PP275 Spatial Data and Analysis, Fall 2019

Lab 7: Remote sensing, minimum distance & spatial Markov chains

Solomon Hsiang

University of California, Berkeley

About this lab

Due: Monday, November 14

- **NOTE:** Please include the lines

  ```
  import random
  random.seed(0)
  import numpy as np
  np.random.seed(0)
  ```

  at the top of your script so that the results from your random number generation will be reproducible. This also means that you should run the entire script top to bottom to generate results before submitting your Lab. This will prevent you from getting different random number results if, say, you ran a section of code multiple times for debugging.

- When referring to computer commands and files, `I will use this font`.

- You may talk with other students about the lab (I encourage you to help one another), but each student is responsible for doing all exercises in the lab themselves and turning in their own write up.

- **Bold text** indicates script output or responses to questions that should be included in your writeup. As always, your code should be included as well. For figures, additional directions (e.g. label your plot) are not bolded, so please read carefully.

- When you are done with the lab, please restart the kernel and run all the codes one more time (press the "≫" button), so that the grader will see a clean notebook. Don't forget to save! Please submit this lab to bCourses as "YOURCAL1ID.ipynb". This will facilitate anonymous grading.

- The class just recently transitioned from Matlab to Python. This is a newly developed assignment, so if you think there is an error or something is unclear, let us know right away. That will be extremely helpful to us and your fellow students.

- This lab requires that you install the following python packages (both are already provided by anaconda so you should already have them if you are using Anaconda python):

1. `requests`, through use of the command line call `conda install -c anaconda requests`
2. `scikit-image`, through use of the command line call `conda install -c conda-forge scikit-image`

- This lab requires that you download the following files and include them in the same working directory as your python script.

1. `clean_line_breaks.py`, which can be downloaded from the bcourses page

- You will also need access to the internet.

1 In the cloud(s)

A lot of data is shared and accessed via *application programming interfaces*, known as APIs. This way, instead of downloading a lot data to a single machine, a computer connected to the internet can query another machine (also on the internet) and retrieve smaller pieces of data right when it needs it. This approach to data storage and access is becoming increasingly popular as government, organizations, and firms amass increasing large data sets that are expensive to store, index, and query (i.e. "big data" problems). Python has some nice interfaces for collecting data from the internet via APIs, so let's do a short problem to see how this works. You'll pull Landsat imagery data down from the cloud(s).
To do this, we'll use a NASA API that assists in the indexing of a Google Earth API, which actually provides the satellite imagery. We'll start by querying the NASA API and describing the image that we want. The NASA server will then compute a query that we can use to request the image we want from the Google server, which will provide us with the final data.
Querying an API might sound fancy if you've never done it, but its really quite simple. You provide your software, either a web browser or Python (or something else) a URL that contains information about the query. This tells your software to ping the server, and the server provides you back with the requested information. An important aspect of using APIs is that the query and the information returned must be in standard forms, otherwise the machines on either end will not know how to read the information from the other. In this exercise, the NASA API will return information stored in Java Script Object Notation[1] (JSON) and the Google API will return a Red-Green-Blue (RGB) 8-bit image. Luckily, Python knows how to translate both into objects that we are familiar working with (structures and arrays) so the translation is pretty easy.
The NASA API is described here

```
https://api.nasa.gov/api.html#earth
```

in the section "Imagery" where you can read the documentation. Basically, you just need to provide the API with the location of the image you want and a date, and it will send you back the image taken by Landsat 8 that is closest to that date (since the satellite doesn't pass over each location each day), along with an estimate for how much of the image is covered in clouds. Let's take a look at campus by using the coordinates:

```
Berkeley_lat = ``37.87''
Berkeley_lon = ``-122.26''
date = ``2014-05-21''
```

which you should set as strings in Python. The last thing you'll need for your query is an API key, which is just a string used to identify the user so the server providing the information on the other end can keep track of an individual's queries (some APIs use this information to restrict the number

[1] See `https://en.wikipedia.org/wiki/JSON` for background if you're interested.

