'])
plt.title('wind.speed max & mean')
plt.show()

plot_long_term_acf(df, 'wind.speed.max.sqrt.x')
plot_long_term_acf(df, 'wind.speed.max.sqrt.x')
```







Little difference between `wind.speed.mean` and `wind.speed.max`.

Feature specific lags with PACF

Get feature-specific lags based on pacf:

```
def get_feature_specific_lags(data, feat, threshold = 0.05, nlags = 50):
  pacf_ = pacf(data[feat])
  pacf_ = np.abs(pacf_)

  pacf_indices = np.where(pacf_ >= threshold)[0][1:]

  return pacf_indices


pc_feats = ['y_des', 'dew.point_des', 'humidity', 'pressure',
            'irradiance', 'za_rad', 'azimuth_cos']
thresholds = {'y_des': 0.05, 'dew.point_des': 0.15, 'humidity': 0.15,
              'pressure': 0.15, 'irradiance': 0.2, 'za_rad': 0.2,
              'azimuth_cos': 0.2,}

for pc_feat in pc_feats:
  lags_ = get_feature_specific_lags(train_df, pc_feat, thresholds[pc_feat])
  print(pc_feat, '\t', lags_)
```

```
y_des    [ 1  2  3  4  5 49]
dew.point_des    [1 2]
humidity         [1 2]
pressure         [1]
irradiance       [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49 50 51 52 53]
za_rad   [ 1  2  3  4  5 34 36 37 38 39 40 41 42 43 49]
azimuth_cos      [ 1  2  3  4  6  7 33 34 35 36 37 38 39 40 41 42 43 44 45 46
 51 52 53]
```

...

Plot and check prophet components

The yearly prophet components have `_yearly` appended to feature names. Similarly, `_daily` is used for daily components and `_des` (for deseasonalised) is used for residuals.

I used flat trends for simplicity and tweaked the transformation, seasonality mode, daily harmonics and yearly harmonics.

The final decomposition parameters:

Feature	Mode	Transform	Daily	Yearly
y	Additive	None	3	3
dew.point	Additive	None	3	3
humidity	Multiplicative	log	2	Auto

Feature	Mode	Transform	Daily	Yearly
pressure	Multiplicative	log	2	2

The mode and transforms agree with work elsewhere.

I tried log, square root and Box-Cox transformations and tried to keep the daily and yearly harmonics reasonably low.

Using a flat trend may not be appropriate for humidity.

The daily seasonalities for `dew.point` and `pressure` each have two peaks. There is some supporting evidence for two daily peaks for atmospheric pressure, but I'm not aware of any similar evidence for dew point.

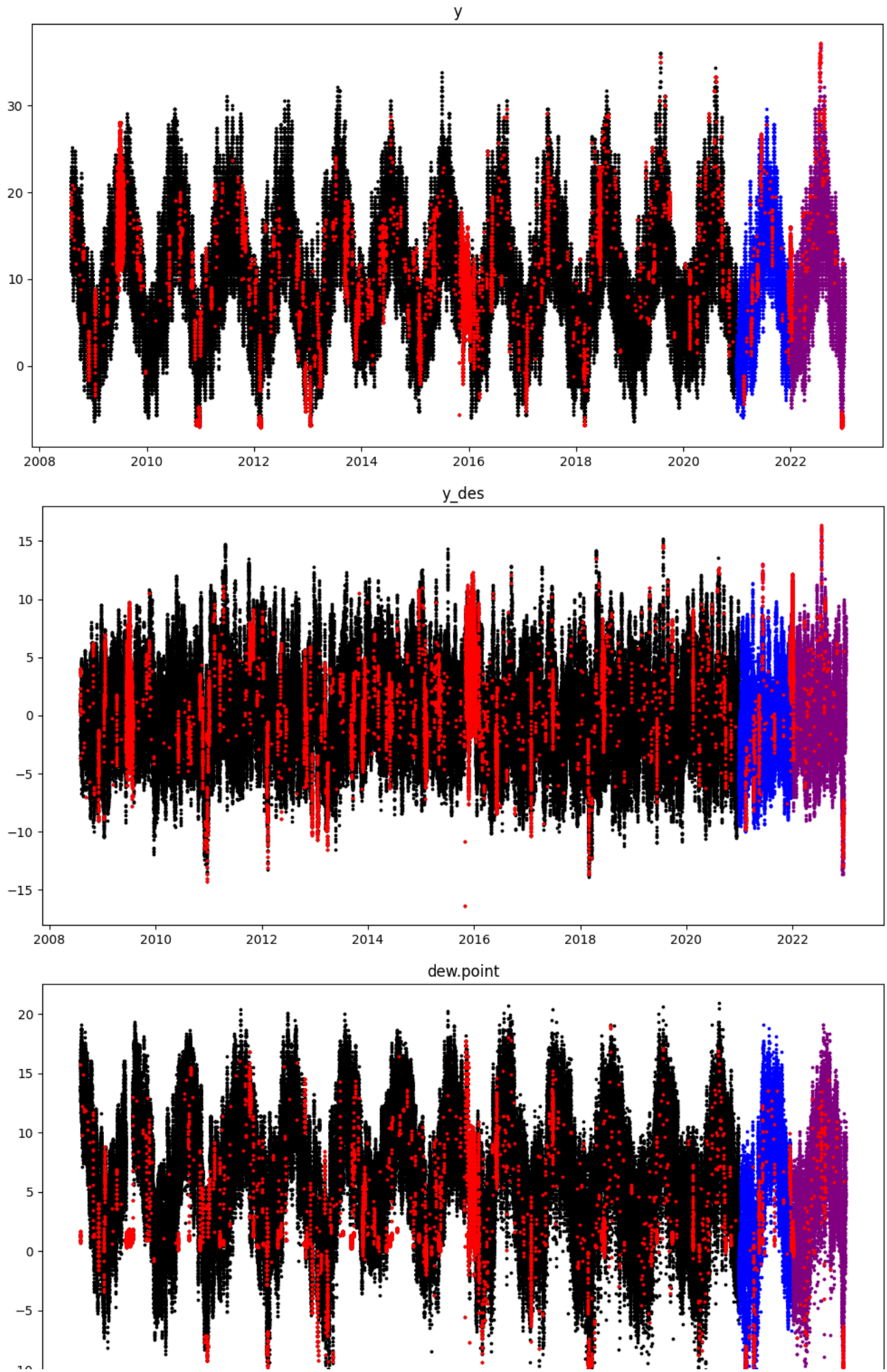
The `humidity` and `pressure` decompositions would benefit from further attention. The daily and yearly components are hundreths of a percent!

The `wind.speed.mean` decomposition was unacceptible. I've commented it out for now. Wind speed may require more sophisticated decomposition techniques like wavelets for example. Surprisingly, others have successfully used multiplicative decomposition with smoothed wind speed data.

We now have a richer set of features for lightgbm to choose from. Also, we can model the deseasonalised temperatures, `y_des`, instead of the more complicated multi-seasonal temperatures `y`. To make forecasts for `y`, `y_yearly` and `y_daily` must be added to the `y_des` predictions.

Finally, plot the historic values for the original and decomposed residual features:

```
vars = ['y', 'dew.point', 'humidity', 'pressure',
       #'irradiance', 'za_rad', 'azimuth_cos',
       ]

for var in vars:
  var_des = var + '_des'
  if var in ['irradiance', 'za_rad', 'azimuth_cos']:
    plot_feature_history(train_df, valid_df, test_df, var)
  else:
    plot_feature_history(train_df, valid_df, test_df, var, missing=True)

  if var in ['y', 'dew.point']:
    plot_feature_history(train_df, valid_df, test_df, var_des, missing=True)
```



A visual comparison of the original and residual features indicates:

- unfortunately `missing` label is not feature specific
- possible longer than annual signal in `y_des`
- possible annual signal remaining in `dew.point_des`
- increasing heteroscadicity in `humidity` and `humidity_des`
- decreasing trend in `humidity` and `humidity_des`
- possible annual signal remaining in `humidity_des`
- `pressure` and `pressure_des` appear similar
- possible annual signal remaining in `pressure_des`

Stationarity Tests

It is worthwhile looking at the stationarity of these time series after decomposition.

A stationary time series should have mean, variance and autocorrelation that do not change over time. This means a flat series, without trend, constant variance, constant autocorrelation structure and no periodic fluctuations (seasonalities).

```
def adf_test(timeseries, NLAGS):
    timeseries = timeseries.dropna()

    if NLAGS is not None:
      dftest = adfuller(timeseries, maxlag = NLAGS)
    else:
      dftest = adfuller(timeseries, autolag="AIC")

    dfoutput = pd.Series(
        dftest[0:4],
        index=["Test Statistic",
               "p-value",
               "#Lags Used",
               "Number of Observations Used",
        ],
    )

    print("Results of Dickey-Fuller Test:")
    for key, value in dftest[4].items():
        dfoutput["Critical Value (%s)" % key] = value

    print(dfoutput)


def kpss_test(timeseries, NLAGS):
    timeseries = timeseries.dropna()

    if NLAGS is not None:
      kpsstest = kpss(timeseries, regression="c", nlags=NLAGS)
    else:
      kpsstest = kpss(timeseries, regression="c", nlags="auto")
```

```
    kpss_output = pd.Series(
        kpsstest[0:3], index=["Test Statistic", "p-value", "Lags Used"]
    )

    print("\nResults of KPSS Test:")
    for key, value in kpsstest[3].items():
        kpss_output["Critical Value (%s)" % key] = value

    print(kpss_output)


def stationarity_tests(df, var, NLAGS=None):
    print(var)
    adf_test(df[var], NLAGS)
    kpss_test(df[var], NLAGS)
    print('\n')


for var in ['y', 'dew.point', 'humidity', 'pressure',
            'irradiance', 'za_rad', 'azimuth_cos']:
    stationarity_tests(train_df, var)
    if var in ['y', 'dew.point']:
        stationarity_tests(train_df.loc['2016-01-12':, :], var + '_des')
```

```
y
Results of Dickey-Fuller Test:
Test Statistic                -1.221669e+01
p-value                        1.131383e-22
#Lags Used                     8.200000e+01
Number of Observations Used    2.176450e+05
Critical Value (1%)           -3.430380e+00
Critical Value (5%)           -2.861553e+00
Critical Value (10%)          -2.566777e+00
dtype: float64

Results of KPSS Test:
Test Statistic            0.24991
p-value                   0.10000
Lags Used               259.00000
Critical Value (10%)      0.34700
Critical Value (5%)       0.46300
Critical Value (2.5%)     0.57400
Critical Value (1%)       0.73900
dtype: float64


y_des
<ipython-input-24-a27d6bd1e297>:31: InterpolationWarning: The test statistic i
look-up table. The actual p-value is greater than the p-value returned.

  kpsstest = kpss(timeseries, regression="c", nlags="auto")
Results of Dickey-Fuller Test:
Test Statistic                  -20.515403
p-value                           0.000000
#Lags Used                       66.000000
Number of Observations Used    87101.000000
Critical Value (1%)               -3.430425
```

```
Critical Value (5%)               -2.861573
Critical Value (10%)              -2.566788
dtype: float64

Results of KPSS Test:
Test Statistic            0.234886
p-value                   0.100000
Lags Used               165.000000
Critical Value (10%)      0.347000
Critical Value (5%)       0.463000
Critical Value (2.5%)     0.574000
Critical Value (1%)       0.739000
dtype: float64


dew.point
<ipython-input-24-a27d6bd1e297>:31: InterpolationWarning: The test statistic i
look-up table. The actual p-value is greater than the p-value returned.

