Routino: Users manual

Python code for doing batch travel time & distance calculations

January 5, 2021

1 The functionality of the program

Routino provides an open source solution for calculating travel time and distance between two locations. Mike Allen created python code in order to make use of Routino to do batch calculations of many routes. This document describes how to use the python code and data files for your own batch calculations of travel times and distances.

This document will describe these following files, in this file structure:

- data_etc
  - profiles.xml
  - uk_postcodes.csv
  - uk_postcodes_additional.csv
- output
  - results.csv
  - pivot_results.py
  - pivoted_results_distance_km.csv
  - pivoted_results_time_min.csv
- batch_process_with_running_save.py
- from_to.csv
- matrix.csv
- unstack_matrix.py

Figure 1 shows an overview of how the files listed above are linked together, and how the datafiles (blue) flow between the python code (yellow).

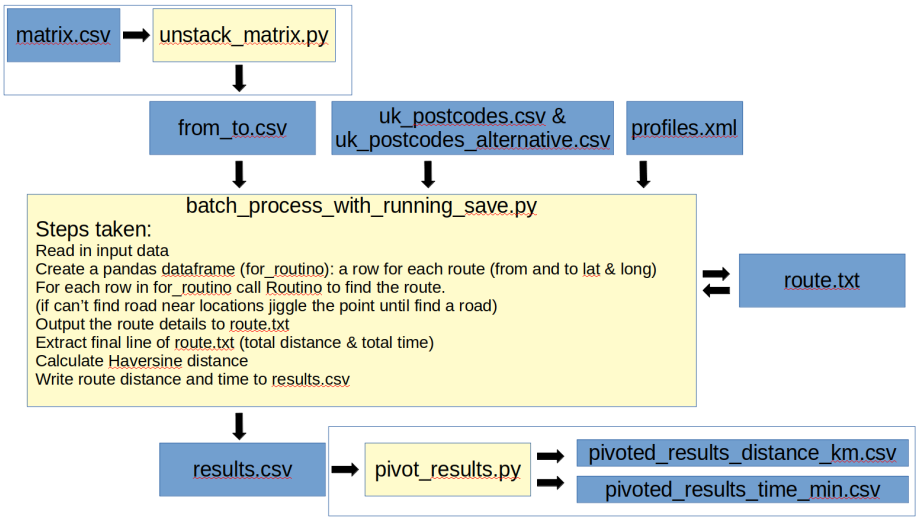

Figure 1: A process diagram to represent the flow of information between files and Python code. Blue squares represent data files. Yellow squares represent Python code. Items in a boarder represent stages only required if you are working with a matrix (calculating a complete set of routes between a set of starting points and a set of ending points)

2 Describing the main program: batch_process_with_running_save.py

The main Python code is *batch_process_with_running_save.py*. This code does the core of the functionality - batch calculates the travel routes between multiple start and end points. This section will describe how it works and how to tailor it for your own purposes.

2.1 The main part of this Python program

The heart of the program is in the function *get_routino_output*. Here is the code (afterwards is a description of the functionality):

```
def get_routino_output(row):
  """Call Routino (pass start/end lat+long"""

  bashCommand = \
  ('routino-router --dir=data_etc --trafig:process_diagramnsport=motorcar
  --quickest --lon1=' + str(row['from_long']) + ' --lat1='
  + str(row['from_lat']) + ' --lon2=' + str(row['to_long'])
  + ' --lat2=' + str(row['to_lat']) + ' --language=en
  --output-text --output-stdout >output/route.txt')

  os.system(bashCommand)
  last_line = extract_last_line('output/route.txt')
  time, distance, error = get_time_distance(last_line)
  haversine_distance = get_haversine_distance(
                                        row['from_lat'], row['from_long'],
                                        row['to_lat'], row['to_long'])
        return (time, distance, error, haversine_distance)
```

Routino is a separate programme that exists externally to this .py file. The .py file calls Routino using the line of code: "os.system(bashCommand)". This is essentially typing a command into the terminal. The command is stored in the variable bashCommand. It is passing the necessary information the routino-router program.

