How can I optimise my walking journey to UCL?

Introduction

This project investigates ways in which I can optimise my walking journey to UCL. In this instance, by optimise I mean to make the journey time as short as possible, so that I can arrive to UCL in time in the morning. The project makes use of location and time data collected from my walking journeys to and from UCL over six months, as well as statistical analysis and visualisation in Python and Mapbox Studio. There is a tentative suggestion that journeys where my walking speed increases are faster than those where my walking speed decreases, though this is not statistically significant. Regions of my typical route that are slower or faster are identified, though data collection on alternative routes would be needed to determine whether route changes could be used to optimise my journey.

Background

I had been collecting data about my walking movements using a fitness tracking app on my smartphone. The idea occurred to me that I could use this data to come up with some interesting conclusions about my walking behaviour. I had some initial thoughts about how my walking behaviour varied – for instance, that I felt that I walked faster in the morning on my way to UCL than in the evening on my way home. I settled on the question of optimising my journeys.

The project was an opportunity to show the interesting insights we can gain when we examine the data collected by our smartphones. Commonly used smartphone apps track our location on average once every three minutes (Dwoskin, 2015). While this data may improve the service companies can provide, over 90% of Americans feel that they have lost control over their personal data (Dwoskin, 2015). It's possible to advocate not using smartphones at all, but as an alternative, we as users can use the data ourselves: by doing this, we gain a greater understanding of what data is being collected, as well as potentially using it to improve our own lives. The concepts used in this project could be extended into a service to be used by other smartphone users to gain similar insights about their own lives.

Walking as transport is an area that is justifiable to study. Walking has a key part to play in transportation in cities such as London, forming at least a small part of all journeys, especially those taken using public transport (Transport for London, 2004, p.13). Walking for transport may be essential to ensuring that city dwellers meet the recommended levels of physical exercise, and gain from the health benefits that this brings (Karusisi et al., 2014, p.1). Planning a city with appropriate provision for walking is necessary to encourage walking. Assumptions about pedestrian speeds are used by pedestrian behaviour modelling software to plan changes to infrastructure supporting walking, so accurate information about walking speeds is important (Al-Azzawi, 2004, p.198). Variation in walking speed is likely to be affected by pedestrian density and width of the pavement, and it may also be affected by whether we are walking in a shopping centre or to or from work (Al-Azzawi, 2004, p.45). However, detailed analysis of walking behaviour for transport is not that common: research often focuses on pedestrian behaviour around large events such as the Notting Hill Carnival, where the benefits of such analysis in informing crowd control decisions are clear (Batty, 2003, pp.98-105). Part of the reason for this limited research may be that accurate path data is difficult to obtain from conventional collection methods such as examining CCTV footage (Batty, 2003, p.97). I hope to show that using tracking data collected by smartphones is an effective means of data collection when investigating walking speeds.

Data collection

The data used in this project was collected over a period of six months using Moves, an iOS application running on an iPhone 5 which I carried with me when walking to and from UCL. This application collects latitude and longitude co-ordinates representing location from the iPhone (ProtoGeo Oy, 2016). Ultimately, this location data is sourced from satellite and mobile network location services (Fleishman, 2011). I used the Moves API (Application Programming Interface) along with the `pymoves` library to collect my data from the Moves app's servers and make it available in Python. First, I had to filter the data, since many of the journeys recorded were not from my hall of residence (St Pancras Way) to UCL or vice versa. After filtering, data on approximately 230 journeys remained, comprising a total of 12,000 data points made up of a pair of latitude and longitude co-ordinates and a time. A sample data point is shown below:

```
{     'lat': 51.534584,
      'lon': -0.13541696,
      'time': '20160325T123604Z'    }
```

The Moves API presented the data in JSON (JavaScript Object Notation) format, a popular format for interchange of data between computer programs (Ecma International, 2013), and I used the `json` library to import this as a Python object for use by my Python programs.

Initial data processing

Since my question concerned my walking speed, I needed to calculate this. In order to find the speed, the distance between each pair of points had to be determined. I calculated the distance between adjacent data points using the `vincenty` function of the `geopy` library, which determines the distance in meters between two points in latitude and longitude co-ordinates (GeoPy Contributors, 2015). I then added this calculated speed to each data point.

I visualised this data by exporting it using the `json` library to a GeoJSON file. GeoJSON is a specification based on JSON which encodes geographical features (Butler et al., 2016). I then displayed this using Mapbox Studio, a web-based map visualisation platform (Mapbox, 2016). This visualisation is shown in Fig. 1.

In this visualisation, the green colour shows speeds of less than 1.6ms^{-1} (3.5mph), the orange colour shows speeds of between 1.6ms^{-1} and 2ms^{-1}, and the red colour shows speeds of more than 2ms^{-1} (4.6mph). It is clear that there is a range of speeds in the route. The visualisation is also a useful check to show that the filtering process has worked as expected: various different routes to UCL from my hall of residence are visible, but there are no journeys to other locations visible.