of queries a user can make, for example). If you don't make a lot of mistakes, you might be able to complete this lab without registering for an API key. The NASA site allows a computer to make 30 queries per hour using the generic api key `DEMO_KEY`. But if you make more than 30 requests, you will need to register for your own key by going here:

```
https://api.nasa.gov/index.html#apply-for-an-api-key
```

in your browser and filling in basic information. After that, you will be able to make up to 1000 requests per hour. You must save your API key as a string in Python. It should look like a long string of numbers and letters, for example, it might look like:

```
my_api_key = ‘‘ihHSc8NcgsVdNKyu3OHIqeKBxsS8OkTmb48jJv8s’’
```

or if you are just using the demo key:

```
my_api_key = ‘‘DEMO_KEY’’
```

You are now ready to construct your first query to the NASA server. First you must construct a URL using the different variables you've stored up to this point. One way is to just build the URL as a string, using the standard format described by NASA. You assign certain fields the values you stored as strings using an '`=`' and seperating fields using '`&`'. The URL is built by simply concatenating several strings:[2]

```
base_url = ‘‘https://api.nasa.gov/planetary/earth/imagery/’’
image_query = base_url+‘‘?lat=’’+Berkeley_lat+‘‘&lon=’’+Berkeley_lon+‘‘&date=’’+date
image_query = image_query+‘‘&api_key=’’+my_api_key
```

This is just a URL, which you can and should try pasting into a web browser (try `print(image_query)` to see this). You will see that NASA sends back a text file in JSON format. But since we don't want to cut and paste back into Python, let's use the package `requests` to query the server and to parse the JSON reply.

```
import requests as re
r = re.get(image_query).
```

Note that making an API request in this format isn't really using the `requests` package as it was designed. Instead, try running the code

```
r = re.get(base_url, params={’lat’:Berkeley_lat,’lon’:Berkeley_lon,
      ’date’:date,’api_key’:my_api_key})
```

and confirm for yourself that this makes the exact same API request (to see this, run the code `r.url` after each request – the url is the same). Take a look at what came back from NASA and is now stored in the structure you named `r.content`, you should see the date and time of the flyover, a URL and an identification number. We only need the URL, which is just the string stored in `r.json()[’url’]`. This URL is a specially formatted query for Google, so we want to take this URL and use it in *another* query, but this time to the Google API. If you take a look at it, you'll see why we wanted NASA to do the formatting for us. Before doing it in Python, try cutting and pasting this URL from the window into your web browser. You should be able to see the satellite image that we are after, since the image file is all that Google sends back.
Now query the Google API from Python with the URL retrieved from the NASA API using the code

[2]If you didn't set the lon and lat values as strings, but instead used numeric values, recall that you will need to use the `str()` command to turn these values into strings for the URL.

below:

```
def download_file(url,local_filename):
      r = re.get(url, stream=True)
      with open(local_filename, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024):
                  if chunk:
                        f.write(chunk)
      return local_filename

nasa_reply_url = r.json()['url']
download_file(nasa_reply_url,''image.png'')
```

This program will save the RGB 8-bit image file as a PNG named `image.png` in the same directory as your python script (feel free to change this!). Now, let's read in the image we have downloaded using the package `scikit-image`.

```
from skimage import io
im = io.imread('image.png')
```

The elements in the array are 8-bit values and they can only take on integer values between 0 and 255. (You can change this, but then it can't be plotted as the same kind of image.) To display the satellite image, use the `matplotlib` imshow function we have used before:

```
import matplotlib.pyplot as plt
plt.imshow(im)
```

1. What is the size of the numpy array `im`?
2. The image array should have three dimensions, pixels run North-South and East-West, while there should be 3 values describing the third dimension of the array. The first layer (index=0) is the layer describing the intensity of red light observed by the satellite (known as the "red channel" or "red band"). The second layer is the green channel, and the third layer is the blue channel (hence, RGB). **Use `imshow` to plot three images of the three channels side-by-side in black and white.** These are essentially the three images you would take with a camera (from space) if you were using a red, green, or blue filter. Make sure to label and title these images properly. **Is it easy to tell them apart?**
3. The image you are looking at shows a limited area that is $0.025° \times 0.025°$, but it's easy to request additional images via API so you can see more. Request the image directly south of the first image and the two images directly north of the first image and **concatenate (i.e. tile) them to make an image that is four times larger in terms of its coverage.**
4. You can plot the red channel vs. green channel values for each pixel separately with the command

   ```
   plt.scatter(im[:,:,0].flatten(), im[:,:,1].flatten(),alpha=0.1,s=1)
   ```

   where the position of each pixel indicates the intensity of the two different colors of light. Also compare the green to blue channels. If you like, you can compare all three channels in a three dimensional scatter plot.

