```
%matplotlib inline

import datetime
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.tsa.api import VAR
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing
from statsmodels.tools.eval_measures import rmse, medianabs


# Reduces variance in results but won't eliminate it :-(
%env PYTHONHASHSEED=0
import random
random.seed(42)
np.random.seed(42)
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is deprecated
  import pandas.util.testing as tm
env: PYTHONHASHSEED=0
```

Baseline Forecasts for Cambridge UK Weather Time Series

Building baseline models for time series analysis of Cambridge UK temperature measurements taken at the University computer lab weather station.

I'm primarily interested in short term temperature forecasts (less than 2 hours) but will include results up to 24 hours in the future.

See my previous work for further details:

- 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

Import Data

The measurements are relatively noisy and there are usually several hundred missing values every year. 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 April 2021 and occur every 30 mins.

```
if 'google.colab' in str(get_ipython()):
    data_loc = "https://github.com/makeyourownmaker/CambridgeTemperatureNotebooks/blob/main/data/CamMetCleanish2021.04.26.cs
else:
    data_loc = "../data/CamMetCleanish2021.04.26.csv"
df = pd.read_csv(data_loc, parse_dates = True)

df['ds'] = pd.to_datetime(df['ds'])
df['y'] = df['y'] / 10
df['wind.speed.mean'] = df['wind.speed.mean'] / 10

df = df.loc[df['ds'] > '2008-08-01 00:00:00',]

print("Shape:")
print(df.shape)
print("\nInfo:")
print(df.info())
print("\nSummary stats:")
display(df.describe())
print("\nRaw data:")
display(df)
print("\n")


def plot_examples(data, x_var):
    """Plot 9 sets of observations in 3 * 3 matrix ..."""

    assert len(data) == 9

    cols = [col for col in data[0].columns if col != x_var]

    fig, axs = plt.subplots(3, 3, figsize = (15, 10))
    axs = axs.ravel()  # apl for the win :-)

    for i in range(9):
```

```
      for col in cols:
        axs[i].plot(data[i][x_var], data[i][col])
        axs[i].xaxis.set_tick_params(rotation = 20, labelsize = 10)

    fig.legend(cols, loc = 'upper center',  ncol = len(cols))

    return None


cols = ['ds', 'y', 'humidity', 'dew.point', 'pressure',
        'wind.speed.mean', 'wind.bearing.mean']
plots  = 9
window = 24
starts = [random.randint(0, np.floor(df.shape[0] / window)) for _ in range(plots)]
p_data = [df.loc[starts[i] * window:starts[i] * window + window, cols]
          for i in range(plots)]
plot_examples(p_data, 'ds')
```

```
Shape:
(225251, 7)

Info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 225251 entries, 2 to 225252
Data columns (total 7 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   ds                 225251 non-null  datetime64[ns]
 1   y                  225251 non-null  float64
 2   humidity           225251 non-null  float64
 3   dew.point          225251 non-null  float64
 4   pressure           225251 non-null  float64
 5   wind.speed.mean    225251 non-null  float64
 6   wind.bearing.mean  225251 non-null  float64
dtypes: datetime64[ns](1), float64(6)
memory usage: 13.7 MB
None
```

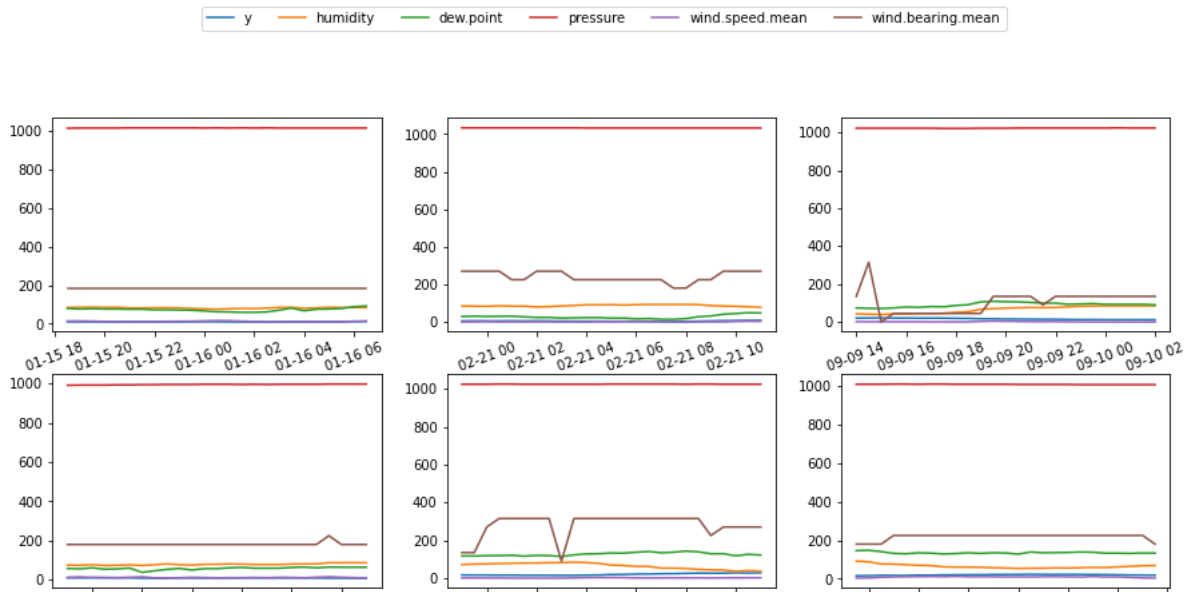
Summary stats:

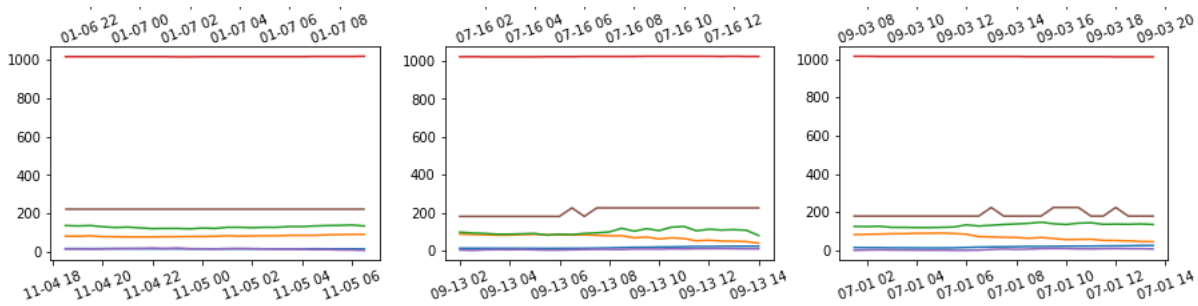
	y	humidity	dew.point	pressure	wind.speed.mean	wind.bearing.mean
count	225251.000000	225251.000000	225251.000000	225251.000000	225251.000000	225251.000000
mean	10.027882	78.619532	58.600183	1014.342062	4.434737	194.974558
std	6.509969	17.308646	51.645273	11.911991	4.011087	82.876095
min	-7.000000	20.000000	-100.000000	963.000000	0.000000	0.000000
25%	5.200000	68.000000	19.000000	1008.000000	1.200000	135.000000
50%	9.600000	83.000000	59.100000	1016.000000	3.500000	225.000000
75%	14.500000	92.000000	97.000000	1022.000000	6.600000	270.000000
max	36.100000	100.000000	209.000000	1051.000000	29.200000	360.000000