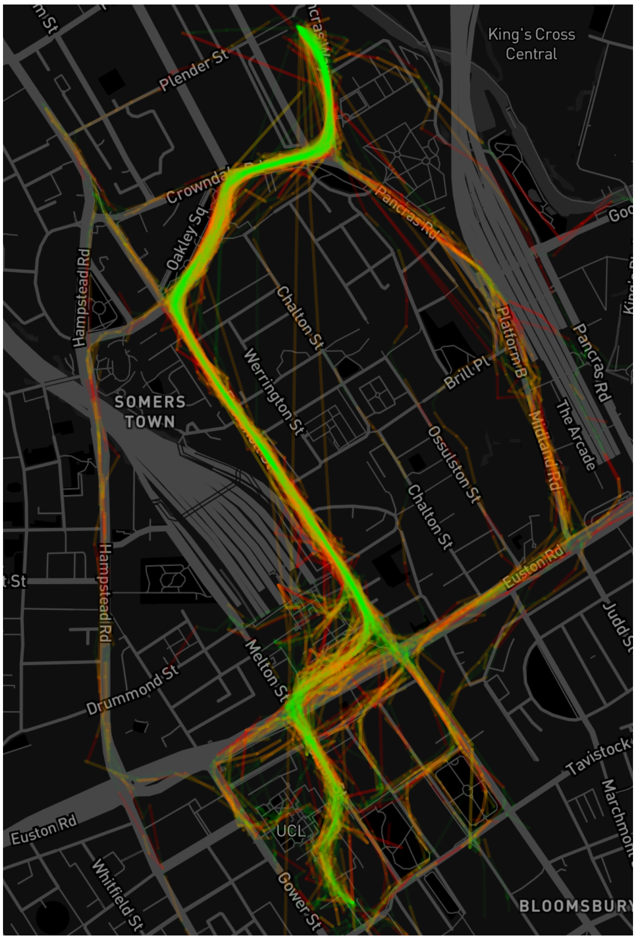

Fig. 1 ***All journeys to and from UCL***
Source: mapbox.com/studio

To examine the data more closely, I plotted the walking speed against time for each journey using `matplotlib`. This showed a lot of speed variation in the data, as well as some unusual behaviour at the beginning and end of many of the journeys. An example is shown in Fig. 2, where there is a walking speed of almost 6ms^{-1}, or over 13mph: this is an impossible walking speed. I believe that this might have been to do with the way that the Moves app identifies the start and end of a walking journey – there may be a delay where the app starts recording the information. Because of these issues, I decided to exclude the first and last speed points of each journey from my analysis, and additionally to implement a moving average. A moving average is a method used to smooth out fluctuations in data. It involves taking the average of the values of a variable over a set number of periods. I chose to use a three-period moving average, since an odd number of periods allows you to plot the average at the centre of the region over which you have averaged (Weisstein, n.d.). In fact, what I implemented is not a true moving average, since the time periods vary (the Moves app does not collect location data at equal intervals), whereas a moving average should have the same averaging period throughout the data, but I chose to average the data in this way because it was substantially easier to implement. As you can see in Fig. 3, many of the fluctuations in the data have been removed.

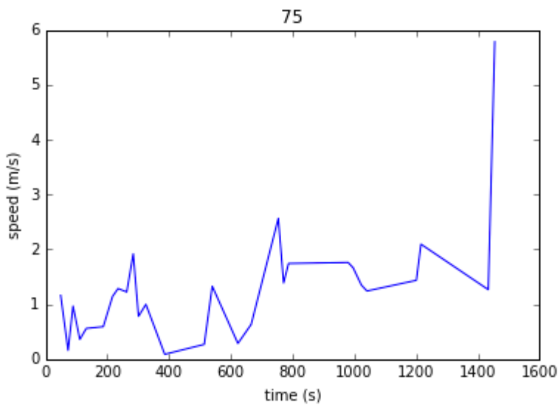

Fig. 2 ***Sample journey: raw data***

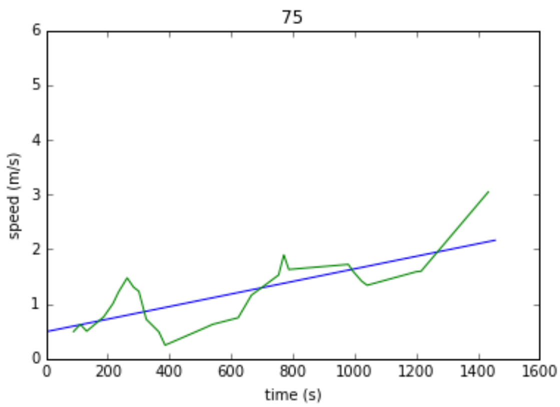

Fig. 3 ***Sample journey: smoothed data and best-fit line***

Data analysis I: getting faster or slower within journeys

One possible way of optimising my journey would be to vary my walking speed. A possible hypothesis is that if I walk faster earlier on in the journey, I could get tired and end up walking slower at the end of the journey, leading to a longer overall journey time; conversely, if I start off walking more slowly then I might conserve energy and be able to walk faster towards the end of the journey, leading to a shorter overall journey time. I will refer to these two types of journey as those with increasing and decreasing speeds respectively. Although my aim was to optimise the overall journey time, it was more reasonable to compare the average speed on these journeys rather than the overall journey time, since the overall journey time is affected by the distance of the journey, which depends on the specific destination at UCL, or the route taken. I also used the data from my journeys both to and from UCL, given the relatively small quantity of data available, and I thought that there was unlikely to be a difference in the effect that the changing speed in a journey would have a similar effect on the average journey speed both to and from UCL.

Formalising this into null (H_0) and alternative (H_1) hypotheses gives the following:

> H_0: There is no difference in average speed between journeys with decreasing and increasing speeds.
> H_1: A journey with increasing speed has a faster average speed than one with a decreasing speed.

I decided to carry out the hypothesis test at the 5% significance level. Before being able to carry out a hypothesis test, I needed to split up the journeys into those that had increasing and decreasing speeds, as well as those which had neither increasing or decreasing speeds. To do this, I then carried out a linear regression analysis on the set of data for each journey. This involved using the `scipy.stats.linregress` function, which returns a best fit line for a set of data, as well as the r-value – the Product Moment Correlation Coefficient (The Scipy community, 2016). The r-value is a measure of the strength of the correlation in the data: a value of -1 indicates a perfect negative correlation, whereas a value of 1 indicates a perfect positive correlation. A value of 0 indicates no correlation (Revision World Networks Ltd., n.d.). Hence, those journeys with a highly positive r-value would indicate that the speed was increasing, and those journeys with a highly negative r-value would indicate that the speed was decreasing. In the right hand graph, you can see the linear fit line for the exemplar journey. The code that I used to accomplish this is below; where there is an ellipsis, some code has been omitted.

```
for commute_number in range(len(commute_journeys)):
...
```

The first line iterates through all the journeys. The next line iterates through all data points in one journey.

```
        for point_number in range(len(trackPoints)-4):
        ...
```