   Using your larger image from question 1.3, try to isolate many of the buildings around Berkeley based on their visual properties. Note that many of the buildings in and around the campus are brighter (nearer to white) than surrounding areas that tend to be green and darker. Recall that bright colors result from a high intensity of all colors combining. So to identify buildings, look for pixels where the red, green and blue channels all take on values greater than 120 (out of a possible 255 using this 8-bit color system). **Adjust your two scatter plots so that pixels that are "bright" by this criteria (red, green and blue channels are all above 120) are colored blue and all remaining pixels are red.**

5. Again using the larger image from question 1.3, create a new array that is a mask for "bright" pixels where red, green and blue channels are all above 120. **Plot this mask using `imshow` in a panel next to your true color image. Did you do a good job isolating the buildings in the image?** If you think you can do better, try varying the brightness cutoffs of the different channels to see if you can improve performance. *Extra optional challenge:* Can you plot the building-pixels you've identified in their normal full color, but plot the rest of the image in white, i.e. erasing everything that's not a building?

2 How many cats could a camera catch if a camera could catch cats?

Counting events or populations that are sparsely distributed in space is difficult, but often we need such estimates for policy purposes and we must construct these estimates with limited resources. Here, we will use Monte-Carlo/Markov chain simulation to help us make estimates about a complex system with only sparse data; as well as to figure out the best way to make observations when we know that our observations will be limited due to cost.

Figure 1: Snow leopard habitat, a difficult region to sample completely. Photo by Steve Winter.

Snow leopards are notoriously difficult to track, they cover enormous ranges, are solitary and live in harsh environments. For these reasons, we have limited observations of snow leopards and directly observing complete snow leopard populations is extraordinarily difficult (see Figure 1). Despite this,

Figure 2: A snow leopard trips a camera trap. Photos by Steve Winter of National Geographic, whose camera traps shot 30,000 frames over ten months just to get a limited number of observations of this species.

we often need counts of these [and other] animals for policy purposes – for example, to test whether conservation or species-mangement policies are effective or not.

Imagine that we are trying to estimate the number of snow leopards in a region using a camera trap (they are expensive and that's all our researcher team can afford). A camera trap is a stationary piece of equipment that will snap a photo every time an animal sets off the motion sensor (see Figure 2). This will give us a count of how often animals are observed at a specific location (assume for now that we cannot distinguish one animal from another in the photographs). We will use simulations to answer two questions. (1) Given that a camera trap in a certain location records a certain number of sightings over a certain time interval, what is the best estimate for the number of animals roaming the region? (2) If we are restricted to using one camera trap to estimate the number of animals in a region, where should we leave the trap?

We will use Monte-Carlo/Markov chain methods to answer Questions (1) and (2) above. This means we will create a model of our world, containing randomness structurally similar to what we expect to find in nature, and will run *many* simulations with it. We will use the results of these simulations to infer something about the world, given our observations[3]. Specifically, we will run our simulation many times to generate probability density functions for the distribution of expected snow leopard observations. For Question (1), given that our camera trap is at position x for time T, we'll compare how probable it was that we observed M snow leopards if there were N wandering around compared to if there were N'. For Question (2), we'll compare how noisy our observations are (for a given N and T) when our camera is at position x vs. x', so that we can optimize the placement of our camera.

[3]We will be using maximum likelihood methods, although the emphasis here is not on the statistical technique but rather its interpretation.