  kpsstest = kpss(timeseries, regression="c", nlags="auto")
Results of Dickey-Fuller Test:
Test Statistic                -1.708259e+01
p-value                        7.688411e-30
#Lags Used                     8.200000e+01
Number of Observations Used    2.176450e+05
```

The relevant statsmodels docs explain the ADF and KPSS tests plus provide help with interpretation of the p-values.

ADF:

- Null hypothesis: the dataset is non-stationary and therefore differencing must be carried out.
- Alternative hypothesis: the dataset has no unit root and is therefore stationary.
- If the null hypothesis failed to be rejected, this test may provide evidence that the series is non-stationary.

KPSS:

- Null hypothesis: the series is trend stationary.
- Alternative hypothesis: the series is not stationary.
- The null and alternate hypothesis for the KPSS test are opposite that of the ADF test.

A Bonferroni-corrected p-value threshold of 0.005 (0.05 / 10) is used.

Feature	ADF	KPSS	Result
y	3.35e-23	0.1	Stationary
y_des	0.0	0.01	Stationary
dew.point	4.6e-29	0.01	Stationary
dew.point_des	0.0	0.01	Stationary
humidity	0.0	0.01	Stationary
humidity_des	0.0	0.01	Stationary
pressure	0.0	0.01	Stationary
pressure_des	0.0	0.01	Stationary

Feature	ADF	KPSS	Result
irradiance	0.0	0.01	Stationary
zenith angle	0.0	0.01	Stationary
azimuth_cos	...	...	...

For all features tested, the ADF null hypothesis is rejected and the KPSS null hypothesis is accepted at the Bonferroni-corrected p-value threshold. All of the features were stationary before and after transformation.

Save default feature sets

Save default train, valid and test data sets for use in main modeling notebook(s).

The graphs above show their are quite a few missing values around the beginning of 2016. Earlier work calculating first differences for the core features showed there was substantial differences before early 2016 and afterwards. So, training data starts after early 2016.

```
params_def = {'verbose':   True,
              'feat_name': 'default_',
              'date_str':  '.2022.09.20',
              'save_and_download': False,
              'save_to_gdrive': False,
             }
```

```
train_mask = (train_df.index >= '2016-01-12')
valid_mask = valid_df.index.year == VALID_YEAR
test_mask  = test_df.index.year  == TEST_YEAR
save_data_and_download_files(train_df.loc[train_mask], 'train', params_def)
save_data_and_download_files(valid_df.loc[valid_mask], 'valid', params_def)
save_data_and_download_files(test_df.loc[test_mask],   'test',  params_def)
```

yyy

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

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

---

Feature Engineering

Feature engineering, or feature extraction, transforms raw data into features for forecasting. It includes the following packages and approaches:

- catch22
- tsfeatures

- bivariate
    - numpy
    - scipy
    - statsmodels
    - dynamic time warping
    - distances
    - kernels

catch22

catch22 is a collection of 22 time-series features which were selected based on their *classification* performance across a collection of 93 real-world time-series classification problems.

- pycatch22
- short feature descriptions
- detailed descriptions with visual depictions of the behavior of features

The `get_feature_selection_scores` function calculates F-statistic tests, Pearson correlation and mutual information between the calculated features and the target variable. These tests assume a linear model. This is not optimal. Don't draw any hasty conclusions from these scores.

Lags considered:

- [6, 12, 24, 48, 96]

Features considered:

- `y_des`
- `dew.point_des`
- `humidity`

`pressure` has some long sequences of single values which produce many NAs. A better deseasonalised `pressure` feature would be worthwhile trying.

catch22 installation is straight-forward:

```
!pip install pycatch22
```

```
Collecting pycatch22
  Downloading pycatch22-0.4.4.tar.gz (49 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.9/
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: pycatch22
  Building wheel for pycatch22 (pyproject.toml) ... done
  Created wheel for pycatch22: filename=pycatch22-0.4.4-cp310-cp310-linux_x86_
  Stored in directory: /root/.cache/pip/wheels/10/67/84/cdce1a956aa218fd5ce5b5
Successfully built pycatch22
```

```
    Installing collected packages: pycatch22
    Successfully installed pycatch22-0.4.4
```

```
from pycatch22 import catch22_all, DN_HistogramMode_5, DN_HistogramMode_10, \
                      CO_f1ecac, CO_FirstMin_ac, CO_HistogramAMI_even_2_5, \
                      CO_trev_1_num, MD_hrv_classic_pnn40, \
                      SB_BinaryStats_mean_longstretch1, \
                      SB_TransitionMatrix_3ac_sumdiagcov, \
                      PD_PeriodicityWang_th0_01, \
                      CO_Embed2_Dist_tau_d_expfit_meandiff, \
                      IN_AutoMutualInfoStats_40_gaussian_fmmi, \
                      FC_LocalSimple_mean1_tauresrat, \
                      DN_OutlierInclude_p_001_mdrmd, \
                      DN_OutlierInclude_n_001_mdrmd, \
                      SP_Summaries_welch_rect_area_5_1, \
                      SB_BinaryStats_diff_longstretch0, \
                      SB_MotifThree_quantile_hh, \
                      SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1, \
                      SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1, \
                      SP_Summaries_welch_rect_centroid, \
                      FC_LocalSimple_mean3_stderr


def c22_call(ser, func):
  '''The pd.Series.to_list() call is necessary.
     Otherwise rol.apply(how) may work.
     Also, np.ndarray.tolist() is necessary.
     Otherwise c22 returns null exception.
     Or, maybe rol.apply(lambda shenanigans with AGGS array)'''
  # return func(data.loc[ser.index, feat_col].to_list())
  return func(ser.tolist())


def get_rolling_catch22_features(data, feat_cols, windows, AGGS, verbose=False):
  '''Probably can't add norm and scale options due to compute time issues'''

  aggs_l = []

  for feat_col in feat_cols:
    print(feat_col)
    for window in windows:
      print('\t', window)

      #if feat_col == 'humidity':
      #  continue
      #elif window == 24 and feat_col == 'pressure':
      #if window == 24 and feat_col == 'pressure':
      #  continue

      rol = data[feat_col].shift(1).rolling(window=window, min_periods=1,)

      for how in AGGS:
```

```
        start_time = timeit.default_timer()
        if verbose:
          print('\t\t', how.__name__, end='')

        c22args = {'func': how}

        agg_ = rol.apply(c22_call, kwargs=c22args, raw=True)
        agg = agg_.copy()
        agg = agg.rename(f'{feat_col}_window_{window}_{how.__name__}')
        aggs_l.append(agg)

        if verbose:
          print('\t', round(timeit.default_timer() - start_time, 2))

  aggs = pd.concat(aggs_l, axis=1)
  aggs = drop_problem_cols(aggs, window)
  aggs = drop_cols_correlated_with_feat_cols(aggs, data[feat_cols])

  return aggs


# feat_cols = [Y_COL]
# windows = [48]
#
# many problems with humidity and train_df - do not use humidity feature
windows = [6, 12, 24, 48, 96]
feat_cols = [Y_COL, 'dew.point_des', 'pressure', 'humidity']
c22_aggs = [DN_HistogramMode_5, DN_HistogramMode_10, CO_f1ecac, \
            CO_FirstMin_ac, CO_HistogramAMI_even_2_5, \
            CO_trev_1_num, MD_hrv_classic_pnn40, \
            SB_BinaryStats_mean_longstretch1, \
            SB_TransitionMatrix_3ac_sumdiagcov, PD_PeriodicityWang_th0_01, \
            CO_Embed2_Dist_tau_d_expfit_meandiff, \
            IN_AutoMutualInfoStats_40_gaussian_fmmi, \
            FC_LocalSimple_mean1_tauresrat, DN_OutlierInclude_p_001_mdrmd, \
            DN_OutlierInclude_n_001_mdrmd, SP_Summaries_welch_rect_area_5_1, \
            SB_BinaryStats_diff_longstretch0, SB_MotifThree_quantile_hh, \
            SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1, \
            SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1, \
            SP_Summaries_welch_rect_centroid, FC_LocalSimple_mean3_stderr]
params_c22 = {'windows':    windows,
              'feat_cols':  feat_cols,
              'aggs':       c22_aggs,
              'agg_func':   get_rolling_catch22_features,
              'verbose':    True,
              'dataset':    'valid',
              'regenerate': True,
              'feat_name':  'catch22_',
              'date_str':   '.2022.09.20',
              'save_and_download': False,
              'save_to_gdrive': False,
             }

train_df_c22, valid_df_c22, test_df_c22 = get_features(train_df,
```

```
                                                        valid_df,
                                                        test_df,
                                                        params_c22)


##############################################################################
# 12 mins for all 22 catch22 functions on valid_df, test_df & train_df
# 305 features added - 0 NAs
#
# windows = [6, 12, 24, 48, 96]
# feat_cols = [Y_COL, 'dew.point_des', 'pressure', 'humidity']
# c22_aggs = [DN_HistogramMode_5, DN_HistogramMode_10, CO_f1ecac, \
#             CO_FirstMin_ac, CO_HistogramAMI_even_2_5, \
#             CO_trev_1_num, MD_hrv_classic_pnn40, \
#             SB_BinaryStats_mean_longstretch1, \
#             SB_TransitionMatrix_3ac_sumdiagcov, PD_PeriodicityWang_th0_01, \
#             CO_Embed2_Dist_tau_d_expfit_meandiff, \
#             IN_AutoMutualInfoStats_40_gaussian_fmmi, \
#             FC_LocalSimple_mean1_tauresrat, DN_OutlierInclude_p_001_mdrmd, \
#             DN_OutlierInclude_n_001_mdrmd, SP_Summaries_welch_rect_area_5_1, \
#             SB_BinaryStats_diff_longstretch0, SB_MotifThree_quantile_hh, \
#             SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1, \
#             SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1, \
#             SP_Summaries_welch_rect_centroid, FC_LocalSimple_mean3_stderr]
# params = {'windows':    windows,
#           'feat_cols':  feat_cols,
#           'aggs':       c22_aggs,
#           'agg_func':   get_rolling_catch22_features,
#           'verbose':    True,
#           'dataset':    'valid',
#           'regenerate': True,
#           'feat_name':  'catch22_',
#           'date_str':   '.2022.09.20', }
```