I then label the points on either side of the starting point.

```
            pointL = trackPoints[point_number_2-1]
            pointR = trackPoints[point_number_2+1]
```

Now, I take the average, using the average function from numpy.

```
            speed_pavg = np.average([point['speed'],pointL['speed'],\
            pointR['speed']])
```

Then, I add the speed to my data about the point.

```
            point['speed_pavg'] = speed_pavg
```

For my regression analysis later, I now add the time and speed to lists.

```
            reference_times.append(point['reference_time'])
            speeds.append(speed_pavg)
```

Before running the regression analysis, I set up a dictionary to add the data obtained from it to my data about that journey.

```
        commute['linr_pavg']={}
        linr = commute['linr_pavg']
```

Then, I simply assigned the values from the regression analysis function to keys in that dictionary. This means that I had all those values available for later use.

```
        linr['slope'], linr['intercept'], linr['r'], linr['p'],\
        linr['std_err'] = stats.linregress(reference_times,speeds)
```

Owing to fluctuations which remained in the data even after having taken moving averages, the r-values for most of the journeys remained low. I decided that I would consider anything where $r > 0.4$ to be a journey with increasing speed, and $r < -0.4$ to be a journey with decreasing speed. Since I was comparing two data sets, I used an unpaired t-test, since this "tests the null hypothesis that the population means related to two independent, random samples from an approximately normal distribution are equal" (StatsDirect Limited, 2016) – this is the null hypothesis shown above.

	Decreasing speed	Increasing speed
	$r < -0.4$	$r > 0.4$
Average speed	1.46 m s^{-1}	1.72 m s^{-1}
Number of journeys (n)	32	19
t-test result	$p = 0.12$	

Although the increasing speed journeys appear to be associated with a faster average speed journey, the t-test reveals that the there is a 12% chance of this result having occurred randomly. Therefore, I cannot conclude at the 5% significance level that starting slower and increasing my speed through the journey will lead to a faster journey. More data would be needed in order to determine whether this is the case – in this case, the fact that 81% of the journeys couldn't be classified into increasing or decreasing speed journeys greatly limited the quantity of data available.

Data analysis II: averaging speeds across all journeys along my typical route

For this part of the project, I wanted to try to find out whether there were particular parts of my normal route between my hall of residence and UCL where I walked particularly slowly, so that I could potentially modify my route to optimise the time it takes me to walk to UCL. This was a more difficult task to accomplish, since all of the journeys had different location co-ordinates associated with them – either due to differences in the routes that I walked, or due to artefacts in the location data (these are visible in Fig. 1). In order to come up with any meaningful conclusions, I would need to restrict myself to a single route, then average my speeds while walking along each part of this route. I mapped out my typical route on Google Maps (see Fig. 4) and exported this route as a Keyhole Markup Language (KML) file, which contained the co-ordinates of points along that reference route.

The next step was to find the point along this reference corresponding to each point I had recorded data about. I did this using the `shapely` library, which would allow me to find the point along a specified line closest to a given point. To find the point closest to a line, I used code written by Kington (2013). Fig. 5 illustrates this principle on another dataset: the blue line is the reference line, and the red dots represent data points and the points on the reference line that they have been mapped to. However, the `shapely` library operates in Cartesian co-ordinates (the Python community, 2015) so I needed to convert the data from latitude and longitude to Cartesian co-ordinates, which I did using the `pyproj` library. I converted the co-ordinates into a system called UTM (Universal Transverse Mercator). This Cartesian co-ordinate system represents latitude and longitude with a maximum error of 0.1%, which was acceptable for this project. The minimum distance between a point and the reference line was calculated, and I only included the data for those points which were less than 100m from the reference line – I thought that this took account of the width of roads and the level of error generally present as a result of the data collection.

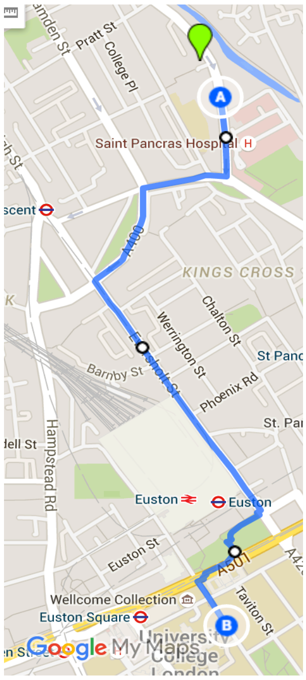

Fig. 4 ***Reference route*. Source: maps.google.com**

I then took averages of the speed in 10m segments along the reference route. For the purposes of this part of the analysis, I separated the data into journeys to and from UCL, since it was the journey to UCL that I was trying to optimise, and certain road features are not symmetric based on the direction in which you are walking (e.g. pedestrian light cycles, which can make a journey take longer depending on the direction in which you are travelling).