Setup

The world we are simulating is 10 km x 10 km grid, with each square kilometer cell indexed by its row and column. Imagine that the habitat beyond this 100 km^2 region is uninhabitable by snow-leopards (i.e. maybe it's not cold enough), so they always remain in this region. A snow leopard's position is only described by which of the square-kilometer cells it is currently in. Begin by planning (on scrap paper) a script to simulate a single snow leopard moving around inside this region. The time-step of our model will be days. Assume that snow leopards never stay in the same cell two days in a row, and they do not travel far in absolute distance each day (since they explore the cell they are in rather extensively). Each day, every snow leopard can move into one of the eight cells surrounding its current position (up, down, left, right or diagonals) so long as those cells are in the 10 km x 10 km habitat region. Of the adjacent cells that a leopard can move into at any time-step, it choses one at random, placing equal probability on each of the available cells. For example, a snow leopard in the middle of the habitat at $t = 0$ will move into any of the eight surrounding cells at $t = 1$ with probability $\frac{1}{8}$. A snow leopard in the corner of the habitat will move into one of the 3 adjacent, habitable cells in the next period, each with a probability of $\frac{1}{3}$. Our research team has been awarded grant to install a single camera trap at position (3,3) for one-hundred days.

Implementation

Begin by breaking the problem up into sub-problems. Each piece of code for each subproblem should be written separately and tested as it is written. This will reduce the amount of time it takes to de-bug your scripts. Plan out your entire program to be as simple as you can make it, test your code frequently as you write it, and be methodological about searching for errors when Python tells you something is wrong. When Python reports an error, be patient and read what Python is telling you is wrong; the error statements are sometimes surprisingly precise.

1. Plan out your program for simulating a single snow leopard moving around the 10 km x 10 km range for a small number of days (< 10). Think about how you will use matrices to keep track of the animal's position, how you will keep track of time, how you will randomly assign the animal to move into one of the adjacent cells[4] and how you will restrict the animal from moving outside of the habitat. This is not a trivial exercise, so plan carefully. Also, be aware that we will eventually be running this program many times ($\sim 10,000$) so it's not a bad idea to think about ways of reducing the number of calculations that you ask the computer to do (eg. the number of "if" statements) and being selective about what information you choose to store[5]. Finally, remember that telling Python to display information as the program runs will be tedious; only ask it to report the time-step every ten or fifty steps.

   Think about ways of using the different dimensions of arrays to store different types of information. Also, it's probably a good idea to write functions wherever you can for operations you will do over and over again (like choosing which cell the snow leopard should move into). I strongly recommend that you plan out your program with heuristic diagrams for how you store information, index things, where functions are used and the order in which your script will make decisions.

   Write a script (perhaps with functions) that will make a snow leopard wander for ten days. **Verify that your code is working properly by turning in a figure illustrating a snow-leopard's path during 10 days (output from one simulation).**

2. **Run your code for 100 days and make a plot of the snow leopard's path in the 10 km x 10 km space over the entire study.**

[4]Hint: to generate a random number between 1 and 8, use `np.random.randint(1,9)`.

[5]For example, recording whether a snow leopard is in each cell at every time-step is 100 bits of information for each period; 100 x 100 days = 1 x 10^4 bits for the whole study; and 1,000 x 10^4 = 10^7 bits if we re-run the simulation 1,000 times to generate a single probability distribution.

3. Once you have code working for one snow leopard, adjust your code so that it can be run for N snow leopards walking around simultaneously (this is where the ability to add an extra dimension to your data structures is convenient). Assume that two snow leopards can inhabit the same 1 km x 1 km cell simultaneously. Because we'll want to vary the number of snow leopards several times, you should define a parameter N (or something similar) at the beginning of the script, and then use this parameter to define how large your data structures are and how many times to call functions (since you'll need to do everything you did for your one-snow-leopard-run N times now). Check that it works for $N = 2$ before trying higher numbers (a good general rule is to always do the simplest test first). Don't turn in this adjusted script just yet, but do **turn in a figure with three subplots showing the paths of three separate snow leopards over the course of the study to show that it works.**

4. When you are confident that your simulation is doing what you want it to do, "set up" our simulated camera in cell (3,3). This means every period, keep track of whether there is a snow leopard in that cell. Because camera traps are elaborate and must be reset after taking a shot (something our field research team can only afford to do daily for this project), we can't take photos of two snow leopards in the same day, so if there is more than one animal in cell (3,3) on a given day, we only take a photo of one. (Alternatively, you can think of our camera as always only recording whether there was at least one snow leopard in the cell on a given day). For every day of the 100-day study, you should have either a zero or a one recorded for whether there was a snow leopard at (3,3). **Demonstrate that your camera is "working" by showing two subplots in a single plot. The first is the paths of 3 animals over the study, with the location of the camera marked with a "*". The second is a time series of the observations made by the camera. Confirm that your record shows the same number of observations as your map of snow leopard paths.**