For the case above, the command is passing this information to routino:

1. the data is stored in the subfolder data_etc
2. from the profiles.xml file use the values for motorcar
3. find the quickest route
4. for starting point use the values from the pandas dataframe row: from_long and from_lat
5. for ending point use the values from the pandas dataframe row: to_long and to_lat
6. setting language to English
7. send the output to the file output/route.txt

Essentially we pass routino the start and end points (as lat and long values) and routino returns a text file (route.txt) that contains a step-by-step instruction of the routing information. Each row contains details of a section of the route, and the total route so far. Figure 2 shows an example of this file.

```
# Creator : Routino - http://www.routino.org/
# Source : Based on OpenStreetMap data from http://www.openstreetmap.org/
# License : http://www.openstreetmap.org/copyright
#
#Latitude       Longitude       Section         Section         Total           Total           Point   Turn    Bearing Highway
#                               Distance        Duration        Distance        Duration        Type
 51.580998       0.134132       0.000 km         0.0 min         0.0 km          0 min          Waypt#1          +2      (null)
 51.580994       0.134235       0.007 km         0.0 min         0.0 km          0 min          Junct    +0      +1      (null)
 51.581148       0.134516       0.027 km         0.0 min         0.0 km          0 min          Junct    -1      -2      (null)
 51.582344       0.117508       1.181 km         0.9 min         1.2 km          1 min          Junct    +2      +0      (null)
 51.582443       0.117559       0.011 km         0.0 min         1.2 km          1 min          Junct    +2      +2      (null)
 51.581192       0.137839       1.410 km         1.1 min         2.6 km          2 min          Junct    +0      +2      (null)
 51.581093       0.141323       0.237 km         0.2 min         2.9 km          2 min          Junct    +0      +2      (null)
 51.592391       0.181704       3.314 km         2.6 min         6.2 km          5 min          Junct    +0      +1      (null)
 51.593051       0.209998       2.051 km         1.6 min         8.2 km          6 min          Junct    +0      +2      (null)
 51.592879       0.212494       0.171 km         0.2 min         8.4 km          7 min          Junct    +0      +1      (null)
 51.592624       0.213219       0.076 km         0.1 min         8.5 km          7 min          Junct    +0      +3      (null)
 51.578675       0.250522       3.079 km         2.7 min        11.6 km          9 min          Junct    +0      +2      (null)
 51.574000       0.275003       1.808 km         1.4 min        13.4 km         11 min          Junct    +0      +3      (null)
 51.572977       0.283810       0.633 km         0.5 min        14.0 km         11 min          Junct    +0      +1      (null)
 51.571829       0.286558       0.392 km         0.3 min        14.4 km         12 min          Junct    +0      -4      (null)
 51.502646       0.268105       7.971 km         4.2 min        22.4 km         16 min          Junct    +0      -4      (null)
 51.500225       0.268463       0.269 km         0.1 min        22.6 km         16 min          Junct    +0      -4      (null)
 51.453506       0.243827       5.695 km         4.5 min        28.3 km         21 min          Junct    +0      -2      (null)
 51.442331       0.238274       1.490 km         1.2 min        29.8 km         22 min          Junct    +0      -4      (null)
 51.436637       0.239062       0.636 km         0.5 min        30.5 km         22 min          Junct    +0      -4      (null)
 51.433292       0.239234       0.371 km         0.3 min        30.8 km         23 min          Junct    +0      -4      (null)
 51.399192       0.205002       4.605 km         2.5 min        35.4 km         25 min          Junct    +0      -3      (null)
 51.392141       0.196401       0.986 km         0.5 min        36.4 km         26 min          Junct    +0      -3      (null)
 51.353903       0.156140       5.276 km         2.8 min        41.7 km         28 min          Junct    +0      -4      (null)
 51.301732       0.149579       6.129 km         3.3 min        47.8 km         32 min          Junct    +0      -3      (null)
 51.272763       0.048537       8.339 km         4.4 min        56.2 km         36 min          Junct    +0      -2      (null)
 51.258601      -0.057665       7.850 km         4.2 min        64.0 km         40 min          Junct    +0      -2      (null)
 51.259432      -0.109492       3.633 km         1.9 min        67.6 km         42 min          Junct    +0      -2      (null)
 51.261051      -0.116243       0.501 km         0.3 min        68.1 km         42 min          Junct    +0      -1      (null)
 51.268476      -0.166283       3.606 km         1.9 min        71.8 km         44 min          Junct    +0      -2      (null)
 51.300768      -0.318135      12.698 km         6.8 min        84.5 km         51 min          Junct    +0      -1      (null)
 51.303149      -0.321824       0.381 km         0.2 min        84.8 km         51 min          Junct    +0      -3      (null)
 51.303122      -0.321871       0.004 km         0.0 min        84.8 km         51 min          Junct    +0      -3      (null)
 51.302179      -0.321684       0.110 km         0.1 min        84.9 km         51 min          Junct    +0      +3      (null)
 51.298735      -0.316510       0.522 km         0.5 min        85.5 km         52 min          Junct    +0      +2      (null)
 51.298681      -0.316291       0.015 km         0.0 min        85.5 km         52 min          Junct    +0      +2      (null)
 51.298703      -0.315756       0.036 km         0.0 min        85.5 km         52 min          Junct    +0      +1      (null)
 51.326727      -0.273706       4.417 km         4.7 min        89.9 km         57 min          Junct    +2      +3      (null)
 51.326414      -0.273006       0.058 km         0.1 min        90.0 km         57 min          Junct    +0      +3      (null)
 51.325619      -0.272587       0.091 km         0.2 min        90.1 km         57 min          Waypt#2
```

