Identifying Rideable Cycle Routes in Duluth, MN through PGRouting and Social Media

Ian Bachman-Sanders, JohnMark Fisher, Brian Robinson, Kimberly Sundeen

Introduction

Upon learning the city of Duluth, MN has limited cycling infrastructure and being introduced to a local cycling organization, Duluth Bikes[1], we decided to create a database of information to help identify existing and potentially rideable, safe, useful, and/or interesting cycle routes in Duluth, MN for our final Geography 574 project. We intend for organizations like Duluth Bikes to utilize this database and subsequent applications for two purposes:

1. Showcase interesting destinations and identify the safest existing bike routes
2. Help city planners decide where to build safe dedicated bike routes that would maximize access to interesting and popular places and businesses in Duluth, MN

We also aimed to serve non-professional users by providing a free and open database application[2] that helps users unfamiliar with the area query the data in order to safely explore Duluth by bike. Our goal of this database is for it to be used to promote biking in and around Duluth, whether it is for commuting to and from work, running errands, or for leisure activities.

While in discussions regarding how to ultimately calculate the most efficient cycling route we were exposed to pgRouting, a PostGIS tool[3]. The pgRouting function that interested us the most was pgr_dijkstra,[4] which minimizes a given 'cost' variable to calculate the optimal path from one point to another using Edsger Dijkstra's algorithm[5]. This tool analyzes every possible route between two points and only returns the shortest outcome, determined by 'cost', to a table of line segments in the order in which they must be traveled. Generally, 'cost' is the length of a line segment; however, it may be manipulated to include other variables. The key to our project was creating a score variable based on social media data as well as various road features that may benefit or inhibit cycling with which to adjust the cost of paths between two points.

[1] DuluthBikes.org

[2] http://grad.geography.wisc.edu/cycleroutes

[3] http://pgrouting.org/

[4] http://docs.pgrouting.org/2.2/en/src/dijkstra/doc/pgr_dijkstra.html

[5] https://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html

Model Design

Based on best data available and various geospatial data manipulation tools, we implemented a database described below by the following logical schema and ER diagram:

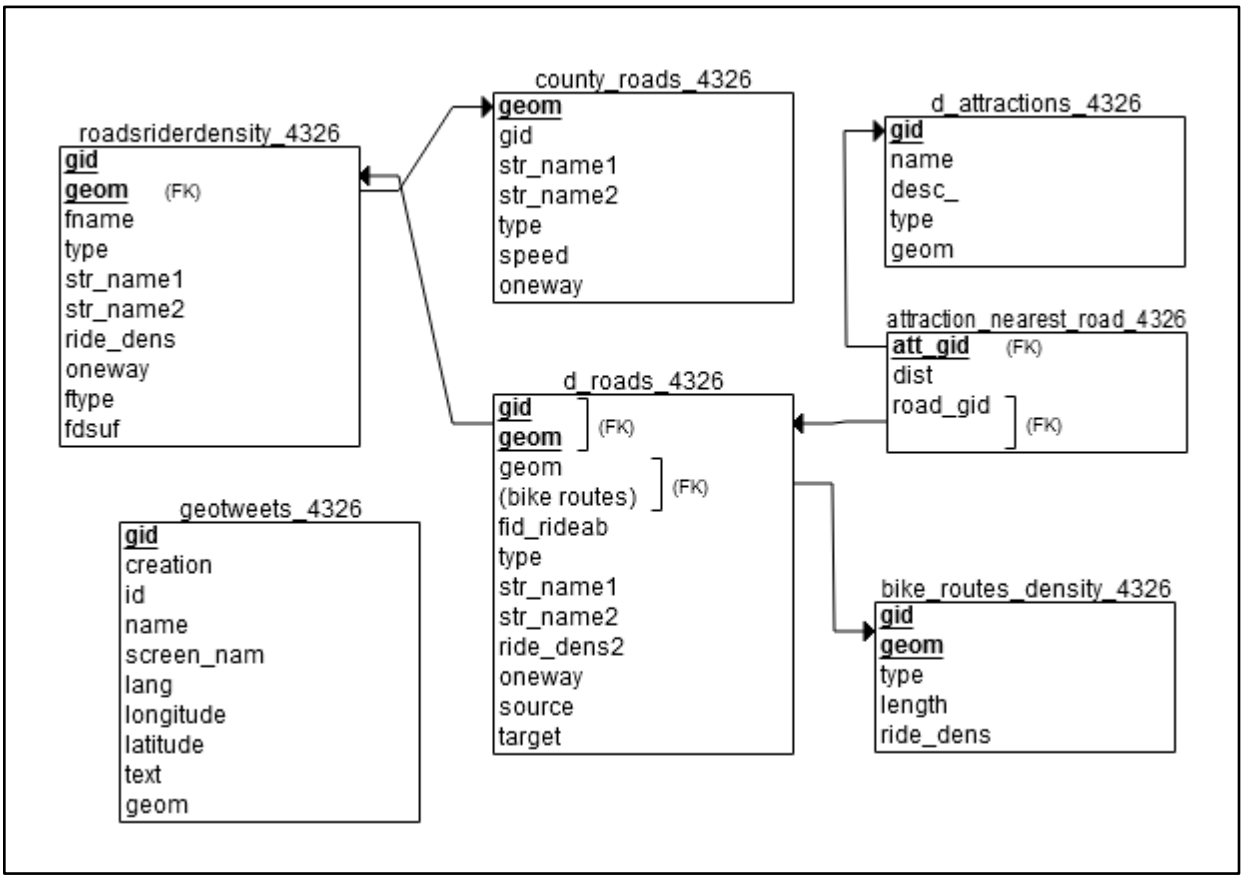

Figure 1: Logical Schema of Cycling Routes Database

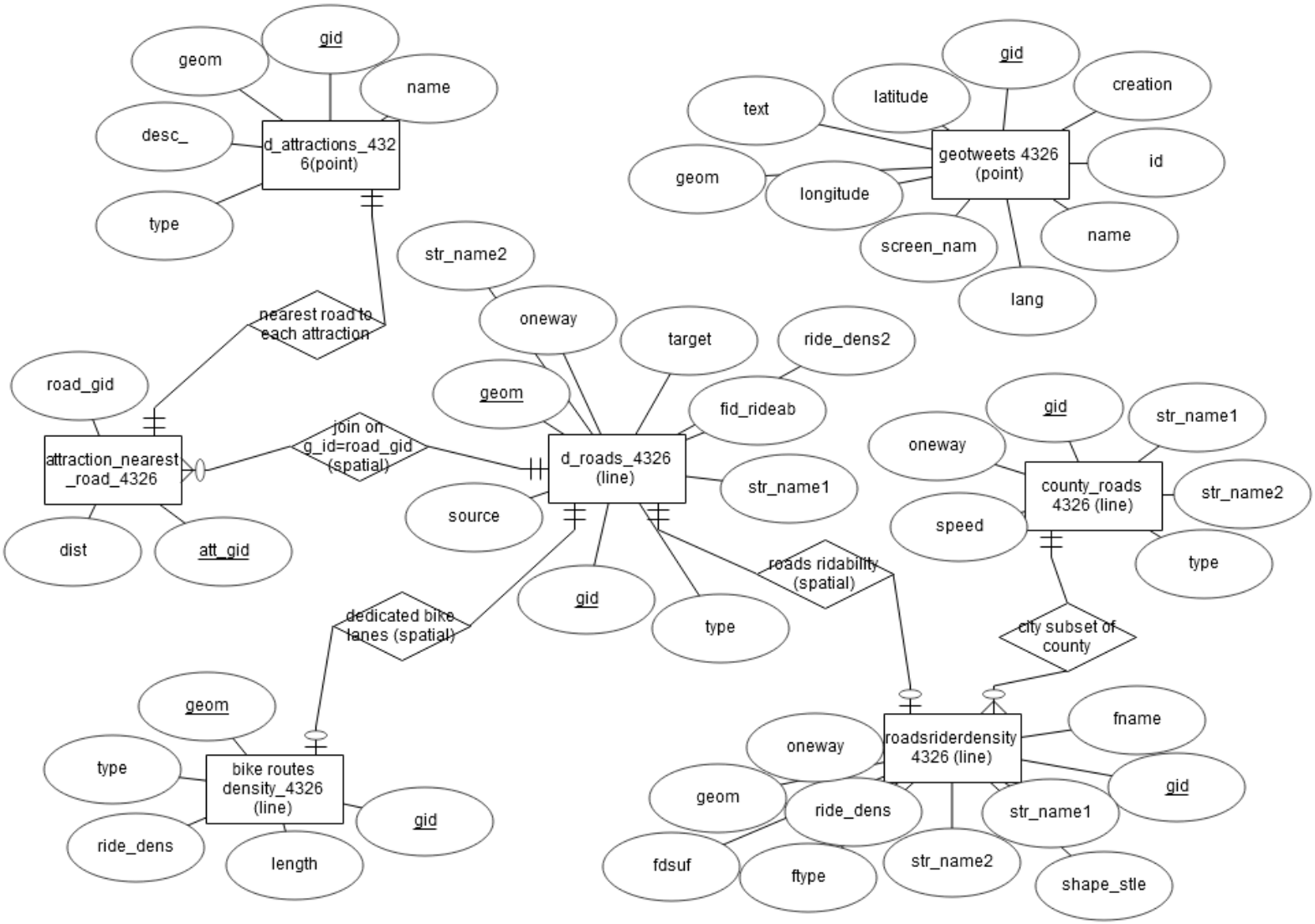
gid
geom
name
desc_
d_attractions_4326(point)
type
gid
text
latitude
creation
geom
longitude
geotweets 4326 (point)
id
screen_nam
name
lang
str_name2
oneway
target
ride_dens2
nearest road to each attraction
road_gid
geom
fid_rideab
join on g_id=road_gid (spatial)
attraction_nearest_road_4326
d_roads_4326 (line)
str_name1
source
dist
att_gid
gid
type
roads ridability (spatial)
gid
str_name1
oneway
county_roads 4326 (line)
str_name2
speed
type
dedicated bike lanes (spatial)
city subset of county
geom
type
bike routes density_4326 (line)
gid
ride_dens
length
fname
oneway
roadsriderdensity 4326 (line)
gid
geom
ride_dens
str_name1
fdsuf
ftype
str_name2
shape_stle

Figure 2: Entity Relationship Diagram of Cycling Routes Database

A brief description of the database's relationships are as follows:

1. *county_roads_4326* - *roadsriderdensity_4326* (spatial): **1...1:0...N** *roadsriderdensity_4326* is a pgRouting ready Duluth-only subset of *county_roads_4326*. Therefore, each street is broken up into even smaller segments at each node for future manipulation by pgr_createTopology[6]. One *county_road_432* segment is associated with 0 (not in Duluth) or many *roadsriderdensity_4326* segments and each *roadsriderdensity_4326* segment must be associated with 1 *county_road_4326* segment.
2. *roadsriderdensity_4326* - *d_roads_4326* (spatial): **0...1:1...1** Every segment in *roadsriderdensity_4326* must be associated with 1 *d_roads_4326* segment, however, there are some bike routes in *d_roads_4326* that do not exist in *roadsriderdensity_4326* (roads/streets/highways only) so each *d_roads_4326* segment is associated with 0 or 1 *roadsriderdensity_4326* segment.
3. *bike_routes_density_4326* - *d_roads_4326* (spatial): **0...1:1...1** Each bike route segment must be associated with one *d_roads_4326* segment but each *d_roads_4326* segment may be associated with 0 or 1 bike route (*d_roads_4326* also contains segments that are not bike routes). Those bike routes that were not along roads and therefore were completely unique had their geometries and other attributes added to *d_roads_4326*.
4. *d_roads_4326* - *attraction_nearest_road_4326* (spatial): **1...1:0...N** Each *d_roads_4326* segment may be associated with 0 or many attractions whereas each attraction must be associated with its nearest *d_roads_4326* segment.
5. *d_attractions_4326* - *attraction_nearest_road_4326*: **1...1:1...1** There is a 1:1 mandatory relationship going both ways since every attraction can only have one nearest road's gid associated with it.

