Exercise 3

Due Friday June 02 2023.

Some files are provided that you need below: E3.zip. **You may not write any loops** in your code (except in one place mentioned below).

CPU Temperature Noise Reduction

I gathered some data on the CPU temperature and usage of one of the computers in my life: sampling every minute (with psutil *(https://pypi.python.org/pypi/psutil)* , if you're wondering). That gives a temperature (in °C), CPU usage (in percent), and one-minute system load (number of processes running/waiting averaged over the last minute). The data is in the provided `sysinfo.csv`.

If you have a look, you will see that there's a certain amount of noise from the temperature sensor, but it also seems like there are some legitimate changes in the true temperature. We would like to separate these as best possible and determine the CPU temperature as closely as we can.

For this question, write a Python program `smooth_temperature.py`. It should take its input CSV file as a command line argument:

```
python3 smooth_temperature.py sysinfo.csv
```

Before you get started, have a look:

```
plt.figure(figsize=(12, 4))
plt.plot(cpu_data['timestamp'], cpu_data['temperature'], 'b.', alpha=0.5)
# plt.show() # maybe easier for testing
plt.savefig('cpu.svg') # for final submission
```

LOESS Smoothing

We can try LOESS smoothing to get the signal out of noise. For this part of the question, we're only worried about the temperature values.

Use the `lowess` function from statsmodels *(http://www.statsmodels.org/stable/generated/statsmodels.nonparametric.smoothers_lowess.lowess.html)* to generated a smoothed version of the temperature values.

Adjust the `frac` parameter to get as much signal as possible with as little noise as possible. The contrasting factors: (1) when the temperature spikes (because of momentary CPU usage), the high temperature values are reality and we don't want to smooth that information out of existence, but (2) when the temperature is relatively flat (where the computer is not in use), the temperature is probably relatively steady, not jumping randomly between 30°C and 33°C as the data implies.

Have a look and see how you're doing:

```
loess_smoothed = lowess(…)
plt.plot(cpu_data['timestamp'], loess_smoothed[:, 1], 'r-')
```

Kalman Smoothing

A Kalman filter will let us take more information into account: we can use the processor usage, system load, and fan speed to give a hint about when the temperature will be increasing/decreasing. The time stamp will be distracting: keep only the four columns you need.

```
kalman_data = cpu_data[['temperature', 'cpu_percent', 'sys_load_1', 'fan_rpm']]
```

To get you started on the Kalman filter parameters, I have something like this:

```
initial_state = kalman_data.iloc[0]
observation_covariance = np.diag([0, 0, 0, 0]) ** 2 # TODO: shouldn't be zero
transition_covariance = np.diag([0, 0, 0, 0]) ** 2 # TODO: shouldn't be zero
transition = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]] # TODO: shouldn't (all) be zero
```

You can choose sensible (non-zero) values for the observation standard deviations here. The value `observation_covariance` expresses how much you believe the *sensors*: what kind of error do you usually expect to see (perhaps based on looking at the scatter plot, or by estimating the accuracy of the observed values). The values in the template above are taken to be standard deviations (in the same units as the corresponding values) and then squared to give variance values that the filter expects.

The `transition_covariance` expresses how accurate *your prediction* is: how accurately can you predict the temperature of the CPU (and processor percent/load), based on the previous temperature and processor usage?

The transition matrix is where we can be more clever. I predict that the “next” values of the variables we're observing will be:

$$\begin{aligned} \textit{temperature} &\leftarrow 0.94 \times \textit{temperature} + 0.5 \times \textit{cpu\_percent} + 0.2 \times \textit{sys\_load\_1} - 0.001 \times \textit{fan\_rpm} \\ \textit{cpu\_percent} &\leftarrow 0.1 \times \textit{temperature} + 0.4 \times \textit{cpu\_percent} + 2.1 \times \textit{sys\_load\_1} \\ \textit{sys\_load\_1} &\leftarrow 0.94 \times \textit{sys\_load\_1} \\ \textit{fan\_rpm} &\leftarrow \textit{fan\_rpm} \end{aligned}$$

These values are the result of doing some regression on the values and a fair bit of manual tweaking. You're free to change them if you wish, but it's probably not necessary.

Experiment with the parameter values to get the best smoothed result you can. The tradeoffs are the same as before: removing noise while keeping true changes in the signal. Have a look:

```
kf = KalmanFilter(…)
kalman_smoothed, _ = kf.smooth(kalman_data)
plt.plot(cpu_data['timestamp'], kalman_smoothed[:, 0], 'g-')
```

Final Output

Add a legend to your plot so we (and you) can distinguish the data points, LOESS-smoothed line, and Kalman-smoothed line. Hint: you saw `plt.legend` in Exercise 1.