```
dataset: valid
y_des
         6
                 DN_HistogramMode_5      0.02
                 DN_HistogramMode_10     0.02
                 CO_f1ecac       0.04
                 CO_FirstMin_ac  0.04
                 CO_HistogramAMI_even_2_5        0.03
                 CO_trev_1_num   0.02
                 MD_hrv_classic_pnn40    0.02
                 SB_BinaryStats_mean_longstretch1        0.02
                 SB_TransitionMatrix_3ac_sumdiagcov      0.06
                 PD_PeriodicityWang_th0_01       0.07
                 CO_Embed2_Dist_tau_d_expfit_meandiff    0.05
                 IN_AutoMutualInfoStats_40_gaussian_fmmi         0.02
                 FC_LocalSimple_mean1_tauresrat  0.07
                 DN_OutlierInclude_p_001_mdrmd   0.22
                 DN_OutlierInclude_n_001_mdrmd   0.23
                 SP_Summaries_welch_rect_area_5_1        0.03
                 SB_BinaryStats_diff_longstretch0        0.02
                 SB_MotifThree_quantile_hh       0.05
                 SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1     0.04
                 SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1  0.04
                 SP_Summaries_welch_rect_centroid        0.03
                 FC_LocalSimple_mean3_stderr     0.02
         12
                 DN_HistogramMode_5      0.02
                 DN_HistogramMode_10     0.02
                 CO_f1ecac       0.08
                 CO_FirstMin_ac  0.09
                 CO_HistogramAMI_even_2_5        0.04
                 CO_trev_1_num   0.03
                 MD_hrv_classic_pnn40    0.02
                 SB_BinaryStats_mean_longstretch1        0.02
                 SB_TransitionMatrix_3ac_sumdiagcov      0.09
                 PD_PeriodicityWang_th0_01       0.08
                 CO_Embed2_Dist_tau_d_expfit_meandiff    0.08
                 IN_AutoMutualInfoStats_40_gaussian_fmmi         0.03
                 FC_LocalSimple_mean1_tauresrat  0.13
                 DN_OutlierInclude_p_001_mdrmd   0.37
                 DN_OutlierInclude_n_001_mdrmd   0.37
                 SP_Summaries_welch_rect_area_5_1        0.05
                 SB_BinaryStats_diff_longstretch0        0.02
                 SB_MotifThree_quantile_hh       0.06
                 SC_FluctAnal_2_rsrangefit_50_1_logi_prop_r1     0.04
                 SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1  0.05
                 SP_Summaries_welch_rect_centroid        0.05
                 FC_LocalSimple_mean3_stderr     0.02
         24
                 DN_HistogramMode_5      0.03
                 DN_HistogramMode_10     0.03
                 CO_f1ecac       0.15
                 CO_FirstMin_ac  0.15
                 CO_HistogramAMI_even_2_5        0.06
                 CO_trev_1_num   0.04
                 MD_hrv_classic_pnn40    0.03
                 SB_BinaryStats_mean_longstretch1        0.03
                 SB_TransitionMatrix_3ac_sumdiagcov      0.16
```

305 additional features were added in approximately 12 minutes (0 NAs).

Features above or between `dew.point_des` and `humidity` in the feature selection scores table may prove useful in future models. The `DN_OutlierInclude_p_001_mdrmd` and `DN_OutlierInclude_n_001_mdrmd` features may prove useful. Other features may be useful for later forecast steps.

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.

---

tsfeatures

Python implementation of the R package tsfeatures (originally by Hyndman and co).

List of available features:

Features		
acf_features	heterogeneity	series_length
arch_stat	holt_parameters	sparsity
count_entropy	hurst	stability
crossing_points	hw_parameters	stl_features
entropy	intervals	unitroot_kpss
flat_spots	lumpiness	unitroot_pp
frequency	nonlinearity	
guerrero	pacf_features	

Description of features.

Features considered:

- `y_des`
- `dew.point_des`
- `humidity`

`pressure` has some long sequences of single values which produce many NAs. A better deseasonalised `pressure` feature would be worthwhile trying.

Lags considered:

- [6, 12, 24, 48]

```
!pip install tsfeatures
```

```
Collecting tsfeatures
  Downloading tsfeatures-0.4.5-py3-none-any.whl (28 kB)
Collecting antropy>=0.1.4 (from tsfeatures)
  Downloading antropy-0.1.6.tar.gz (17 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting arch>=4.11 (from tsfeatures)
  Downloading arch-6.1.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_6
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 916.4/91
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: scikit-learn>=0.23.1 in /usr/local/lib/python3.
Requirement already satisfied: statsmodels>=0.13.2 in /usr/local/lib/python3.1
Collecting supersmoother>=0.4 (from tsfeatures)
  Downloading supersmoother-0.4.tar.gz (233 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 233.8/23
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-package
Collecting numba>=0.57 (from antropy>=0.1.4->tsfeatures)
  Downloading numba-0.57.1-cp310-cp310-manylinux2014_x86_64.manylinux_2_17_x86
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.6/3
Collecting stochastic (from antropy>=0.1.4->tsfeatures)
  Downloading stochastic-0.7.0-py3-none-any.whl (48 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 48.1/
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/di
Collecting llvmlite<0.41,>=0.40.0dev0 (from numba>=0.57->antropy>=0.1.4->tsfea
  Downloading llvmlite-0.40.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42.1/
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
Building wheels for collected packages: antropy, supersmoother
  Building wheel for antropy (pyproject.toml) ... done
  Created wheel for antropy: filename=antropy-0.1.6-py3-none-any.whl size=1687
  Stored in directory: /root/.cache/pip/wheels/98/22/06/e91d7bb213c7133d5e2eb3
  Building wheel for supersmoother (setup.py) ... done
  Created wheel for supersmoother: filename=supersmoother-0.4-py3-none-any.whl
  Stored in directory: /root/.cache/pip/wheels/cf/6a/f2/3d545d90957029bf34179c
Successfully built antropy supersmoother
Installing collected packages: supersmoother, llvmlite, stochastic, numba, ant
  Attempting uninstall: llvmlite
    Found existing installation: llvmlite 0.39.1
    Uninstalling llvmlite-0.39.1:
      Successfully uninstalled llvmlite-0.39.1
  Attempting uninstall: numba
    Found existing installation: numba 0.56.4
    Uninstalling numba-0.56.4:
      Successfully uninstalled numba-0.56.4
Successfully installed antropy-0.1.6 arch-6.1.0 llvmlite-0.40.1 numba-0.57.1 s
```

```
from tsfeatures import entropy, unitroot_kpss, unitroot_pp, stability,     \
                       sparsity, nonlinearity, lumpiness, hurst, guerrero, \
                       flat_spots, crossing_points, arch_stat, \
                       holt_parameters, acf_features, heterogeneity, \
                       hw_parameters, intervals, pacf_features, stl_features
from scipy.signal import periodogram, welch
from multiprocessing import Pool, cpu_count
from numba import jit


# NOTE: Tried using functools.partial to create 4 spectral_moment functions
#       Problems with missing __name__ attribute with partial functions :-(
#       Additionally, there are alternatives to using signal.welch -
#       FFT for one - there pros and cons should be considered
# @jit(forceobj=True)
def spectral_moment_1(y):
    f, Pxx_den = welch(y)
    sm = 0
    n = len(f)

    for i in range(0, n):
        sm += Pxx_den[i] * f[i]

    return sm


# @jit(forceobj=True)
def spectral_moment_2(y):
    sm = 0
    order = 2
    f, Pxx_den = welch(y)
    n = len(f)

    for i in range(0, n):
        sm += Pxx_den[i] * f[i] ** order

    return sm


# @jit(forceobj=True)
def spectral_moment_3(y):
    sm = 0
    order = 3
    f, Pxx_den = welch(y)
    n = len(f)

    for i in range(0, n):
        sm += Pxx_den[i] * f[i] ** order

    return sm


# @jit(forceobj=True)
```

```
def spectral_moment_4(y):
    sm = 0
    order = 4
    f, Pxx_den = welch(y)
    n = len(f)

    for i in range(0, n):
        sm += Pxx_den[i] * f[i] ** order

    return sm


# TOO SLOW - DO NOT USE!
# Use spectral_moment_{1,2,3,4} instead
# @jit(forceobj=True)
#def spectral_moments(y):
#    sm_1 = sm_2 = sm_3 = sm_4 = 0
#    f, Pxx_den = welch(y)
#    n = len(f)
#
#    for i in range(0, n):
#        sm_1 += Pxx_den[i] * f[i] ** 1
#        sm_2 += Pxx_den[i] * f[i] ** 2
#        sm_3 += Pxx_den[i] * f[i] ** 3
#        sm_4 += Pxx_den[i] * f[i] ** 4
#
#    return {'1': sm_1, '2': sm_2, '3': sm_3, '4': sm_4}


@jit
def peak2peak(y):
    return np.ptp(y)
# Faster than np.ptp - https://stackoverflow.com/a/40184053/100129
# df['y_window_48_min_max_diff'] = df['y'].rolling(48, min_periods=1).agg(['min',


@jit
def peak_location(y):
    return np.argmax(y)


@jit
def trough_location(y):
    return np.argmin(y)


@jit(forceobj=True)
def spectral_entropy(x, freq = 1):
    """
    Getting normalized Shannon entropy of power spectral density.
    PSD is calculated using scipy's periodogram.

    Args:
        x: The univariate time series array in the form of 1d numpy array.
```

```
        freq: int; Frequency for calculating the PSD via scipy periodogram.
              Default value is 1.

    Returns:
        Normalized Shannon entropy.
    """

    # calculate periodogram
    _, psd = periodogram(x, freq)

    # calculate shannon entropy of normalized psd
    psd_norm = psd / np.sum(psd)
    entropy = np.nansum(psd_norm * np.log2(psd_norm))

    return -(entropy / np.log2(psd_norm.size))


@jit(forceobj=True)
def std1st_der(x):
    """
    Calculating std1st_der: the standard deviation of the first derivative of the
    Reference: https://cran.r-project.org/web/packages/tsfeatures/vignettes/tsfeat

    Args:
        x: The univariate time series array in the form of 1d numpy array.

    Returns:
        The standard deviation of the first derivative of the time series.
    """

    std1st_der = np.std(np.gradient(x))
    return std1st_der


@jit(nopython=True)
def histogram_mode(x, nbins = 10):
    """
    Measures the mode of the data vector using histograms with a given number of b
    Reference: https://cran.r-project.org/web/packages/tsfeatures/vignettes/tsfeat

    Args:
        x: The univariate time series array in the form of 1d numpy array.
        nbins: int; Number of bins to get the histograms. Default value is 10.

    Returns:
        Mode of the data vector using histograms.
    """

    cnt, val = np.histogram(x, bins=nbins)
    return val[cnt.argmax()]


@jit(nopython=True)
def binarize_mean(x):
```

```
    """
    Converts time series array into a binarized version.
    Time-series values above its mean are given 1, and those below the mean are 0.
    Return the average value of the binarized vector.
    Reference: https://cran.r-project.org/web/packages/tsfeatures/vignettes/tsfeat

    Args:
        x: The univariate time series array in the form of 1d numpy array.

