Time-Series Analysis of Urban Sensing with Big Data and Machine Learning

Jan Jakowski[1]

[1] University of Tennessee, Knoxville TN 37996, USA
Email: jasjakowski@gmail.com

Abstract.

Recent Advances in large scale data storage and computational advancement have led to the opportunity to combine data quantification with large urban sensing. A key part of this process involves the creation of systems that can be used to score and classify the similarity of different dates and potentially apply them to models. The development of statistical correlation technology into Python and the Pandas libraries for large scale data management and analysis have provided an opportunity to analyze the data on a large scale. Specifically, by analyzing the closest days between similarities, future energy usage patterns could also benefit from having a metric used to compare and contrast similar dates and weeks. This approach focuses on the use of previously developed technologies in a smart home neighborhood to monitor similarities in energy patterns with the large scale data challenge issues. By using existing methods for data analysis through a toolkit developed in 2019 [UrbSys'19,50-68(2019)], this method offers a unique approach at providing an opportunity to analyze large data sets. Specifically, by reformatting the data and developing the existing code to handle a larger database, similarities between different dates and visualizations can be employed to showcase how the temperature changes within time. Jupyter-notebook was used for its ability to share comments and markdown with inline code. The data was primarily broken down into a section with 12 days, 2 months, and the entire set with a focus on temperature, wind speed, and rainfall.

Keywords: Big Data, Data Challenge, Urban Sensing, Machine Learning

1 Introduction

1.1 Jupyter Notebook and Initial Code Use of Libraries

Due to its convenience and utility, Jupyter notebook was primarily used for the development of the code to model similarities in time ranges [1].

```
import pandas as pd
from math import sqrt
import seaborn as sns
import pandasql as ps
from scipy.spatial import distance
from scipy.stats import spearmanr
from scipy.stats import kendalltau
```

Fig. 1. Overview of some functional libraries utilized in analysis of dates

Initially, the pandasql library was used for querying to Python's Pandas dataframes which contained most of the information. The dataset provided in challenge 3 contained information on 16 parameters, from latitude and longitude to Global Horizontal Irradiance (RadDif_Wm-2). While this information was abundant, the data had to be cleaned and converted into a format that could be analyzed.

1.2 Unix script and data cleaning formulation

The code we had developed last year contained information that was passed to the program in a specific format. This included an "id" measurement similar to the OID, a "micro_id" that contained a specific measurement type, a "timestamp" similar to that provided in the data challenge set and the measurement value.

```
"id","micro_id","timestamp","val1",
519548492,2985,"2019-04-04 00:02:00","0.00"
```

In contrast, the data challenge's 16 columns with 20 million rows each containing unique data posed a unique challenge in breaking down the data for analysis. As a result, the data had to be reformatted in a way so as to be readable by the rewritten code. This was initially attempted using SQL's data formatting commands and through Pandas's SQL extension library Pandasql. However, the lack of a space between the time in the file provided was causing SQL not to register the timestamps being entered as "time" objects. As a result, a basic Unix script was run to parse the data and consolidate it into a form that was more readable by the program:

```
mycol=2  ;   cut  -d','  -f${mycol}  ${myfile}  | sed -ne   '/time/p' -ne  "s/_/ /p"      | awk '{printf("\"%s\"\n",$0)}'       > col-${mycol}     #   "time"
#mycol=2  ;   cut  -d','  -f${mycol}  ${myfile} > col-${mycol}                                                                  #=> col-2 <== "time"
mycol=3  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-3 <== "lat"
mycol=4  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-4 <== "lon"
#mycol=5  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-5 <== "temp_K"
mycol=5  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("%s\n",$1)}'        > col-${mycol}     #=> col-5 <== "temp_K"
mycol=6  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-6 <== "dewpt_K"
mycol=7  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-7 <== "RH_pct"
#mycol=8  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-8 <== "pres_Pa"
mycol=8  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-8 <== "pres_Pa"
mycol=9  ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-9 <== "RadDir_Wm-2"
mycol=10 ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-10 <== "RadDif_Wm-2"
mycol=11 ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-11 <== "Longwave_Wm-2"
mycol=12 ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-12 <== "ShortwaveNorm_Wm-2"
mycol=13 ;   cut  -d','  -f${mycol}  ${myfile}    | awk '{printf("\"%s\"\n",$1)}'    > col-${mycol}     #=> col-13 <== "Shortwave_Wm-2"
```