When you submit, make sure your program is not popping up a window, but **saves the plot as `cpu.svg`** with the data points, LOESS smoothed line, and Kalman smoothed line: `plt.savefig('cpu.svg')`.

GPS Tracks: How Far Did I Walk?

Sensor noise is a common problem in many areas. One sensor that most of us are carrying around: a GPS receiver in a smartphone (or a smartwatch or other wearable). The data that you get from one of those devices is often filtered automatically to remove noise, but that doesn't mean there isn't noise inherent in the system: GPS can be accurate to about 5m *(http://www.gps.gov/systems/gps/performance/accuracy/)* .

Modern GPS systems do much better than 5m accuracy for various reasons: more accurate satellites/receivers, augmenting with other location information, and internal noise filtering that happens before you ever see the data. In the data for this assignment, noise has been artificially added so the errors in the measurements are human-scale and visible.

I recorded some tracks of myself walking with Physics Toolbox Sensor Suite *(https://play.google.com/store/apps/details?id=com.chrystianvieyra.physicstoolboxsuite&gl=US)* . They are included as corresponding `*.gpx` files (for latitude/longitude data) and `*.csv` files (for the other fields collected).

GPX files are XML files that contain (among other things) elements like this for each observation:

```
<trkpt lat="49.28022235" lon="-123.00543652"><time>…</time>…</trkpt>
```

The question I want to answer is simple: **how far did I walk?** The answer to this can't be immediately calculated from the tracks, since the noise makes it look like I ran across the street, crossed back, backed up, jumped forward, …. I actually walked in mostly-straight lines, as one does. On the other hand, we can't just take the difference between the starting and ending points: I didn't walk a completely straight line either.

For this question, write a Python program `calc_distance.py` that does the tasks described below. See the included `calc_distance_hint.py`. Your program must take the paths of the GPX and CSV files on the command line, so this command would read the first example data:

```
python3 calc_distance.py walk1.gpx walk1.csv
```

Read the XML

Since the GPX files are XML-based, you'll need to use an XML library to read them. Pick one of xml.dom.minidom *(https://docs.python.org/3/library/xml.dom.minidom.html)* (with xml.dom *(https://docs.python.org/3/library/xml.dom.html)*) or xml.etree.ElementTree *(https://docs.python.org/3/library/xml.etree.elementtree.html)* (both of which are in the Python standard library). Dealing with XML namespaces in ElementTree can be tricky. Here's a hint:

```
parse_result.iter('{http://www.topografix.com/GPX/1/0}trkpt')
```

[You may **not** use a GPX-parsing library: working with XML is an expectation of this exercise.]

You will need to extract the latitude and longitude and time from each `<trkpt>` element. We can ignore the elevation and other fields. **Create a DataFrame** with columns `'datetime'`, `'lat'` and `'lon'` holding the observations. [It's certainly possible to do this without loops, but you may write a loop to iterate as you read the file/elements for this part.]

The `'datetime'` should be a NumPy `datetime64`, which you can convert from the original strings (in a Pandas series) like this:

```
data['datetime'] = pd.to_datetime(data['datetime'], utc=True)
```

Read the CSV and Combine

There are many columns in the CSV file, but the ones we care about are the compass readings: it would be useful to know which direction I was walking in, and we could use that to better predict the location when noise filtering.

The compass direction is given to us as *x* and *y* components of the phone's magnetometer *(https://en.wikipedia.org/wiki/Magnetometer)* . These values would need some trigonometry to get a heading *(https://en.wikipedia.org/wiki/Heading_(navigation))* , but we can work with them directly.

For this assignment, the timestamps (`<time>` in the GPX file and `datetime` in the CSV file) will match **exactly**. Thus if you have a DataFrame `points` containing the GPX data and with the datetime as `datetime64`, you can read the CSV file and incorporate the relevant fields like this:

```
points = points.set_index('datetime')
sensor_data = pd.read_csv(input_csv, parse_dates=['datetime']).set_index('datetime')
points['Bx'] = sensor_data['Bx']
points['By'] = sensor_data['By']
```

You aren't required to use that hint, but it's a pretty good hint.

Calculate Distances

To get from latitude/longitude points to distances, we'll need some trigonometry: the haversine formula *(https://en.wikipedia.org/wiki/Haversine_formula)* in particular. You can find more implementation-friendly descriptions of the haversine calculation *(http://stackoverflow.com/questions/27928/calculate-distance-between-two-latitude-longitude-points-haversine-formula/21623206)* online, of course. (But remember AcademicHonesty in your code: if you take any code from somewhere else, we **always** expect a reference.)

Write a function `distance` that takes a DataFrame as described above, and returns the distance (in metres) between the latitude/longitude points (without any noise reduction: we'll do that next).