    Returns:
        The binarized version of time series array.
    """

    return np.mean(np.asarray(x) > np.mean(x))


@jit
def zero_proportion(x):
  return np.count_nonzero(x == 0) / len(x)


# \@jit Fails :-(
@jit(nopython=True)
def binned_entropy(x, max_bins=30):
    hist, _ = np.histogram(x, bins=max_bins)
    probs = hist / len(x)
    # binned_entropy = - sum(p * np.math.log(p) for p in probs if p != 0)
    binned_entropy = 0.0
    for p in probs:
      if p != 0:
        binned_entropy += p * np.math.log(p)

    return - binned_entropy


def tsf_call(ser, func, data, feat_col):
    '''
    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windo
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''
    tsf_dict = func(ser)
    return list(tsf_dict.values())[0]


def tsf_call_mv(ser, func, data, feat_col, window, agg):
    '''Last line adapted from https://stackoverflow.com/a/62717384/100129

    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windo
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''
```

```
    tsf_dict = func(data.loc[ser.index, feat_col])

    tsf_cols = [feat_col + '_window_' + str(window) + '_' + func.__name__ +
                '_' + tsf_key for tsf_key in tsf_dict.keys()]

    agg.loc[ser.index, tsf_cols] = list(tsf_dict.values())

    return 1


def get_tsf_agg(*args):
    (data, rol, feat_col, window, verbose, how) = args

    if verbose:
      print('\t\t', how.__name__)

    mv_funcs = ['acf_features', 'heterogeneity', 'holt_parameters',
                'hw_parameters', 'intervals', 'pacf_features', 'stl_features']
    np_funcs = ['binned_entropy', 'spectral_moment', 'spectral_moment_1',
                'spectral_moment_2', 'spectral_moment_3', 'spectral_moment_4',
                'peak2peak',
                'peak_location', 'trough_location', 'spectral_entropy',
                'std1st_der', 'histogram_mode', 'binarize_mean',
                'zero_proportion']

    tsfargs = {'func':     how,
               'data':     data,
               'feat_col': feat_col,
              #'window':   window
              }

    if how.__name__ in mv_funcs:
        agg = pd.DataFrame(index=data.index)
        tsfargs['agg']    = agg
        tsfargs['window'] = window
        rol.apply(tsf_call_mv, kwargs=tsfargs, raw=False)
    elif how.__name__ in np_funcs:
        agg = rol.apply(how, raw=True)
        tsfargs['window'] = window
        feat_name = get_tsf_feat_name(tsfargs)
        agg = agg.rename(feat_name)
    else:
        agg = rol.apply(tsf_call, kwargs=tsfargs, raw=True)
        tsfargs['window'] = window
        feat_name = get_tsf_feat_name(tsfargs)
        agg = agg.rename(feat_name)

    return agg


def get_tsf_feat_name(tsfargs):
    feat_col = tsfargs['feat_col']
    window   = tsfargs['window']
```

```
    how      = tsfargs['func']

    feat_name = f'{feat_col}_window_{window}_{how.__name__}'

    return feat_name


def get_rolling_tsfeatures(data, feat_cols, windows, AGGS, shifts = None, verbose=
  aggs_l = []

  pool_args = []

  for feat_col in feat_cols:
    if verbose:
      print(feat_col)

    for window in windows:
      if verbose:
        print('\twindow =', window)

      rol = data[feat_col].shift(1).rolling(window=window, min_periods=1,)

      start_time = timeit.default_timer()

      N_CORES = cpu_count()
      num_cores = np.min([N_CORES, len(AGGS)])
      pool = Pool(num_cores)

      #pool_args = []
      for agg in AGGS:
        pool_args.append([data, rol, feat_col, window, verbose, agg])

      # approx 10m 30s - pool.starmap_async very similar runtime
      # 22m 41s without multiprocessing
      # aggs = pd.concat([aggs, pool.starmap(get_tsf_agg, tuple(pool_args))], axis
      # aggs = pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1)
      # aggs_l.append(pool.starmap(get_tsf_agg, tuple(pool_args)))
      #aggs_l.append(pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1
      #pool.close()
      #pool.join()

      #if verbose:
      #  print('\t', round(timeit.default_timer() - start_time, 2))

  aggs_l.append(pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1))
  pool.close()
  pool.join()

  aggs = pd.concat(aggs_l, axis=1)
  aggs_nas = aggs.isna().sum().sum()
  print('aggs_nas:', aggs_nas)
  aggs = drop_problem_cols(aggs, window, drop_cor=False)

  # WARN: shifted features will be highly correlated with underlying features!
```

```
  for agg_name in aggs.columns:
    for shift_ in shifts:
      if shift_ == 0:
        continue
      else:
        new_name = agg_name + '_shift_' + str(shift_)
      # if verbose:
      #   print('\t', new_name)

      aggs[new_name] = aggs[agg_name].shift(shift_)

  aggs_nas = aggs.isna().sum().sum()
  print('aggs_nas:', aggs_nas)

  aggs = drop_cols_correlated_with_feat_cols(aggs, data[feat_cols])

  return aggs


windows = [6, 12, 24, 48]
feat_cols = [Y_COL, 'dew.point_des', 'humidity',]
shifts = [0]
# tsf_aggs = [unitroot_kpss, entropy]  # fast and usually included after filterinç
# functions roughly ordered by runtime
# hurst, guerrero, holt_parameters are too slow
# sparsity, crossing_points did not work up to window = 144
# stability, lumpiness work with window >= 24
tsf_aggs = [intervals,
            acf_features,
            pacf_features,
            stl_features,
            hw_parameters, arch_stat, heterogeneity,
            nonlinearity,
            unitroot_pp, lumpiness,
            ## spectral_moment_1, spectral_moment_2,
            ## spectral_moment_3, spectral_moment_4,
            flat_spots, stability, entropy,
            binned_entropy,
            unitroot_kpss,
            peak2peak,
            peak_location, trough_location, spectral_entropy, std1st_der,
            histogram_mode, binarize_mean, zero_proportion
            ]
params_tsf = {'windows':    windows,
              'shifts':     shifts,
              'feat_cols':  feat_cols,
              'aggs':       tsf_aggs,
              'agg_func':   get_rolling_tsfeatures,
              'verbose':    True,
              'dataset':    'valid',
              'regenerate': True,
              'feat_name':  'tsfeatures_',
              'date_str':   '.2022.09.20',
              'save_and_download': False,
```

```
             'save_to_gdrive': False,
            }

train_df_tsf, valid_df_tsf, test_df_tsf = get_features(train_df,
                                                       valid_df,
                                                       test_df,
                                                       params_tsf)


#############################################################################
# approx 1 hr 30 mins - added 246 features - 0 NAs
#
# windows = [6, 12, 24, 48]
# feat_cols = [Y_COL, 'dew.point_des', 'humidity',]
# shifts = [0]
# tsf_aggs = [intervals,
#             acf_features,
#             pacf_features,
#             stl_features,
#             hw_parameters, arch_stat, heterogeneity,
#             nonlinearity,
#             unitroot_pp, lumpiness,
#             flat_spots, stability, entropy,
#             binned_entropy,
#             unitroot_kpss,
#             peak2peak,
#             peak_location, trough_location, spectral_entropy, std1st_der,
#             histogram_mode, binarize_mean, zero_proportion]
# params_tsf = {'windows':    windows,
#               'shifts':     shifts,
#               'feat_cols':  feat_cols,
#               'aggs':       tsf_aggs,
#               'agg_func':   get_rolling_tsfeatures,
#               'verbose':    True,
#               'dataset':    'valid',
#               'regenerate': True,
#               'feat_name':  'tsfeatures_',
#               'date_str':   '.2022.09.20',
#               'save_and_download': True,}
```

```
dataset: valid
y_des
        window = 6
        window = 12
        window = 24
        window = 48
dew.point_des
        window = 6
        window = 12
        window = 24
        window = 48
humidity
        window = 6
        window = 12
        window = 24
        window = 48
                intervals
                lumpiness
                spectral_entropy
                flat_spots
                hw_parameters
                binned_entropy
                zero_proportion
                unitroot_pp
                trough_location
                spectral_entropy
                intervals
                unitroot_kpss
                peak2peak
                peak_location
                trough_location
                spectral_entropy
                std1st_der
                stability
                entropy
                std1st_der
                histogram_mode
                histogram_mode
                binarize_mean
                binned_entropy
                zero_proportion
                lumpiness
                binarize_mean
                intervals
                zero_proportion
                unitroot_kpss
                intervals
                std1st_der
                peak2peak
                peak_location
                histogram_mode
                trough_location
                stl_features
                binarize_mean
                flat_spots
                entropy
                binned_entropy
                unitroot_kpss
```

262 additional features were added in approximately 2 hours.

Features above or between `dew.point_des` and `humidity` in the feature selection scores table may prove useful in future models. The `intervals_mean` and `histogram_mode` features appear promising. Other features may be useful for later forecast steps.

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.

TODO Add `pressure` to `feat_cols` - run time is already too long :-(

TODO Add 96 to `windows` - run time is already too long :-(

TODO Move fastest functions to rolling stats feature set

- `peak_location`, `trough_location`, `spectral_entropy`, `std1st_der`, `histogram_mode`, `binarize_mean`, `zero_proportion` are canditates for moving
- faster version of `peak2peak` is also a candidate
- none of these functions are part of the tsfeatures package

TODO Consider replacing [nixtla tsfeatures]() package with [tsfeatures from kats package]()

- faster - most functions are numba jitted
- additional features - changepoints etc
- not all features are identical
    - flat_spots - nixtla gives number, kats gives max length

---

Expansion of intervals_mean and pacf features

The intervals_mean and some of the pacf features from the tsfeatures package are highly beneficial to the lightgbm models. Here I expand their use over a wider range of window values.

```
SC_FluctAnal_2_dfa_50_1_2_logi_prop_r1  0.04
```

```
from tsfeatures import intervals, pacf_features  # acf_features
from multiprocessing import Pool, cpu_count


def tsf_call_mv(ser, func, data, feat_col, window, agg):
    '''Last line adapted from https://stackoverflow.com/a/62717384/100129

    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windc
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''

    tsf_dict = func(data.loc[ser.index, feat_col])

    tsf_cols = [feat_col + '_window_' + str(window) + '_' + func.__name__ +
                '_' + tsf_key for tsf_key in tsf_dict.keys()]
```

```
    agg.loc[ser.index, tsf_cols] = list(tsf_dict.values())

    return 1


def get_tsf_agg(*args):
    (data, rol, feat_col, window, verbose, how) = args

    if verbose:
      print('\t\t', how.__name__)

    mv_funcs = ['acf_features', 'heterogeneity', 'holt_parameters',
                'hw_parameters', 'intervals', 'pacf_features', 'stl_features']
    np_funcs = ['binned_entropy', 'spectral_moment', 'spectral_moment_1',
                'spectral_moment_2', 'spectral_moment_3', 'spectral_moment_4',
                'peak2peak',
                'peak_location', 'trough_location', 'spectral_entropy',
                'std1st_der', 'histogram_mode', 'binarize_mean',
                'zero_proportion']

    tsfargs = {'func':     how,
               'data':     data,
               'feat_col': feat_col,
              #'window':   window
              }

    if how.__name__ in mv_funcs:
        agg = pd.DataFrame(index=data.index)
        tsfargs['agg']    = agg
        tsfargs['window'] = window
        rol.apply(tsf_call_mv, kwargs=tsfargs, raw=False)
    elif how.__name__ in np_funcs:
        agg = rol.apply(how, raw=True)
        tsfargs['window'] = window
        feat_name = get_tsf_feat_name(tsfargs)
        agg = agg.rename(feat_name)
    else:
        agg = rol.apply(tsf_call, kwargs=tsfargs, raw=True)
        tsfargs['window'] = window
        feat_name = get_tsf_feat_name(tsfargs)
        agg = agg.rename(feat_name)

    return agg


def get_tsf_feat_name(tsfargs):
    feat_col = tsfargs['feat_col']
    window   = tsfargs['window']
    how      = tsfargs['func']

    feat_name = f'{feat_col}_window_{window}_{how.__name__}'

    return feat_name
```

```
def get_rolling_tsfeatures(data, feat_cols, windows, AGGS, shifts = None, verbose=
  aggs_l = []

  pool_args = []

  for feat_col in feat_cols:
    if verbose:
      print(feat_col)

    for window in windows:
      if verbose:
        print('\twindow =', window)

      rol = data[feat_col].shift(1).rolling(window=window, min_periods=1,)

      start_time = timeit.default_timer()

      N_CORES = cpu_count()
      num_cores = np.min([N_CORES, len(AGGS)])
      pool = Pool(num_cores)

      #pool_args = []
      for agg in AGGS:
        pool_args.append([data, rol, feat_col, window, verbose, agg])