Fig. 2. Unix splitting script for data parsing

The benefits of using Unix's shell script over SQL or Pandas were two-fold: computational simplicity and improved runtime[2]. Since the awk command from Unix runs on

a “line-by-line” basis, it is much was personally found to run much faster on a portable laptop than SQL connected to a database.

2 Data Formulation and Focus

Using the aforementioned awk script, the data had been correctly formatted so as to have timestamps readable by Python and SQL’s time class. However, the data had still to be formatted in a way that could be easily readable by the program. In addition, the most significant attributes from every measurement were chosen, namely Temperature, Windspeed, and Rainfall. These attributes were selected as most significant due to limited hardware specifications. Additionally, the dataset was trained using SKLearn’s preprocessing tools. Other Machine Learning (ML) processes are also used in the construction of the datasets base analysis. However, they also serve as attributes that could have the most significant effect on energy usage [3]. As a result of differences between the Connected Neighborhood code and the Data Challenge Data Format, the code was both rewritten and data reformulated to accommodate for new data forms.

```
#map_dict = {'2987': 'Rain Fall', '3496': 'GHI', '2994': 'AMBIENT TEMP',
#             '3497': 'Weather Station Heartbeat',
#            '3498': 'Weather Station Global Horizon I',
#             '3500': 'Weather Station Barometric Press',
#             '2985': 'Wind Speed',
#             '3499': 'Weather Station Daily Rain Total',
#             '2988': 'ON PLANE IRRADIANCE SENSOR',
#             '3495': 'Weather Station Relative Humidity'}
```

Fig. 3. Initial mapping dictionary from Connected Neighborhood project

Since the data was initially read in line-by-line with only one attribute (rainfall, humidity, etc.) at a time, with a corresponding mapping code (micro_id), both the code Connected Neighborhood and data from Challenge 3 had to be redeveloped with a novel mapping of attributes specific to the challenge. In particular, an mapping with micro_ids in the 1000s was chosen to differentiate from previous micro_ids which tended to be between two and three thousand.

2.1 Potential Data Expansion and Solution

As a result of the single attribute reading for every data value, the data tended to expand three times. This was a direct consequence of every line that contained information about rainfall, temperature, and windspeed being printed stored on three distinct lines:

```
1,1001,"2015-01-01 00:00:00",267.953430
2,1002,"2015-01-01 00:00:00",8
3,1003,"2015-01-01 00:00:00",0
```

Although using the awk command to reformulate the data into the separate segments caused the data size to increase somewhat, this was resolved by redeveloping the code to optimize for the three current attributes.

Specifically, the micro_id's were chosen so that as the lines were read in, they could easily and efficiently be filled in to the corresponding value they represented. Since temperature corresponds to 1001, windspeed corresponds to 1002, and rainfall to 1003, the mapping fills in the appropriate attribute that is measured for each timestamp.

```
if  micro_id == '1001':
    try:
        keys_inserted_v2[timestamp]
        query = "UPDATE microgrid_weather_v2 SET val_1001=%s WHERE date='%s'"%(val1, timestamp)
        c.execute(query)
    except:

        query = "INSERT INTO microgrid_weather_v2 VALUES ('%s',%s,0,0)"%(timestamp, val1)
        c.execute(query)
        keys_inserted_v2[timestamp] = True
        print(query)

if  micro_id == '1002':
    try:
        keys_inserted_v2[timestamp]
        query = "UPDATE microgrid_weather_v2 SET val_1002=%s WHERE date='%s'"%(val1, timestamp)
        c.execute(query)
    except:

        query = "INSERT INTO microgrid_weather_v2 VALUES ('%s',0,%s,0)"%(timestamp, val1)
        c.execute(query)
        keys_inserted_v2[timestamp] = True
        print(query)
```

Fig. 4. Micro_id application and formulation of table entries.