This can be done with DataFrame.shift *(http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shift.html)* (to get adjacent points into the same rows) and a few expressions on the arrays. If you haven't noticed before, NumPy has implemented useful mathematical operations *(https://docs.scipy.org/doc/numpy/reference/routines.math.html)* on arrays.

For the record, I get:

```
>>> points = pd.DataFrame({
    'lat': [49.28, 49.26, 49.26],
    'lon': [123.00, 123.10, 123.05]})
>>> distance(points).round(6)
11217.038892
```

In your main program, print out the literal distance described in the GPX file, rounded to two decimal places, like this:

```
dist = distance(points)
print(f'Unfiltered distance: {dist:.2f}')
```

Kalman Filtering

Kalman predictions rely on a "prediction". Our prediction for the "next" values will be:

$$\begin{aligned} latitude &\leftarrow latitude + (5 \times 10^{-7} \times Bx) + (34 \times 10^{-7} \times By) \\ longitude &\leftarrow longitude + (-49 \times 10^{-7} \times Bx) + (9 \times 10^{-7} \times By) \\ Bx &\leftarrow Bx \\ By &\leftarrow By \end{aligned}$$

You also need to specify several other parameters. These facts will help get you there:

- Around Vancouver, one degree of latitude or longitude is about 10^5 meters. That will be a close enough conversion as we're estimating error…
- While GPS can be much more accurate, we assume it is accurate to about 5 metres. Artificial noise has been added to the data to make this problem possible. (This implies a value for `observation_covariance`.)
- I have no prior knowledge of where the walk started, but the default *(https://en.wikipedia.org/wiki/Null_Island)* is probably very far off. The first data point (`points.iloc[0]`) is probably a much better guess. (`initial_state_mean`)
- The predictions for latitude and longitude implicitly assume that the walking pace is constant. If I speed up or slow down, the prediction has no way to take that into account. So we expect these predictions to be good, but not perfect. (`transition_covariance`)
- The predictions for Bx and By are "they don't change" which probably isn't a great prediction. (`transition_covariance`)

Use these assumptions to create a Kalman filter and reduce the noise in the data. Create a new DataFrame with this data and calculate the "true" distance I walked.

Print the distance in metres, again to two decimal places. There is no "correct" answer to give here: closer to reality is better.

```
smoothed_points = smooth(points)
smoothed_dist = distance(smoothed_points)
print(f'Filtered distance: {smoothed_dist:.2f}')
```

Final output should be as in the provided `calc_distance.txt` (but with a more realistic filtered distance).

Viewing Your Results

Once you create the smoothed track in a DataFrame, you can call the provided `output_gpx` to save it to a file. **Create a GPX file** `out.gpx` as a side-effect of your program.

A GPX can be viewed online with MyGPSFiles *(http://www.mygpsfiles.com/en/)* , or with GpsPrune *(https://activityworkshop.net/software/gpsprune/download.html)* (Ubuntu package `gpsprune` or download the Java), or in Windows with GPS Track Editor *(http://www.gpstrackeditor.com/)* .

Have a look at your results: are you smoothing too much or too little? **Tweak the parameters** to your Kalman filter to get the best results you can.

So you can see the kind of results you might expect, I have included a screenshot of a track (but not one of the ones you have) and its smoothed result in MyGPSFiles.

Questions

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

1. When smoothing the CPU temperature, do you think you got a better result with LOESS or Kalman smoothing? What differences did you notice?
2. In the CSV files, you might have also noticed other data about the observations: accelerometer (acceleration in x, y, z directions), gyroscope (rate of turning, pitch, roll, yaw). How could those have been used to make a better prediction about the “next” latitude and longitude?
3. [Optional, because it takes more linear algebra than is prerequisite to this course] The transition matrix for the GPS Kalman filter had a sub-matrix that was `[[5e-7, 34e-7], [-49e-7, 9e-7]]` (which was found by regression to the true values). If you multiply that by `3e5`, it looks *a lot* like a rotation matrix by 285°. The magnetic declination *(https://www.magnetic-declination.com/)* around Vancouver is about 15°. Explain that part of the transition matrix.
4. [Optional, because it's interesting but not *that* interesting] In your `calc_distance.py`, temporarily set the `transition_covariance` values for the latitude and longitude to be very small (like 1/1000 of the corresponding `observation_covariance` values). This will give you a track in `out.gpx` that is basically "just the predictions, ignore the measurements". Have a look at the tracks on a viewer that includes a map (so you can see the track relative to roads/sidewalks). What can you infer about the predictions we're making?

Submitting

Submit your files through CourSys for Exercise 3.

Updated Wed May 10 2023, 20:27 by ggbaker.