      # approx 10m 30s - pool.starmap_async very similar runtime
      # 22m 41s without multiprocessing
      # aggs = pd.concat([aggs, pool.starmap(get_tsf_agg, tuple(pool_args))], axis
      # aggs = pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1)
      # aggs_l.append(pool.starmap(get_tsf_agg, tuple(pool_args)))
      #aggs_l.append(pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1
      #pool.close()
      #pool.join()

      #if verbose:
      #  print('\t', round(timeit.default_timer() - start_time, 2))

  aggs_l.append(pd.concat(pool.starmap(get_tsf_agg, tuple(pool_args)), axis=1))
  pool.close()
  pool.join()

  aggs = pd.concat(aggs_l, axis=1)
  aggs_nas = aggs.isna().sum().sum()
  print('aggs_nas:', aggs_nas)
  aggs = drop_problem_cols(aggs, window, drop_cor=False)

  # WARN: shifted features will be highly correlated with underlying features!
  for agg_name in aggs.columns:
    for shift_ in shifts:
      if shift_ == 0:
        continue
      else:
        new_name = agg_name + '_shift_' + str(shift_)
```

```
      # if verbose:
      #   print('\t', new_name)

      aggs[new_name] = aggs[agg_name].shift(shift_)

  aggs_nas = aggs.isna().sum().sum()
  print('aggs_nas:', aggs_nas)

  aggs = drop_cols_correlated_with_feat_cols(aggs, data[feat_cols])

  return aggs


windows = [12, 18, 24, 30, 36, 42, 48, 54, 60, 96]
# shifts = [6, 12, 24, 48, 60, 96]
shifts = []
feat_cols = [Y_COL]
tsf_aggs = [intervals,
           #acf_features,
            pacf_features,
           ]
params_int = {'windows':    windows,
              'shifts':     shifts,
              'feat_cols':  feat_cols,
              'aggs':       tsf_aggs,
              'agg_func':   get_rolling_tsfeatures,
              'verbose':    True,
              'dataset':    'valid',
              'regenerate': True,
              'feat_name':  'intervals_etc_',
              'date_str':   '.2022.09.20',
              'save_and_download': False,
              'save_to_gdrive': False,
             }

train_df_int, valid_df_int, test_df_int = get_features(train_df,
                                                       valid_df,
                                                       test_df,
                                                       params_int)


###############################################################################
# approx 50 mins - added 40 features - 0 NAs
#
# windows = [12, 18, 24, 30, 36, 42, 48, 54, 60, 96]
# shift   = []
# feat_cols = [Y_COL]
# tsf_aggs = [intervals, pacf_features]
# params_int = {'windows':    windows,
#               'shifts':     shifts,
#               'feat_cols':  feat_cols,
#               'aggs':       tsf_aggs,
#               'agg_func':   get_rolling_tsfeatures,
#               'verbose':    True,
```

```
#                'dataset':    'valid',
#                'regenerate': True,
#                'feat_name':  'intervals_etc_',
#                'date_str':   '.2022.09.20',
#                'save_and_download': True,}
```

```
dataset: valid
y_des
        window = 12
        window = 18
        window = 24
        window = 30
        window = 36
        window = 42
        window = 48
        window = 54
        window = 60
        window = 96
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
                 intervals
                 pacf_features
aggs_nas: 176640
columns with null values:
y_des_window_12_intervals_intervals_mean      0
y_des_window_12_pacf_features_x_pacf5         0
y_des_window_12_pacf_features_diff1x_pacf5    0
y_des_window_12_pacf_features_diff2x_pacf5    0
y_des_window_18_intervals_intervals_mean      0
y_des_window_18_pacf_features_x_pacf5         0
y_des_window_18_pacf_features_diff1x_pacf5    0
y_des_window_18_pacf_features_diff2x_pacf5    0
y_des_window_24_intervals_intervals_mean      0
y_des_window_24_pacf_features_x_pacf5         0
y_des_window_24_pacf_features_diff1x_pacf5    0
y_des_window_24_pacf_features_diff2x_pacf5    0
y_des_window_30_intervals_intervals_mean      0
y_des_window_30_pacf_features_x_pacf5         0
y_des_window_30_pacf_features_diff1x_pacf5    0
y_des_window_30_pacf_features_diff2x_pacf5    0
y_des_window_36_intervals_intervals_mean      0
y_des_window_36_pacf_features_x_pacf5         0
y_des_window_36_pacf_features_diff1x_pacf5    0
y_des_window_36_pacf_features_diff2x_pacf5    0
y_des_window_42_intervals_intervals_mean      0
y_des_window_42_pacf_features_x_pacf5         0
y_des_window_42_pacf_features_diff1x_pacf5    0
y_des_window_42_pacf_features_diff2x_pacf5    0
y_des_window_42_pacf_features_diff2x_pacf5_50_0_logi_prop_r1     0.44
```

40 additional features were added in approximately 50 mins.

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.

Permutation importance experiments

Permutation feature importance measures the increase in the prediction error of the model after we shuffle the feature values. It can be used for feature selection.

- sklearn permutation importance
- Permutation Importance vs Random Forest Feature Importance (MDI)
- Permutation Importance with Multicollinear or Correlated Features

Here I run some permutation importance experiments with a simple and fast ridge regression model.

```
from sklearn.linear_model import Ridge
from sklearn.inspection import permutation_importance


def get_pi_data(train_df, valid_df, pi_cols, pi_lags, pred_step, y_col=Y_COL):
  all_cols = pi_cols + [y_col]
  train = train_df[all_cols].dropna()
  valid = valid_df[all_cols].dropna()

  # Add lagged features
  train = train.assign(**{
    f'{col}_lag_{lag}': train[col].shift(lag)
    for lag in pi_lags
    for col in all_cols
  })
  valid = valid.assign(**{
    f'{col}_lag_{lag}': valid[col].shift(lag)
    for lag in pi_lags
    for col in all_cols
  })

  train = train.dropna()
  valid = valid.dropna()

  y_train = train[y_col].shift(-pred_step).dropna()
  X_train = train.head(y_train.shape[0])
  # X_train = train[pi_cols + [y_col]].tail(y_train.shape[0])

  y_val = valid[y_col].shift(-pred_step).dropna()
  X_val = valid.head(y_val.shape[0])
  # X_val = valid[pi_cols + [y_col]].tail(y_val.shape[0])

  return X_train, y_train, X_val, y_val


def print_permuation_importances(model, train, X_val, y_val):
```

```
  scoring = ['r2', 'neg_mean_absolute_error', 'neg_root_mean_squared_error']
  r_multi = permutation_importance(model, X_val, y_val,
                                   n_repeats = 30,
                                   random_state = 0,
                                   scoring = scoring)

  for metric in r_multi:
    print(f"\n{metric}")
    r = r_multi[metric]
    for i in r.importances_mean.argsort()[::-1]:
      if r.importances_mean[i] - 2 * r.importances_std[i] > 0:
        print(f"    {train.columns[i]:<20}"
              f"\t{r.importances_mean[i]:.3f}"
              f" +/- {r.importances_std[i]:.3f}")


# pred_step = 48
pi_lags = [1, 2, 3, 48]
pi_cols = ['dew.point_des', 'humidity', 'pressure', 'irradiance',
          #'wind.speed.mean.sqrt.x', 'wind.speed.mean.sqrt.y',
          #'dT_dH', 'dT_dP', 'dT_dTdp', 'y_des_shadow'
          ]

for pred_step in range(1, 4):
  X_train, y_train, X_val, y_val = get_pi_data(train_df,
                                               valid_df,
                                               pi_cols,
                                               pi_lags,
                                               pred_step)

  model = Ridge(alpha=1e2).fit(X_train, y_train)
  r2s = round(model.score(X_val, y_val), 6)
  preds = model.predict(X_val)
  rmses = round(rmse_(y_val, preds), 6)
  maes  = round(mae_(y_val, preds), 6)
  print('\n\n', pred_step, r2s, rmses, maes)

  print_permuation_importances(model, X_train, X_val, y_val)
```

```
    1 0.981094 0.501796 0.357047

   r2
       y_des                    2.338 +/- 0.021
       irradiance_lag_1         0.345 +/- 0.002
       irradiance               0.165 +/- 0.001
       irradiance_lag_2         0.081 +/- 0.001
       y_des_lag_3              0.040 +/- 0.000
       pressure_lag_1           0.026 +/- 0.000
       pressure                 0.024 +/- 0.000
       y_des_lag_1              0.010 +/- 0.000
       pressure_lag_3           0.006 +/- 0.000
       pressure_lag_2           0.006 +/- 0.000
       irradiance_lag_48        0.005 +/- 0.000
       humidity_lag_3           0.003 +/- 0.000
```

```
    y_des_lag_2                 0.002 +/- 0.000
    humidity                    0.002 +/- 0.000
    dew.point_des_lag_3         0.001 +/- 0.000
    humidity_lag_1              0.001 +/- 0.000
    irradiance_lag_3            0.000 +/- 0.000
    dew.point_des               0.000 +/- 0.000
    dew.point_des_lag_2         0.000 +/- 0.000
    y_des_lag_48                0.000 +/- 0.000
    humidity_lag_2              0.000 +/- 0.000
    dew.point_des_lag_48        0.000 +/- 0.000

neg_mean_absolute_error
    y_des                       4.086 +/- 0.020
    irradiance_lag_1            1.182 +/- 0.007
    irradiance                  0.759 +/- 0.005
    irradiance_lag_2            0.478 +/- 0.004
    y_des_lag_3                 0.338 +/- 0.003
    pressure_lag_1              0.241 +/- 0.002
    pressure                    0.235 +/- 0.002
    y_des_lag_1                 0.117 +/- 0.002
    pressure_lag_3              0.081 +/- 0.001
    pressure_lag_2              0.066 +/- 0.001
    irradiance_lag_48           0.049 +/- 0.001
    humidity_lag_3              0.033 +/- 0.001
    y_des_lag_2                 0.022 +/- 0.001
    humidity                    0.020 +/- 0.001
    dew.point_des_lag_3         0.012 +/- 0.001
    humidity_lag_1              0.008 +/- 0.000
    irradiance_lag_3            0.003 +/- 0.000
    y_des_lag_48                0.001 +/- 0.000
    dew.point_des               0.001 +/- 0.000
    dew.point_des_lag_2         0.001 +/- 0.000
    humidity_lag_2              0.001 +/- 0.000
    dew.point_des_lag_48        0.000 +/- 0.000

neg_root_mean_squared_error
    y_des                       5.102 +/- 0.025
    irradiance_lag_1            1.701 +/- 0.007
    irradiance                  1.063 +/- 0.006
    irradiance_lag_2            0.653 +/- 0.004
    y_des_lag_3                 0.384 +/- 0.003
```

...

Bivariate features

Features from catch22 and tsfeatures are derived from single variables. Features can also be extracted from two or more variables. Here I consider only two variables. Initially based on, https://towardsdatascience.com/a-step-by-step-guide-to-feature-engineering-for-multivariate-time-series-162ccf232e2f, by Vitor Cerqueira, but greatly expanded.

Bivariate feature types considered here include: correlations, distances, dynamic time warping, cross power spectral density moments and custom features.

correlations:

- Pearson - np_corr custom function using np.corrcoef
- ccf
    - cross-correlation from statsmodels
- ccovf
    - cross covariance from statsmodels

distances:

- energy
- wasserstein aka earth mover's distance

dynamic time warping:

- dynamic time warping
- dtaidistance
    - Time series distances: Dynamic Time Warping (fast DTW implementation in C)

cross power spectral density moments:

- what is cross spectral density
- csd from scipy
    - cross spectral density is a spectrum
- tells you how two signals are roughly correlated in time through their respective power spectra
- only considering first moment (corresponds to the mean) for now
- **TODO** consider higher moments

custom features:

- `seasonal_strength`
    - seasonal strength in Forecasting Principles and Practice
- `signaltonoise`
    - signal-to-noise statistic
    - difference in mean of features over sum of standard deviations
    - not to be confused with signal to noise ratio

The numba jit decorator is used to compile the `signal_strength`, `signaltonoise` and `np_corr` custom functions. This greatly reduces their run time.

Additional pairwise distance measures from sklearn are calculated in the next sub-section.

Features considered:

- all bivariate combinations from
    - `y_des`
    - `dew.point_des`

    - humidity
    - pressure

Lags considered:

- [6, 12, 24, 48]

Bivariate functions considered:

- signaltonoise
- seasonalstrength
- np_corr
- ccf
- ccovf
- energy_distance
- wasserstein_distance
- csd_moment
- dtw_dist