Figure 2: The contents of the *route.txt* file

As you can see, the final line of this file contains the total distance and time travelled for the whole route. For our purposes, we are only interested in this information. So for each route, we will extract this information from the final line of this file, write it to the output file, and then overwrite the contents of the route.txt file with the next journey. And repeat.

One of the input files Routino uses is *profiles.xml*. This contains the parameters for the transport selected via the command that was passed to routino. In our case this is *motorcar*. Figure 3 shows the associated properties for motorcar that routino will use to calculate the route:

```
<profile name="motorcar" transport="motorcar">
  <speeds>
    <speed highway="motorway"      kph="112" />
    <speed highway="trunk"         kph="76" />
    <speed highway="primary"       kph="76" />
    <speed highway="secondary"     kph="70" />
    <speed highway="tertiary"      kph="64" />
    <speed highway="unclassified"  kph="51" />
    <speed highway="residential"   kph="38" />
    <speed highway="service"       kph="25" />
    <speed highway="track"         kph="12" />
    <speed highway="cycleway"      kph="0" />
    <speed highway="path"          kph="0" />
    <speed highway="steps"         kph="0" />
    <speed highway="ferry"         kph="25" />
  </speeds>
  <preferences>
    <preference highway="motorway"      percent="100" />
    <preference highway="trunk"         percent="100" />
    <preference highway="primary"       percent="90" />
    <preference highway="secondary"     percent="80" />
    <preference highway="tertiary"      percent="70" />
    <preference highway="unclassified"  percent="60" />
    <preference highway="residential"   percent="50" />
    <preference highway="service"       percent="40" />
    <preference highway="track"         percent="0" />
    <preference highway="cycleway"      percent="0" />
    <preference highway="path"          percent="0" />
    <preference highway="steps"         percent="0" />
    <preference highway="ferry"         percent="20" />
  </preferences>
  <properties>
    <property type="paved"         percent="100" />
    <property type="multilane"     percent="60" />
    <property type="bridge"        percent="50" />
    <property type="tunnel"        percent="50" />
    <property type="footroute"     percent="45" />
    <property type="bicycleroute"  percent="45" />
  </properties>
  <restrictions>
    <oneway obey="1" />
    <turns  obey="1" />
    <weight limit="0.0" />
    <height limit="0.0" />
    <width  limit="0.0" />
    <length limit="0.0" />
  </restrictions>
</profile>
```