Raw data:

	ds	y	humidity	dew.point	pressure	wind.speed.mean	wind.bearing.mean
2	2008-08-01 00:30:00	19.5	65.75000	119.150000	1014.416667	1.150000	225.0
3	2008-08-01 01:00:00	19.1	49.75000	79.200000	1014.384615	1.461538	225.0
4	2008-08-01 01:30:00	19.1	66.17875	106.600000	1014.500000	1.508333	225.0
5	2008-08-01 02:00:00	19.1	58.50000	99.250000	1014.076923	1.430769	225.0
6	2008-08-01 02:30:00	19.1	66.95000	121.883333	1014.416667	1.133333	225.0
...	...	...	...	...	...	...	...
225248	2021-04-25 23:00:00	3.6	61.00000	-32.000000	1028.000000	1.400000	45.0
225249	2021-04-25 23:30:00	3.6	64.00000	-26.000000	1028.000000	2.600000	45.0
225250	2021-04-26 00:00:00	3.6	58.00000	-39.000000	1028.000000	4.300000	45.0
225251	2021-04-26 00:30:00	3.2	62.00000	-34.000000	1027.000000	5.400000	45.0
225252	2021-04-26 01:00:00	3.2	62.00000	-34.000000	1027.000000	4.200000	45.0

225251 rows × 7 columns





Data Processing and Feature Engineering

The data must be reformatted before model building.

The following steps are carried out:

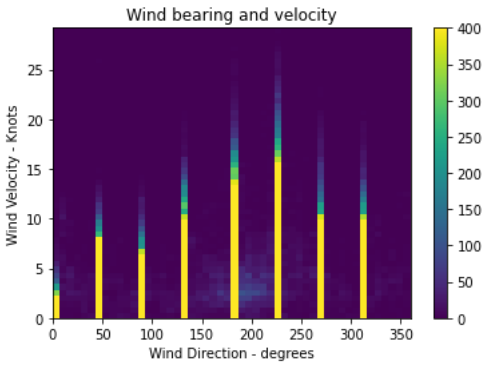
- Wind direction and speed transformation
- Time conversion
- Train test data separation
- Data windowing
    - Some baseline methods operate on initial training windows of data

Wind direction and speed transformation

The `wind.bearing.mean` column gives wind direction in degrees but is categorised at 45 degree increments, i.e. 0, 45, 90, 135, 180, 225, 270, 315. Wind direction shouldn't matter if the wind is not blowing.

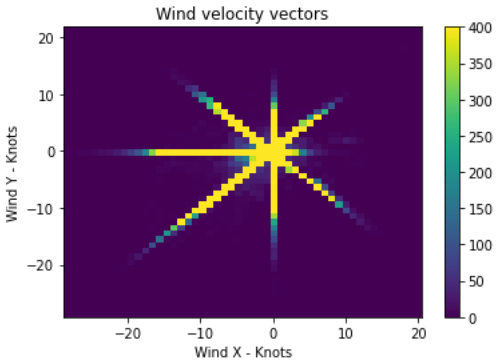
The distribution of wind direction and speed looks like this:

```
plt.hist2d(df['wind.bearing.mean'], df['wind.speed.mean'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind Direction - degrees')
plt.ylabel('Wind Velocity - Knots')
plt.title('Wind bearing and velocity');
```



Convert wind direction and speed to x and y vectors, so the model can more easily interpret them.

```
wv = df['wind.speed.mean']

# Convert to radians
wd_rad = df['wind.bearing.mean'] * np.pi / 180

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

df_orig = df

plt.hist2d(df['wind.x'], df['wind.y'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind X - Knots')
plt.ylabel('Wind Y - Knots')
plt.title('Wind velocity vectors');
```



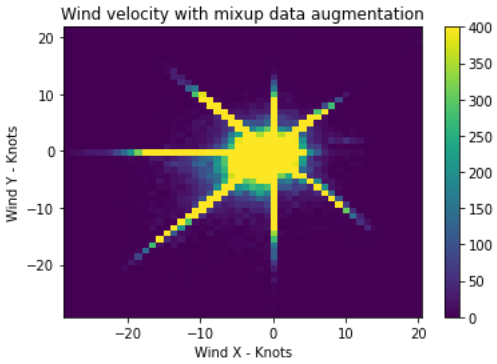
Better, but not ideal. Data augmentation with the mixup method is carried out below.

From the mixup paper: "mixup trains a neural network on convex combinations of pairs of examples and their labels".

Further details on how I apply mixup to time series are included in the Window data section of my keras_mlp_fcn_resnet_time_series.ipynb notebook.

Here is an illustration of the improvement in wind velocity sparsity with mixup augmentation.

```
def mixup(data, alpha = 1.0, factor = 1):
    batch_size = len(data) - 1

    data['epoch'] = data.index.astype(np.int64) // 10**9

    # random sample lambda value from beta distribution
    l   = np.random.beta(alpha, alpha, batch_size * factor)
    X_l = l.reshape(batch_size * factor, 1)

    # Get a pair of inputs and outputs
    y1  = data['y'].shift(-1).dropna()
    y1_ = pd.concat([y1] * factor)

    y2  = data['y'][0:batch_size]
    y2_ = pd.concat([y2] * factor)

    X1  = data.drop('y', 1).shift(-1).dropna()
    X1_ = pd.concat([X1] * factor)

    X2  = data.drop('y', 1)
    X2  = X2[0:batch_size]
    X2_ = pd.concat([X2] * factor)

    # Perform mixup
    X = X1_ * X_l + X2_ * (1 - X_l)
    y = y1_ * l   + y2_ * (1 - l)

    df = pd.DataFrame(y).join(X)
    df = data.append(df).sort_values('epoch', ascending = True)
    df = df.drop('epoch', 1)

    df = df.drop_duplicates(keep = False)

    return df


df_mix = mixup(df.loc[:, ['y','wind.x','wind.y']], factor = 2)
plt.hist2d(df_mix['wind.x'], df_mix['wind.y'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind X - Knots')
plt.ylabel('Wind Y - Knots')
plt.title('Wind velocity with mixup data augmentation');
```



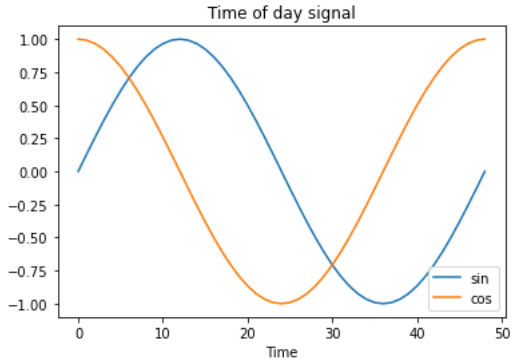
Mixup improves the categorical legacy of the wind velocity data. Unfortunately, if outliers are present their influence will likely be reinforced.