```
y_des_window_24_interval_mean_IN_AutoMutualInfoStats_40_gaussian_fmmi          1.42
```

```
!pip install dtaidistance
```

```
Collecting dtaidistance
  Downloading dtaidistance-2.3.11-cp310-cp310-manylinux_2_17_x86_64.manylinux2
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.9/2
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-package
Installing collected packages: dtaidistance
Successfully installed dtaidistance-2.3.11
```

```
y_des_window_36_partialautocorr_diff1_pacf_0_FC_LocalSimple_mean3_stderr  0.34
```

```
from scipy.signal import csd
from scipy.stats import pearsonr, spearmanr, kendalltau, energy_distance, \
                        wasserstein_distance
from statsmodels.tsa.stattools import ccf, ccovf
from dtaidistance import dtw
from numba import jit


def dtw_dist(x, y):
    '''Dynamic time warping distance

    dtw.distance_fast call means cannot be used with numba :-(

    https://github.com/wannesm/dtaidistance
    '''

    # distance_fast requires an array as input with the double type
    s1 = np.array(x, dtype=np.double)
    s2 = np.array(y, dtype=np.double)

    return dtw.distance_fast(s1, s2, use_pruning=True)


# TODO: Consider higher moments
```

```
# TODO: Try jitting this with @jit(nopython=True) or @jit(forceobj=True)
def csd_moment(x, y, order=1):
    '''Cross power spectral density, Pxy, using Welch's method

    csd call means cannot be used with numba :-(

    https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.csd.html
    '''

    f, Pxy_den = csd(x, y, nperseg=len(x))
    csd_ = 0
    n = len(f)

    for i in range(0, n):
        csd_ += Pxy_den[i] * f[i] ** order

    return np.abs(csd_)


@jit(nopython=True)
def signaltonoise(arr, min_window=6):
    '''Not to be confused with signal to noise ratio'''
    a = arr[:, 0]
    b = arr[:, 1]

    if len(arr) < min_window:
        return np.nan
    else:
        return np.abs(np.mean(a) - np.mean(b)) / (np.std(a) + np.std(b))


@jit(nopython=True)
def seasonal_strength(arr, min_window=6):
  '''res_var - residual variable, seas_var - seasonal
  https://otexts.com/fpp2/seasonal-strength.html
  '''
  res_var  = arr[:, 0]
  seas_var = arr[:, 1]

  if len(arr) < min_window:
    return np.nan
  else:
    return 1 - np.var(res_var) / (np.var(seas_var + res_var))


@jit(nopython=True)
def np_corr(arr):
    '''...'''
    a = arr[:, 0]
    b = arr[:, 1]
    return np.corrcoef(a, b)[0, 1]


def biv_call(ser, func, data, col1, col2):
```

```
    '''
    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windo
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''
    return func(data.loc[ser.index, col1], data.loc[ser.index, col2])


# TODO: Investigate taking min, max, range, mean, std ... instead of first element
def biv_call_cc(ser, func, data, col1, col2):
    '''
    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windo
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''
    return func(data.loc[ser.index, col1], data.loc[ser.index, col2])[0]


def biv_call_sci_stats(ser, func, data, col1, col2, window, agg):
    '''Last line adapted from https://stackoverflow.com/a/62717384/100129

    agg = pd.DataFrame(list(pd.Series(data[feat_col].shift(1).rolling(window=windo
    works for single value and multi-value functions returning dicts :-)
    no need for tsf_call functions
    must rename columns and set index
    '''
    biv_vals = func(data.loc[ser.index, col1], data.loc[ser.index, col2])
    biv_cols = [col1 + '_' + col2 + '_window_' + str(window) + '_' + func.__name__
    agg.loc[ser.index, biv_cols] = biv_vals.statistic
    return 1


def get_rolling_bivariate_features(data, feat_cols, windows, AGGS, verbose=False):

  stats_cc  = ['ccf', 'ccovf']
  sci_stats = ['pearsonr', 'spearmanr', 'kendalltau']
  np_numba  = ['signaltonoise', 'seasonal_strength', 'np_corr']
  col_combs = list(itertools.combinations(feat_cols, 2))
  aggs = pd.DataFrame(index=data.index)

  for col1, col2 in col_combs:
    if verbose:
      print(col1, '-', col2, end='')

    for window in windows:
      if verbose:
        print('\t', window)

      # Should use data[[col1, col2]]... but
      # if use data[col1]... then can rougly HALVE run time but
      # must adapt biv_call... functions to use
      # func(data.loc[ser.index, col1], data.loc[ser.index, col2]) and
```

```
    # use agg = agg.rename(...) instead of agg = agg[col1].rename(...)
    # rol = data[[col1, col2]].shift(1).rolling(window=window, min_periods=1)
    rol = data[col1].shift(1).rolling(window=window, min_periods=1)

    for how in AGGS:
      #if (col1 == 'irradiance' or col2 == 'irradiance') and \
      #   how.__name__ == 'pearsonr':
      #  continue

      # Avoid division by zero errors
      if (col1 in ['humidity', 'pressure'] and \
          col2 in ['humidity', 'pressure'] and \
          how.__name__ == 'signaltonoise' and \
          window < 24):
        continue

      if verbose:
        print('\t\t', how.__name__, end='')

      bivargs = {'func': how, 'data': data, 'col1': col1, 'col2': col2}
      start_time = timeit.default_timer()

      if how.__name__ in sci_stats:
        agg = pd.DataFrame(index=data.index)
        bivargs['window'] = window
        bivargs['agg'] = agg
        rol.apply(biv_call_sci_stats, kwargs=bivargs, raw=False)
      # elif how.__name__ in stats_cc or how.__name__ in sci_sig:
      elif how.__name__ in stats_cc:
        agg = rol.apply(biv_call_cc, kwargs=bivargs)
        agg = agg.rename(f'{col1}_{col2}_window_{window}_{how.__name__}_0')
        # agg = agg[col1].rename(f'{col1}_{col2}_window_{window}_{how.__name__}_
      elif how.__name__ in np_numba:
        # WARN: Do not overwrite rol here - use rol_numba instead
        rol_numba = data[[col1, col2]].shift(1).rolling(window=window, min_perio
        agg = rol_numba.apply(how, engine='numba', raw=True)
        agg = agg[col1].rename(f'{col1}_{col2}_window_{window}_{how.__name__}')
      else:
        agg = rol.apply(biv_call, kwargs=bivargs)
        agg = agg.rename(f'{col1}_{col2}_window_{window}_{how.__name__}')
        # agg = agg[col1].rename(f'{col1}_{col2}_window_{window}_{how.__name__}

      aggs = aggs.join(agg)
      if verbose:
        print('\t', round(timeit.default_timer() - start_time, 2))


# TODO Move this immediately after the deseasonalisation
# bivariate eg feat_col - seas_col

windows = [48, 96]

for feat_col in feat_cols:
  if '_des' in feat_col:
```

```
      if verbose:
        print(feat_col, end='')

      how = seasonal_strength
      seas_col = feat_col.replace('_des', '_yhat')
      bivargs = {'func': how, 'data': data, 'col1': feat_col, 'col2': seas_col}

      for window in windows:
        # Division by zero with seasonal_strength when window =  6 on valid_df
        # Division by zero with seasonal_strength when window = 12,24 on train_df
        if window < 48:
          continue

        if verbose:
          print('\t', window, '\t', how.__name__, end='')

        start_time = timeit.default_timer()

        rol = data[[feat_col, seas_col]].shift(1).rolling(window=window, min_perio
        agg = rol.apply(how, raw=True, engine='numba')

        agg = agg[feat_col].rename(f'{feat_col}_window_{window}_{how.__name__}')
        aggs = aggs.join(agg)

        if verbose:
          print('\t', round(timeit.default_timer() - start_time, 2))


  aggs = drop_problem_cols(aggs, window)
  aggs = drop_cols_correlated_with_feat_cols(aggs, data[feat_cols])

  return aggs


# Division by zero with seasonal_strength when window =  6 on valid_df
# Division by zero with seasonal_strength when window = 12 on train_df
# feat_cols = [Y_COL, 'dew.point_des']
# windows = [6, 48]
# feat_cols = [Y_COL, 'dew.point_des', 'humidity_des', 'pressure',
#              'irradiance']
# biv_aggs = [signaltonoise]
feat_cols = [Y_COL, 'dew.point_des', 'humidity', 'pressure']
# feat_cols = ['humidity', 'pressure']
windows = [6, 12, 24, 48]
biv_aggs = [signaltonoise,
            np_corr,
            ccf, ccovf,
            energy_distance,
            wasserstein_distance,
            # spearmanr, kendalltau
            csd_moment,
            dtw_dist,
           ]
params_biv = {'windows':    windows,
```

```
              'feat_cols':  feat_cols,
              'aggs':       biv_aggs,
              'agg_func':   get_rolling_bivariate_features,
              'verbose':    True,
              'dataset':    'valid',
              'regenerate': True,
              'feat_name':  'bivariate_',
              'date_str':   '.2022.09.20',
              'save_and_download': False,
              'save_to_gdrive': True,
             }

train_df_biv, valid_df_biv, test_df_biv = get_features(train_df,
                                                       valid_df,
                                                       test_df,
                                                       params_biv)


################################################################################
# approx 6 hours 15 mins - added 103 features (0 NAs)
#
# feat_cols = [Y_COL, 'dew.point_des', 'humidity', 'pressure']#, 'irradiance']
# windows = [6, 12, 24, 48]
# biv_aggs = [signaltonoise,
#             np_corr,
#             ccf, ccovf,
#             energy_distance,
#             wasserstein_distance,
#             # spearmanr, kendalltau
#             csd_moment,
#             dtw_dist,]
# params = {'windows':    windows,
#           'feat_cols':  feat_cols,
#           'aggs':       biv_aggs,
#           'agg_func':   get_rolling_bivariate_features,
#           'verbose':    True,
#           'dataset':    'valid',
#           'regenerate': True,
#           'feat_name':  'bivariate_',
#           'date_str':   '.2022.09.20',}
```

