PyOSMRoute

Installation

The **Python 3** package `pyosmroute` depends on two Python modules: `numpy` and `psycopg2`. Both are available via `pip` (may be `pip3` on Mac).The interface to the `pyosmroute` package is the package itself, imported like any Python module:

Setting up the OSM Database

The second input needed by `pyosmroute` methods is a Postgres database created by osm2pgsql. The input for this is a .pbf or .osm file acquired from the various distributors of OSM data on the web. It is accessible by country here. If multiple countries are desired, it is necessary to use the Osmosis tool to merge together bordering countries (or overlapping PBF files). This can be done using the following command:

```
osmosis --readpbf my_file.pbf --read-pbf my_other_file.pbf --merge --write-pbf combined.pbf
```

Adding the PBF file to the database uses the following command:

```
osm2pgsql -s -C 1600 -H localhost -d DB_name -U DB_user -W combined.pbf
```

The most important argument is the **-s** parameter, which leaves a copy of the raw nodes/ways in the database. This data is used by the routing and matching function extensively. The **-C** parameter just specifies an amount of memory to be used (in MB). The **-W** parameter prompts for a password, so it may be desirable to have no password on this database for the preparation phase.

From the Command Line

It's possible to use the `matchcsv.py` file (in this folder) from the command line to match CSVs and write/aggreate output. This is mostly useful for running the script from R, which has more interctive capabilities for mapping. The usage for the command line interface looks like this:

```
> python3 matchcsv.py --help

usage: Run route matching on CSVs containing date/time, latitude, and longitude information.
       [-h] [-r] [-o OUTPUT] [--writepoints] [--writesegs]
       [--processes PROCESSES] [--chunksize CHUNKSIZE] [-v]
       infile

positional arguments:
  infile                Directory containing or a single CSV file with GPS
                        Time (UTC), Latitude, and Longitude columns.

optional arguments:
  -h, --help            show this help message and exit
  -r, --recursive       Walk directory recursively
  -o OUTPUT, --output OUTPUT
                        Specify summary output file, use '.csv' or '.tsv'
                        extension.
```

```
  --writepoints         Write point matches to FILE_osmpoints.csv
  --writesegs           Write all segment matches to FILE_osmsegs.csv
  --processes PROCESSES
                        Specify number of worker processes.
  --chunksize CHUNKSIZE
                        Specify the multiprocesing chunksize parameter.
  -v, --verbose         Verbose debug output.
```

Note that for this to work, you'll have to have your `dbconfig.py` setup (see below).

From R

The command line interface is best run straight from the `testmatch.R` or `testmatch.Rmd`, which are already setup to handle the output and display it complete with scalebar and basemap. This is the preferred method for debugging the script. There is an RStudio project already setup for the directory.

From Python

The package is loaded in Python like any normal Python module:

```
import pyosmroute as pyosm
```

The DataFrame

At the heart of passing large amounts of data is the `DataFrame` class (here it would be accessed as `pyosm.DataFrame`). Usually this is accessed by the `read_csv()` function, which returns a `DataFrame`.

```
gpsdata = pyosm.read_csv("example-data/test/2016-03-02 17_37_41_Car - Normal Drive_Android_start.csv", s
gpsdata.head()
```

Time (UTC)

Latitude

Longitude

2016-03-02 17:37:51

45.0917807

-64.36975685

2016-03-02 17:37:53

45.09165327

-64.36969589

2016-03-02 17:37:54

45.09168978

-64.36977029

2016-03-02 17:37:55

45.09167523

-64.36973768

2016-03-02 17:37:56

45.09165241

-64.36973143

2016-03-02 17:37:57

45.09169459

-64.36969024

The `DataFrame` class supports most of the methods that the `pandas.DataFrame` class contains, but is much more lightweight (running a `pyosmroute.DataFrame` is about twice as fast in this context as the `pandas` version). Some quick examples:

```
"Latitude" in gpsdata
```

```
True
```

```
gpsdata.iloc[3]
```

```
{'Latitude': 45.09167523,
 'Longitude': -64.36973768,
 'Time (UTC)': '2016-03-02 17:37:55'}
```

```
gpsdata.iloc[3, :]
```

