**5 things that make my job as a Data Scientist easier**

**After working as a Data Scientist for a year, I am here to share some things I learnt along the way that I feel are helpful and have increased my efficiency. Hopefully some of these tips can help you in your journey :)**



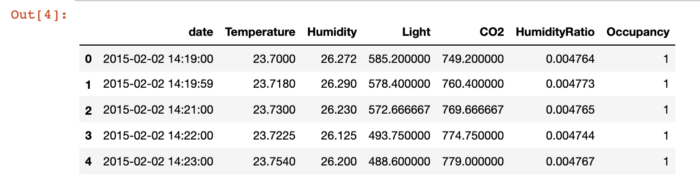
Photo by [Boitumelo Phetla](https://unsplash.com/@writecodenow?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

1. **Time Series Data Processing with Pandas**

If you work with time series data, chances are you have spent a significant amount of time accounting for missing records or aggregating data at a particular temporal granularity either via SQL queries or writing custom functions. Pandas has a very efficient [resample function](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.resample.html) that can help you process the data at a particular frequency by simply setting the DataFrame index to be the timestamp column.

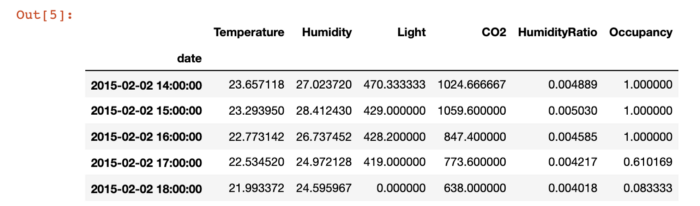
I am going to use the room occupancy dataset to give an example of this function. You can find the dataset [here](https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+). This dataset records observations at a minute level.

import pandas as pd  
data = pd.read\_csv('occupancy\_data/datatest.txt').reset\_index(drop = True)  
data.head(5)



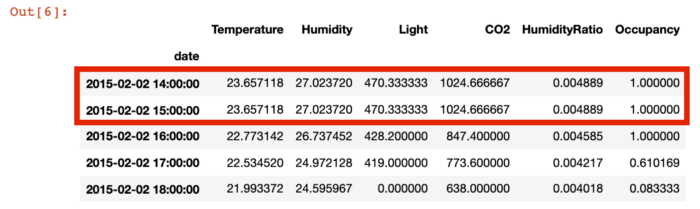
First, I show a simple aggregation one can do to get metrics at an hourly level.

data.index = pd.to\_datetime(data['date'])  
pd.DataFrame(data.resample('H').agg({'Temperature':'mean',  
 'Humidity':'mean',  
 'Light':'last',  
 'CO2':'last',  
 'HumidityRatio' : 'mean',  
 'Occupancy' : 'mean'})).head(5)



Though this dataset is not sparse, in the real world one often encounters data which has missing records. It is important to account for those records since you might want to put in 0 values if there were no records or use the previous or next time steps for imputation. Below, I removed records for hour 15 to show how you can use the hour 14 timestamp to impute the missing value:

data = pd.read\_csv('occupancy\_data/datatest.txt').reset\_index(drop = True)data\_missing\_records = data[~(pd.to\_datetime(data.date).dt.hour == 15)].reset\_index(drop = True)data\_missing\_records.index = pd.to\_datetime(data\_missing\_records['date'])data\_missing\_records.resample('H', base = 1).agg({'Temperature':'mean',  
 'Humidity':'mean',  
 'Light':'last',  
 'CO2':'last',  
 'HumidityRatio' : 'mean',  
 'Occupancy' : 'mean'}).fillna(method = 'ffill').head(5)

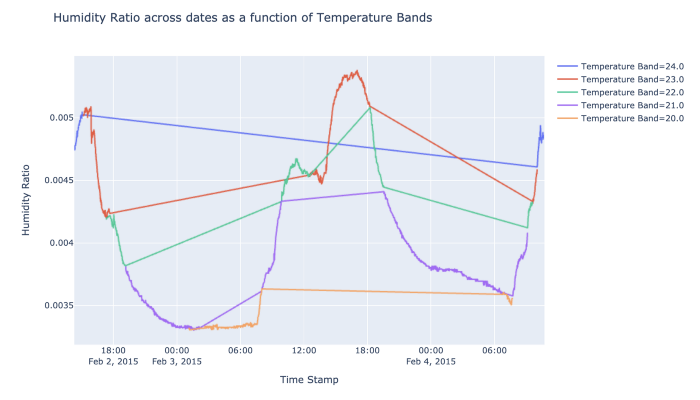


2. **Fast Visualization via Plotly Express**

From analysis to model training to model reporting, visualizations are often required. Especially with time series graphs, I noticed I was spending a lot of time trying to customize the size and angle of my x-axis ticks in matplotlib. After I switched to using Plotly Express, I cut down the time I spent in making graphs looking cleaner/crisper by around 70%. And if I want to implement specific details in my visuals I can still do that by using Plotly Graph Objects. Additionally, Plotly offers a lot of easy options via Express like setting group colors in plots which results in more powerful visualizations.

import plotly.express as px  
data['Temp\_Bands'] = np.round(data['Temperature'])  
fig = px.line(data, x = 'date',  
 y = 'HumidityRatio',  
 color = 'Temp\_Bands',  
 title = 'Humidity Ratio across dates as a function of  
 Temperature Bands',  
 labels = {'date' : 'Time Stamp',  
 'HumidityRatio' : 'Humidity Ratio',  
 'Temp\_Bands' : 'Temperature Band'})  
fig.show()

Using the occupancy dataset mentioned above, I used Plotly Express to create line plots with color grouping. We can see how easy it to create these plots with just two functions.