```
dataset: valid
y_des - dew.point_des    6
                  signaltonoise   6.35
                  np_corr         13.16
                  ccf     25.14
                  ccovf   23.51
                  energy_distance         20.49
                  wasserstein_distance    20.2
                  csd_moment      26.02
                  dtw_dist        20.13
         12
                  signaltonoise   0.02
                  np_corr         0.05
                  ccf     24.91
                  ccovf   23.28
                  energy_distance         20.57
                  wasserstein_distance    20.37
                  csd_moment      26.15
                  dtw_dist        20.14
         24
                  signaltonoise   0.02
                  np_corr         0.05
                  ccf     25.17
                  ccovf   23.43
                  energy_distance         20.27
                  wasserstein_distance    20.26
                  csd_moment      26.42
                  dtw_dist        20.0
         48
                  signaltonoise   0.03
                  np_corr         0.06
                  ccf     25.2
                  ccovf   23.29
                  energy_distance         20.72
                  wasserstein_distance    20.75
                  csd_moment      26.79
                  dtw_dist        20.24
y_des - humidity         6
                  signaltonoise   0.02
                  np_corr         0.05
                  ccf     24.96
                  ccovf   23.16
                  energy_distance         20.29
                  wasserstein_distance    20.12
                  csd_moment      25.82
                  dtw_dist        20.0
         12
                  signaltonoise   0.02
                  np_corr         0.05
                  ccf     24.91
                  ccovf   22.96
                  energy_distance         20.26
                  wasserstein_distance    20.24
                  csd_moment      25.97
                  dtw_dist        19.79
         24
    ...          ...  signaltonoise   0.03       ...         ...              ...
                      hwshparameters
```

103 additional features (0 NAs) were added in approximately 6 hours 15 mins.

Features above or between `dew.point_des` and `humidity` in the feature selection scores table may prove useful in future models. The `y_des_humidity_window_6_energy_distance`, `y_des_humidity_window_12_wasserstein_distance`, `y_des_humidity_window_48_energy_distance` and `y_des_humidity_window_24_energy_distance` features may prove useful. Other features may be useful for later forecast steps.

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.

TODO add 96 to `windows` - run time is already way too long :-(

- consider swapping with one of the existing `windows` values

TODO add `irradiance` to `feat_cols` - run time is already way too long :-(

- consider swapping with one of the existing `feat_cols` values

TODO The `signaltonoise` and `seasonal_strength` functions are much faster to caluclate than the other bivariate functions

- they should be moved to the rolling stats feature set which contains other fast functions

TODO consider adding later

- acovf autocovariance from statsmodels

Bivariate features - sklearn distances and kernels

sklearn provides pairwise distances and pairwise kernels:

- pairwise metrics
- kernel approximation

The joblib package is used to run some feature calculations in parrallel. This reduces run time but not as much as I hoped.

Pre-processing functions transform raw features into a representation that may be more suitable for forecasting.

Pre-processing considered:

- Z-score (mean and deviation scaling)

Min max scaling is implemented but not used. Earlier experiments with Box-Cox transformations were not beneficial.

Distances considered:

- euclidean
- canberra
- cosine

- hamming
- sqeuclidean

Kernels considered:

- linear
- rbf
- sigmoid
- poly - 2 and 3

Features considered:

- all bivariate combinations from
    - `y_des`
    - `humidity`
    - `dew.point_des`
    - `pressure`

```
interpolate: pressure_window_48_sc_FluctAnal_2_rsrangefit_50_1_logi_prop_r1
energy distance        20.78
```

```
from scipy.stats import zscore
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics.pairwise import pairwise_kernels
from sklearn.metrics import pairwise_distances
from joblib import Parallel, delayed
from multiprocessing import cpu_count


def get_bivariate_poly_kern(x, Y, deg, verbose=False):
  print('\t poly_', deg, sep='', end='')

  N_CORES = cpu_count()
  bs = int(np.ceil(len(x) / N_CORES))

  start_time = timeit.default_timer()

  poly_vals = Parallel(n_jobs=N_CORES, batch_size=bs, pre_dispatch='all', verbose=
                      delayed(pairwise_kernels)(x[i].reshape(-1, 1),
                                                Y[i].reshape(-1, 1),
                                                metric='poly',
                                                degree=deg)
                      for i in range(len(x))
  )

  if verbose:
    print('\t', round(timeit.default_timer() - start_time, 2))

  return np.concatenate(poly_vals, axis=1)[0]


def get_bivariate_feature_name(x_col, y_col, func, scale, z_score):
  fn = f'{y_col}_{x_col}'

  if scale is True:
```

```
    fn += f'_scaled'
  elif z_score is True:
    fn += f'_zscore'

  fn += f'_{func}'

  return fn


def get_scaled_values(X, col, scale, z_score):
  if scale is True:
    scaler = MinMaxScaler()
    vals = scaler.fit_transform(X[[col]])
  elif z_score is True:
    vals = zscore(X[col])
  else:
    vals = X[col].values

  return vals


def get_bivariate_dists_kerns(data, x_cols, aggs,
                              y_col=Y_COL,
                              scale=False, z_score=False,
                              verbose=True):

  colnames = [y_col, *x_cols]
  X = data[colnames].dropna()
  col_combs = list(itertools.combinations(colnames, 2))
  feat_set = {}

  all_dists = ['euclidean', 'l2', 'l1', 'manhattan', 'cityblock', 'canberra',
               'chebyshev', 'correlation', 'cosine', 'hamming', 'minkowski',
               'sqeuclidean']
  all_kerns = ['linear', 'poly', 'rbf', 'laplacian', 'sigmoid'] #, 'cosine']

  pw_dists = [agg for agg in aggs if agg in all_dists]
  pw_kerns = [agg for agg in aggs if agg in all_kerns]

  for col1, col2 in col_combs:
    if verbose:
      print(col1, '-', col2)
    Y = get_scaled_values(X, col1, scale, z_score)
    x = get_scaled_values(X, col2, scale, z_score)

    for pw_dist in pw_dists:
      start_time = timeit.default_timer()
      if verbose:
        print('\t', pw_dist, end='')

      # slightly faster than for loop
      # slower than cdist but it OOM crashes :-(
      N_CORES = cpu_count()
      bs = int(np.ceil(len(x) / N_CORES))
```

```
      dist_vals = Parallel(n_jobs=N_CORES, batch_size=bs, pre_dispatch='all', verb
                           delayed(pairwise_distances)(x[i].reshape(-1, 1),
                                                       Y[i].reshape(-1, 1),
                                                       metric=pw_dist)
                           for i in range(len(x))
      )

      feat_name = get_bivariate_feature_name(col2, col1, pw_dist, scale, z_score)
      feat_set[feat_name] = np.concatenate(dist_vals, axis=1)[0]

      if verbose:
        print('\t', round(timeit.default_timer() - start_time, 2))

    for pw_kern in pw_kerns:
      if pw_kern == 'poly':
        feat_name = get_bivariate_feature_name(col2, col1, pw_kern + '_2', scale,
        feat_set[feat_name] = get_bivariate_poly_kern(x, Y, 2, verbose)
        feat_name = get_bivariate_feature_name(col2, col1, pw_kern + '_3', scale,
        feat_set[feat_name] = get_bivariate_poly_kern(x, Y, 3, verbose)
      else:
        start_time = timeit.default_timer()
        if verbose:
          print('\t', pw_kern, end='')

        kern_vals = Parallel(n_jobs=N_CORES, batch_size=bs, pre_dispatch='all', ve
                             delayed(pairwise_kernels)(x[i].reshape(-1, 1),
                                                       Y[i].reshape(-1, 1),
                                                       metric=pw_kern)
                             for i in range(len(x))
        )

        feat_name = get_bivariate_feature_name(col2, col1, pw_kern, scale, z_score
        feat_set[feat_name] = np.concatenate(kern_vals, axis=1)[0]

        if verbose:
          print('\t', round(timeit.default_timer() - start_time, 2))

  bivar_feats = pd.DataFrame(feat_set, index=X.index)
  del feat_set

  bivar_feats = drop_problem_cols(bivar_feats, 0)
  bivar_feats = drop_cols_correlated_with_feat_cols(bivar_feats, data[colnames])

  return bivar_feats


# pw_dists = ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
# pw_dists = ['euclidean', 'l2', 'l1', 'manhattan', 'cityblock', 'canberra',
#             'chebyshev', 'correlation', 'cosine', 'hamming', 'minkowski',
#             'sqeuclidean']
# pw_kerns = ['linear', 'poly', 'rbf', 'laplacian', 'sigmoid', 'cosine']
# pw_dists = ['cosine',]
# pw_kerns = ['rbf',]
# feat_cols = ['humidity_des']
```

```
pw_dists  = ['euclidean',   'canberra', 'cosine',   'hamming', 'sqeuclidean']
pw_kerns  = ['linear',  'rbf', 'sigmoid', 'poly']
feat_cols = ['humidity', 'dew.point_des', 'pressure', 'irradiance']
params_pair = {'windows':    [],
               'feat_cols':  feat_cols,
               'aggs':       pw_dists + pw_kerns,
               'agg_func':   get_bivariate_dists_kerns,
               'scale':      False,
               'z_score':    True,
               'verbose':    True,
               'dataset':    'valid',
               'regenerate': True,
               'feat_name':  'pairwise_',
               'date_str':   '.2022.09.20',
               'save_and_download': False,
               'save_to_gdrive': False,
              }

train_df_pair, valid_df_pair, test_df_pair = get_features(train_df,
                                                          valid_df,
                                                          test_df,
                                                          params_pair)


##############################################################################
# approx 18 mins - added 88 features (0 NAs)
#
# pw_dists  = ['euclidean', 'canberra', 'cosine',   'hamming', 'sqeuclidean']
# pw_kerns  = ['linear',    'rbf', 'sigmoid', 'poly']
# feat_cols = ['humidity', 'dew.point_des', 'pressure']
# params = {'windows':    [],
#           'feat_cols':  feat_cols,
#           'aggs':       pw_dists + pw_kerns,
#           'agg_func':   get_bivariate_dists_kerns,
#           'scale':      False,
#           'z_score':    True,
#           'verbose':    True,
#           'dataset':    'valid',
#           'regenerate': True,
#           'feat_name':  'pairwise_',
#           'date_str':   '.2022.09.20', }
```