Database Implementation

We utilized the University of Wisconsin's geospatial server and had a PostgreSQL database named 'cycleroutesdb' created for us. A SSH login secures access, and data manipulated and queried through pgAdmin. Both PostGIS and pgRouting extensions were added to our database management system by the Geography Department's IT Manager, Jay Scholz.

Each entity in the database required data collection, organization, recalculation, and geospatial manipulation. The majority of the data in our table is unique to our project, and either directly calculated

[6] http://docs.pgrouting.org/2.2/en/src/topology/doc/pgr_createTopology.html

by our work or generated by manipulating data we created. The process to develop each entity to its final iteration went as follows:

county_roads_4326 Table Development

In order to identify potential cycle routes, at the very least we needed road data for the city of Duluth, MN. After discussions with Mike Casey, a representative from Duluth Bikes, we decided to use St. Louis County's shapefile of all roads within the county[7]. This was used as a foundation for creating other entity types and the only manipulation we performed was projecting it to Spatial Reference System Identifier (SRID) 4326 (WGS 1984) from its original SRID 32615 (WGS 1984 / UTM Zone 15N) to display on a web application that relies on this database. We generally imported shapefiles using the pgAdmin "PostGIS Shapefile and DBF Loader" graphic user interface (GUI) setting SRID to 4326. However, the following command line statements may have been used in the command prompt[8]:

```
shp2pgsql -s 4326 -I county_roads_4326 public.county_roads_4326 >
county.sql
psql -h 144.92.235.47 -d cycleroutesdb -U cycleroutesuser -f county.sql
```

-s: to SRID
-I: creates a GiST index on the geometry column
-h: database server host
-d: database name
-U: user name
-f: SQL file called

roadsriderdensity_4326 Table Development

The project required only select fields in the county roads shapefile and only those roads within Duluth city limits. As a result, we removed excess fields (outlined in Appendix) and used a definition query, "`"CITYLEFT" = 'Duluth' OR "CITYRIGHT" = 'Duluth'`" in ArcMap 10.4 to select Duluth city roads, exporting the results as a new shapefile, *roadsriderdensity_4326*. This file is used for pgRouting least-cost calculations. However, the pgRouting algorithm can only process line strings, and *roadsriderdensity_4326* was stored as multiline strings. We used the "Split Line at Vertices" tool in ArcMap 10.4 to resolve this issue. As a result, each line segment in *county_roads_4326* is associated with 0 (not in Duluth city) to many segments (each connecting vertices of the original multiline string) in *roadsriderdensity_4326*.

bike_routes_density_4326 Table Development

[7] http://www.stlouiscountymn.gov/LAND-PROPERTY/Maps/Data
[8] http://www.bostongis.com/pgsql2shp_shp2pgsql_quickguide.bqg

We also wanted data regarding existing bike lanes in Duluth, whether they were on a road shared with cars or completely separate, such as trails through a park or along the shore of Lake Superior. This data was collected and given to us by Duluth Bikes in a shapefile. The St. Louis County road file also included some of these bike paths, but was not complete, and we removed any bike paths from the road shapefiles to avoid redundancies. Once bike routes and road segments were established, rider density could be calculated based on a combinations of variables stored within the *bike_routes_density_4326* and *roadsriderdensity_4326* and outside sources.

d_roads_4326 Table Development

The tables *bike_routes_density_4326* and *roadsriderdensity_4326* were used to adjust the distance-cost calculated in the Dijkstra algorithm. This score changes the cost of a road segment by being set as the divisor of distance in the cost calculation. We used a scale of 0.0001-100 (non 0) to describe rider density. Therefore, if a segment is 100 meters long and has a rider density score of 100, its cost is 1 (100/100 = 1). If a similar 100 meter segment has a score of 1, its cost is 100 (100/1 = 100). The Dijkstra algorithm would then select the 1 cost segment because it is the lesser of the two.

The detailed process used to calculate rider density cost score, 'ride_dens2' in *d_roads_4326*, can be found in the Appendix. In brief, *ride_dens2* is the result of a multivariate analysis, the result of which was stored in polylines. Central to this analysis was a global heatmap created by the app Strava[9] which shows bike rider density on roads and paths based on its users. Strava is a fitness tracking application and social media platform which utilizes GPS to record activities. It calculates biking or running speed, elevation, and distance traveled based on data captured through phones or other personal GPS devices. Strava annually updates their heatmap based on data from millions of users to convey exactly where its users most frequently bike or run. In our case, we limited the extent to Duluth and filtered for only biking activities[10]. We processed the heatmap in Photoshop to create a monochromatic .tiff file, which we then georeferenced using ArcMap, and classified using the range of color saturation in the .tiff to measure density. Then we overlaid Duluth roads and bike routes separately onto the .tiff raster, created buffers around each segment, and calculated zonal statistics of the average rider density for each segment. Finally, we joined that data to the road/route shapefiles, creating *roadsriderdensity_4326* and *bike_routes_density_4326*. If a road was widely used (high density) in Strava, we considered it a good option for a cycling. If there was low usage (low density), then we assumed the road is not safe or conducive for cycling and we considered it a poor option. Segments were additionally scored based on

[9] https://www.strava.com/

[10] http://labs.strava.com/heatmap/#12/-92.13032/46.78184/blue/bike

their speed limit and road type to avoid interstates where cycling is illegal or highways where cars travel quickly and can be unsafe.

Finally, the shapefile *d_roads_4326* was created based on a spatial join between *bike_routes_density_4326* and *roadsriderdensity_4326*. Therefore, *d_roads_4326* contains all highways, roads, roads with bike lanes, and stand-alone bike lanes and their "ridability" scores within the city of Duluth. The following image shows all bike paths (green) and where they are in relation to our final working roads file, *d_roads_4326* (purple):