For now, I'm not using the wind vector or mixup augmentation with any of the baseline methods. See my keras_mlp_fcn_resnet_time_series.ipynb notebook for an illustration of the profoundly beneficial improvement mixup provides with this dataset.

Time conversion

Convert `ds` timestamps to "time of day" and "time of year" variables using `sin` and `cos` functions.

```
# Convert to secs
date_time   = pd.to_datetime(df['ds'], format = '%Y.%m.%d %H:%M:%S')
timestamp_s = date_time.map(datetime.datetime.timestamp)

day  = 24 * 60 * 60
year = (365.2425) * day

df['day.sin']  = np.sin(timestamp_s * (2 * np.pi / day))
df['day.cos']  = np.cos(timestamp_s * (2 * np.pi / day))
df['year.sin'] = np.sin(timestamp_s * (2 * np.pi / year))
df['year.cos'] = np.cos(timestamp_s * (2 * np.pi / year))

plt.plot(np.array(df['day.sin'])[49:98])
plt.plot(np.array(df['day.cos'])[49:98])
plt.xlabel('Time')
plt.legend(['sin', 'cos'], loc = 'lower right')
plt.title('Time of day signal');
```



The yearly time components may benefit from a single phase shift so they align with the seasonal temperature peak around the end of July and temperature trough around the end of January. Similarly, the daily components may benefit from small daily phase shifts.

I use these time components with the multi-variate VAR method.