3. **Speed up pandas apply() via Swifter**

I sometimes run into long wait times for processing pandas columns even with running code on a notebook with a large instance. Instead, there is an easy one word addition that can be used to speed up the apply functionality in a pandas DataFrame. One only has to import the library swifter.

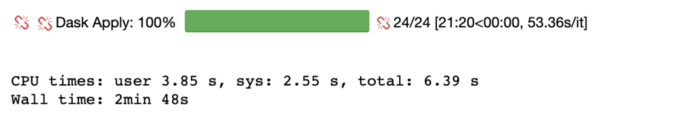
def custom(num1, num2):  
   
 if num1 > num2:  
 if num1 < 0:  
 return "Greater Negative"  
 else:  
 return "Greater Positive"  
 elif num2 > num1:  
 if num2 < 0:  
 return "Less Negative"  
 else:  
 return "Less Positive"  
 else:  
 return "Rare Equal"import swifter   
import pandas as pd  
import numpy as npdata\_sample = pd.DataFrame(np.random.randint(-10000, 10000, size = (50000000, 2)), columns = list('XY'))

I created a 50 million rows DataFrame and compared the time taken to process it via swifter apply() vs the vanilla apply(). I also created a dummy function with simple if else conditions to test the two approaches on.

%%timeresults\_arr = data\_sample.apply(lambda x : custom(x['X'], x['Y']), axis = 1)

https://miro.medium.com/max/700/1*ni1QpCjoKiJTYYfAhJzLXw.png

%%timeresults\_arr = data\_sample.swifter.apply(lambda x : custom(x['X'], x['Y']), axis = 1)



We are able to reduce the processing time by 64.4% from 7 minutes 53 seconds to 2 minutes 38 seconds.

4. **Multiprocessing in Python**

While we are on the topic of decreasing time complexity, I often end up dealing with datasets that I wish to process at multiple granularities. Using multiprocessing in python helps me save that time by utilizing multiple workers.

I demonstrate the effectiveness of multiprocessing using the same 50 million rows data frame I created above. Except this time I add a categorical variable which is a random value selected out of a set of vowels.

import pandas as pd  
import numpy as np  
import randomstring = 'AEIOU'data\_sample = pd.DataFrame(np.random.randint(-10000, 10000, size = (50000000, 2)), columns = list('XY'))  
data\_sample['random\_char'] = random.choices(string, k = data\_sample.shape[0])  
unique\_char = data\_sample['random\_char'].unique()

I used a for loop vs the Process Pool executor from concurrent.futures to demonstrate the runtime reduction we can achieve.

%%timearr = []for i in range(len(data\_sample)):  
   
 num1 = data\_sample.X.iloc[i]  
 num2 = data\_sample.Y.iloc[i]  
   
 if num1 > num2:  
 if num1 < 0:  
 arr.append("Greater Negative")  
 else:  
 arr.append("Greater Positive")  
 elif num2 > num1:  
 if num2 < 0:  
 arr.append("Less Negative")  
 else:  
 arr.append("Less Positive")  
 else:  
 arr.append("Rare Equal")

https://miro.medium.com/max/700/1*RyiANqsRAmQLWQclut5R6w.png

def custom\_multiprocessing(i):  
   
 sample = data\_sample[data\_sample['random\_char'] == \  
 unique\_char[i]]  
   
 arr = []  
   
 for j in range(len(sample)):  
 if num1 > num2:  
 if num1 < 0:  
 arr.append("Greater Negative")  
 else:  
 arr.append("Greater Positive")  
 elif num2 > num1:  
 if num2 < 0:  
 arr.append("Less Negative")  
 else:  
 arr.append("Less Positive")  
 else:  
 arr.append("Rare Equal")  
   
 sample['values'] = arr  
   
 return sample

I created a function that allows me to process each vowel grouping separately:

%%time   
import concurrentdef main():  
 aggregated = pd.DataFrame()  
   
 with concurrent.futures.ProcessPoolExecutor(max\_workers = 5) as executor:  
 results = executor.map(custom\_multiprocessing, range(len(unique\_char)))if \_\_name\_\_ == '\_\_main\_\_':  
 main()

https://miro.medium.com/max/700/1*FLZihJGsx3OhMv00gBKZNw.png

We see a reduction of CPU time by 99.3%. Though one must remember to use these methods carefully since they will not serialize the output therefore using them via grouping can be a good means to leverage this capability.

5. **MASE as a metric**

With the rise of using Machine Learning and Deep Learning approaches for time series forecasting, it is essential to use a metric **NOT** just based on the distance between predicted and actual value. A metric for a forecasting model should use errors from the temporal trend as well to evaluate how well a model is performing instead of just point in time error estimates. Enter [Mean Absolute Scaled Error](https://robjhyndman.com/publications/another-look-at-measures-of-forecast-accuracy/)! This metric that takes into account the error we would get if we used a random walk approach where last timestamp’s value would be the forecast for the next timestamp. It compares the error from the model to the error from the naive forecast.

def MASE(y\_train, y\_test, pred):   
   
 naive\_error = np.sum(np.abs(np.diff(y\_train)))/(len(y\_train)-1)   
 model\_error = np.mean(np.abs(y\_test - pred))return model\_error/naive\_error

If MASE > 1 then the model is performing worse than a random walk. The closer the MASE is to 0, the better the forecasting model.