Figure 3: The motorcar properties contained in the *profiles.xml* file

2.2 How the code runs in sequence

The program starts at this line of code:

```
if __name__ == '__main__':
```

Throughout the program the code prints it's progress. The first one of these is at the very start:

```
print(datetime.datetime.now().strftime("%H:%M:%S"), 'Start')
```

For cases in the program where a route can not be calculated by routino, the straight line distance between two points on the surface of a sphere is used (the haversine distance). This is then increased by a factor to take into account of the curvature of the roads (haversine_tortuosity_correction), and a constant speed of travel is used (assumed_speed_when_imputing_kph)

```
haversine_tortuosity_correction = 1.262
assumed_speed_when_imputing_kph = 48.3
correction_for_all_times = 1.000
```

As the program calculates each route, it writes the route result (along with any error catchers) to the output file as it goes along. Here we open the output file.

```
output_filename = 'output/results.csv'
results_header = ('from_postcode,to_postcode,from_lat,from_long,to_lat,' +
'to_long,from_loc_identified,to_loc_identified,' +
'route_locs_identified,time_min,distance_km,imputed,no_route\n')

# Open output file
with open(output_filename, 'w') as myfile:
myfile.write(results_header)
```

Note: To run multiple versions of Routino at once, make sure change the name of the two output files (results.csv and route.txt), otherwise will get errors as the route.txt will be written to and read by many programs, and so the values will be assigned to the wrong reference ID.

2.3 Input files to edit to represent your case

There are three files that you need to edit so that the program will run for your case.

2.3.1 from_to.csv

This file contains all of the routes that Routino will calculate, one route per row. This input file needs to be a csv file with two columns. The first is the reference ID for the from location, and the second is the reference ID for the to location.

```
# Load data
    print(datetime.datetime.now().strftime("%H:%M:%S"), 'Loading data')
    from_to = load_from_to('from_to.csv')
```

Note: If you are working with a travel matrix (calculating a complete set of routes between a set of starting points and a set of ending points) then there exists a separate .py file (*unstack_matrix.py*) which can help you to create the *from_to.csv* file. That .py file takes as an input all the starting locations, and all the ending locations, and creates the required input file for each pairwise combinations.

2.3.2 uk_postcodes.csv (& uk_postcodes_additional.csv)

There exists two further input files (*data_etc/ukpostcodes.csv* & *data_etc/ukpostcodes_additional.csv*) that contain three columns. The first column is the reference ID for the location, the second is the corresponding latitude, and the third is the corresponding longitude. The function

load_postcode_lat_long() joins these two csv files into a single pandas dataframe, removes any spaces in the reference ID and drops any duplicates.

This pandas dataframe is used to look up the reference ID (that is contained in the *from_to.csv* file, and get it's latitude and longitude values. If your routes contain a reference ID that does not already exist in either of these files, you need to add a new ID together with it's latitude and longitude to the *data_etc/ukpostcodes_additional.csv* file.

This code was initially set up to calculate routes between two postcodes (hence the variables names call this location reference ID "postcodes"), but any unique ID can be used to reference a location so long as that unique ID exists in the lookup file along with it's corresponding latitude and longitude.

```
postcodes = load_postcode_lat_long()
```

2.3.3 profile.xml

As previously mentioned, this file contains all of the parameters used for the different modes of transport (speed on road types, and preferences for road types).

2.4 Preprocessing stage

This step populates a pandas dataframe with the to-from locations represented as their lat-long values. Done in function *merge_postcodes_with_lat_long()*, and the resulting pandas dataframe is called *for_routino*.