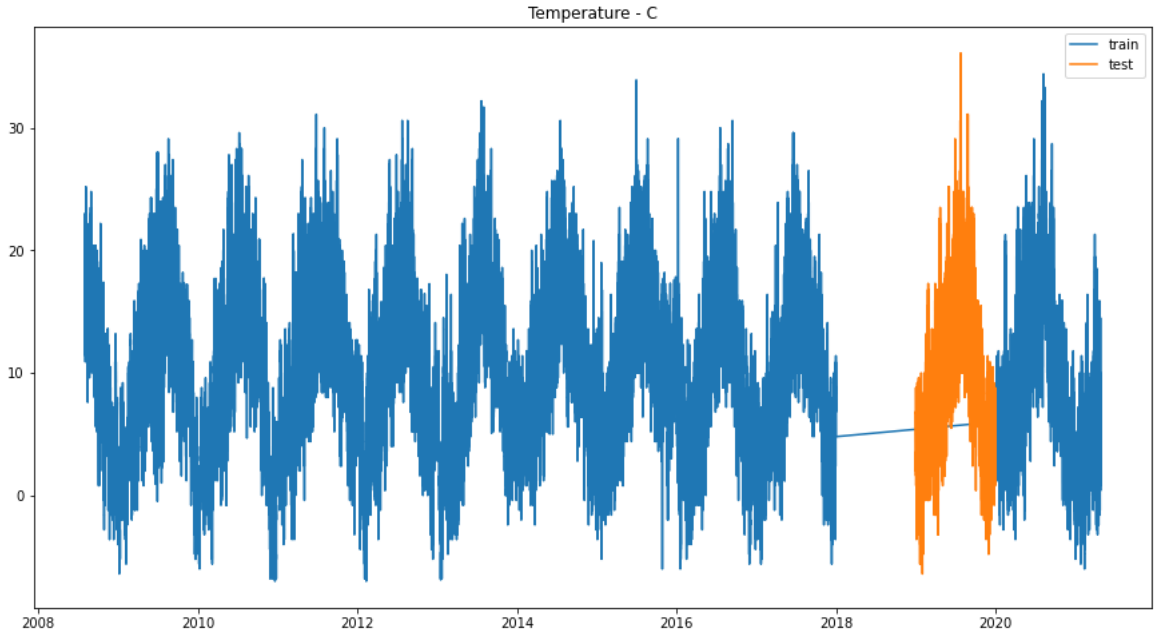
Split data¶

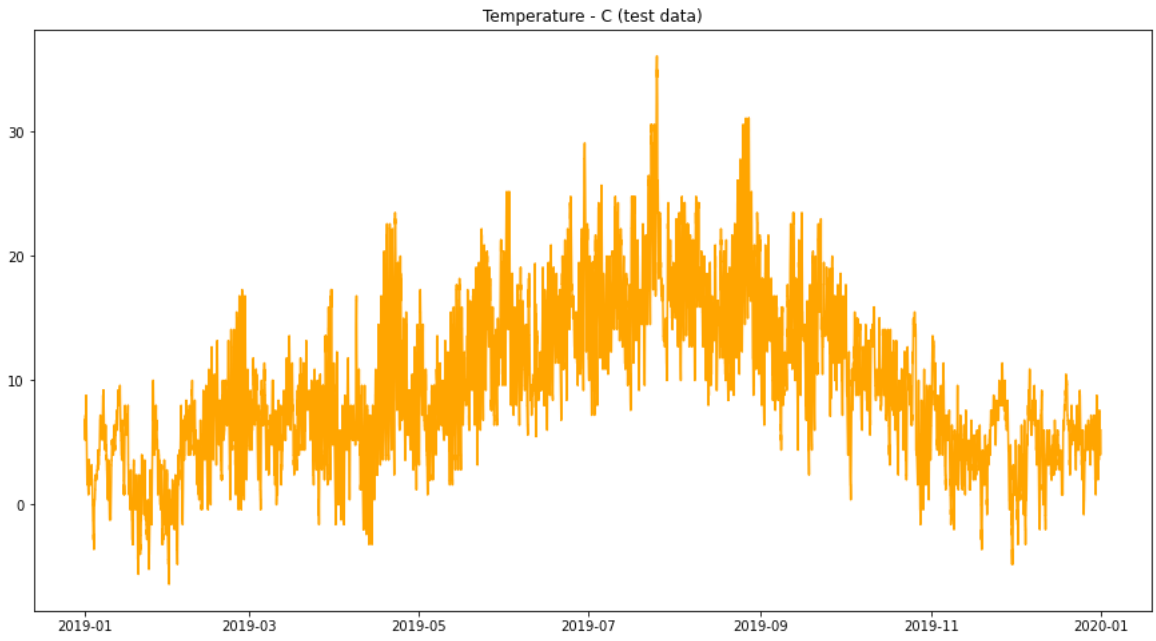
I use data from 2018 for validation, 2019 for testing and the remaining data for training. These are entirely arbitrary choices. This results in an approximate 84%, 8%, 8% split for the training, validation, and test sets respectively.

```
keep_cols = ['y', 'humidity', 'dew.point', 'pressure', 'wind.x', 'wind.y',
             'day.sin', 'day.cos', 'year.sin', 'year.cos', 'ds']

df['year'] = df['ds'].dt.year
train_df = df.loc[(df['year'] != 2018) & (df['year'] != 2019)]
valid_df = df.loc[df['year'] == 2018]
test_df  = df.loc[df['year'] == 2019]
```

```
train_df = train_df.drop('year', axis = 1)  # inplace = True gives SettingWithCopyWarning
valid_df = valid_df.drop('year', axis = 1)  # ...
test_df  = test_df.drop('year',  axis = 1)
df       = df.drop('year',       axis = 1)

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

plt.figure(figsize = (15, 8))
plt.plot(train_df.ds, train_df.y)
plt.plot(test_df.ds,   test_df.y)
plt.title('Temperature - C')
plt.legend(['train', 'test'])
plt.show()

plt.figure(figsize = (15, 8))
plt.plot(test_df.ds, test_df.y, color='orange')
plt.title('Temperature - C (test data)')
plt.show();
```

```
df.drop shape:  (225251, 13)
train shape:    (190023, 13)
valid shape:    (17655, 13)
test shape:     (17573, 13)
```





The training data is used to calculate the seasonal average values.

Baselines Methods

This notebook is being developed on Google Colab primarily with the statsmodels package.

I will not use complicated methods just for comparison of baseline methods. Also, I don't want to spend a great deal of time optimising hyperparameter settings. This means I will use simpler methods and default settings as far as possible.

Model diagnostics will not be checked. Some of the models built may have issues.

Runtime may become an issue for complicated methods.

Univariate baseline methods:

- mean - simple average
- naive - persistent
    - commonly used in the meteorology literature
- seasonal average - historical average
- SES - Simple Exponential Smoothing
- HWES - Holt Winter's Exponential Smoothing

I would also like to look at results from, the AutoReg and STL functions but they are unavailable in the current version of statsmodels (0.10.2) installed on Google Colab (as of 28/04/21). The alternative AR function is depracated. ARIMA models would be the next logical choice but often require model checking and tuning. Installing packages on Google Colab is straight-forward. Upgrading packages may also be straight-forward, but it's probably more productive to move onto multi-variate baselines instead.

Multivariate baseline methods:

- VAR - Vector AutoRegression

Forecast horizons:

- next 30 mins - 1 step ahead
- next 2 hours - 4 steps ahead
- next 24 hours - 48 steps ahead
- horizon is abbreviated to `h` in the interrim results tables

Some forecast methods are limited to single step forecasts. For example, simple average and seasonal average.

Metrics considered:

- rmse - root mean squared error
- mae - median absolute error

Univariate Baseline Methods

Firstly, the mean baseline:

```
# sa - simple average
sa = train_df.y.mean()
rmse_sa = rmse(test_df.y, sa)
mae_sa  = medianabs(test_df.y, sa)

print("simple average:", round(sa, 2))
print("rmse:", round(rmse_sa, 2))
print("mae: ", round(mae_sa,  2))

cols = ['type', 'method', 'metric', 'horizon', 'value']
metrics_h = []  # h for horizon

for i in range(1, 49):
  metrics_h.append(dict(zip(cols, ['univariate', 'mean', 'rmse', i, rmse_sa])))
  metrics_h.append(dict(zip(cols, ['univariate', 'mean',  'mae', i,  mae_sa])))

metrics = pd.DataFrame(metrics_h, columns = cols)
```

```
    simple average: 10.09
    rmse: 6.42
    mae:  4.89
```

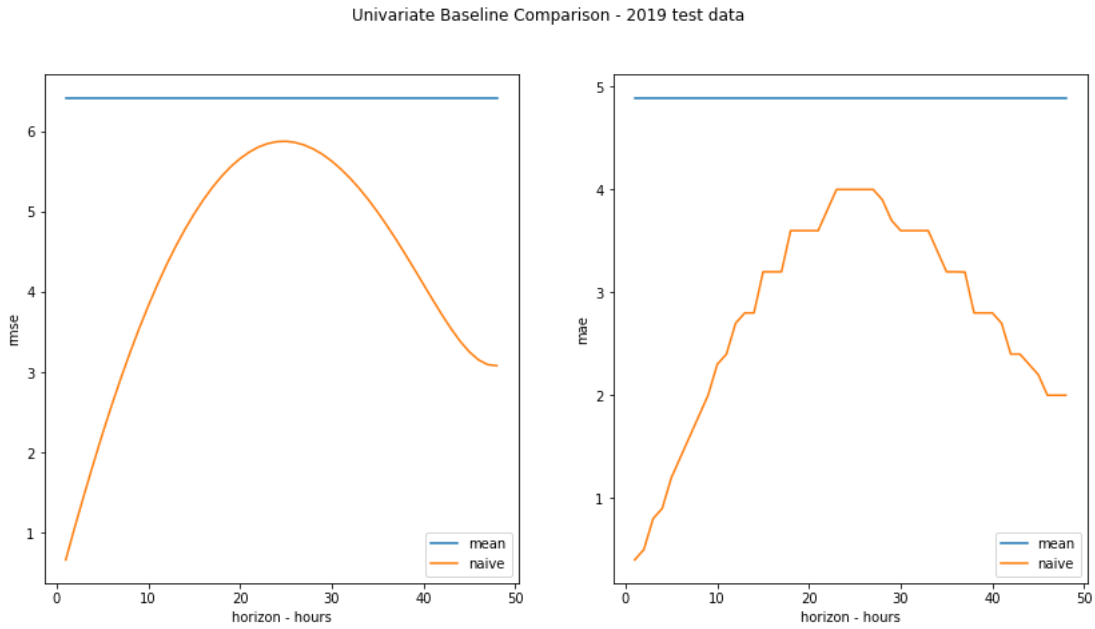
The mean baseline has equivalent metric values across all forecast horizons.

