Exercise 4

Due Friday June 09 2023.

Some files are provided that you need below: E4.zip. **You may not write any loops** in your code (with one exception noted below).

Merging Walking Data

Exercise 3 lied about the GPS data you used last week: it got a bunch of cleanup before you saw it, and was actually the result of combining two data sources: data from my phone's sensors recorded with Physics Toolbox Sensor Suite *(https://play.google.com/store/apps/details?id=com.chrystianvieyra.physicstoolboxsuite&gl=US)* and metadata from a GoPro recording *(https://github.com/gopro/gpmf-parser)* of the same walk. I made a video explaining how it was collected *(https://youtu.be/CW71z5iAs8s)* if you're interested.

The Physics Toolbox Suite data (`phone.csv.gz`) does not record date/time values, but only seconds offset from the start of the recording. The GoPro data has a `timestamp` field with a precise date and time (all of the other files). We need to combine these two recordings to get the GoPro's GPS data with the phone's magnetometer data: that was necessary to do last week's problem. (In the interest of transparency: I *could* have recorded the GPS data with my phone as well, and not had to integrate these files. But I didn't, thus creating this exercise question.)

Combining the Data

Your program `combine_walk.py` should take the *directory* containing the input files and destination directory for output files on the command line:

```
python3 combine_walk.py walk1 output
```

This should read the files from the GoPro (`walk1/accl.ndjson.gz`, `walk1/gopro.gpx`) from that directory, and the data from the phone (`walk1/phone.csv.gz`). See the provided `combine_walk_hint.py` that gives example code for the reading/writing of data.

Looking at the data sets, I see a few problems right away: (1) the times of the different measurements are different because the sensors are read at different times, (2) there are more readings than we need and a lot of noise. We can start to attack both problems by grouping nearby observations and averaging the readings. But also, (3) the phone data doesn't have date/time values, only a number of seconds from the start time.

For now we will assume that the phone data starts at exactly the same time as the accelerometer data (and refine this assumption below). That is, we can create a timestamp in the phone data like this, with to_timedelta *(https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_timedelta.html)* :

```
offset = 0
first_time = accl['timestamp'].min()
phone['timestamp'] = first_time + pd.to_timedelta(phone['time'] + offset, unit='sec')
```

[We will choose a better value for `offset` below.]

To unify the times, we will aggregate using 4 second bins (i.e. round to the nearest 4 seconds, hint 1 *(https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.dt.round.html)*), then group on the rounded-times, and average all of the other values (hint 2 *(https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html?highlight=groupby#pandas.DataFrame.groupby)* , hint 3 *(https://numpy.org/doc/stable/reference/generated/numpy.mean.html)*).

If you do this correctly, you should have three DataFrames (accelerometer, GPS, phone sensors) with identical keys/labels on the rows (perhaps ignoring a few rows where the two devices didn't start/stop at exactly the same time).

Correlating Data Sets

Combining the data sets hasn't really been done correctly yet because **I did not press record at the exact same time** on both devices. The first data point on the phone will be offset from the first on the GoPro by a few seconds.

Timestamps are consistent within the data extracted from the GoPro (`accl.ndjson.gz`, `gopro.gpx`).

There are fields that have been measured in both devices: if we can guess the offset time between the two data sets, we should see those values align. They won't be *exactly* the same because they aren't exactly the same sensors, but they should correlate.

We will look at acceleration in the *x*-axis: `gFx` in the phone data, and `x` from the accelerometer data. We need a time offset that can be applied to the phone data so that the `time` values from there will be "real" times as in the GoPro data.

That is, instead of working with `phone['time']`, we need to take our input as `phone['time'] + offset`. The time offset will be at most 5 seconds, and we would like to find it accurate to one decimal place. That is, `offset` will be a value chosen from `np.linspace(-5.0, 5.0, 101)`.

Your code should find the `offset` with the highest cross-correlation *(https://en.wikipedia.org/wiki/Cross-correlation)* between the two accelerometer readings (after the "four second bins" is done to the phone data). The cross-correlation is basically the dot-product of the two fields: multiply the corresponding `'gFx'` and `'x'` values, and sum the results.

You may use one loop (like "`for offset in np.linspace(-5.0, 5.0, 101)`") here.