2.5 Find fastest route and output result to *results.csv*

For each row in the pandas dataframe *for_routino*, pass the latitude & longitude values for both the start and end points to routino, and it will return the travel time, travel distance and haversine distance. It will also return an error flag to capture whether it was not possible to calculate the route.

```
# Call Routino
travel_time, distance, error, haversine_distance =
                                        get_routino_output(row)
```

If no route was found (error flag return True), this could be due to no road being identified at the start or end locations.

The code tries to deal with this by moving the start/end points a small amount to find a nearby road. These small location adjustments are stored in the list of lists called *offsets*, and incrementally moves the point along each of the four cardinal directions (N, S, E and W). If a road is found (by making these small location adjustments) then calculate the route from that point.

If no road was found after moving the locations, use the Haversine distance and report that the route is imputed.

Write the result to the output file *results.csv*. The values written to the output file are:

1. **from_postcode**: starting location as reference ID
2. **to_postcode**: ending location as reference ID
3. **from_lat**: latitude of the starting location
4. **from_long**: longitude of the starting location
5. **to_lat**: latitude of the ending location
6. **to_long**: longitude of the ending location
7. **from_loc_identified**: if "1", then could not locate a road at the start location. Need to use haversine calculation
8. **to_loc_identified**: if "1", then could not locate a road at the end location. Need to use haversine calculation
9. **route_locs_identified**: Set to "1" if either of the previous two columns are "1". Need to use haversine calculation
10. **time_min**: the time taken for the route
11. **distance_km**: the distance of the route
12. **imputed**: if "1", then no route found. Use straight line and adjust (Haversine)
13. **no_route**: if "1", use straight line and adjust (Haversine)

Then move on to the next row in the pandas dataframe *for_routino*, and calculate the route for the next set of to-from locations.

At the end of the program, remove the route.txt file that was created by Routino.

3 Formatting the output file

The output file *results.csv* has the values for a route per row.

If your data represents a travel matrix (a complete set of routes from a set of starting locations and a set of end locations), then use *pivot_results.py* to put the output values back into a matrix format. Two output files are produced: one for distance; one for time.

- output
  - time_min_pivoted_results.csv
  - distance_km_pivoted_results.csv

4 Calibration

It is advised that the values from Routino are calibrated against 50 of the routes as calculated by Google.

Manually select 50 of your routes that cover the spread of the distances & durations across all of the routes.

Here I will talk about the route distances, but repeat this for both of the distance and durations.

Calculate the correction factor for each of the 50 routes. Where the correction factor is calculated as:

$$correction factor = Google\_distance / Routino\_distance$$

Calculate the median correction factor.

Also plot a scatter chart of the correction factor against the Routino distance. See Figure 4 to demonstrate this.

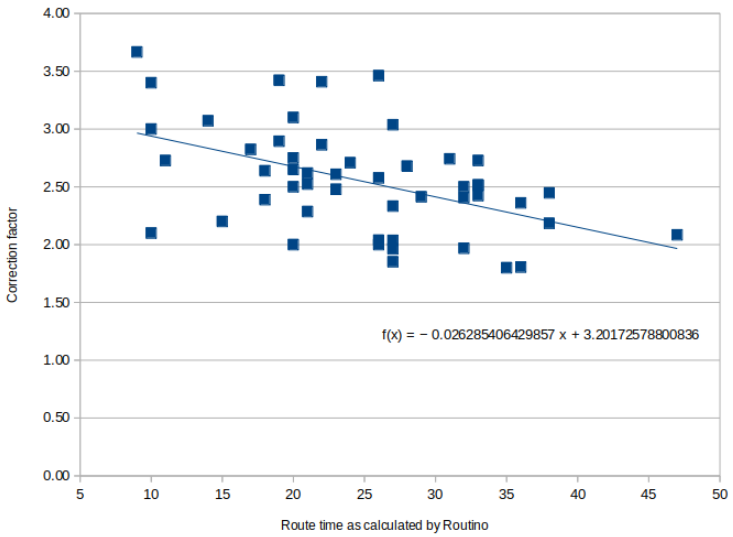

Figure 4: Plotting the correction factor against the Routino values

Decide whether it's reasonable to use the median of the correction factors for all, or whether there's a relationship such that the correction factor is dependant on the length of the route.

