Exercise 3

Due Friday June 04 2021.

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:

$$
\begin{aligned}
\mathit{temperature} &\leftarrow 0.97 \times \mathit{temperature} + 0.5 \times \mathit{cpu\_percent} + 0.2 \times \mathit{sys\_load\_1} - 0.001 \times \mathit{fan\_rpm} \\
\mathit{cpu\_percent} &\leftarrow 0.1 \times \mathit{temperature} + 0.4 \times \mathit{cpu\_percent} + 2.2 \times \mathit{sys\_load\_1} \\
\mathit{sys\_load\_1} &\leftarrow 0.95 \times \mathit{sys\_load\_1} \\
\mathit{fan\_rpm} &\leftarrow \mathit{fan\_rpm}
\end{aligned}
$$

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 5 m *(http://www.gps.gov/systems/gps/performance/accuracy/)* but it seems unlikely that your phone will do that well.

I recorded some tracks of myself walking with GPS Logger for Android *(https://play.google.com/store/apps/details?id=com.mendhak.gpslogger&hl=en)* . They are included as `*.gpx` files.

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

```
<trkpt lat="49.28022235" lon="-123.00543652">…</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 path of a `.gpx` file on the command line, like this:

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

Read the XML

Since the input 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 from each `<trkpt>` element. We can ignore the elevation, time, and other fields. **Create a DataFrame** with columns `'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.]

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:

```
points = read_gpx(sys.argv[1])
print('Unfiltered distance: %0.2f' % (distance(points)))
```

Kalman Filtering

For the Kalman filter, we need to specify several parameters. The units of measurement in the data are degrees latitude and longitude.

- 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 accurate to about 5 metres, the reality seems to be several times that: maybe 15 or 20 metres with my phone. (This implies a value for `observation_covariance`.)
- Without any other knowledge of what direction I was walking, we must assume that my current position will be the same as my previous position. (`transition_matrices`)
- I usually walk something like 1 m/s and the data contains an observation about every 10 s. (`transition_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.

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)
print('Filtered distance: %0.2f' % (distance(smoothed_points)))
```

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 GPX files, you might have also noticed other data about the observations: time stamp, course (heading in degrees from north, 0–360), speed (in m/s). How could those have been used to make a better prediction about the "next" latitude and longitude? [Aside: I tried, and it didn't help much. I think the values are calculated from the latitude/longitude by the app: they don't really add much new information.]

Submitting

Submit your files through CourSys for Exercise 3.

Updated Wed April 07 2021, 09:16 by ggbaker.