Time (UTC)

Latitude

Longitude

2016-03-02 17:37:55

45.09167523

-64.36973768

```
len(gpsdata)
```

```
176
```

```
for colname in gpsdata:
    print(colname)
```

```
Time (UTC)
Latitude
Longitude
```

```
for row in gpsdata.itertuples():
    # do something with each row...
    pass

# set columns
gpsdata["newcol"] = [lat+5 for lat in gpsdata["Latitude"]]
gpsdata.head()
```

Time (UTC)

Latitude

Longitude

newcol

2016-03-02 17:37:51

45.0917807

-64.36975685

50.0917807

2016-03-02 17:37:53

45.09165327

-64.36969589

50.09165327

2016-03-02 17:37:54

45.09168978

-64.36977029

50.09168978

2016-03-02 17:37:55

45.09167523

-64.36973768

50.09167523

2016-03-02 17:37:56

45.09165241

-64.36973143

50.09165241

2016-03-02 17:37:57

45.09169459

-64.36969024

50.09169459

```
# delete columns
del gpsdata["newcol"]
```

The `DataFrame` object does quite a bit, but at heart it's just a collection of related information. This implementation is not really designed to be used in an analysis environment but is lightweight and is more or less interface compatible with the `pandas` version such that `pandas` objects can be used as input without breaking the code.

Loading the OSM Database

This database is accessed via the `PlanetDB` class, usually instantiated by `get_planet_db()`. It is possible to specify login credentials within `get_planet_db()`, however it is more efficient to specify them in the `dbconfig.py` file in the `pyosmroute` folder. Contained in the folder is a `dbconfig.example.py` that specifies the format:

```
DB_USER = "osm"
DB_PASSWORD = "osm"
DB_HOST = "localhost"
DB_NAME = "osm"
```

Obviously, you'll want to change this to fit the configuration of the target machine. You can then get a database object like this:

```
planetdb = pyosm.get_planet_db()
planetdb.connect()
planetdb.is_connected()
```

```
True
```

Don't forget to `planetdb.disconnect()` when you're finished. Connecting to and disconnecting from the database takes enough time that it's worth maintaining a single connection per process.

Road Matching Methods

There are two primary road-matching methods in the `pyosmroute` package: `on_road_percent()` and `osmmatch()`. The `on_road_percent()` function takes a `PlanetDB` and a `DataFrame` with the GPS information.

```
pyosm.on_road_percent(planetdb, gpsdata)
```

```
0.95454545454545459
```

If your `DataFrame` contains non-standard column names (i.e. not `Latitude` and `Longitude`), you can pass these in as parameters as well:

```
pyosm.on_road_percent(planetdb, gpsdata, latitude_column=1, longitude_column=2)
```

```
0.95454545454545459
```

It's also possible to specify a radius (the default is 15 metres):

```
[pyosm.on_road_percent(planetdb, gpsdata, radius=radius) for radius in (5, 10, 15, 20)]
```

```
[0.67613636363636365, 0.8125, 0.95454545454545459, 0.96590909090909094]
```

It may be worth cleaning points by distance to get this paramter, which can be done using the `cleanpoints()` function.

```
cleandf = pyosm.cleanpoints(gpsdata, min_distance=30)
[pyosm.on_road_percent(planetdb, cleandf, radius=radius) for radius in (5, 10, 15, 20)]
```

```
[0.58620689655172409,
 0.7931034482758621,
 0.96551724137931039,
 0.96551724137931039]
```

The `cleanpoints()` function also takes some other parameters involving max and min velocities and lat/lon columns like `on_road_percent()`. This function is used internally in the `osmmatch()` function; is it not necessary to clean data before putting it into the function.