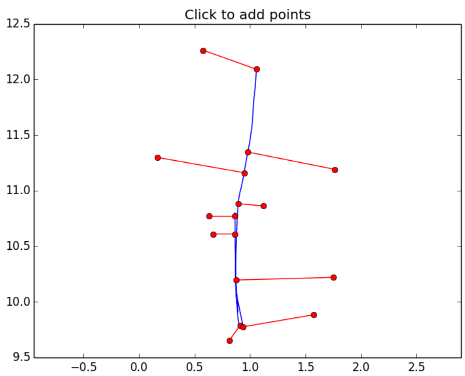

Fig. 5 ***Mapping points to a reference line*** **(Kington, 2013)**

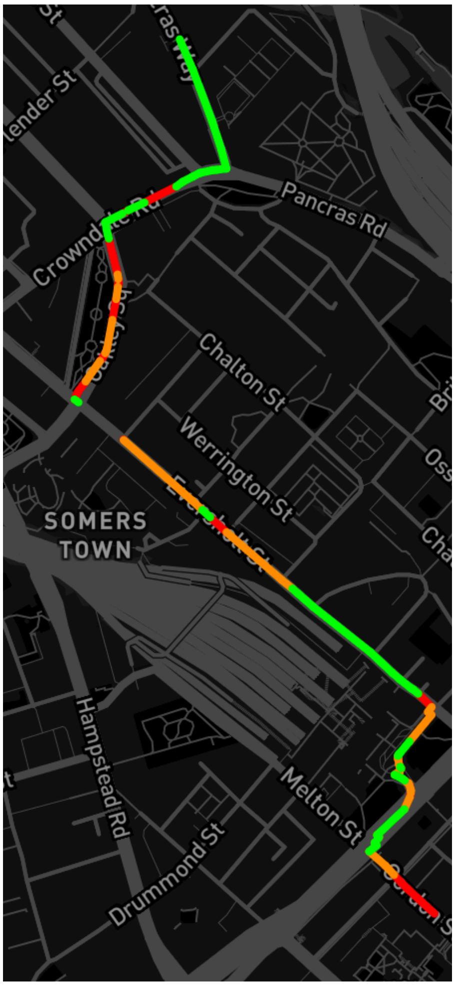

Fig. 6 ***Averaged speeds on journeys from UCL***
Source: mapbox.com/studio

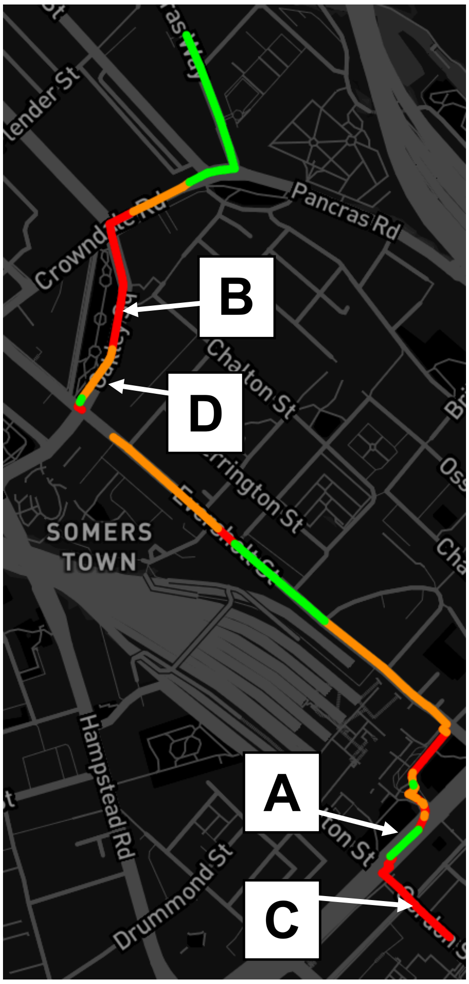

Fig. 7 ***Averaged speeds on journeys to UCL***
Source: mapbox.com/studio

Note: in these figures, green represents speeds under 1.4ms^{-1} (3.1mph), orange represents speeds from 1.4ms^{-1} to 1.6ms^{-1}, and red represents speeds in excess of 1.6ms^{-1} (3.5mph).

I will focus on Fig. 7, since these are the journeys to UCL. Some speed variations in particular locations are visible. For instance, there is a slower speed of 1.2-1.3ms^{-1} around the Euston Road/Gordon Street junction (marked *A*), probably due to me often crossing the Euston Road at this point, which involves waiting for pedestrian lights to change or for a suitable gap in traffic. However, it seems unlikely that I can optimise my journey to avoid this slower speed segment, since I have to cross the Euston Road at some point in my journey. It also appears that the slower speeds around this point are followed by faster speeds on Gordon street (marked *C*) which suggests that a small delay in my journey due to traffic lights may not actually affect my overall journey speed substantially.

In contrast, the stretches along the northern part of Oakley Square (marked *B*) have a faster average walking speed, over 1.6ms^{-1}, which makes sense because there are no obstructions to the journey at this point. There is a slower speed immediately to the south of this section (marked *D*), of 1.4ms^{-1}, which I believe is due to the the presence of an entrance to a housing estate at this point. Once again it seems unlikely that it would be possible to optimise my journey by avoiding this part of the route, since other alternative routes such as Midland Road have other features that slow down a journey (i.e. lack of a pedestrian crossing on the road which results in having to walk around crash barriers).

At the southern end of Eversholt Street, there is a speed of 1.4ms^{-1}, which may be as a result of a busier pavement with more pedestrians at this point, as I get closer to central London. Once again, there are no obvious changes that could be made at this point in order to optimise my journey.

In general, the speed differences visible are quite small in relation to the noise in the data – the method of data collection that was used has inherent inaccuracies which affected the results, even when averages were taken across all of the data that I had available.

Other factors that may be influential

Since I thought that I walked faster on my way to UCL than back from UCL, I thought I would check whether this was in fact the case (although this is not relevant to me being able to optimise my journey). I split the data into the two sets of walking to and from UCL, computed the average speeds and used a t-test at the 5% significance level to calculate the probability of this difference between the two sets of journeys, which gave me the following results:

Average speed of walk to UCL	3.39mph
Average speed of walk from UCL	3.20mph
t-test result	p=0.0004

So I can certainly conclude at the 5% significance level that I walk faster to UCL than I walk from UCL, although this difference is not substantial (I estimate that it makes a difference of just 76 seconds to the total journey time).

It is also possible that the weather conditions along the route may affect how long my journey takes. It was beyond the scope of this project to examine the role that weather conditions played, but it would be relatively feasible, since there are data sources available which allow you to request weather data for a given point in space and time (The Dark Sky Company LLC, 2016). However, this would not provide insights around optimising the journeys that were taken, since it is not possible to modify the weather in order to optimise your journey.

Conclusions

There is no statistically significant difference between increasing and decreasing my walking speed through a journey, although the data I have collected suggests that there may be a difference. More data would be needed to confirm whether or not this is the case. If more data supports this suggestion, then I could optimise my journey by walking slower at the beginning of the journey and faster at the end of the journey.