Table 1. Using the three micro_ids (1003 not pictured above), the code runs relatively quickly and sorts the table entries according to the timestamp provided. If the previous entries would already exist as was assumed, the entry would be updated with the corresponding value. As a result, a table with one or two entries already existing would have the third simply added and not take up an additional row.

3 SQL Usage and Database Engine Setup.

As Fig. 4 mentions, the entries themselves stored are listed in a table. However, notably, the table itself does not exist within a dataframe, but rather a .db "database".

```
def build_db():
    conn = sqlite3.connect('microgrid_local_all_data.db')
    c = conn.cursor()

    # Create table


    try:
        c.execute('''drop table microgrid_weather_v2''')
    except:
        pass



    #***********
    c.execute('''CREATE TABLE microgrid_weather_v2
                 (date DATETIME primary key ,
                  val_1001 real, val_1002 real, val_1003 real) ''')
    c.execute('''CREATE INDEX date_index_v2 ON microgrid_weather_v2(date);''')


    #***************

    #list_of_microids_v2 = ['lat', 'lon', 'temp_K' , 'dewpt_K', 'RH_pct', 'pres_Pa', 'RadDir_Wm-2', 'RadDif_Wm-2',
    #        'Longwave_Wm-2', 'ShortwaveNorm_Wm-2', 'Shortwave_Wm-2', 'WindDir_deg', 'WindSpd_ms-1', 'RainDpth_mm']
    list_of_microids_v2 = ['1001', '1002', '1003']


    keys_inserted_v2 = {}
```

Fig. 5. Table setup as a database file that exists on hard disk. Within database, table is created to store entries with columns containing temperature, wind speed, and rainfall.

As depicted in the figure with table entries, the use of a database file stems from previous research work into Connected Neighborhoods. However, by adapting rewriting existing code structures and repurposing the data challenge data, a unique approach to solving some data analysis problems and comparing similar dates is presented.

3.1 Querying Directly Into Database through SQL

While Python presents several helpful libraries with objects such as dataframes and accompanying functions, the use of a database to store the table values presents an opportunity to use SQL queries to communicate with the table and get the derived entry values.

```
def get_array_v2(date, col_idx):
    array = []
    conn = sqlite3.connect('microgrid_local_all_data.db')
    c = conn.cursor()
    query = "SELECT * FROM microgrid_weather_v2 where strftime('%M', date)%15==0 and date like '"+date+"%'order by date
    #print("--------------------------")
    #print(c.execute(query))
    for row in c.execute(query):
        array.append(row[col_idx])
    conn.close()
    return array


def pandas_sql_v2(date):
    conn = sqlite3.connect('microgrid_local_all_data.db')
    df = pd.read_sql("SELECT * FROM microgrid_weather_v2 where strftime('%M', date)%15==0 and date like '"+date+"%'orde
                     con=conn)
    return df
```

Fig. 6. Basic Query Functions in Python that return a dataframe based on date attributes

Within the table, a modified “string” is injected into the dataframe from the database connection established. Every time data is required, the connection must be established, the data retrieved, and the connection closed again. As such, building up the database

takes a significant portion of the code, but once data is present within a pandas data-frame, analysis can continue.