Notes on Logging

The `pyosmroute` module uses Python's `logging` module to do its logging, but won't do any logging unless you explicitly tell it to do so. Now we can see a little more of what the `cleanpoints()` function is actually doing. This is logged to `sys.stderr` by default, but you can pass in any keyword arguments to be passed to `logging.basicConfig()`.

```
pyosm.config_logger()
cleandf = pyosm.cleanpoints(gpsdata, min_distance=30)
```

Map Matching

Map matching is accessed via the `osmmatch()` method. Its basic usage and ouput are straightforward (except `xte`, which stands for cross track error, or distance from the route).

```
stats, points, segs = pyosm.osmmatch(planetdb, gpsdata)
stats
```

```
{'cleaned_points': 29,
 'gps_distance': 959.58746357071789,
 'in_points': 176,
 'match_time': 0.289517879486084,
 'matched_points': 29,
 'matched_proportion': 1.0,
 'mean_xte': 3.896052922553324,
 'result': 'ok',
 'segment_distance': 1014.4255969295274,
 'started': '2016-03-10 05:59:23 +0000',
 'summary_time': 0.007563114166259766}
```

The `points` and `segs` output are both `DataFrames`. The `points` output describes the specific segment matched to each GPS point that was used to perform matching. Columns are described here:

alongtrack: How far from **node1** towards **node2** the nearest point on the segment is.

bearing: The bearing of the segment (from **node1** to **node2).

dist_from_route: How far the point on the route is from the actual GPS point.

distance: The distance from **node1** to **node2**.

name: The value of the *name* tag. Useful for debugging.

node1, **node2**: The OSM ID number for the node at the start and end, respectively.

oneway: `True` if the route can only be traced from **node1** to **node2**.

segment: Which segment in the **wayid**.

typetag: The value of the *highway* tag for **wayid** (e.g. `motorway`, `trunk`, etc.)

xte: The cross-track error of the GPS point with regard to **node1** and **node2**. Mostly the same as **dist_from_route**.

p1, p2: The locations of **node1** and **node2**.

pt_onseg: The location of the GPS point if it were on the segment.

gps...: All the columns passed into the original `DataFrame`. Guaranteed are **gps_Latitude**, **gps_Longitude**, **gps___datetime** (as a Python `datetime` object), **gps___bearing, gps___distance, and** gps___original_index**, which refers to the row index of the GPS point with regard to the original `DataFrame`.

waytag...: Values of the **wayid** tags.

```
points.head()
```

alongtrack

bearing

dist_from_route

distance

name

node1

node2

oneway

segment

typetag

wayid

weight

xte

p1_lon

p1_lat

p2_lon

p2_lat

pt_onseg_lon

pt_onseg_lat

gps_Latitude

gps_Longitude

gps_Time (UTC)

gps___bearing

gps___datetime

gps___distance

gps___original_index

gps___rotation

gps___velocity

waytag_lanes

waytag_source

waytag_name

waytag_highway

waytag_surface

2.23854439378

74.6450684114

15.8905509385

3.1957372431

None

3624069719

3624069724

False

20

unclassified

357012198

1

15.8905507538

-64.3697307537

45.091637562

-64.3696914973

45.0916451723

-64.3697032554

45.0916428929

45.0917807

-64.36975685

2016-03-02 17:37:51

66.6926429519

2016-03-02 17:37:51

nan

0
1.92087908237
1.26275096175
2
NRCan-CanVec-10.0
unclassified
paved
0.723006684459
348.718992698
6.67381561091
2.73967742269
None
3624069825
3624069834
False
40
unclassified
357012198
1
6.67381558494
-64.3693023471
45.091898403
-64.3693091743
45.0919225655
-64.3693041488
45.0919047795
45.09189303
-64.36938752
2016-03-02 17:38:16
114.714620011
2016-03-02 17:38:16
31.5687740437
21
2.65564234267
1.25486265933
2