5. If your camera is working properly, we're ready to begin our Monte-Carlo simulations. The idea is this: *during our 100-day study, our camera recorded five snow leopard sightings* and we want to estimate what this means in terms of the number of snow leopards wandering around our 10 km x 10 km range. Clearly there is at least one. However, if there was just one, would it be probable that we would observe it five times over 100 days? Would it be more or less probable to have made five sightings if there were two or three wandering around? If there were fifty snow leopards, intuition suggests that *only* five observations would be very unlikely. So in between $N = 1$ and $N = 50$, there is some N that maximizes the probability (likelihood) that we would have observed $M = 5$ sightings. Our goal is to find this $\hat{N}$.

   This general approach is known as *maximum likelihood estimation*, although often a researcher will *assume* a probability distribution function describing the likelihood of observed events rather than constructing a simulation to *build* these distributions "from scratch" (as we are doing here). Formally, we are looking for

   $$\hat{N} = argmax_N \text{ Prob(number of observations} = M = 5|N).$$

   To construct this estimate, take the following steps

   (a) Pick an N' to start with (1 is a good choice), this is the number of snow leopards that we'll assume are wandering around.

   (b) Based on this value for N', run the 100-day simulation 1,000 separate times. After each run, record the number of times that the camera snapped a photo of a snow leopard (this is M for that run).

   (c) A histogram of M across these 1,000 simulations is an approximation of the probability distribution of M conditional on N'. Since we know $M = 5$ in the actual study, compute the probability that we would have observed $M = 5$ if $N = N'$. Record this probability (written as $\text{Prob}(M = 5|N = N')$).

(d) Add one to N' and repeat (b)-(c). Do this for many values of N' (until $N' = 12$ or so is a good idea).

(e) Plot the probability of obtaining five snow leopard sightings ($M = 5$) for each value of N' between 1 and 12 (or whatever maximum value you chose). The value of N' that has the highest likelihood is $\hat{N}$, your best estimate for N.

What is the most likely number of snow leopards that are in our study area?

Turn in a plot that compares all the histograms you estimate in (c) for each value of N' (e.g. you could make multiple subplots using `plt.hist` or make a 3-D plot of likelihoods over M and N' using methods we have seen before). Also turn in a plot of the probability that $M = 5$ against N' that you made in (e).

IMPORTANT: Before you implement the above procedure, make sure to simplify your script as much as possible. Earlier, you were recording the position of each snow leopard every day. We did this because we had to verify that our code was working correctly. But for our Monte-Carlo procedure, each day we only need to record whether there was a snow leopard in (3,3) or not (using a one or zero). Recording the position of all the animals each time-step will slow down the program unnecessarily, so disable whatever parts of the script that record this information (you should comment it out so that I can see where it used to be).

ALSO NOTE: If you try to store too much information, e.g. try to define a matrix that has trillions of elements, Python may crash. But it may not tell you that it has crashed (yes, this is annoying). Instead it will just look like it is doing computation. Your Monte-Carlo runs may take some time to finish, so it may be hard to tell whether Python has crashed or is just doing a long computation. For this reason, it's always a good idea to ask Python to print *something* every so often (eg. to display which iteration it has completed once every fifty iterations) just so you can verify that your program is running properly.

6. The above procedure makes it clear that uncertainty is a serious issue, especially when our observations are sparse. A clever researcher, however, can design experiments in ways that reduce this uncertainty. Is it possible that by changing the location of the camera-trap, we could get a more precise estimate of N? Use your simulation to answer consider this question. Determine whether our original location, (3,3), or (1,1) or (5,5) produce more precise or noisier samples. Do this by running the simulation 1,000 times for $N = 3$ snow leopards for each of these camera positions (don't forget that you've already done this for (3,3)). **Compare the variances of the distributions for $M_{(x,y)}$, the number of sightings made by the camera at (x, y). Which location provides an expected distribution with the smallest variance?** (The function `np.var()` will be useful). **Which location produces a distribution with the largest variance? Interpret what this means. If you had done this simulation before the field experiment was conducted, where would you recommend placing the camera? Why?**