3.2 Statistical correlational function

In order to provide a correlational metric that takes into account different similarities in time-series data, an aggregate correlational function was used in the Connected Neighborhood Project. This code, while useful, was applied to the different attributes formed in the data provided. Within the scope of the data challenge, the adapted data and code takes into account the larger albeit more focused (temperature, windspeed, and rainfall) dataset.

```
mae_vals = []
euclidean_vals = []
pearson_vals = []
spearman_vals = []
kendall_vals = []
dtw_vals = []

date1 = list_dates[0]
date2 = list_dates[1]

df_day1 = pandas_sql(date1)#(str(i[0]))
df_day2 = pandas_sql(date2)#(str(i[1]))

for idx in list(df_day1.columns)[1:]: # This will iterate all columns
    array_1 = df_day1[idx].values
    array_2 = df_day2[idx].values

    if len(array_1) == len(array_2):
        data_type = map_dict[idx.split('val_')[1]]
        dtw_val = DTWDistance(array_1, array_2)
        mae_val = MAE(array_1, array_2)
        euclidean_val = euclidean(array_1, array_2)
        pearson_val = pearson(array_1, array_2)
        spearman_val = spearman(array_1, array_2)
        kendall_val = kendall(array_1, array_2)
        data.append([date1, date2, data_type, dtw_val, mae_val, euclidean_val, pearson_val, spearman_val, kendall_v
```

Fig. 7. Compound Aggregate Correlational Function

The "comparison function" take into account Spearman and Kendall-Tau similarities in developing analysis [4,5]. Additionally, the use of multiple metrics for comparison serves as a sort of baseline "average" between different correlational metrics. This baseline includes a Pearson [6] and Kendall [4] rank correlational coefficient for scoring different sets such as pairs of dates.

4 Application of Similar Dates and DataFrame

As a "list of dates ver. 2" list is used to store the different unique dates appearing in the database, this data is fed into the statistical correlational functions and between all possible combinations of dates. The effect of this is the "ranking" of every date to every other date and finding the most similar pairs between the two.

	score_weight
count	55.000000
mean	0.143408
std	0.086202
min	0.015581

25%	0.064280
50%	0.124590
75%	0.227194
max	0.316413

Table 2. Ranking different metrics for a small sample set of 11 days

The code is initially run on a small sample set of 12 days (January 1- 11) in order to develop a simple baseline. While the correlations are note exceedingly strong, they serve as a useful starting point before taking in larger sets of data.

Additionally, a dataframe is used to store the rankings, which appear symmetric with respect to the diagonal.

```
new_score_df = new_score_df.fillna('0')
new_score_df = new_score_df.sort_index(axis=1)
new_score_df = new_score_df.sort_index(axis=0)
```

```
new_score_df
```

	2015-01-01	2015-01-02	2015-01-03	2015-01-04	2015-01-05	2015-01-06	2015-01-07	2015-01-08	2015-01-09	2015-01-10	2015-01-11	2015-01-12
2015-01-01	0	0.10901	0.126419	0.0379738	0.201439	0.16153	0.0902833	0.264853	0.0990668	0.262058	0.221307	0
2015-01-02	0.10901	0	0.084649	0.06311	0.310303	0.06545	0.256058	0.0541252	0.214121	0.062954	0.117711	0
2015-01-03	0.126419	0.084649	0	0.0307958	0.114644	0.220689	0.0829068	0.187817	0.0865679	0.1909	0.233082	0
2015-01-04	0.0379738	0.06311	0.0307958	0	0.079169	0.128875	0.201547	0.12459	0.244037	0.107355	0.0202969	0
2015-01-05	0.201439	0.310303	0.114644	0.079169	0	0.0550348	0.242308	0.0385769	0.17598	0.0687218	0.145482	0
2015-01-06	0.16153	0.06545	0.220689	0.128875	0.0550348	0	0.0464247	0.235517	0.0155806	0.242814	0.256816	0
2015-01-07	0.0902833	0.256058	0.0829068	0.201547	0.242308	0.0464247	0	0.0499177	0.272696	0.0561505	0.105085	0
2015-01-08	0.264853	0.0541252	0.187817	0.12459	0.0385769	0.235517	0.0499177	0	0.0483153	0.316413	0.242431	0
2015-01-09	0.0990668	0.214121	0.0865679	0.244037	0.17598	0.0155806	0.272696	0.0483153	0	0.0457116	0.127988	0
2015-01-10	0.262058	0.062954	0.1909	0.107355	0.0687218	0.242814	0.0561505	0.316413	0.0457116	0	0.243772	0
2015-01-11	0.221307	0.117711	0.233082	0.0202969	0.145482	0.256816	0.105085	0.242431	0.127988	0.243772	0	0
2015-01-12	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 8. DataFrame entires provided in tabular format scoring rankings