My walking speeds around the Euston Road area were slow, but it is not possible to avoid crossing the Euston Road, so there is no way of optimising my journey in this respect. I did walk more slowly along Eversholt Street, but an optimisation of my journey based on avoiding this area would involve a complete change of route. Whether another route would lead to a more optimal journey would need to be assessed by collecting more data about other routes, i.e. by walking along these routes.

Limitations

The main limitation in this project was the quantity of data available. At first, it appeared that there was quite a lot of data available, but as the project went on, it became clear that, for instance, when the data was narrowed down to those journeys with increasing or decreasing speeds, the quantity of data and the level of differences that exist are quite small, and it is difficult to make meaningful conclusions.

Improving this situation would involve collecting more data. In order to better optimise my own journey, this would involve collecting more data about my own walking patterns, which would be time-consuming, although certainly possible. The difficulty with this is that any suggestions for optimising my journey might be irrelevant by the time I could make statistically significant conclusions, as I could be graduating from UCL by this time. The other option would be to collect more data from other people undertaking similar or even different journeys, since the way that different people behave may be similar.

There are also some issues regarding the quality of the data. Owing to the means of data collection using satellite positioning methods, the location data is not particularly precise, and this manifests itself as fluctuations in speeds along the journeys. The method I used to rectify these fluctuations, a moving average, can introduce artefacts into the data (Goodman, 2013), which could have affected the conclusions (a journey could have been misclassified as having increasing or decreasing speed).

Implications

In principle, the data analysis that was carried out by me for this project could be extended to other data sets without intervention: hence, it would be possible to package this into a web service or app to analyse movement data, which could be used by anyone interested in finding out about their walking behaviour. This would require more work, however, and would require care around other people's data, since location data can be sensitive. With appropriate consent from the individuals using the service, this data could be aggregated across users it would be possible to gain more useful information about walking speeds in different locations. Indeed, the use of data collected by mobile phones to feed into traffic models has been suggested this year by one London Mayoral candidate, Sian Berry, who said that "there's loads of apps we could use that would give us much more fine-grained data to feed into a new model." (London Reconnections, 2016). The use of such data around walking speeds and the journeys that people take has the potential to be influential in informing cities' transport planning decisions.

Bibliography

Al-Azzawi, M. 2004. Factors affecting pedestrian walking speeds [online]. Thesis (PhD), Napier University. Available from: http://researchrepository.napier.ac.uk/2749/ [Accessed 29 April 2016].

Batty, M. 2003. Advanced spatial analysis: extending GIS. In: P. Longley and M. Batty, eds. Advanced spatial analysis: the CASA book of GIS. Redlands, CA: ESRI Press, pp. 81-108.

Butler, H. et. al. 2016. GeoJSON specification [online]. Available from: http://geojson.org/geojson-spec.html [Accessed 29 April 2016].

Dwoskin, E. 2015. Apps track users—once every 3 minutes. *Wall Street Journal*. [online] 24 Mar. Available from: http://www.wsj.com/articles/apps-track-usersonce-every-3-minutes-1427166955 [Accessed 29 April 2016].

Ecma International. 2013. Standard ECMA-404: the JSON data interchange format [online]. Available from: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf [Accessed 29 April 2016].

Fleishman, G. 2011. How the iPhone knows where you are [online]. Available from: http://www.macworld.com/article/1159528/smartphones/how-iphone-location-works.html [Accessed 29 April 2016].

GeoPy Contributors, 2015. Welcome to GeoPy's documentation! — GeoPy 1.10.0 documentation [online]. Available from: https://geopy.readthedocs.io/en/1.10.0/#module-geopy.distance [Accessed 30 April 2016].

Goodman, G., 2013. Data corruption by running mean 'smoothers' [online]. Available from: https://judithcurry.com/2013/11/22/data-corruption-by-running-mean-smoothers/ [Accessed 2 May 2016].

Karusisi, N. et. al. 2014. Environmental conditions around itineraries to destinations as correlates of walking for transportation among adults: the RECORD cohort study. *PLoS One* [online], 9 (5). Available from: doi:10.1371/journal.pone.0088929 [Accessed 29 April 2016].

Kington, J. 2013. python - find minimum distance from point to complicated curve [online]. Available from: http://stackoverflow.com/questions/19101864/find-minimum-distance-from-point-to-complicated-curve#answers-header [Accessed 1 May 2016].

London Reconnections. 2016. The politics of integration: talking transport with Sian Berry [online]. Available from: http://www.londonreconnections.com/2016/the-politics-of-integration/ [Accessed 2 May 2016].

Mapbox. 2016. Mapbox studio [online]. Available from: https://www.mapbox.com/mapbox-studio/ [Accessed 29 April 2016].

ProtoGeo Oy. 2016. How does Moves work? [online]. Available from: http://movesapp.zendesk.com/hc/en-us/articles/201098676-How-does-Moves-work- [Accessed 29 April 2016].

the Python community. 2015. Shapely 1.5.15: python package index [online]. Available from: https://pypi.python.org/pypi/Shapely [Accessed 1 May 2016].

Revision World Networks Ltd. n.d. The product moment correlation coefficient [online]. Available from: http://revisionmaths.com/advanced-level-maths-revision/statistics/product-moment-correlation-coefficient [Accessed 1 May 2016]

StatsDirect Limited. 2016. Unpaired (two sample) t test [online]. Available from: http://www.statsdirect.com/help/default.htm#parametric_methods/unpaired_t.htm [Accessed 1 May 2016].

The Dark Sky Company LLC. 2016. The dark sky forecast API [online]. Available from: https://developer.forecast.io/docs/v2#time_call [Accessed 2 May 2016].

The Scipy community. 2016. scipy.stats.linregress — SciPy v0.17.0 reference guide [online]. Available from: http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html [Accessed 1 May 2016].

Transport for London. 2004. Making London a walkable city: the walking plan for London [online]. London. Available from: http://www.rudi.net/files/walking-plan-2004.pdf [Accessed 1 May 2016].

Weisstein, E.W. n.d. Moving Average [online]. Available from: http://mathworld.wolfram.com/MovingAverage.html [Accessed 13 April 2016].