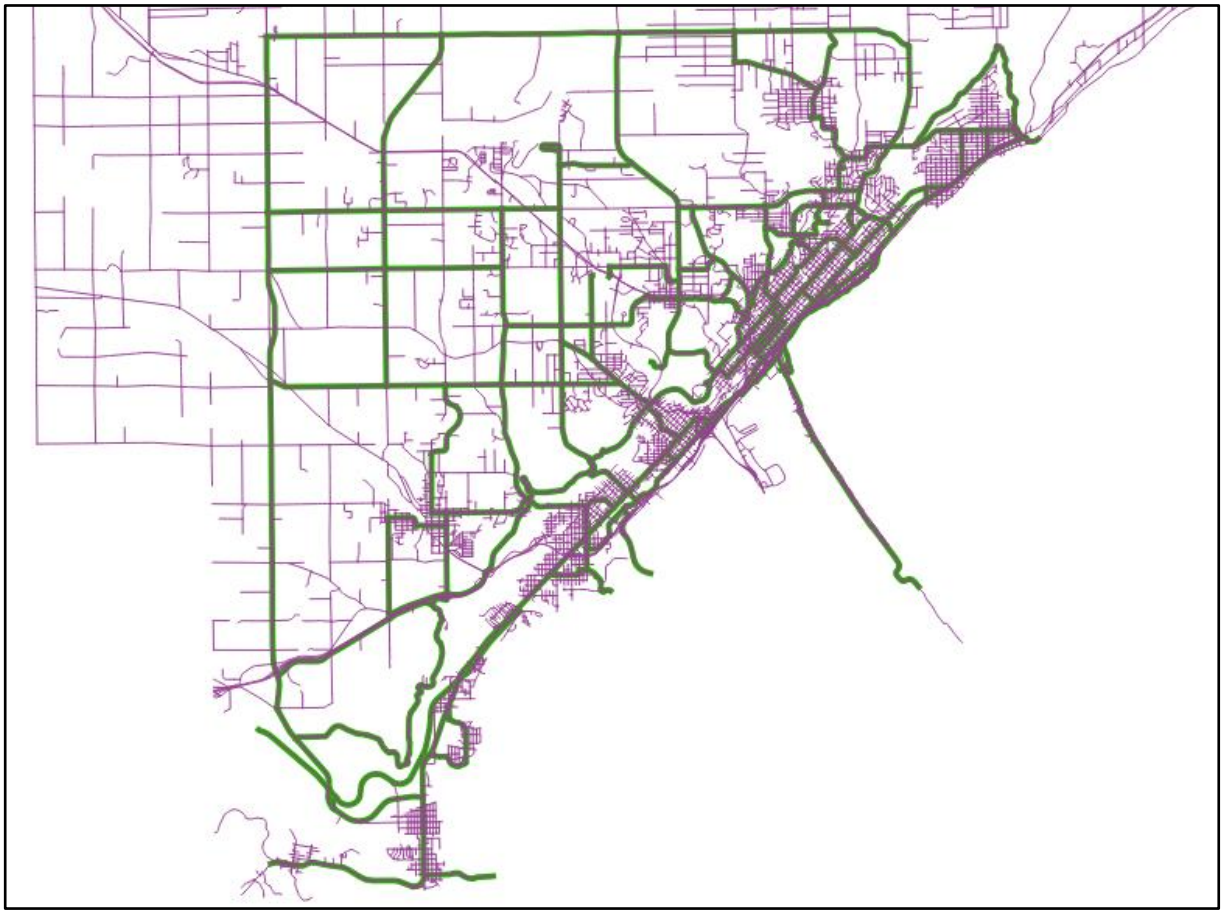

Figure 3: Bike paths and roadways in Duluth, Minnesota

Now that *d_roads_4326*, *bike_routes_density_4326*, and *roadsriderdensity_4326* were ready to be added to the database, we used pgAdmin again, as we did with *county_roads_4326*, to import the shapefile through the program GUI. One important item to note is that an Import Option change was required to make sure they were imported as single line shapefiles rather than multilines, checking the following box:

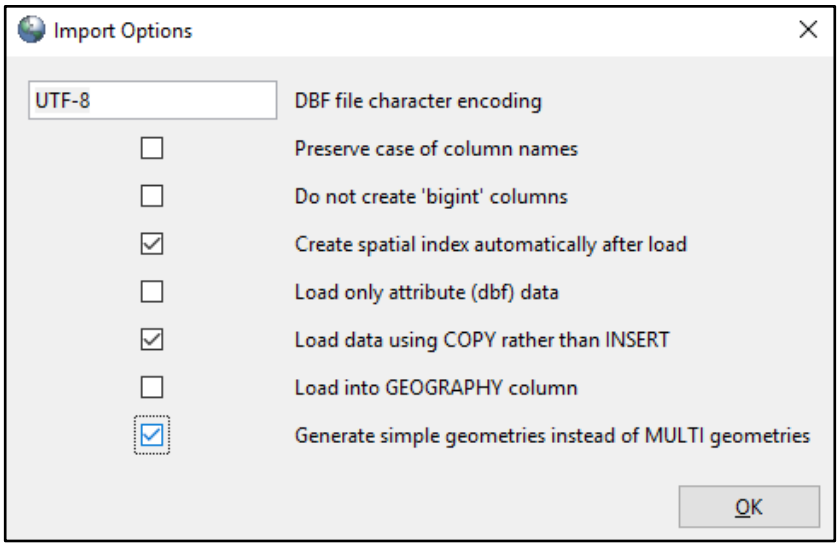

Figure 4: Using pgAdmin PostGIS Shapefile and DBF Loader GUI options to import simple linestrings

However, the same shapefiles could have been imported using command line statements similar to the following (note the "-S" which makes sure they are "simple" geometries instead of MULTI):

- `shp2pgsql -s 4326 -I -S d_roads_4326 public.d_roads_4326 > d_roads.sql`
  `psql -h 144.92.235.47 -d cycleroutesdb -U cycleroutesuser -f d_roads.sql`
- `shp2pgsql -s 4326 -I -S roadsriderdensity_4326 public.roadsriderdensity_4326 > dens.sql`
  `psql -h 144.92.235.47 -d cycleroutesdb -U cycleroutesuser -f dens.sql`
- `shp2pgsql -s 4326 -I -S bike_routes_density_4326 public.bike_routes_density_4326 > bike.sql`
  `psql -h 144.92.235.47 -d cycleroutesdb -U cycleroutesuser -f bike.sql`

d_roads_4326 and pgRouting Table Development

Once *d_roads_4326* was successfully added to the database including its 'ride_dens2' field, it needed to be formatted for pgRouting functionality. This involved adding two fields and creating a topology, using the following SQL statements:

```
alter table d_roads_4326 add column source integer;
alter table d_roads_4326 add column target integer;
select pgr_createTopology('d_roads_4326', 0.0001, 'geom', 'gid');
```