The Dataframe table containing information is useful to construct a “heatmap” of sorts based on rankings from 0 to 1 which also serve as a useful visualization for the data challenge.

4.1 Consequences for Visualizations

Due to the time-series similarity function that ranked dates based on their “closeness” to one another on weather patterns in temperature, rainfall, and windspeed, the Data-Frame with rankings can also be represented as a “heatmap”. Dynamic time warpong can be used to compare different time-series datasets even if the amount of data is different in each [7]. Since there are gaps in the data throughout the year and especially towards the end of each month, this approach for comparison can be used with the data challenge set for ranking the closeness between different time ranges. In addition, the most similar dates can be plotted to validate their similarities.

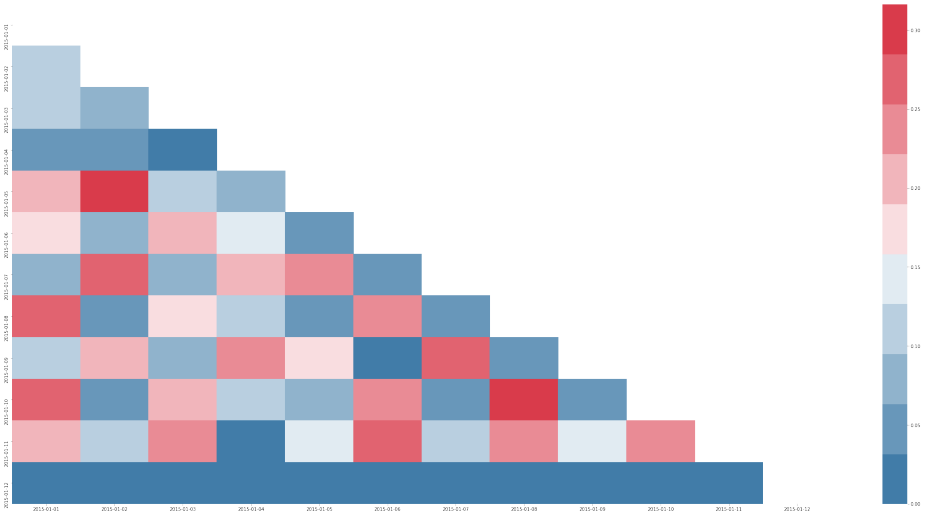

Fig. 9. Heatmap of Similar Dates in January (1st-11th)

The heat map representation is another "human friendly" visualization for the data challenge that can help researchers visualize their own time series data. However, a plot of similar dates by hour shows a fuller picture of similar dates on a less macroscopic level.