Appendices

The following pages (including this page) are appendices containing all code used in the project and should not be regarded as part of the six page assessed submission.

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar  6 10:47:51 2016

@author: REDACTED
"""

#This program imports the relevant data from the Moves API
#It calculates the speeds at all points within each commute
#It then exports the data to a GeoJSON file for plotting

from moves import Moves
from geopy.distance import vincenty
import datetime as dt
import json

#Input the authorisation information for the Moves API
m = Moves("REDACTED","REDACTED","REDACTED")

# Set up a list of the locations we're interested in.
my_locs = [
    {
        'name': 'home',
        'lat' : 51.5376610267,
        'lon' : -0.1334586581
        },
    {
        'name': 'work',
        'lat' : 51.5245335649,
        'lon' : -0.134065590859
        }
    ]

# A function to check if the distance between two locations
# is less than a set distance
def compare_dim(one,two,d):
    dist = vincenty(one, two).meters
    print dist
    if dist<d:
        return True
    else:
        return False


def compare_loc(sample, rules):
#Check if a location matches one of the locations in the list
    sample_lat = float(sample['lat'])
    sample_lon = float(sample['lon'])
    sample_loc = (sample_lat,sample_lon)
    for rule in rules:
        rule_loc = (rule['lat'],rule['lon'])
```

```python
        if compare_dim(rule_loc,sample_loc,500) == True:
                return rule['name']
    return False

def append_path(any_list):
#Add a path to the list of paths
    any_list.append({
        'from_location': from_location_name,
        'data': move_segment['activities'][0],
        })

#This is the access token that allows us to access the data from Moves
access_token = "REDACTED"

#Set up a list for the dates
dates_list = []
#Start date here
start_date = dt.date(2015,9,1)
#Step for data requests
step = dt.timedelta(6)
today = dt.date.today()

#Go through all the dates from the start date to today
for x in range(100):
    if (start_date + step)<today:
        date_set = [start_date, start_date + step]
        start_date += step
        dates_list.append(date_set)
    else:
        date_set = [start_date, today]
        dates_list.append(date_set)
        break

#Set up a list for us to dump all the journeys
commute_journeys=[]

#COLLECTING ALL THE DATA:

for date_set in dates_list:

    #Get the data from the Moves API.
    #Date converted into the way Moves wants it.
    from_date = date_set[0].strftime('%Y%m%d')
    to_date = date_set[1].strftime('%Y%m%d')

    #Here we're calling the moves library.
    data = m.get_storyline(access_token,from_date,to_date, 'true')

    #Now that we have the data, let's pick out what we want
    for day in range(len(data)):
        segments = data[day]['segments']
```

```
        if segments != None:

            counter = 0

            for segment in segments:

            #We're only interested in journeys that start at a place
                if segment['type'] == 'place':

                    #See if the location is one of the ones in the list
                    #then name it as that.
                    #compare_loc will return False if there is no match
                    from_location_name = \
                    compare_loc(segment['place']['location'],my_locs)

                    #Only proceed if there is a valid location name
                    if from_location_name != False:

                        #Only proceed if there are enough segments
                        if len(segments)>counter+2:
                            #Name the subsequent segments.
                            move_segment = segments[counter+1]
                            to_segment = segments[counter+2]

                            #Check if the segments are of the right type
                            #(We want a sequence of place-move-place)
                            if move_segment['type'] == 'move':
                                if to_segment['type'] == 'place':

                                    to_location = \
                                    to_segment['place']['location']
                                    to_location_name = \
                                    compare_loc(to_location, my_locs)

                                    #If they are valid pairs,
                                    #add to the list of commutes.
                                    if from_location_name == 'home':
                                        if to_location_name == 'work':
                                            append_path(commute_journeys)
                                    if from_location_name == 'work':
                                        if to_location_name == 'home':
                                            append_path(commute_journeys)
                counter += 1

#Function to extract the date and time from a Moves output
def time_parser(time):
    time = str(time)
    y=int(time[0:4])
    m=int(time[4:6])
    d=int(time[6:8])
    h=int(time[9:11])
```

```python
    mi=int(time[11:13])
    s=int(time[13:15])
    time=dt.datetime(y,m,d,h,mi,s)
    return time

#Set up a suitable GeoJSON formatted dictionary for the results
results = {
    u'type': u'FeatureCollection',
    u'features': []
    }

#Alias the relevant bit
export_features = results['features']

#Loop through the commutes

for walk in range(len(commute_journeys)):

    reference_time = 0

    #Alias trackPoints
    trackPoints = commute_journeys[walk]['data']['trackPoints']

    for point in range(len(trackPoints)-1):
        #Get lat and lon into the right format
        #for geopy to take distances
        from_loc = (trackPoints[point]['lat'],trackPoints[point]['lon'])
        to_loc = (trackPoints[point+1]['lat'],trackPoints[point+1]['lon'])

        #Use vincenty to determine the distance between the two locations
        distance = vincenty(from_loc, to_loc).meters

        #Use time_parser to extract the times from the Moves data
        from_time = time_parser(trackPoints[point]['time'])
        to_time = time_parser(trackPoints[point+1]['time'])
        #Find the time taken between the two locations
        time_taken = (to_time - from_time).total_seconds()
        #Fix the time taken if it =0
        if time_taken == 0:
            time_taken = 1

        reference_time += time_taken

        #Find the speed - this is given in m/s
        speed = distance/time_taken
        print speed

        #write the speed to the first trackPoint in the pair
        trackPoints[point]['speed']=speed
        trackPoints[point]['reference_time']=reference_time

```

```
    #Now we're looking to write to a GeoJSON file.
    for point in range(len(trackPoints)-1):
        line_data = trackPoints[point]
        line_data2 = trackPoints[point+1]

        #create a suitable GeoJSON object with the speed as a property
        line = {
                        u'geometry':
                            {
                                u'type':u'LineString',
                                u'coordinates': [[line_data['lon'], \
                                line_data['lat']],\
                                [line_data2['lon'],line_data2['lat']]]
                            },
                        u'type': u'Feature',
                        u'properties':
                            {
                            u'speed': line_data['speed']
                            }
                }