We have previously found that a constant value (the median) can be used for the correction factor for distance, but that the correction factor for time tends to be larger for the shorter routes, and so for those cases an equations (the trend line) is used. This can be linear, or a n order polynomial (as was the case for Scotland as the data showed an obvious kink/curve in the relationship).

If the trendline in Figure 4 is used, it will give a correction factor of 3.2017 when the route is of length 0, and for each unit increase in length, the correction factor will reduce by 0.026285.

Ensure after these corrections that matrix (calculating a complete set of routes between a set of starting points and a set of ending points)the min value returned is 1.

5 Using the travel matrix in a geographical modelling project

Program: *geographical_modelling_scenarios.py*

If your routes form a travel matrix (where you have calculated a complete set of routes between a set of starting points and a set of ending points), then this section will describe how to use the program *geographical_modelling_scenarios.py*, and the output file *time_min_pivoted_results.csv* (or *distance_km_pivoted_results.csv* - both from *pivot_results.py*) in order to model some geographical scenarios.

The code treats all of the "from" locations as where a patient will begin their journey, and each of the "to" locations are where they can access a service. The scenarios will explore the impact of having a different combination of these services open and receiving patients.
The code will create the full set of scenarios that will cover every possible combination of the services as "open".

For each of the scenarios, the code will allocate each of the patient locations ("from") to the nearest open service ("to") location.

5.1 Input files

This program requires 2 input files. Here is the file structure this program requires:

- data_etc
  - activity_data.csv
- output
  - pivoted_results_distance_km.csv
  - pivoted_results_time_min.csv
  - scenario_results.csv
- geographical_modelling_scenarios.py

5.1.1 Input file: activity_data.csv

This is a csv file with two columns and a row per "from" location. The "from_postcode" contains the reference ID for the "from" location. The "activity" column contains the number of patients that are at that location.

The column titles must be kept as "from_postcode" and "activity". This file

provides the program with information about the number of patients at each of the "from" locations. The program will calculate weighted averages based on these activity values. If only one patient comes from each location, populate all of the values in column "activity" with 1.

5.1.2 Input file: time_min_pivoted_results.csv

Change the code in the .py file to read in the equivalent distance file, if ever necessary.
This is the output file from pivot_results.py (which is the program that takes the Routino output as it's input file).

5.2 Output files

The program creates 1 output file: *scenario_results.csv*. It has a row per scenario. The number of columns depends on the number of "to" locations there are.

There are 4 columns to describe the full system:

- **median_travel**: median travel of all the patients in the whole system
- **max_travel**: max travel of all the patients in the whole system
- **95pctl_travel**: 95th percentile travel of all the patients in the whole system
- **activity_within_30**: total activity in the whole system within the threshold travel (in this case, it's 30 as shown in the column title)

There is then a column per "to" location to show whether for that scenario it is open:

- **location_n**: whether the "to" location was open in the scenario (1 = open)

And finally there are 6 columns per "to" location, to describe the group of patients that are in it's catchment (where n is the number of the "to" location). These 6 columns are:

- **median_n**: median travel for the patients that attended this "to" location
- **max_n**: max travel for the patients that attended this "to" location
- **95pctl_n**: 95th percentile travel for the patients that attended this "to" location
- **activity_n**: the number of patients that attended this "to" location
- **activity_within_threshold_n**: the number of patients that attended this "to" location within the threshold travel.

6 Mapping

Please consult the tutorial `https://youtu.be/QEZFnERpZxY` (YouTube channel HSMA: "Session 6B : An introduction to GeoPandas") to follow and apply GeoPandas to the results of this dataset.