NRCan-CanVec-10.0

unclassified

paved

31.0891424822

341.476092119

1.59826207608

38.7551944322

None

2542694962

3624069741

False

5

service

247371551

1

1.59827339691

-64.3690744445

45.0913455811

-64.3692312904

45.0916760572

-64.3692002652

45.0916106867

45.09160612

-64.36921957

2016-03-02 17:38:28

164.951409631

2016-03-02 17:38:28

34.5198996316

33

3.20753363665

3.45601032391

service

-0.0

341.476092119

7.94447404041

38.7551944322

None
2542694962
3624069741
False
5
service
247371551
1
7.94445882325
-64.3690744445
45.0913455811
-64.3692312904
45.0916760572
-64.3690744445
45.0913455811
45.09132275
-64.36917034
2016-03-02 17:38:35
175.657759108
2016-03-02 17:38:35
31.7454619601
40
0.158127083996
5.32494236069
service
19.6727191847
356.365949552
2.01841889933
23.1416456661
None
2542694952
2542694954
False
2
service
247371551

1

2.01841957925

-64.3691163959

45.0908576956

-64.3691350808

45.0910653947

-64.3691322799

45.0910342606

45.09103311

-64.36915794

2016-03-02 17:38:40

166.848934639

2016-03-02 17:38:40

32.2212499875

45

-1.28637246974

1.65780809845

service

14.8047532051

340.698757115

5.99144896527

30.5303607331

None

2542694950

2542694952

False

1

service

247371551

1

5.99143750947

-64.3689878469

45.0905985622

-64.3691163959

45.0908576956

-64.3690501828

45.090724221

45.09074203

-64.36897815

2016-03-02 17:39:15

124.202860318

2016-03-02 17:39:15

35.3100868925

79

-2.06725787148

1.38043172741

service

The `segs` output is similar, but contains a complete list of the segments that make up the route. Only several columns are new here.

direction: Either 1, -1, or 0. 1 if the way was traced from **node1** to **node2**, -1 if the route was traced from **node2** to **node2**, or 0 if neither.

nodetag. . . : Similar to waytag, but refers to the tags of **node2**. Signals, if in the OSM database, would be referred to here.

```
segs.head()
```

wayid

segment

node1

node2

typetag

name

distance

bearing

p1_lon

p1_lat

p2_lon

p2_lat

direction

waytag_lanes

waytag_source

waytag_name

waytag_highway

waytag_surface

357012198

20

3624069719

3624069724

unclassified

None

3.1957372431

74.6450684114

-64.3697307537

45.091637562

-64.3696914973

45.0916451723

1

2

NRCan-CanVec-10.0

unclassified

paved

357012198

21

3624069724

3624069728

unclassified

None

3.1726935802

76.2417253587

-64.3696914973

45.0916451723

-64.3696522409

45.0916519581

1

2

NRCan-CanVec-10.0

unclassified

paved

357012198

22

3624069728
3624069731
unclassified
None
3.1994999765
74.4015176906
-64.3696522409
45.0916519581
-64.3696129845
45.0916596952
1
2
NRCan-CanVec-10.0
unclassified
paved
357012198
23
3624069731
3624069735
unclassified
None
3.1889364185
74.6115184091
-64.3696129845
45.0916596952
-64.369573818
45.0916673055
1
2
NRCan-CanVec-10.0
unclassified
paved
357012198
24
3624069735
3624069740

unclassified

None

3.19081452708

74.4894312542

-64.369573818

45.0916673055

-64.3695346515

45.0916749791

1

2

NRCan-CanVec-10.0

unclassified

paved

357012198

25

3624069740

3624069743

unclassified

None

3.32184257816

73.2191900744

-64.3695346515

45.0916749791

-64.3694941374

45.0916836041

1

2

NRCan-CanVec-10.0

unclassified

paved

Road Matching Parameters

Match timestamped GPS points to roads in the OSM database. The matching is based a Hidden Markov Model with emission probabilities based on the distance to the road segment, and transition probabilities based on the difference between the GPS distance between two points and what the driving distance would be. This model is explained in Microsoft Research paper by Paul Newson and John Krumm entitled 'Hidden Markov Map Matching Through Noise and Sparseness'. This implementation of what is explained in the paper has two differences. First, emission probability has a component that is based on the difference between