3 As the crow flies and as the railcar rolls...

Our apologies, we are still in the process of converting some of the cleaning code for this exercise from Matlab to Python. However, the first two problems hopefully should be enough to keep you entertained for now. Please check back in in a day or two for exercise 3 (which will be posted in an updated version of this pdf).
The government of South Africa is considering building a major grain storage facility to buffer the national economy from the impacts of future climate change. There are many parameters to consider when deciding where to place the facility, but transport costs is a major factor. Your job is help the government understand the transport distances involved with placing a facility at different nodes in

the nation's rail network. Because grain must be carried by train, your analysis will focus on minimum distances along the railroad network.
As policy-makers consider different facility locations, they will need to see the distribution of transport distances for that location. However, because they do not yet know all the factors they will ultimately consider, they have not asked you to optimize the placement of the facility. Instead, they have asked you to develop an interactive tool that will allow them to quickly analyze any potential site by simply specifying its location. The tool's specification is that a user must be able to tell your program a location and your program should output (1) a map depicting transport distances to all other nodes in the network and (2) a histogram of these distances. (For this problem it is okay to compute distances in degrees, but you can convert to km if you like).

1. Begin by downloading the shapefile for the railroad network in South Africa at `www.diva-gis.org`. Import the data the usual way with using the command

   ```
   import geopandas as gpd
   df = gpd.read_file('ZAF_rrd/ZAF_rails.shp')
   ```

   and make sure that you understand the structure of the data. Each of the 497 objects in `s` can be called using the same indexing as earlier shapefiles, however now each object is a polyline describing a railroad track linkage between two stations (nodes) in the railroad network.

   You can plot a single railroad-link of the network by running the following code.

   ```
   index = 0 ### feel free to change this
   plt.plot(df['geometry'][index].xy[0], df['geometry'][index].xy[1])
   ```

   Another simple way to do this that we have seen before is by just running
   `df['geometry'][index]`

   However, there are sometimes issues with data that you get from public sources, since not everybody in the world has the same standards. It turns out that there are some line breaks hidden at strange spots in this data where they shouldn't be (it's very hard to find them by hand), and if you leave them in the raw data you'll have trouble writing code that works on the entire data set. To save you time on this data-cleaning issue, I've written a short script that will clean the data for you by hunting down all the rogue errors. To clean the data, run the following lines of code:

   ```
   from clean_line_breaks import clean_line_breaks
   df = clean_line_breaks(df)
   ```

   Using the script `clean_line_breaks.py` included with this lab (feel free to read through the script so you understand what it's doing, it might be useful to write something similar when you are cleaning data for your final project). This will overwrite the contaminated data with the clean data. If you run it a second time with the clean data, nothing should change since there should not be any extraneous errors remaining after cleaning.

   Plot the entire network. (Don't turn in this map just yet)

2. We don't have a separate list of the railroad station locations, but we can figure out where they are by simply inspecting the ends of each link. For example if you type

```
line_lats, line_lons = df['geometry'][4].xy
plt.plot(line_lats[0],line_lons[0], color='blue', marker='o')
```

you will plot the station at the start of the fifth link as a closed blue circle and if you type

```
plt.plot(line_lats[-1],line_lons[-1], color='red', marker='^')
```

you will plot the station at the end of the fifth link as a red triangle (see the matplotlib documentation for more ideas).

Add to your loop commands that will map each start and end node using these symbols (overlaid on the network of railroads. **Turn in this plot.**

3. You will notice that many stations are plotted [at least] twice, since a station may be both a start and end node for a railroad segment. But in computing minimum distances between locations, you need to consider each node only once. This means you need to create a list of stations where each station only appears once. You can then use this list of stations to compute an adjacency matrix (and the associated distance matrix).