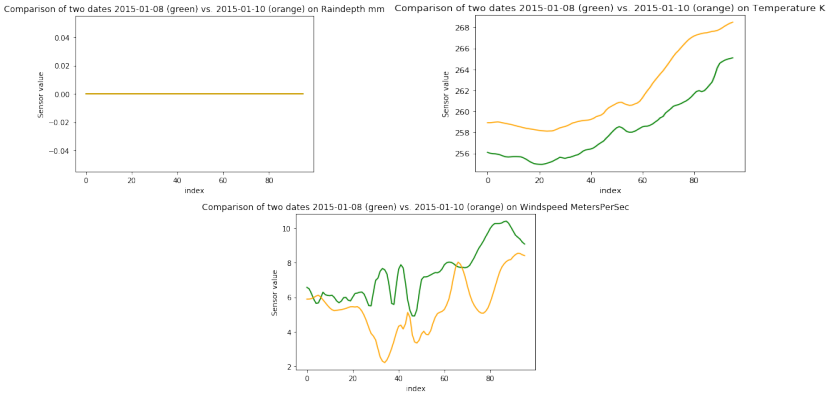

Fig. 10. Time series Plotting Of Similar Dates and their associated values

As can be seen above, the most similar dates of January 8th and 10th both contain similar heating patterns to within 3 Kelvin most of the day, no rainfall, and sporadic wind speeds. These patterns help visually validate the similarities in weather and provide a useful outlook on how different dates compare.

5 Expansion to entire dataset

While the expansion to the larger dataset provided with 20 million lines does increase the file size and time of completion, the results help provide a larger pool of potential comparisons between dates.

5.1 Adapting to 5 Million Lines

Upon analysis of the size of the dataset provided, it was concluded that an iterative process to monitor the performance of the code would be an optimal solution at gauging its performance. As such, initially, only 200,000 lines were initially chosen to build up the database. However, this data was only sufficient for a little more than a day's worth of data. As such, an analysis could not properly run without larger comparison. In order to accommodate for this, the file size was increased to allow for more lines and the process was re-evaluated again

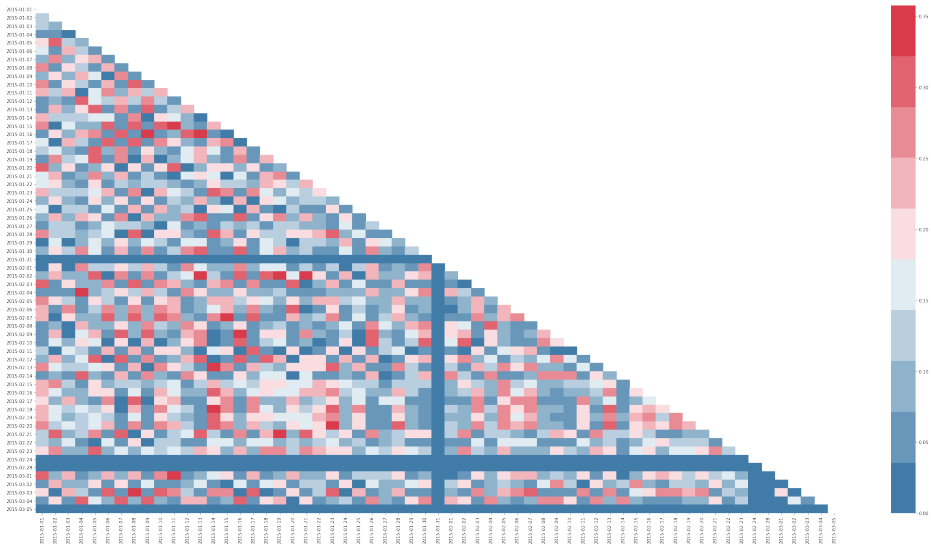

Fig. 11. Heatmap of Most Similar Dates January 1- March 5, 2015

The above heatmap serves as an expansion of the smaller dataset from the first 12 days in January. As can be seen, most of the dates are not similar to one another. Yet a few of the days, most notable January 1st and 15th contain a high correlation to each other. Additionally, the lack of data towards the end of the month (as explained in the Reddit Ask Me Anything session) can start to be seen in the bands going along the grid. Furthermore, the two most similar pairs (January 1&15 and January 4 & February 4) are also shown to have consistent temperature, similar wind patterns and no rainfall as shown in the figure below

Fig. 12. Time-series graph of two most similar dates for time from January 1 to March 5

After running the larger time-series graph it become apparent that the code will scale to larger application. As a result, it was applied to the entire data-set and used to provide a fuller picture of analysis with visualization.