What this does is creates two new fields, 'source' and 'target', which are used to hold integer values for each line segment. When using pg_Dijkstra, the user must supply both a 'source' and 'target' value to their query. This identifies to pgRouting where to start and end the route, so it can identify how to get from point A ('source') to point B ('target') most efficiently. Creating topology generates a new table in our database called *d_roads_4326_verticies_pgr*, which is used for calculation purposes when pg_Dijkstra runs. It is populated with values and then wiped clean during the processing of a query.

geotweets_4326 Table Development

The remaining spatial datasets are both point shapefiles. One of which, *geotweets_4326*, is completely stand alone with no spatial relationships to the other datasets. This is a collection of tweets within the Duluth, MN area collected from Twitter's public stream throughout the spring using the Twitter API[11]. This, along with the *d_attractions_4326* dataset, may be used by a cycler to decide where interesting places may be in Duluth and also adds another social media factor to our database. Twitter allows developers to access their public data via a streaming endpoint. We used the python library Tweepy[12] in consultation with Scott Farly[13] of the University of Wisconsin, Madison, geography department. The resulting program (see Appendix) could be run at any time to collect geotagged tweets from the Duluth area and write them to a CSV file, which could be easily converted to a shapefile. These tweet points will be used for various spatial queries relating them to roads or attractions. Similarly, this was uploaded using the PostGIS GUI shapefile loader.

d_attractions_4326 Table Devlopment

The final two entities in the database are *d_attractions_4326*, an independent point shapefile, and its intermediate, dependent table *attraction_nearest_road_4326*, relating it to *d_roads_4326*. *d_attractions_4326* houses information on the top 22 points of interest in Duluth, MN as listed on TripAdvisor[14]. Each attraction's latitude, longitude, type of attraction, and a brief description, were added to a table. Utilizing the lat/lon values, we were able to create a shapefile from this in ArcMap using the "Make XY Event Layer" tool. Finally, this was uploaded using the PostGIS GUI shapefile loader. Similar to the tweets, attractions are designed to encourage exploration and be used as destinations by users, and will be leveraged using a variety of spatial queries.

attraction_nearest_road_4326 Table Development

This cycling app is designed, in part, to encourage users to cycle from one attraction to another. In order to facilitate this functionality, we built a permanent table connecting each attraction to a road segment in *d_roads_4326*. The following query was used to create this table based on existing fields in *d_roads_4326* and *d_attractions_4326*.

```
create table attraction_nearest_road_4326 (
road_gid integer,
att_gid integer,
dist double precision,
```

[11] https://dev.twitter.com/streaming/public

[12] https://github.com/tweepy/tweepy

[13] https://github.com/scottsfarley93/twitterOnDemand

[14] https://www.tripadvisor.com/Attractions-g43018-Activities-Duluth_Minnesota.html#ATTRACTION_SORT_WRAPPER

```
primary key(att_gid));
```

To populate this table, the following spatial query was run 22 times, altering the "`where d.gid= x`" portion each time to calculate the nearest road segment to each attraction.

```
insert into attraction_nearest_road_4326 (road_gid, att_gid, dist)
select r.gid as road_gid, d.gid as att_gid, ST_Distance(d.geom,r.geom)
as dist from d_attractions_4326 as d, d_roads_4326 as r where d.gid = 1
order by dist limit 1
```

Once completed, we have the ability to join these tables during client interactions without running the spatial distance calculation to find the nearest road/attraction.

With all of our shapefiles and tables properly stored in our database, we could begin answering our research questions about safely and quickly cycling to interesting places in Duluth using pgRouting, spatial, and non-spatial SQL queries.

Database Research Questions

The following set of queries answer various research questions as well as other interesting and potentially useful pieces of information in our database or future research.

- Identify which street (no bike lane) cyclists use the most in Duluth.

```
select str_name1, ride_dens from roadsriderdensity_4326
where ride_dens = (select max(ride_dens) from roadsriderdensity_4326)
```

Commonwealth Ave with a density of 99.102

- Identify how many tweets there were within 1 mile of a given attraction.

```
select count(name) from geotweets_4326 as t
where ST_Dwithin ((ST_Transform(t.geom,3857)), (select
ST_Transform(geom,3857)
from d_attractions where name = 'Duluth Depot'),1609)
```

44 total tweets. The name of the attraction may be edited accordingly.

- Identify how one would most efficiently cycle from the western end of the Skyline Parkway to the eastern end purely based on distance (QUERY 1) and then by distance *and* 'ride_dens2', our "ridability" score variable (QUERY 2). Compare the lengths of these route options.

QUERY 1:

```
SELECT geom FROM pgr_dijkstra(
'SELECT gid as id, source, target, st_length(geom) as cost FROM
d_roads_4326',
(select source from d_roads_4326 where gid = (
select road_gid from attraction_nearest_road_4326 where att_gid = (
select gid from d_attractions_4326 where name = 'Skyline Parkway' and desc_
like '%western end%'))), (select source from d_roads_4326 where gid = (select
road_gid from attraction_nearest_road_4326 where att_gid = (select gid from
```

```
d_attractions_4326 where name = 'Skyline Parkway' and desc_ like '%eastern
end%'))), false, false)
as di JOIN d_roads_4326 pt ON di.id2 = pt.gid;
```

Then, only a small part of this query needs to be edited in order to incorporate our "ridability" score, 'ride_dens2', setting it as the divisor of st_length:

QUERY 2:

```
...st_length(geom)/ride_dens2 as cost...
```

Since these queries produce a rather long list of road segments, the best way to understand the results is to visualise them. To do so, we utilized the QGIS "DB Manager" tool for querying a PostGIS database and then displaying the results as a shapefile as long as the output contains the input file's geometry field. The following image shows the distance-only route (blue) compared to our route (red) on top of Duluth's streets and bike routes (purple).