   To assemble this list of station locations, you first need to be sure not to leave out any stations, so start by creating a list (which may include duplicates) of all start and end stations (for each link). For each station we need a lat and lon coordinate. After you assemble this list, you'll check it for duplicate stations. First assemble the list of coordinates for start and end stations:

```
start_lats, start_lons = [], []
end_lats, end_lons = [], []
for i in range(len(df)):
     line_lats, line_lons = df['geometry'][i].xy
     start_lats.append(line_lats[0])
     start_lons.append(line_lons[0])
     end_lats.append(line_lats[-1])
     end_lons.append(line_lons[-1])
```

   Then concatenate the lat vectors and lon vectors (vertically) so that you have two long vectors, one lats and one lons. These two vector can then be concatenated (horizontally) so that each row describes the lat and lon of each station. You could do this with the single command:

```
list_with_duplicates = np.array(list(zip(start_lats+end_lats, start_lons+end_lons)))]
```

   The reason you must create a matrix with two columns (rather than two one column matrices like we normally work with) is that to identify unique stations we want to match duplicate stations based on both their lat and lon coordinate. To pull out a list of unique station positions, use the command `np.unique` with the `axis=0` option specified:

```
stations = np.unique(list_with_duplicates, axis=0)
```

   The matrix `stations` now has only one row for each station, with the first column describing the latitude of each station and the second column describing the longitude of each station. For the rest of the problem, treat the index of a station in this list as the unique index for that station. Check for yourself that you have gotten all the stations by plotting them on the rail network using only the matrix `stations`. **How many unique stations are in the network?**

4. Now that you have a list of the station positions, you are ready to compute the adjacency matrix and the corresponding distance matrix. Begin by first constructing the adjacency matrix L describing which stations are directly linked to other stations. Start by constructing a large matrix of zeros where each station gets a row and column. Then go through the rail links one by one and determine which station is the start station, which is the end station, and add a one to the matrix to record that the two are stations are connected. Once you have added a one to the adjacency matrix for each rail link then you know that you have the entire matrix since there can't be any more links. The indexing here gets a little tricky here because you have to search through the nodes to find which one is at the start/end of each link:

   ```
   L = np.zeros((len(stations),len(stations)))
   for i in range(len(df)):  ### index of link
        line_lats, line_lons = df['geometry'][i].xy
        for j in range(len(stations)):  ### index of node
             if np.sum([line_lats[0], line_lons[0]] == stations[j,:])  == 2:
                  start_j = j
             if np.sum([line_lats[-1], line_lons[-1]] == stations[j,:])  == 2:
                  end_j = j
        L[start_j,end_j] = 1
        L[end_j,start_j] = 1
   ```

   Once this runs, **check that the adjacency matrix looks like you expect using `imshow`.** For this adjacency matrix, notice that we did not fill in the diagonal, i.e. we do not list that each location is connected to itself.

   Now adjust this code to build the distance matrix D, where instead of using a one/zero to denote whether two stations are connected, you list the distance between two locations if they are connected (computed along the rail link that the connects them) and infinity if they are not connected (you should not use zero to denote that two stations are not connected because you might later confuse this with zero distance, i.e. that the two stations are very close together). Hint: you may wish to write a function that computes the travel distance along a rail link – this will likely save you time since each rail link is a polyline with a different number of segments describing it, and you will have to replicate this calculation over 400 times.

   Visualize D using `imshow`, does it look the way you expected?

   For the following parts of the problem (building the program for the South African government) you will use D as the starting point, i.e. you don't need to recompute D each time the program runs (you may want to save it and have your program load it each time it runs – try pickling it).

5. Now that you have simplified the network into its distance matrix, implement the algorithm to compute the shortest distance from an arbitrary station `k` (the potential location for the grain facility) to all other stations in the network. Follow the steps in the algorithm covered in class. Check that the output is correct for some short and obvious distances. Once you think it is working correctly for longer distances, check by having it **plot the shortest path between stations 1 and 200 overlaid on the complete rail network**. Your code should output a single vector describing the distance from `k` to each station along the shortest path.

6. Finally, build the user interface that the South African government requested. When your program runs, it should start by prompting the user to input a potential station site using the `input` function:

   ```
   k = input('Station for facility?  Enter 1 to '+str(len(stations))+':')
   ```

It should then compute the minimum distance for station `k` to all other station locations. It should then output a plot with two subplots. The first should show each station as a marker overlaid on the rail network, with stations colored based on their distance from the grain facility. The second subplot should show a histogram of these distances. Make sure these plots are labeled well so that when government planners work with your program, they know exactly what they are looking at in these plots.

Check that your code works by running it three times, trying out grain facilities at stations 100, 250, and 360. Turn in all three of these final plots.