5.2 Scaling to Full Dataset

Despite some initial issues with adapting new micro_id's and building the database, the code was re-written to accommodate for the larger dataset. This final version (also provided with paper/appendix)

Heatmap from All Data

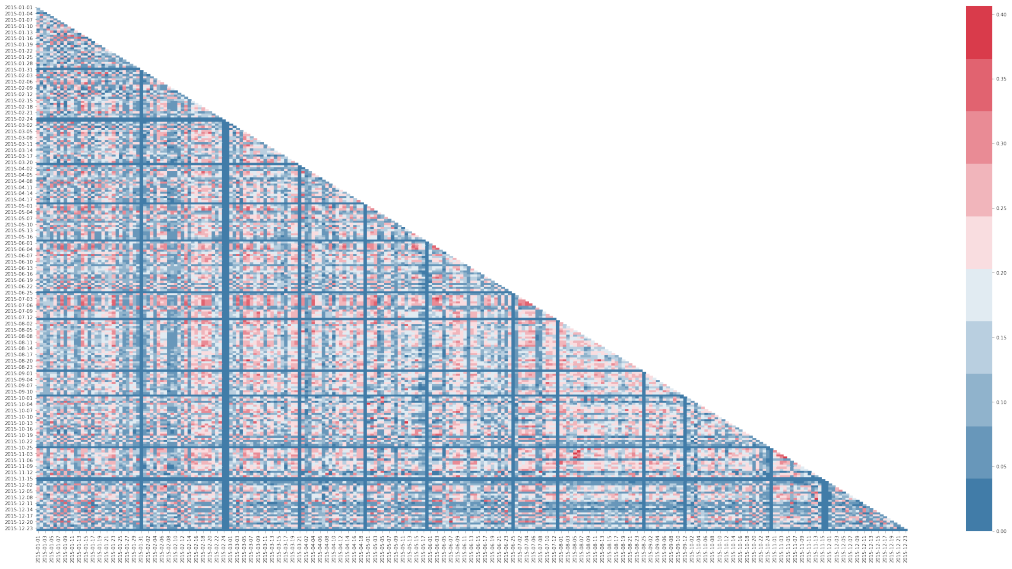

Fig. 13. All data heatmap showing similarities between various pairs of dates

As described in 5.1 and discussed in the Data Challenge AMA session, the "lack of data" time periods become visible towards the end of each month. They surface as horizontal and vertical bands blue bands of low correlation on the heatmap. The heatmap is another visualization tool that can be useful to researchers studying data weather patterns at regular intervals.

All Data Time-Series Graph Comparison

Fig. 14. Graph of most similar weather dates from entire urban sensing data set

After running the entire comparison code on the most similar dates to generate a heatmap, the most similar dates pairs of March 5 & October 13 and February 7 & December 3 are generated. It is interesting to note both comparatively how these date pairs are based on data that is far apart and computationally for the size of comparisons required. Initially, it was hypothesized that the most similar dates would likely be those that are close to one another. This was seemingly supported through the analysis of weather from the 5 Million lines of data from January to March where January 11 and 15 are found to be the closest. However, by running the dataset on the full set, it was found that the closest dates are often separated by several months as was the case with March and October. This yields an interesting computational question: whether the slight increase in similarity from the entire dataset warrants greater computational cost of comparing multiple dates. Visually, both the highest scoring pair from the dataset with 5 Million lines and the 20 million lines seem to contain quite similar data on an hour-by-hour basis. Yet the number of comparisons for n dates

$$\text{Comp}(n) = n + n\text{-}1 + n\text{-}2 + \ldots + 3 + 2 + 1 \quad (1)$$

Tends toward

$$\frac{n*(n+1)}{2} \quad (2)$$

Since the total number of comparisons to be made is one less than the date before it until the first date must be compared, which has n comparisons itself. Overall, this basic application of Big-O can also be used to analyze the benefits vs drawbacks of such an approach for analysis.