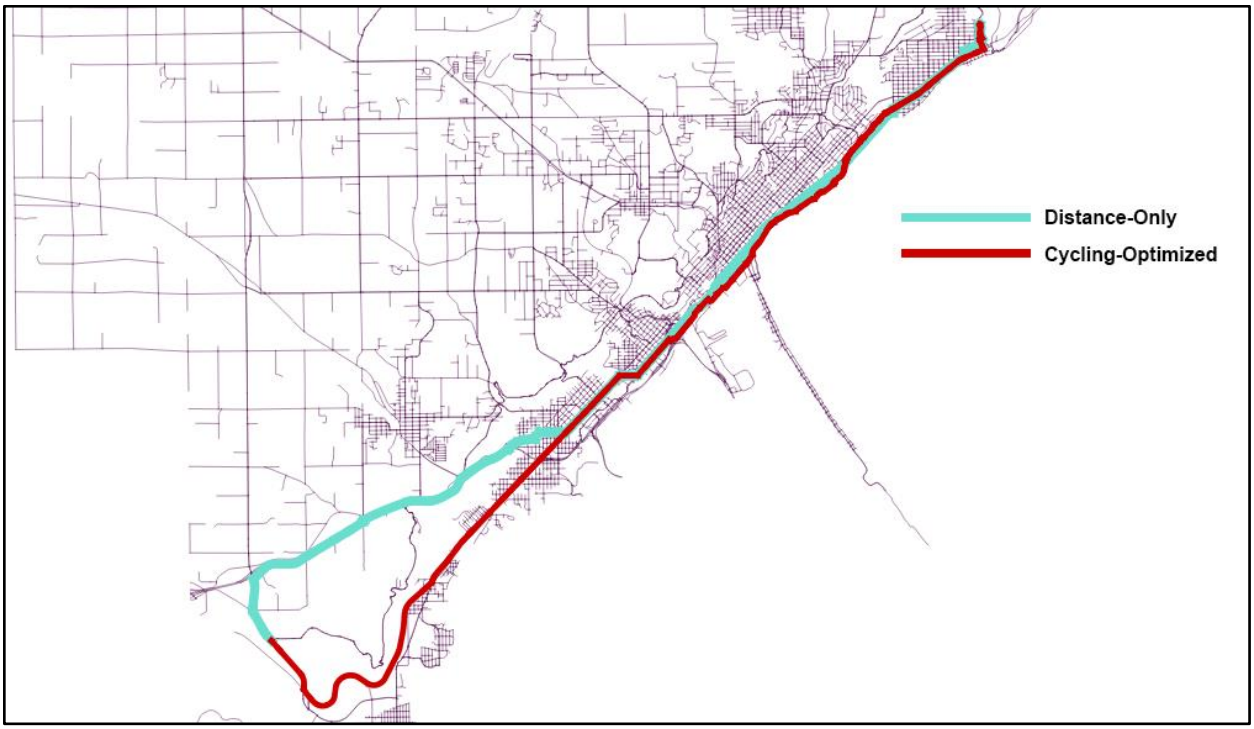

Figure 5: Comparing pgRouting results for a test query optimized by distance and then augmented by our riding density data

Again, slightly altering the very first line of the query allows us to see the actual length of each route. We transform this to SRID 32615 for two reasons:

1. It is the SRID of our original master *county_roads* shapefile (from which *d_roads_4326* was ultimately derived) before it was projected to 4326 so it will be more accurate.
2. The length output is easier to understand in meters than decimal degrees.

```
SELECT SUM(st_length(st_transform(geom,32615))) FROM pgr_dijkstra(...
```

Without our score factored in: 28,761 meters or 17.9 miles

With our score factored in: 31,330 meters or 19.4 miles

Therefore, our calculations would recommend that you travel 8% farther than the most direct route in order to avoid high risk areas and maximize time on roads and paths frequented by other riders.

- Given any lat/lon value in Duluth, find its nearest road that is not a bike path or bike lane. We will use (46.831146 N, 92.058057 W) as our point for this example and `st_transform` where necessary to return the distance in meters.

```
select str_name1,
st_distance((st_transform(geom,32615)),(st_transform((st_setsrid((st_makepoin
t(-92.058057,46.831146)),4326)),32615))) as dist from d_roads_4326
where str_name1 <> 'On Street'
and str_name1 <> 'Bike Route'
order by dist limit 1
```

Valley Dr at a distance of 44.017 meters.

To illustrate how accurate this PostGIS SQL query is, the following Google Maps screenshot shows the distance from our point to the nearby Valley Dr at a distance of 137.51 feet or 41.9 meters.

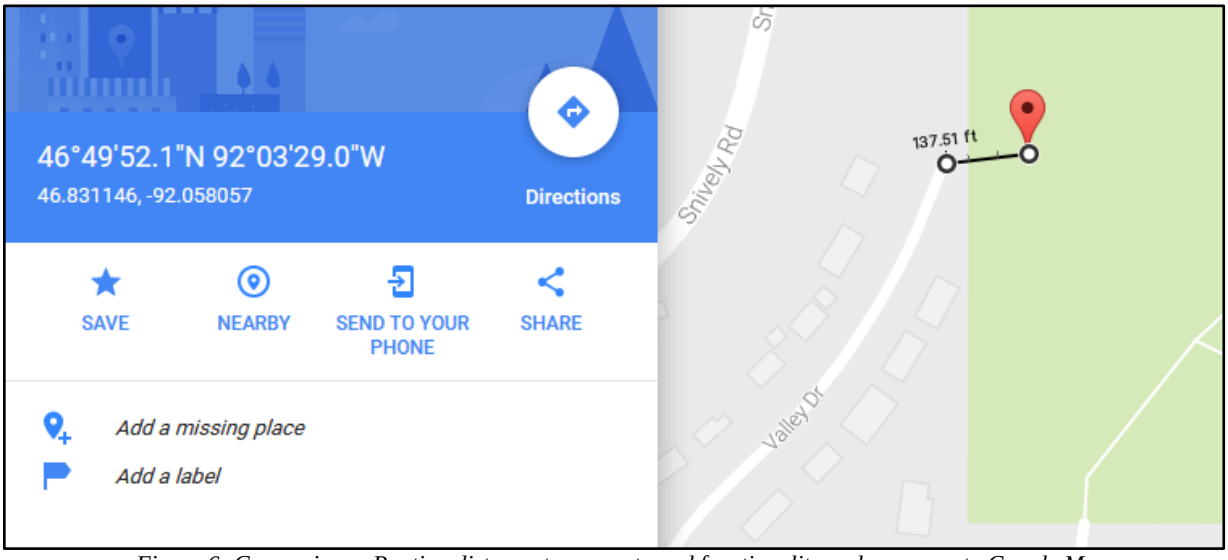

Figure 6: Comparing pgRouting distance-to-nearest-road functionality and accuracy to Google Maps