Secondly, the naive baseline:

```
def get_naive_metrics(data, horizon, method):
  y_hat = data.copy()
  y_hat['naive'] = y_hat.y.shift(horizon)
```

```
  y_hat = y_hat.iloc[horizon:]  # remove na value
  naive_rmse = rmse(y_hat.y, y_hat.naive)
  naive_mae  = medianabs(y_hat.y, y_hat.naive)

  return [naive_rmse, naive_mae]


def plot_metrics(metrics, main_title):
  fig, axs = plt.subplots(1, 2, figsize = (14, 7))
  fig.suptitle(main_title)
  axs = axs.ravel()  # APL ftw!

  methods = metrics.method.unique()

  for method in methods:
    df = metrics.query('metric == "rmse" & method == "%s"' % method)
    axs[0].plot(df.horizon, df.value)
  axs[0].set_xlabel("horizon - hours")
  axs[0].set_ylabel("rmse")
  axs[0].legend(methods)

  for method in methods:
    df = metrics.query('metric == "mae" & method == "%s"' % method)
    axs[1].plot(df.horizon, df.value)
  axs[1].set_xlabel("horizon - hours")
  axs[1].set_ylabel("mae")
  axs[1].legend(methods);


def update_metrics(metrics, test_data, method, get_metrics, model=None):
  metrics_h = []

  if method in ['SES', 'HWES']:
    horizons = [i for i in range(4, 49, 4)]
    horizons.insert(0, 1)
  else:
    horizons = range(1, 49)

  if method in ['VAR']:
    variates = 'multivariate'
  else:
    variates = 'univariate'

  print("h\trmse\tmae")
  for h in horizons:
    if method in ['VAR']:
      rmse_h, mae_h = get_metrics(test_df, h, method, model)
    else:
      rmse_h, mae_h = get_metrics(test_df, h, method)

    metrics_h.append(dict(zip(cols, [variates, method, 'rmse', h, rmse_h])))
    metrics_h.append(dict(zip(cols, [variates, method,  'mae', h,  mae_h])))

    if h in [1, 4, 48]:
      print(h, "\t", round(rmse_h, 2), "\t", round(mae_h, 2))

  print("\n")

  metrics_method = pd.DataFrame(metrics_h, columns = cols)
  metrics = metrics.append(metrics_method)

  return metrics


metrics = update_metrics(metrics, test_df, 'naive', get_naive_metrics)
plot_metrics(metrics, "Univariate Baseline Comparison - 2019 test data")
```

```
h       rmse    mae
1        0.66    0.4
4        1.86    0.9
48       3.08    2.0
```



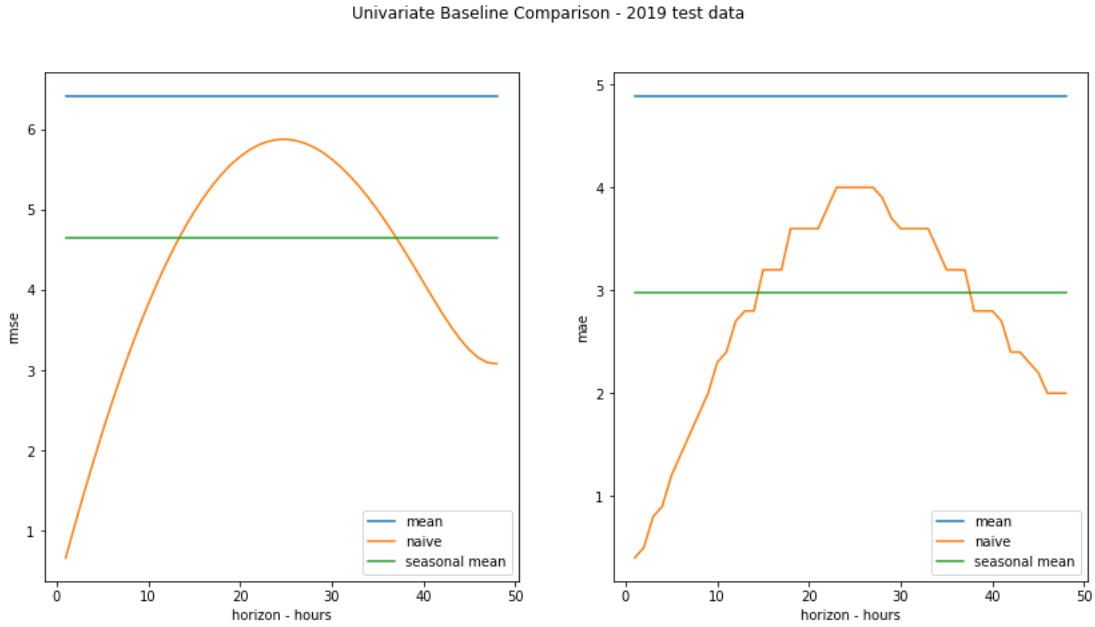
Obviously, the naive method is an improvement over the simple average.

The daily seasonality is clearly evident in the naive method.

Third, seasonal average:

```
train_df['doy'] = train_df.ds.dt.strftime('%j')
train_df['hms'] = train_df.ds.dt.strftime('%H:%M:%S')

# ha - historical average
ha = pd.DataFrame(train_df.groupby(['doy', 'hms']).y.mean())
ha.reset_index(inplace=True)

rmse_ha = rmse(test_df.y[:17568],      ha.y)
mae_ha  = medianabs(test_df.y[:17568], ha.y)

print("rmse:", round(rmse_ha, 2))
print("mae: ", round(mae_ha,  2))

metrics_h = []  # h for horizon

for h in range(1, 49):
  metrics_h.append(dict(zip(cols, ['univariate', 'seasonal mean', 'rmse', h, rmse_ha])))
  metrics_h.append(dict(zip(cols, ['univariate', 'seasonal mean',  'mae', h,  mae_ha])))

metrics_ha = pd.DataFrame(metrics_h, columns = cols)
metrics = metrics.append(metrics_ha)

plot_metrics(metrics, "Univariate Baseline Comparison - 2019 test data")
```

```
rmse: 4.65
mae:  2.97
```



The historical average, or seasonal average, is superior to the simple average. It also illustrates how the naive method is struggling with the daily seasonality component.