        #Append this object to the list
        export_features.append(line)

#Save everything to a file
export_file = open('export_file.json','w')
json.dump(commute_journeys,export_file)
export_file.close()

#Convert all results to JSON
json_export_file = open('new_data.json','w')
json.dump(results,json_export_file)
json_export_file.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Apr  8 10:57:53 2016

@author: REDACTED
"""
#THIS PROGRAM TAKES MOVING AVERAGES
#AND THEN RUNS REGRESSION ANALYSIS

#Import relevant libraries
import json
from scipy import stats
import numpy as np

#Open the file and load it as an object
exported_file = open('export_file_4.json','r')
commute_journeys = json.load(exported_file)
exported_file.close()

#Looping through all the commutes
for commute_number in range(len(commute_journeys)):

    reference_times = []
    speeds = []

    commute = commute_journeys[commute_number]
    trackPoints = commute['trackPoints']

    if len(trackPoints)>=0:

        #We have a reduced number of points
        #(we can't period average at the edges)
        for point_number in range(len(trackPoints)-4):
            #This will be the central point
            point_number_2 = point_number + 2
            point = trackPoints[point_number_2]
            #Points one to the left and right
            pointL = trackPoints[point_number_2-1]
            pointR = trackPoints[point_number_2+1]

            #Take the average
            speed_pavg = np.average([point['speed'],pointL['speed'],\
            pointR['speed']])
            #Put the average in to the data about the point
            point['speed_pavg'] = speed_pavg
            reference_times.append(point['reference_time'])
            speeds.append(speed_pavg)

        if reference_times != []:
            ref_test = True

```

```
        #Linear regression based on the moving average values
        commute['linr_pavg']={}
        linr = commute['linr_pavg']
        linr['slope'], linr['intercept'], linr['r'], linr['p'],\
        linr['std_err'] = stats.linregress(reference_times,speeds)

#Write to a file.
exported_file = open('export_file_5.json','w')
json.dump(commute_journeys, exported_file)
exported_file.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  7 11:05:55 2016

@author: REDACTED
"""

#THIS PROGRAM PLOTS GRAPHS BASED ON MOVING AVERAGES
#AND LINEAR REGRESSION

#Import the relevant libraries
import json
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

#Open the file
exported_file = open('export_file_5.json','r')
commute_journeys = json.load(exported_file)
exported_file.close()

#Set up lists we can put the sorted commutes into
speeds_down = []
speeds_up = []
speeds_neutral = []

#Function for putting the data into a form that we can plot
#Add this data to a list.
x=0
y=0
reference_times = []
neutral_commutes = []
speeds = []
def process_data(speeds_list):

    print commute['i_d']
    reference_times = []
    speeds = []

    for point in range(len(trackPoints)-3):
        if 'speed_pavg' in trackPoints[point+2]:
        #Put all the speed data into a list, paired with times
        #for speed/time plot
            reference_times.append(trackPoints[point+2]['reference_time'])
            speeds.append(trackPoints[point+2]['speed_pavg'])

    # 100 linearly spaced numbers
    x = np.linspace(0,commute['duration'],100)

    #A straight line based on the linear regression numbers
    y = commute['linr_pavg']['intercept'] + \
```

```
    commute['linr_pavg']['slope']*x

    plot()
    speeds_list.append(speed)

#A function for running the plot
def plot():
    #Set up a new figure for each commute
    plt.figure(commute['i_d'])
    #Label it so we can see it
    plt.title(commute['i_d'])
    #Label the axes
    plt.xlabel("time (s)")
    plt.ylabel("speed (m/s)")
    #Plot the line
    plt.plot(x,y)
    #Plot the data
    plt.plot(reference_times,speeds)
    #Set the axis values
    plt.axis([0,1600,0,6])

#Selecting the data that we've identified as increasing/decreasing
#speed journeys
for commute_no in range(len(commute_journeys)):
    commute = commute_journeys[commute_no]
    trackPoints = commute['trackPoints']

    #Only process those journeys that have had a period average
    #(some were too short for this to take place)
    if 'linr_pavg' in commute:

        speed = commute['distance']/float(commute['duration'])

        #speeds_down will be those which  have r<-0.4
        if commute['linr_pavg']['r']<-0.4:

            process_data(speeds_down)

        #speeds_up will be those which have r>0.4
        elif commute['linr_pavg']['r']>0.4:

            process_data(speeds_up)

        #everything else goes into the speeds_neutral list
        else:

            process_data(speeds_neutral)
            neutral_commutes.append(commute_no)

#Printing averages and t-test results for whatever we want
def compare(one,two):
```

```

    print np.average(one)
    print np.average(two)
    print stats.ttest_ind(one, two, axis=0, equal_var=True)

compare(speeds_down, speeds_up)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  7 10:44:57 2016

@author: REDACTED
"""

#THIS PROGRAM TRANSFORMS THE DATA FROM LAT/LON TO XY

import json
import pyproj as pp

exported_file = open('export_file_2.json','r')
commute_journeys = json.load(exported_file)
exported_file.close()

p1 = pp.Proj(proj='latlong',datum='WGS84')
p2 = pp.Proj(proj="utm",zone=30,datum='WGS84')

for commute_number in range(len(commute_journeys)):
    commute = commute_journeys[commute_number]
    trackPoints = commute['trackPoints']
    for point_number in range(len(trackPoints)):
        point = trackPoints[point_number]
        point['x'],point['y']=pp.transform(p1,p2,point['lat'],point['lon'])

exported_file = open('export_file_3.json','w')
json.dump(commute_journeys,exported_file)
exported_file.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  7 10:44:57 2016

@author: REDACTED
"""

#THIS PROGRAM FINDS THE POINT ON THE REFERENCE LINE
#WHICH IS CLOSEST TO EACH DATA POINT

import json
import shapely.geometry as geom


#Open the two files and import the data
exported_file = open('export_file_5.json','r')
commute_journeys = json.load(exported_file)
exported_file.close()

exported_file = open('datum_lines_xy','r')
datum_points = json.load(exported_file)
exported_file.close()

datum_for_plot = []