- Which road is nearest "Canal Park"? (Similar query to the one that was used to populate *attraction_nearest_road_4326*)

```
select str_name1 as street, d.name as attraction
from d_attractions_4326 as d, d_roads_4326 as r
where d.name = 'Canal Park' order by ST_Distance(d.geom,r.geom) limit 1
```

Canal Park Drive (see following image from QGIS)

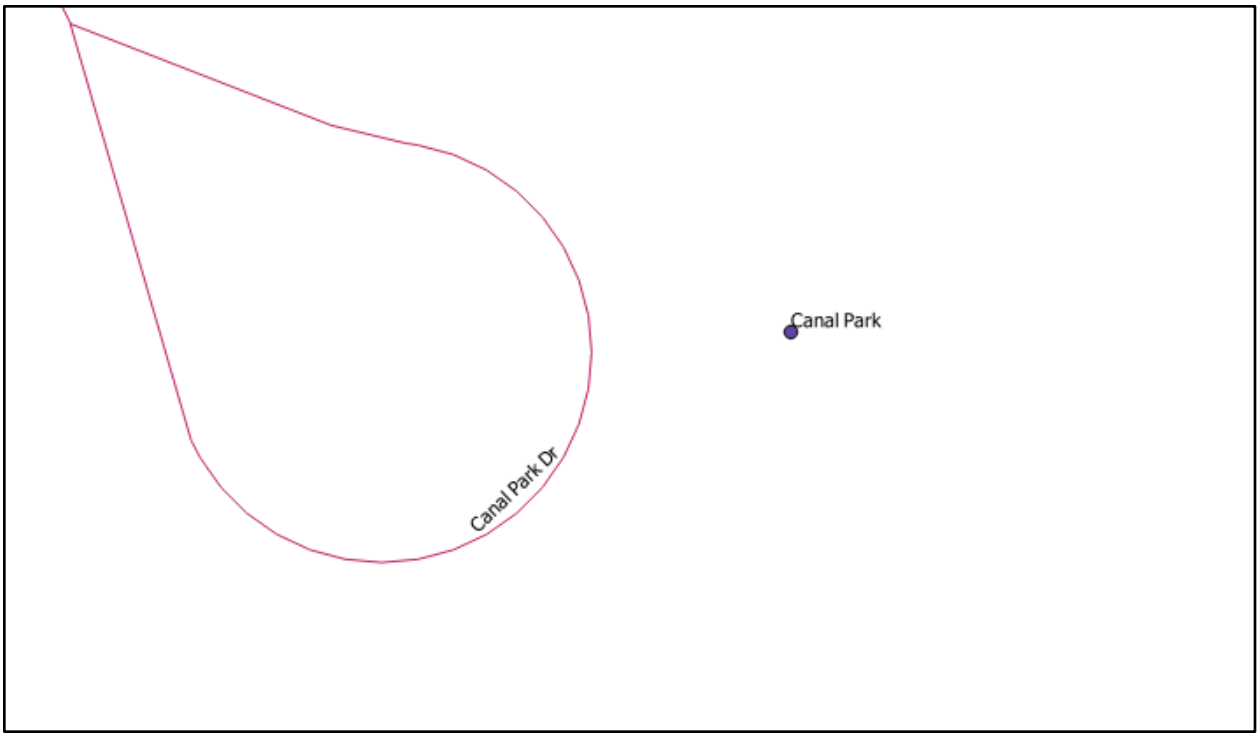

Figure 7: pgRouting find-nearest-road functionality

Appendix

Rider Density Score Creation Process

Analyzing Strava routes- Heatmap average (standardizing based on mean values in buffer):

1. Stitch monochromatic Strava Heatmap together
2. Monochrome to grayscale:
    a. Window-Image Analysis - clip Strava.png to create a copy
    b. Properties > Functions > Grayscale Function

3. Make sure roads layer splits at every intersection- Advanced Editing > Planarize Lines
4. Create a buffer that approximates the width of heatmap (15m each side)
    a. Suggest flat-ended buffers, so that intersections with high-value streets don't throw off values.
5. Zonal Statistics to table, using buffer as zone, pixel value of Strava as raster
6. Add Join > join zonal statistics table to the road file/route network based on FID

Bike Routes Handling

1. Use Tools > Extend Lines to connect any missing connections
2. Make sure roads layer splits at every intersection- Advanced Editing > Planarize Lines
3. Clean data- catch trailing roads (1 meter overlap over intersection) missed connections…
4. Buffer on segments- 15m, flat ends
5. Add Strava.tif
6. Zonal Statistics to table, based on FID
    a. Add Field Join between table and shapefile, based on FID

Bike Routes and Roads Combination

7. Standardize field names between roads and bike paths (type, etc)
8. Combine road/bike paths
9. Split at all intersections- Advanced Editing> Planarize Lines

Beyond Strava- Weight Assignments

- Bike Path Density = Ride_Dens2 (rescaled below- Bike Path Density Specifications)
- Road Dens =Ride_Dens2 * Surface Type Coefficient * Speed Limit Coefficient * Road Type Coefficient * Street Name 1 Coefficient (calculations below- Road Density Specifications)

Bike Path Density Specifications:

- Scall data so that the max bike path value was 100, in order to give bike paths an advantage
- From ESRI support[15]: Rescaled grid = [(grid - Min value from grid) * (Max scale value - Min scale
  value) / (Max value from grid - Min value from grid)] + Min scale value

Road Density Specifications:

- Coefficients for calculating rider density score:
    - Surface Type:

```
      Gravel     0.8
      Else  1
-Speed Limit:
       <=30      1
       35   0.9
       40   0.7
       45   0.6
       50   0.4
       55   0.3
```