Once you have the data correctly combined, `print` the best time offset for the phone data (as in the hint code), and produce files `walk.gpx` and `walk.csv` in the output directory containing the fields that were necessary for exercise 3 (as in the hint code).

Note: you will not get exactly the same values as the dataset provided for exercise 3. The technique is the same, but the artificially-added noise is not.

Cities: Temperatures and Density

This question will combine information about cities from Wikidata *(https://www.wikidata.org/wiki/Q24639)* with a different subset of the Global Historical Climatology Network *(https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-ghcn)* data.

The question I have is: **is there any correlation between population density and temperature**? I admit it's an artificial question, but one we can answer. In order to answer the question, we're going to need to get population density of cities matched up with weather stations.

The program for this part should be named `temperature_correlation.py` and take the station and city data files in the format provided on the command line. The last command line argument should be the filename for the plot you'll output (described below).

```
python3 temperature_correlation.py stations.json.gz city_data.csv output.svg
```

The Data

The collection of weather stations is quite large: it is given as a line-by-line JSON file that is gzipped. That is a fairly common format in the big data world. You can read the gzipped data directly with Pandas and it will automatically decompress as needed:

```
stations = pd.read_json(stations_file, lines=True)
```

The `'avg_tmax'` column in the weather data is °C×10 (because that's what GHCN distributes): it needs to be divided by 10. The value is the average of TMAX values from that station for the year: the average daily-high temperature.

The city data is in a nice convenient CSV file. There are many cities that are missing either their area or population: we can't calculate density for those, so they can be removed. Population density is population divided by area.

The city area is given in m^2, which is hard to reason about: convert to km^2. There are a few cities with areas that I don't think are reasonable: exclude cities with area greater than 10000 km^2.

Entity Resolution

Both data sets contain a latitude and longitude, but they don't refer to exactly the same locations. A city's “location” is some point near the centre of the city. That is very unlikely to be the exact location of a weather station, but there is probably one nearby.

Find the weather station that is closest to each city. We need it's `'avg_tmax'` value. This takes an $O(mn)$ kind of calculation: the distance between every city and station pair must be calculated. Here's a suggestion of how to get there:

- Write a function `distance(city, stations)` that calculates the distance between **one** city and **every** station. You can probably adapt your function from the GPS question last week. [1]
- Write a function `best_tmax(city, stations)` that returns the best value you can find for `'avg_tmax'` for that one city, from the list of all weather stations. Hint: use `distance` and `numpy.argmin` (or `Series.idxmin` or `DataFrame.idxmin`) for this. [2]
- Apply that function across all cities. You can give extra arguments when applyling in Pandas like this: `cities.apply(best_tmax, stations=stations)`.

[1] When working on the collection of stations, make sure you're using Python operators on Series/arrays or NumPy ufuncs *(https://docs.scipy.org/doc/numpy/reference/ufuncs.html#math-operations)* , which are much faster than `DataFrame.apply` or `np.vectorize`. Your program should take a few seconds on a reasonably-fast computer, not a few minutes. [Aside: Python operators on Series/arrays are overloaded to call those ufuncs.]

[2] Note that there are a few cities that have more than one station at the same minimum distance. In that case, we'll use the station that is **first** in the input data. That choice happens to match the behaviour of `.argmin` and `.idxmin`, so if you ignore the ambiguity, you'll likely get the right result.

Output

Produce a scatterplot of average maximum temperature against population density (in the file given on the command line).

Let's give some nicer labels than we have in the past: see the included `sample-output.svg`. The strings used are beautified with a couple of Unicode characters: `'Avg Max Temperature (\u00b0C)'` and `'Population Density (people/km\u00b2)'`.

Questions

Answer these questions in a file `answers.txt`. [Generally, these questions should be answered in a few sentences each.]

1. Based on your results for the last question, do you think daily temperatures are a good way to predict population density? Briefly explain why or why not.
2. Several of the (larger) data files were was kept compressed on disk throughout the analysis, so they needed to be decompressed ever time we ran our program: that seems inefficient. Why might this be faster than working with an uncompressed `.json` or `.csv` data?

Submitting

Submit your files through CourSys for Exercise 4.

Updated Fri June 09 2023, 12:18 by ggbaker.