#Make these into a list
#Loop through all points in the datum
for point_no in range(len(datum_points)):
    datum_point = datum_points[point_no]
    x = datum_point['x']
    y = datum_point['y']

    #Append everything into a list of the form that shapely wants.
    datum_for_plot.append((x,y))

#Now call that a line.
line = geom.LineString(datum_for_plot)

#Looping through all the commutes
for commute_no in range(len(commute_journeys)):
    commute = commute_journeys[commute_no]
    trackPoints = commute['trackPoints']

        #Now set up all the points as shapely objects
        #THIS IS THE CODE FROM KINGTON (2013)
        #(with changes)
            for point_no in range(len(trackPoints)-1):
            trackPoint = trackPoints[point_no]
            point = geom.Point(trackPoint['x'], trackPoint['y'])

            #Determine the distance from each point to the datum line
```

```
            distance = line.distance(point)

            trackPoint['datum']=distance

            #If it's less than 100m, then put into the trackPoint data
            #some info about the datum location
            if distance<100:
                point_on_line = line.interpolate(line.project(point))
                trackPoint['datum']= {'x':point_on_line.x,\
                'y':point_on_line.y, 'datum':line.project(point)}

#Write to a file.
exported_file = open('export_file_6.json','w')
json.dump(commute_journeys, exported_file)
exported_file.close()
```

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  7 10:44:57 2016

@author: REDACTED
"""

#This program splits the datum line into segments,
#and then averages the speeds of all journeys matching these segments

#Import the relevant libraries
import json
import shapely.geometry as geom
import numpy as np
import pyproj as pp

#Open the data about the journeys
exported_file = open('export_file_6.json','r')
commute_journeys = json.load(exported_file)
exported_file.close()

#Set the length of the datum (5km) and create empty lists for the speeds
datum_length = 5000
datum_speeds = [0] * datum_length
datum_averages = [0] * datum_length

for datum_point in range(int(datum_length/10)):

    #Split the datum into 10m long sections
    min = 10 * (float(datum_point))
    max = 10 * (float(datum_point + 1))

    #Create an empty list for each of the sections
    datum_speeds[datum_point]=[]

    #Looping through all the commutes and all the trackPoints
    for commute_no in range(len(commute_journeys)):
        commute = commute_journeys[commute_no]

        #Only include those which started at home
        if commute['from_location'] == 'home':
            trackPoints = commute['trackPoints']
            for point_no in range(len(trackPoints)):
                trackPoint = trackPoints[point_no]

                #Only proceed if we've already assigned datum
                #information to the trackPoint
                if 'datum' in trackPoint and \
                type(trackPoint['datum'])==dict:
                    datum = trackPoint['datum']['datum']

```

```
                    #If in the range, append the speed
                    #to the list of speeds
                    if datum > min and datum < max:
                        datum_speeds[datum_point].append(trackPoint['speec

    #Take an average of the speeds and assign it to the list of averages
    datum_averages[datum_point] = np.average(datum_speeds[datum_point])

#Set up a suitable GeoJSON formatted dictionary for the results
results = {
    u'type': u'FeatureCollection',
    u'features': []
    }

#Alias the relevant bit
export_features = results['features']

#Open the datum file
exported_file = open('datum_lines_xy','r')
datum_points = json.load(exported_file)
exported_file.close()

#Make these into a list
#Loop through all points in the datum
datum_for_plot=[]

for point_no in range(len(datum_points)):
    datum_point = datum_points[point_no]
    datum_averages[point_no]={'speed':datum_averages[point_no]}
    x = datum_point['x']
    y = datum_point['y']
    datum_averages[point_no]['x']=datum_point['x']
    datum_averages[point_no]['y']=datum_point['y']

    #Append everything into a list of the form that shapely wants.
    datum_for_plot.append((x,y))

#Now call that a line in shapely.
line_string = geom.LineString(datum_for_plot)

#Set up the two projections
p1 = pp.Proj(proj='latlong',datum='WGS84')
p2 = pp.Proj(proj="utm",zone=30,datum='WGS84')

#Convert the x and y to lat and lon
for item_no in range(len(datum_averages)):
    item = datum_averages[item_no]
    if type(item) == dict:
        item['lat'],item['lon']=pp.transform(p2,p1,item['x'],item['y'])

#Add everything to the GeoJSON dictionary
```

```
for point in range(len(datum_averages)-1):

        line_data = datum_averages[point]
        line_data2 = datum_averages[point+1]
        if type(line_data2) == dict and type(line_data) == dict:

            #create a suitable GeoJSON object with the speed as a property
            line = {
                        u'geometry':
                            {
                                u'type':u'LineString',
                                u'coordinates': [[line_data['lon'],     \
             line_data['lat']],[line_data2['lon'],line_data2['lat']]]
                            },
                        u'type': u'Feature',
                        u'properties':
                            {
                            u'speed': line_data['speed']
                            }
                    }

            #Append this object to the list
            export_features.append(line)

#Write to a file.
exported_file = open('datum_speeds_home.json','w')
json.dump(results, exported_file)
exported_file.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 15 17:02:33 2016

@author: REDACTED
"""

#THIS PROGRAM TAKES AVERAGES OF JOURNEYS
#TO AND FROM UCL

#Import relevant libraries
import numpy as np
from scipy import stats

#Here, I was lazy and just pasted in the data rather than importing a file

home =[]
work = []

#Split up the journeys into going to and from UCL
for item in saved_data:
    speed = item['data']['distance']/item['data']['duration']
    if item['from_location']=='home':
        home.append(speed)
    if item['from_location']=='work':
        work.append(speed)

#Now print out some statistical tests on the data
#Averages
print np.average(home)
print np.average(work)

#Standard deviations
print np.std(home)
print np.std(work)

#T-test on the two sets
print stats.ttest_ind(home, work, axis=0, equal_var=True)
```