the direction of the segment and the direction based on the two surrounding GPS points. Second, solving the Hidden Markov Model implements a 'lookahead' parameter, such that a next step can be chosen based on looking several steps into the future (see the viterbi_lookahead parameter). Driving distances are based on the pyroutelib2 library (https://github.com/gaulinmp/pyroutelib2), although considerable modifications had to be made to accommodate the needs of this function (mostly driving distances between adjacent segments and connecting to the database instead of reading a downloaded XML). The `osmmatch()` function takes the following arguments:

```
osmmatch(db, gpsdf, lat_column="Latitude", lon_column="Longitude", unparsed_datetime_col=0,
             searchradius=50, minpoints=10, maxvel=250, sigmaZ=10, beta=10.0, maxiter=1,
             minpointdistance=30, paramter_window=3, bearing_penalty_weight=1, viterbi_lookahead=1,
             lazy_probabilities=True, points_summary=True, segments_summary=True):
```

db: a PlanetDB object, as generated by get_planet_db() or by instantiating the object yourself.

gpsdf: A DataFrame (either `pyosmroute.DataFrame` or `pandas.DataFrame`, although the former is about twice as fast) of GPS points with at least date/time, longitude, and latitude columns.

lat_column: A column reference to gpsdf specifying which column contains the latitude information.

lon_column: A column reference to gpsdf specifying which column contains the longitude information.

unparsed_datetime_col: A column reference to gpsdf specifying which column contains the datetime information. The format must be '2016-03-01 20:59:46' (extra letters are stripped).

searchradius: The radius around each GPS point to search for roads. The original paper uses a radius of 200 metres, around 50 seems to work well though.

minpoints: After cleaning the GPS data such that there are data points ever 'minpointdistance' metres, if fewer than this number of points remain, no matching is performed. For debugging it's good to keep this low, but realistically any trip with less than about 20 points isn't worth matching.

maxvel: The maximum assumed velocity (m/s). This value is used to terminate routing between two points once the route would require the driver to travel over this speed. GPS points are noisy enough that this must be about twice the maximum speed. Anything under 250 caused unnecessary gaps during testing.

sigmaZ: The standard deviation of GPS error (metres). A higher value places less emphasis on how close a point is to any given segment.

beta: The standard deviation of the difference between the GPS distance and the driving distance between two points on a segment. Increasing this allows for less efficient routes to be considered.

maxiter: Problematic points can be removed, but this requires that transition probabilities are calculated twice, which can be slow.

minpointdistance: Prior to performing analysis, GPS data are cleaned such that no two points are closer than this distance. The original paper suggests setting this value to sigmaZ * 2, so far sigmaZ * 3 has produced better results.

paramter_window: Velocities, rotations, and bearings are calculated according to this window. 3 is the default value.

bearing_penalty_weight: Use this to increase or decrease the effect of a bearing difference on the emission probability of a point. A number between 0 and 1 is preferred, although higher than 1 will not cause errors.
viterbi_lookahead: The length of the path to consider when making decisions about the best path through the Hidden Markov Model. 0 is fastest, 1 produces better results, and 2 appears to be quite slow.

lazy_probabilities: True if only transition probabilities that are used should be calculated. If lookahead is 0, this is significantly faster. If lookahead is 1 or greater, most transition probabilties are used, so this does not lead to a performance increase.

points_summary: True if the list of point/segment matches should be returned, False otherwise.

segments_summary: True if the complete list of segments should be returned, False otherwise.

Disconnect from the database!

Don't forget!

```
planetdb.disconnect()
```

```
True
```