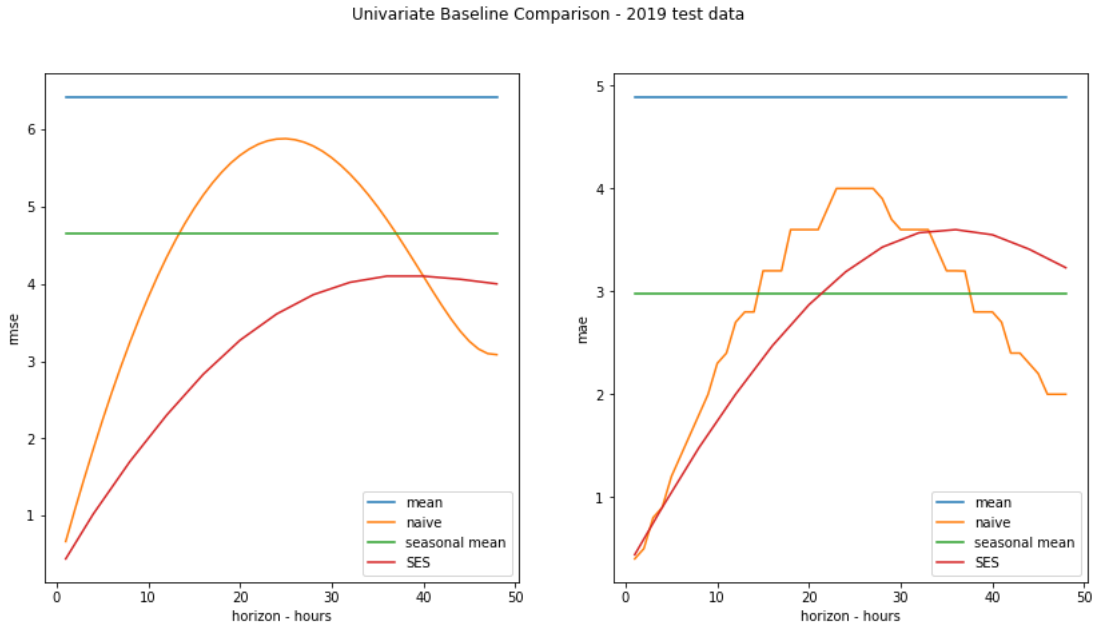
Fourth, simple exponential smoothing:

- Run time is increasing
  - I only run every 4th horizon value for SES and all subsequent methods
  - SES takes approx. 15 mins to complete
- Default settings for `SimpleExpSmoothing` function
  - See [statmodels SimpleExpSmoothing documentation] for details
- Using 48 temperature measurements for the initial training window

```
%%time

def es_rolling_cv(data, horizon, method):
    i =  0
    x = 48  # initial training window
    h = horizon
    rmse_roll, mae_roll = [], []

    while (i + x + h) < len(data):
        train_ts = data[(i):(i + x)].y.values
        test_ts  = data[(i + x):(i + x + h)].y.values
        # print("train:", train_ts, "\ntest:", test_ts)  # DEBUG

        if method == 'SES':
          model_roll = SimpleExpSmoothing(train_ts).fit()
        elif method == 'HWES':
          model_roll = ExponentialSmoothing(train_ts,
                                            seasonal_periods = 48,
                                            seasonal = 'add',
                                            ).fit()

        y_hat = model_roll.forecast(h)
        # print("yhat:", y_hat)  # DEBUG

        rmse_i = rmse(test_ts, y_hat)
        mae_i  = medianabs(test_ts, y_hat)
        # print("rmse:", rmse_i, "\nmae:", mae_i, "\n")  # DEBUG
        rmse_roll.append(rmse_i)
        mae_roll.append(mae_i)

        i = i + 1

    return [np.mean(rmse_roll).round(2), np.mean(mae_roll).round(2)]


metrics = update_metrics(metrics, test_df, 'SES', es_rolling_cv)
plot_metrics(metrics, "Univariate Baseline Comparison - 2019 test data")
```

h	rmse	mae
1	0.44	0.44
4	1.02	0.9
48	4.0	3.23

```
CPU times: user 15min 25s, sys: 47 s, total: 16min 12s
Wall time: 15min 1s
```



Clearly, the SES method is an improvement over the naive method for forecast horizons less than approx. 18 hours.

It may be possible to optimise the SES method by setting initial training window length `train_window` and/or the `initialization_method` parameter. See [statsmodels docs] for `initialization_method` options.

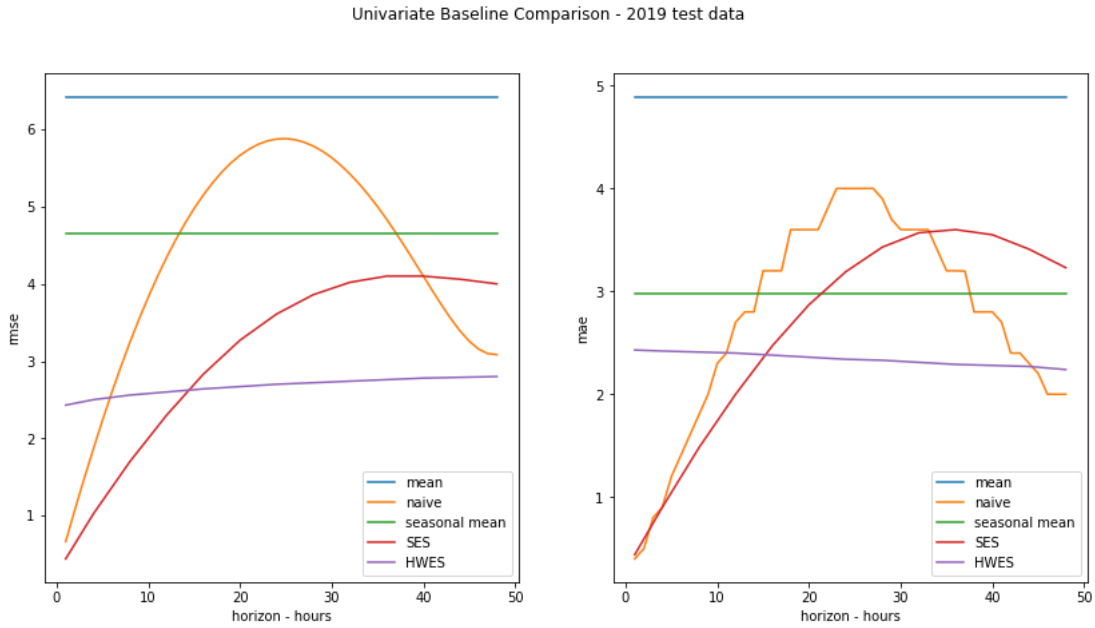
Fifth, Holt-Winters exponential smoothing:

- Run time is getting quite long
    - I run every 4th horizon value

    - Still takes approx. 30 mins
- `seasonal_periods` is set to 48 (24 half-hourly observations)
    - yearly seasonality is not accounted for
    - `ExponentialSmoothing` function does not support multiple seasonalities
- `seasonal` is set to additive
- Temperature measurements are mostly cyclic with little or no trend across the years
    - `trend` and `damped_trend` are set to default values of `None`
- See [statmodels ExponentialSmoothing documentation] for details

```
%%time

metrics = update_metrics(metrics, test_df, 'HWES', es_rolling_cv)
plot_metrics(metrics, "Univariate Baseline Comparison - 2019 test data")
```

```
    h       rmse    mae
    /usr/local/lib/python3.7/dist-packages/statsmodels/tsa/holtwinters.py:924: RuntimeWarning: divide by zero encountered in
      aic = self.nobs * np.log(sse / self.nobs) + k * 2
    /usr/local/lib/python3.7/dist-packages/statsmodels/tsa/holtwinters.py:929: RuntimeWarning: invalid value encountered in d
      aicc = aic + aicc_penalty
    /usr/local/lib/python3.7/dist-packages/statsmodels/tsa/holtwinters.py:930: RuntimeWarning: divide by zero encountered in
      bic = self.nobs * np.log(sse / self.nobs) + k * np.log(self.nobs)
    1       2.43    2.43
    4       2.5     2.42
    48      2.8     2.24


    CPU times: user 30min 31s, sys: 1min 13s, total: 31min 44s
    Wall time: 30min 15s
```



On the whole, HWES is an improvement over the other methods even if short horizon forecasts are worse than the naive and SES methods.

The daily cyclic component present in the naive and SES results is not an issue with the HWES method as would be expected.

It's puzzling that the mean absolute error decreases slightly as the horizon increases. Adding error bars may partially address this issue.

It's also worth noting:

- There are convergence warnings if the initial training window is not equal to the seaonality period of 48
- Opportunities for HWES parameter tuning include the `use_boxcox` and `initialization_method` options

---

Multivariate Baseline Methods

Considering two models:

- VAR - Vector Auto-regression

VAR models capture the linear interdependencies among multiple time series.

Firstly, VAR:

- treating temperature (`y`), `humidity`, `dew.point` and `pressure` as endogenous varaibles
- not using `wind.x` or `wind.y` in model

    - Granger causality tests suggest poor performance for `wind.x` and `wind.y` measurements
    - mixup data augmentation may give a different result
- using daily and yearly sinusoidal time variables as exogenous data
    - `day.cos`, `day.sin`, `year.cos`, `year.sin`
    - not shown in this notebook but VAR model without these exogenous variables performs worse; particularly for longer horizon forecasts
- limiting order of lagged variables to 12
- only calculating rmse and mae for `y`
    - not looking at forecast performance for other endogenous variables at this time
- approx. 1 hour run time
- See [statmodels VAR documentation] for details

```
from statsmodels.tsa.stattools import grangercausalitytests

maxlag = 12
test   = 'ssr_chi2test'

def grangers_causation_matrix(data, variables, test='ssr_chi2test', verbose=False):
    """Check Granger Causality of all possible combinations of the Time series.
    The rows are the response variable, columns are predictors.  The values in the table
    are the P-Values.  P-Values lesser than the significance level (0.05), implies
    the Null Hypothesis that the coefficients of the corresponding past values is
    zero, that is, the X does not cause Y can be rejected.

    data      : pandas dataframe containing the time series variables
    variables : list containing names of the time series variables.

    From https://www.machinelearningplus.com/time-series/vector-autoregression-examples-python/
    """

    df = pd.DataFrame(np.zeros((len(variables), len(variables))), columns=variables, index=variables)

    for c in df.columns:
        for r in df.index:
            test_result = grangercausalitytests(data[[r, c]], maxlag=maxlag, verbose=False)
            p_values = [round(test_result[i + 1][0][test][1], 4) for i in range(maxlag)]

            if verbose: print(f'Y = {r}, X = {c}, P Values = {p_values}')

            min_p_value  = np.min(p_values)
            df.loc[r, c] = min_p_value

    df.columns = [var + '_x' for var in variables]
    df.index   = [var + '_y' for var in variables]

    return df


vars = ['y', 'humidity', 'dew.point', 'pressure'] # ,'wind.x', 'wind.y']

gcm = grangers_causation_matrix(test_df, vars)
display(gcm)
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/base/model.py:1752: ValueWarning: covariance of constraints does not h
  'rank is %d' % (J, J_), ValueWarning)
```

	y_x	humidity_x	dew.point_x	pressure_x
y_y	1.0	0.0000	0.0	0.0
humidity_y	0.0	1.0000	0.0	0.0
dew.point_y	0.0	0.0000	1.0	0.0
pressure_y	0.0	0.0017	0.0	1.0

WARNING: The function used to check Granger causality does not take multiple testing problems into account - ignore the *questionable* documentation regarding significance tests.

Onwards to model building:

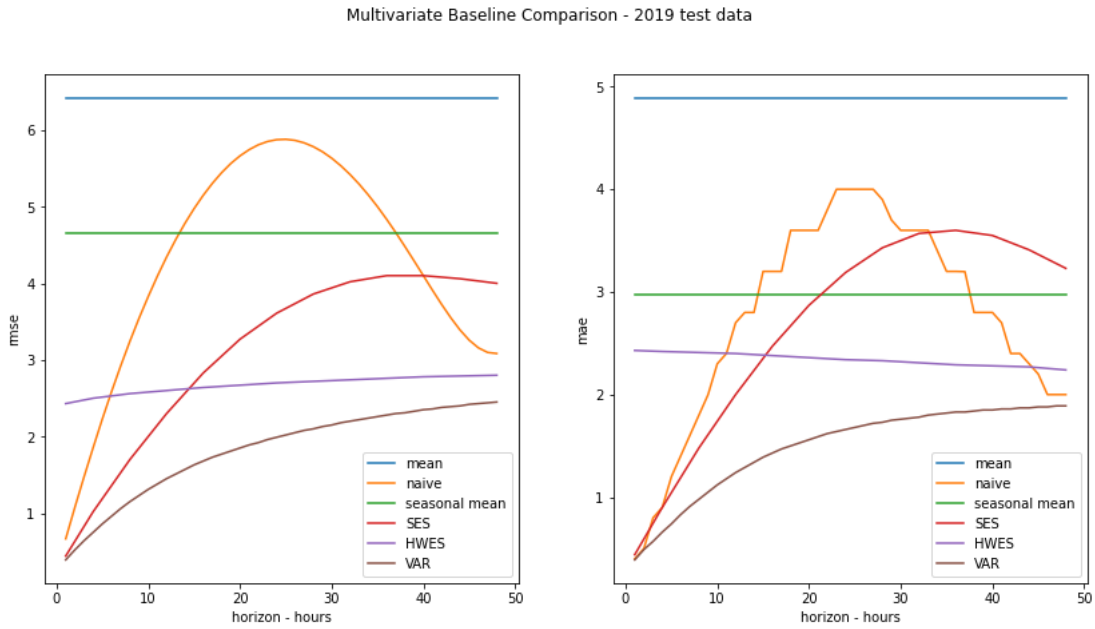
```
%%time

# build model on train_df
endo_vars = ['y', 'humidity', 'dew.point', 'pressure'] # better model without wind.x & wind.y
```

```
exog_vars = ['day.cos', 'day.sin', 'year.cos', 'year.sin']
endo_df = train_df[endo_vars]
exog_df = train_df[exog_vars]

var_model = VAR(endog = endo_df, exog = exog_df)
var_fit   = var_model.fit(maxlags = 12, ic = 'aic')
print(var_fit.summary())


# rolling_cv with pre-trained model
def var_rolling_cv(data, horizon, method, model):
    lags = model.k_ar  # lag order
    i = lags
    h = horizon
    rmse_roll, mae_roll = [], []
    endo_vars = ['y', 'humidity', 'dew.point', 'pressure']
    exog_vars = ['day.cos', 'day.sin', 'year.cos', 'year.sin']
    #print("\ti:", i, "\th:", h, "\tlags:", lags)  # DEBUG

    while (i + h) < len(data):
        test_df = data[endo_vars].iloc[i:(i + h)]
        endo_df = data[endo_vars].iloc[(i - lags):i].values
        exog_df = data[exog_vars].iloc[i:(i + h)]
        #print("\tendo:", endo_df.shape, "\texog:", exog_df.shape)  # DEBUG
        # display(endo_df)  # DEBUG

        y_hat = model.forecast(endo_df, exog_future = exog_df, steps = h)
        #print("yhat:", y_hat)  # DEBUG

        preds = pd.DataFrame(y_hat, columns = endo_vars)

        rmse_i = rmse(test_df.y,      preds.y)
        mae_i  = medianabs(test_df.y, preds.y)
        # print("rmse:", rmse_i, "\nmae:", mae_i, "\n")  # DEBUG
        rmse_roll.append(rmse_i)
        mae_roll.append(mae_i)

        i = i + 1

    return [np.mean(rmse_roll).round(2), np.mean(mae_roll).round(2)]


metrics = update_metrics(metrics, test_df, 'VAR', var_rolling_cv, var_fit)
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tsa/base/tsa_model.py:215: ValueWarning: An unsupported index was prov
  ' ignored when e.g. forecasting.', ValueWarning)
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Fri, 10, Sep, 2021
Time:                     17:18:26
--------------------------------------------------------------------
No. of Equations:         4.00000    BIC:                    4.67359
Nobs:                     190011.    HQIC:                   4.66560
Log likelihood:      -1.52118e+06    FPE:                    105.875
AIC:                      4.66226    Det(Omega_mle):         105.757
--------------------------------------------------------------------
Results for equation y
================================================================================
                    coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------------
const                -1.544437         0.137736          -11.213           0.000
exog0                -0.226294         0.003149          -71.873           0.000
exog1                 0.134527         0.002785           48.302           0.000
exog2                -0.153552         0.003575          -42.951           0.000
exog3                -0.024625         0.002600           -9.470           0.000
L1.y                  0.977004         0.003159          309.282           0.000
L1.humidity          -0.009401         0.000580          -16.214           0.000
L1.dew.point          0.004889         0.000260           18.836           0.000
L1.pressure          -0.018217         0.002418           -7.534           0.000
L2.y                  0.040475         0.004089            9.899           0.000
L2.humidity           0.010012         0.000692           14.465           0.000
L2.dew.point         -0.002872         0.000313           -9.170           0.000
L2.pressure           0.014847         0.003250            4.568           0.000
L3.y                  0.017295         0.004115            4.203           0.000
L3.humidity          -0.001800         0.000701           -2.569           0.010
L3.dew.point          0.001771         0.000316            5.596           0.000
L3.pressure           0.012485         0.003274            3.814           0.000
L4.y                  0.004718         0.004116            1.146           0.252
L4.humidity           0.004475         0.000701            6.384           0.000
L4.dew.point         -0.001763         0.000317           -5.570           0.000
L4.pressure          -0.003827         0.003275           -1.169           0.243
L5.y                 -0.027403         0.004119           -6.653           0.000
L5.humidity          -0.001565         0.000702           -2.230           0.026
```

```
L5.dew.point      -0.000117      0.000317      -0.370      0.712
L5.pressure        0.000103      0.003275       0.032      0.975
L6.y               0.002218      0.004118       0.539      0.590
L6.humidity        0.004751      0.000702       6.771      0.000
L6.dew.point      -0.002048      0.000317      -6.465      0.000
L6.pressure        0.000590      0.003275       0.180      0.857
L7.y              -0.006846      0.004118      -1.662      0.096
L7.humidity        0.001522      0.000702       2.168      0.030
L7.dew.point      -0.001229      0.000317      -3.879      0.000
L7.pressure        0.005164      0.003275       1.577      0.115
L8.y              -0.024975      0.004119      -6.064      0.000
L8.humidity       -0.002039      0.000702      -2.905      0.004
L8.dew.point       0.001246      0.000317       3.933      0.000
L8.pressure       -0.011264      0.003275      -3.439      0.001
L9.y               0.002985      0.004116       0.725      0.468
L9.humidity        0.000780      0.000701       1.113      0.266
```

```
plot_metrics(metrics, "Multivariate Baseline Comparison - 2019 test data")
```



Forecasts from the multi-variate VAR model surpass all univariate methods in this notebook at all horizons tested. Obviously the combination of endogenous measurements and exogenous time components is superior to temperature measurements alone. Additionally, the VAR model may have utility for predicting the other endogenous variables.

Tuning lags should be the best approach to further improve VAR forecast accuracy. It would surprise me if lags in multiples of 48 were not significant, for at least the previous 2 or 3 days and perhaps up to a week. Meaning, the next set of lags to try would include combinations from: {1-12, 1-24, 1-48} and {48, 96, 144, 192, 240, 288, 336}.

Optimising time component frequencies and phase shifts would likely aid in reducing forecast errors and would be applicable to other methods.

Conclusion

Here is a brief summary of univariate and multivariate methods plus some potential future work.

Univariate methods:

- Best method - HWES
- Exponential smoothing models performed well
    - As noted in the literature exponential smoothing approaches can beat more sophisticated methods

Multi-variate methods:

- Best method - VAR

Univariate compared to multivariate methods:

- VAR forecasts were substantially better than HWES on 2019 test data on all horizons tested

In conclusion, the VAR forecasts make a good baseline to beat.

Future work could include:

- add error bars, confidence intervals etc
- consider additional metrics such as Brier scores
    - not MAPE because temperature can be 0 and below