[15] http://support.esri.com/technical-article/000008671

```
      >60   0
-Road Type:
      HWY   0.1
      BYP   0.1
      ENT   0.1
      Else  1
```

- Street Name *a stand in for road type, in the case that FNAME is not defined. There is no overlap between type and name coefficients

```
      I-535 0.0001
      I-35  0.0001
      RAMP  0.0001
      Hwy 2 0.1
      Hwy 23     0.1 *includes an on-street bike path
      Hwy 53     0.1
      Hwy 61     0.1
      Hwy 194 0.1
```

- Functions for coefficients above - ArcMap python:
  - Surface Type:

```
def surfCalc(comment):
  if (comment=="SURF_TYPE Gravel"):
    return 0.8
  else:
    return 1
-SPEED_MPG:
def speedCalc(speed):
  if(int(speed)<=30): return 1
  elif(int(speed)==35): return 0.9
  elif(int(speed)==40): return 0.7
  elif(int(speed)==45): return 0.6
  elif(int(speed)==50): return 0.4
  elif(int(speed)==55): return 0.3
  else: return 0.0001
```

  - FTYPE

```
badType = ["Byp","Ent","Hwy"]
def typeCalc(type):
  if type in badType: return 0.1
  else: return 1
```

  - STR_NAME1

```
badName = ["I-535","I-35"]
badishName = ["Hwy 2","Hwy 23","Hwy 53","Hwy 61","Hwy 194"]
def nameCalc(name):
  if name in badName: return 0.0001
  elif "RAMP" in name: return 0.0001
  elif "Ramp" in name: return 0.0001
  elif name in badishName: return 0.1
  else: return 1
```

- Deleted Fields:
  - FDPRE (Road name directional prefix primarily used in emergency dispatch CAD.)
  - FDSUF (Road name directional suffix primarily used in emergency dispatch CAD.)
  - FNAME (Road name used by St Louis county (not complete)

    - STR_Name3 (3rd optional street name. Kept #2 in case requrired for step-by-step directions)
    - STR_Name4
    - LFADD,LTADD,RFADD,RTADD (Address range, from or to, based on geometry. No longer accurate because streets were bisected)
    - CITYLEFT,CITYRIGHT(City left side of street, or right side)
    - LCITY,RCITY,LZIP,RZIP,CNTY_CODE,STATE (Same as above, just city code, zip, county#,state)
    - TYPE (emergency reference)
      DESIGNATIO
    - ROUTE2 (Alternative route number or historical route number)
    - Route_Num (route number, redundant)
    - PWATYPE
    - SEGMENTUUI (unique segment identifier, but doesn't carry over to bike path shapefile)
    - CreatedDt, Creator,EditedDt (has no bearing on our application)
    - P_MOD_L through F_OVRIDE_R (police and fire modifier/override codes, mostly empty and not applicable)
    - ROUTE1 (used for calculations, then deleted)
    - COMMENT (used for surface calculations, then deleted)
    - SPEED_MPH (used for speed calculations, then deleted)
    - SegmentUUI: assumedly Segment UUID, a unique id for each segment in the shapefile.
- Explanation of field definitions: LINK
- St. Louis County, MN: LINK

Twitter Extraction Python Script

```
##Based on code from Tweepy https://github.com/tweepy/tweepy
##Scott Farly https://github.com/scottsfarley93

print "importing extensions..."
import os
import tweepy
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import json # for storing and writing out data
import csv #for writing the data
import time #to track when tweets were posted
print "extensions imported!"

print "setting outfile..."
#TEST outfile
outfileName = os.path.normpath("C:/programming/database/tweets3.csv/")
print "outfile set!"

#Consumer key
consumerKey = "HIDDEN"
consumerSecret = "HIDDEN"
#API keys
accessToken = "HIDDEN"
accessTokenSecret = "HIDDEN"

print "setting up csv and logging session start time"
```

```
#set up the csv for writing (and create it, if need be)
f = open(outfileName, 'a') #for appending, and will create it if it doesn't
exist.
f.write("TERM:
'cycling','bicycling','cyclecommute','bicyclecommute','biking','mtb','bike
trail', 'bike path'\n")
f.write("STARTED: ")
f.write(str(time.time())) #tracks the time the stream started before logging
data
f.write("\n")
fieldnames = ['creation','id','name','screen_name','lang', 'coord', 'text']
csvWriter = csv.writer(f)
csvWriter.writerow(fieldnames)
print "csv formatted!"

#authorize the API access
auth = tweepy.OAuthHandler(consumerKey, consumerSecret)
auth.set_access_token(accessToken, accessTokenSecret)
api = tweepy.API(auth)
print "Authorized."
print "Running..."

print "Establishing methods"
#STEP 1 set up the Tweepy StreamListener to check messages coming from
Twitter
#include methods to capture the information you want
class TwitterNetListener(tweepy.StreamListener):

#method detecting new statuses- very basic
    def on_status(self, status):
       print(status.text)

#search for data- code to parse out geotags by Scott Farly
    def on_data(self, data):
#only write data to csv if coordinates are successfully created (try method)
       try:
             d = json.loads(data)
             creation = d['created_at']
             id = d['id']
             text = d['text'].encode("UTF-8")
             name = d['user']['name'].encode("UTF-8")
             screen_name = d['user']['screen_name'].encode("UTF-8")
             lang = d['user']['lang']
             coord = d['coordinates']
             row = creation, id, name, screen_name, lang, coord, text
#add an if statement to check if coordinates are null before writing
#otherwise you get a tweet without coords
             if coord != "":
                   csvWriter.writerow(row)
       except Exception as e:
             print str(e)
       return True

#in the event of error connecting to API, report
    def on_error(self, status_code):
       print(status_code)
       return True
```

```
#catches timeout errors
    def on_timeout(self):
       print "Timeout..."
       return True

print "methods established!"

print "starting stream..."
#STEP 2 create a stream object
netListener = TwitterNetListener()
twitterNetStream = tweepy.Stream(api.auth,netListener)
twitterNetStream.new_session()
print "stream object created..."
#STEP 3 start the twitter stream!
twitterNetStream.filter(track=['cycling','bicycling','cyclecommute','bicyclec
ommute','biking','mtb','bike trail', 'bike path'], async = True)
print "stream started!"
```