```
dataset: valid
y_des - humidity
         euclidean       3.13
         canberra        1.54
         cosine  2.24
         hamming         1.45
         sqeuclidean     1.46
         linear  1.46
         rbf     1.99
         sigmoid         1.36
         poly_2  1.29
         poly_3  1.59
y_des - dew.point_des
         euclidean       1.45
         canberra        1.48
         cosine  2.29
         hamming         1.52
         sqeuclidean     1.35
         linear  1.49
         rbf     1.71
         sigmoid         1.32
         poly_2  1.64
         poly_3  1.63
y_des - pressure
         euclidean       1.8
         canberra        1.3
         cosine  2.11
         hamming         1.32
         sqeuclidean     1.55
         linear  1.33
         rbf     2.04
         sigmoid         1.55
         poly_2  1.37
         poly_3  1.49
y_des - irradiance
         euclidean       1.38
         canberra        1.45
         cosine  1.99
         hamming         1.52
         sqeuclidean     1.4
         linear  1.47
         rbf     1.72
         sigmoid         1.35
         poly_2  1.46
         poly_3  1.31
humidity - dew.point_des
         euclidean       1.56
         canberra        1.36
         cosine  2.3
         hamming         1.44
         sqeuclidean     1.49
         linear  1.28
         rbf     1.75
         sigmoid         1.3
         poly_2  1.5
         poly_3  1.27
humidity - pressure
         euclidean       1.39
```

88 additional features were added in approximately 18 minutes (0 NAs).

Features above or between `dew.point_des` and `humidity` in the feature selection scores table may prove useful in future models. Some of the `dew.point_des_irradiance_zscore_*`, `y_des_humidity_zscore_*`, `y_des_irradiance_zscore_*` and `y_des_dew.point_des_zscore_*` features may prove useful. Other features may be useful for later forecast steps.

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.

TODO It may be worthwhile comparing the paired_distances function from sklearn against the pairwise_distances function used above. `paired_distances` computes the paired distances between X and Y and `pairwise_distances` compute the distance matrix from vector arrays X and Y.

TODO Compare zscore and min-max scaling. Potentially expand scaling usage to other feature sets.

TODO Explore applying standard transformations

- power transform
- Box-Cox, sigmoid, logit etc
- apply separately to each pair of features
    - a metric to compare quality of transformations would be worth searching for
    - normality, density of data points etc
- again, this could be expanded to other feature sets

Conclusion

The conclusion is separated into the following sections:

1. Feature sets summary
2. What worked
3. What didn't work
4. Rejected ideas
5. Future work

The commit history for

- gradient_boosting.ipynb
- tsfresh_feature_engineering.ipynb
- feature_engineering.ipynb

provides details for some of the points summarised below.

Note There are some TODOs scattered throughout the main text.

---

1. Feature Sets Summary

Summary of runtime and number of features in each feature set.

Feature set	Run time	Features
meteorology-based	0 mins	9
solar-based	2 mins	4
default	0 mins	109
roll_stats	1 min	111
cross_corr	1 min	29
catch22	12 mins	305
tsfeatures	1 hour 30 mins	246
intervals_etc	50 mins	40
bivariate	6 hours 15 mins	103
pairwise	18 mins	88
---	---	---
Total	9 hours 10 mins	1031

Linear interpolation was used to fill sequences of 12 (6 hours) or less consecutive NAs. Some features were removed due to longer sequences of NAs. 12 is a somewhat arbitrary cutoff. There are probably a few redundant features between feature sets.

So far, the meteorology-based features have not proven useful. This is a bit surprising. The solar-based calculated features (`irradiance`, `za_rad`, `azimuth_cos`) used as future covariates in the darts lightgbm notebook proved beneficial. The meteorology-based and solar-based features are included in the default feature set which also includes seasonal decompositions, wind related features and some other experiments. The meteorology-based and solar-based features are not available separately.

The `intervals_mean`, and to a lesser extent `pacf`, features from the tsfeatures package (intervals_etc feature set) were very useful in the darts lightgbm notebook. pacf-based features are not that surprising. Further investigation may be justified for the intervals-based features. It seems to be an indication of positive demand.

The `get_feature_selection_scores` function calculates F-statistic tests, Pearson correlation and mutual information between the calculated features and the target variable:

- these tests assume a linear model
- statistics calculated for single year of data
    - future work: consider averaging statistics across multiple years
- only for single step ahead target variable
    - future work: expand to multi-step target variable forecasts
- lagged features not considered
- intervals_mean features from tsfeatures were highly ranked
- what is the multiple testing corrected threshold value?

- features above or between `dew.point_des` and `humidity` in the feature selection scores table may prove useful in future models

TODO Compare `get_feature_selection_scores` feature rankings with permutation importance experiments ...

Unfortunately, when combined these features sets exceed the git LFS free usage tier. The default and cross_corr datasets are available on github repo because they are below the 50 MB limit. The other feature sets are stored on google drive, which is not publicly shared.

y des.- humidity

---

2. What worked:
    - switching to forecasting deseasonalised temperature instead of temperature
        - and adding yearly, daily and trend component to forecast
    - adding solar future covariate features
        - irradiance - beneficial
        - zenith angle - beneficial
        - azimuth angle - beneficial
    - plotting temperature, humidity, dew point and pressure in phase space
        - interactive plotly express scatter3d plots worked very well
        - still some obvious outliers present :-(
        - otherwise, these features look periodic and deterministic
        - nothing obviously chaotic, such as the lobe switching seen in the Lorenz attractor
        - last relevant commit before removal
    - missing data annotation
        - particularly `missing` feature for plots
        - and `mi_filled` (multiple imputation filled) feature for models
            - perhaps this indicates the multiple imputation is problematic
        - `hist_average` (historic average) also appeared in a few models
            - again this may indicate problems
        - this is less of an issue after most recent data cleaning
    - decomposing temperature and dew.point
        - into constant mean, daily seasonality, yearly seasonality plus residuals
        - prophet is overkill for this and too slow
            - could be replaced with some Fourier terms and scipy optimisation code
    - decomposing temperature and dew.point with UnobservedComponents
        - note that it supports additive seasonality only

        - this worked well but I didn't use it
            - decompositions were very similar to prophet decompositions
            - apart from a couple of high variance months at the start of decompositons
            - like prophet, pressure and humidity decompositions were questionable
        - it was probably fast enough but almost exceeded available RAM on Colab
            - **update** switching to method='powell' appears to fix this
        - see last relevant commit before removal
        - see also
            - Detrending, Stylized Facts and the Business Cycle
            - Seasonality in time series data
    - Fourier-based seasonal decomposition
        - *daily* autocorrelation plots comparing `y` with `y_des` and `dew.point` with `dew.point_des` indicate remaining seasonality
        - possibly some winter versus summer daily variation remains
            - neither prophet or statsmodels UCM multi-seasonal decompositions capture this seasonal variation
        - Fourier-based daily and yearly decomposition including phase and amplitude changes on averaged data managed to capture increased variation in summer but less so in winter
            - yielding the deseasonalised `y_des_fft` and `dew.point_des_fft` features
            - lightgbm models built with `y_des_fft` and `dew.point_des_fft` produced results which are statistically indistinguishable from models built with `y_des` and `dew.point_des`
            - Fourier-based decomposition was much faster than Prophet decomposition
            - despite the longer run time I decided to use only the Prophet decomposition due to it's relative simplicity
            - Fourier-based decomposition across the entire time series (non-averaged data) gave poor results
                - bad qq plot and other diagnostics
                - this approach also requires constantly updating the decomposition before forecasting to avoid data leakage
            - last relevant commit before removal
    - joblib and numba
        - joblib parallelisation reduced run time

        - numba jit compilation reduced run time
        - cacheing (memoization) experiments may prove worthwhile
    - apache parquet files
        - substantially reduced file size compared to compressed csv

```
pressure_irradian wasserstein_distance 0  99.12
```

---

3. What didn't work:
    - plotting Takens's embedding
        - Takens's theorem
        - no insight gained
        - didn't try to optimise delay setting
        - last relevant commit before removal
    - spline-based time components
    - higher frequency sinusoidal time components
    - phase-shifted sinusoidal time components
    - unsurprisingly, given the above, darts future cyclic encoders
        - last relevant commit
    - calculated meterology-based features
        - mixing ratio and absolute humidity - not beneficial
            - both have high correlation with dew point
                - absolute humidity 0.962
                - mixing ratio 0.96
        - vapour pressure - not beneficial
        - vapour pressure deficit - not beneficial
        - air density - not beneficial
        - water vapour concentration - not beneficial
        - specific humidity - not beneficial
        - potential temperature - not beneficial
    - smoothing `humidity` with simple moving average
    - tsfresh features
        - some subset of these features may be useful
        - tsfresh feature extraction was ran early on in the process
        - it is probably worthwhile re-examining these features
        - last commit before removal
        - see also:

        - tsfresh sub-section in Future Work section below
        - Conclusion in the tsfresh_feature_engineering.ipynb notebook
    - simple rolling statistical features
        - rolling min, max, mean, standard deviation, skew and kurtosis over 12 and 48 step windows
        - mercifully fast to compute though
    - mixup
        - darts TimeSeries requires `freq` be specified
        - won't work with irregularly time spaced mixup data augmentation
        - should work between years or days though
    - seasonal decomposition
        - `humidity`, `pressure`, `wind.x` and `wind.y`
            - using simple additive daily mean plus residuals
            - relative humidity, by definition, is right-censored at 100 %
            - still surprised I couldn't generate a decent `pressure` decomposition
            - wind is caused by pressure differences
        - `pressure`
            - EMD - empirical mode decomposition
                - Similar to the Fourier transform, empirical mode decomposition (EMD) can be used to decompose signals into a finite number of components
                - Empirical mode decomposition
                - EMD-signal package
                - Disappointing results for `pressure`
                - The components (IMFs or intrinsic mode functions) seem highly irregular
                - Perhaps it is necessary to average the annual measurements first or scale and center the data or tune some hyperparameter or there are tricks unknown to me
                    - See this paper: New insights and best practices for the successful use of Empirical Mode Decomposition, Iterative Filtering and derived algorithms
                - Wind speed may have been a better choice to decompose because it is nonstationary and nonlinear which EMD is designed to work with
                - last relevant commit before removal
            - SSA - singular spectral analysis

- Singular spectrum analysis (SSA) can help with the decomposition of time series
- Implemented in the pyts package
- Used the SingularSpectrumAnalysis function to decompose `pressure`
- Again, disappointing results for `pressure`
- The components (SSAs) seem highly irregular
- Perhaps it is necessary to average the annual measurements first or scale and center the data or tune some hyperparameter or there are tricks unknown to me
- last relevant commit before removal

- statsmodels STL and MSTL
  - The statsmodels package has two LOESS (locally estimated scatterplot smoothing) based functions to extract smooth estimates of three components (season, trend and residuals)
    - STL - Season-Trend decomposition using LOESS
    - MSTL - Season-Trend decomposition using LOESS for multiple seasonalities
  - first STL run and MSTL run
    - I tried seasonal decomposition with MSTL once and STL once
      - MSTL was horrendously slow due to the long annual seasonality (17,532)
    - Again, disappointing results for `pressure`
    - The trend and seasonal components seem irregular
    - Perhaps it is necessary to average the annual measurements first and/or more hyperparameter tuning is required
    - last relevant commit before removal of MSTL run and first STL run
  - second STL run
    - this notebook example shows using the trend to capture a cycle and the seasonality to capture a sinusoidal pattern with some amplitude change
    - I created an annual temperature average series but the trend was still irregualr
    - unfortunately no alpha smoothing parameter to tune and no window length parameter to tune
    - exponential smoothing did not help smooth the irregularity in the annual cycle

        - last relevant commit before removal
    - adding ground heat flux
        - may have high feature importance due to being co-linear with `y` target variable
        - for simplicity I ignored daily and yearly ground reservoir temperature variations
        - along with solar irradiance it is part of the radiative flux balance equation at the surface of the earth
        - see
            - surface energy balance
            - my ParametricWeatherModel repository
            - lecture notes summary including daily ground reservoir temperature variations
    - wind features
        - none of the wind features were beneficial
        - wind is caused by differences in atmospheric pressure
            - I presume this is a process that is downstream from temperature changes
        - autocorrelation plots indicated
            - relatively weak short term seasonality
            - almost no long term seasonality - similar to `pressure`
            - `wind.speed.max` has slightly stronger seasonality than `wind.speed.mean`
            - wind `x` components had stronger seasonality than the `y` components
    - declination angle
        - from solarpy
        - not useful in lightgbm models
    - adding climate indices from NOAA
        - North atlantic oscillation - `nao-monthly` monthly and `nao-daily` daily
        - Multivariate ENSO Index (`mei`)
        - neither feature proved beneficial
        - complex non-linear interactions between climate indices
        - if one or more are leading indicators then leading by how much?
            - almost certainly varys
        - last commit before removal
    - multi-dimensional similarity between years
        - calculated using both Procrustes and Hausdorff methods
        - also included color-coded rmse between annual temperature distributions
        - worked without problem but not that insightful