Big O Analysis:

Since the full dataset contains ~ 20 million lines compared to the 5 million trained in the January-March dataset, it contains roughly 4n more dates than the latter. So the method requires,

$$\frac{4*(4+1)}{2} = 10$$

or 10 times as many comparisons despite only containing 4 times as much data. As such, this is a questions each researcher using such approaches will have to address on their own and adapt to their own situation. However, such a toolset for visualizing different data sets and pairs can be useful for visualizing large datasets like those in the data challenge.

6 Conclusion/Ramifications

While mapping the time series data of a particular set of dates does increase the number of combinations that must be compared to one another, sufficient advancements in personal computing have allowed the large data set to be run on a 8gb ram laptop from 2014 with sufficient help from an external hard drive to store the data set. While the read/write time could be a factor in increasing wait times for the analysis to execute, it can be reasonably assumed that more ram or a solid state drive, or upscaling to a larger architecture/server could help the process execute faster.

6.1 Ramifications to Existing Software

Nonetheless, from redevelopment of an existing project, it has been found that developing code to monitor time-series data can be performed agnostic to the system language. The development of time-series analysis code can serve to further benefit urban sensing efforts and provide a unique baseline into the comparison of different dates ranges.

The code can also be generalized into generic dates ranges and select based on minute, month, or week as was done within the connected neighborhood project.

Overall, even though the Connected Neighborhood Project and Data Challenge focused on different buildings in different parts of the country, this analysis has found that the collections of data for date range analysis in urban sensing can be generalized to any size needed and many of the functions are language agnostic (between pandas, awk, and SQL).

6.2 Future work

An interesting future point of research would be the continuation of developing machine learning technologies that can be trained and tested on large data sets such as data challenge 3. The current program analysis could be further built upon to incorporate such elements in the form of more user friendly functions. These functions could be

available to researchers in need of customized functions much like pandas has the .Series and .toNumpy function for conversion between different library types. As the market for smart home devices has increased, new algorithms to measure their performance can be of great use to engineers and researchers alike [3]. These tools could be extended to fit their needs in the future

Acknowledgements

Special Thanks to Msc. Supriya Chinthavali and the 2019 Connected Neighborhood team for hosting me summer 2019. The data analysis in this paper are partially based on the 2019 Connected Neighborhood code.

References

[1] Supriya Chinthavali, Varisara Tansakul, Sangkeun Lee, Anika Tabassum, Jeff Munk, Jan Jakowski, Michael Starke, Teja Kuruganti, Heather Buckberry, Jim Leverette, *Quantification of Energy Cost Savings through Optimization and Control of Appliances within Smart Neighborhood Homes,* in UrbSys'19: Proceedings of the 1st ACM International Workshop on Urban Building Energy Sensing, Controls, Big Data Analysis, and Visualization, Nov. 13-14, 2019, New York pp. 50-68, [DOI: 10.1145/3363459.3363535]

[2] Free Software Foundation, Inc.: *The GNU AWK User's Guide*, 1989. Accessed on: Jul. 2, 2020. [Online] Available: https://www.gnu.org/software/gawk/manual/gawk.html

[3] EIA. [n. d.]. EIA's residential energy survey now includes estimates for more than 20 new end uses. https://www.eia.gov/todayinenergy/detail.php?id=36412

[4] Hervé Abdi. 2007. The Kendall rank correlation coefficient. Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA (2007), 508–510.

[5] Thomas D Gauthier. 2001. Detecting trends using Spearman's rank correlation coefficient. Environmental forensics 2, 4 (2001), 359–362.

[6] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In Noise reduction in speech processing. Springer, 1–4.

[7] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. Knowledge and information systems 7, 3 (2005), 358–386.

Appendix

The code for the data challenge, "Data_challenge_3_code_Jan_Jakowski_All_Data.ipynb" is being attached along with this paper. Also, the shell script "dosplit" used for cleaning the data is also being provided for reference