DOCUMENTATION

Python package: spatial_access 1.0.0

COMPUTING TRAVEL TIMES AND SPATIAL ACCESS METRICS AT SCALE

Authors: Irene Farah, Julia Koschinsky, Logan Noel
Code Development: Logan Noel
Contact: Julia Koschinsky (spatial@uchicago.edu)

Center for Spatial Data Science
University of Chicago

July 30, 2019

Overview

This report contains the technical documentation for the Python package *spatial_access 1.0.0.* The package is designed to compute travel times and spatial access metrics at scale. The following chapters are a summary of the Jupyter notebooks that provide an overview of how the travel times and spatial access metrics are computed and that help users navigate the code's main functions and parameters.

Downloads

Spatial-access 1.0.0 PyPi Package: https://pypi.org/project/spatial-access/

Code: https://github.com/GeoDaCenter/spatial_access

REST API: https://github.com/GeoDaCenter/spatial_access_api

Notebooks:
https://github.com/GeoDaCenter/spatial_access/tree/master/docs/notebooks

Input Data and Results: https://github.com/GeoDaCenter/spatial_access/tree/master/data. Note that the capacity field in the destinations file is not real but only for demo purposes.

The data folder contains the input_data needed to estimate the metrics under sources (for origins) and destinations (for destinations). In output_data, the matrices folder will store the estimated symmetric and asymmetric matrices. The models folder will contains the results of the models' analyses. Finally, figures will store the results of maps and plots calculated during the process.

Acknowledgments

Developed by Logan Noel at the University of Chicago's Center for Spatial Data Science (CSDS) with support from the University of Chicago's Center for Spatial Data Science and the Public Health National Center for Innovations (PHNCI). Research assistance of Shiv Agrawal, Caitlyn Tien and Richard Lu is gratefully acknowledged.

TABLE OF CONTENTS

APPENDIX

Chapter 1. Methodology

Travel Time Matrices and Spatial Access Metrics

Purpose and Structure of Notebooks

Across disciplines, spatial accessibility indicators allow you to address many questions, like who does and does not live within reach of specific amenities/services or where they might be spatial mismatches between supply and demand of these services.

The purpose of this notebook is to present the methodology for 1) efficiently and transparently estimating network-based travel times or distances at scale (p2p module), and, based on this, 2) for generating the spatial access and coverage metrics, especially the access score (Model module).

This notebook explains how the Python modules p2p and Models work that are part of the spatial access package (https://pypi.org/project/spatial-access/). The p2p module is part of an open-source backend infrastructure for estimating network-based travel times for three travel modes: walking, driving, and biking. You can also read in a travel time matrix generated in OpenTrip Planner (otp). These **travel time matrices** serve as the input for the **access** and **coverage metrics** in Models.py to identify potential spatial access gaps. Access metrics are attributes of points of **origins** while coverage metrics are attributes of the **destination** points.

The next notebook (3_Travel_Time_Matrix.ipynb (./3_Travel_Time_Matrix.ipynb)) walks you through the computation of the travel time matrix, followed by three notebooks with demos of the metrics on spatial access (4_Access_Metrics.ipynb (./4_Access_Metrics.ipynb)), coverage (5_Coverage_Metrics.ipynb (./5_Coverage_Metrics.ipynb)), and two-stage floating catchment areas (6_TSFCA.ipynb (./6_TSFCA.ipynb)).

As an example, we analyze health facilities in the City of Chicago with public data (http://makosak.github.io/chihealthaccess/index.html). You are also encouraged to use your own data. We highlight the ***parameters*** you can specify as options for your own data.

Motivation

Why did we decide to create a new package for computing travel times at scale?

Compared to alternative state-of-the-art options, this package computes access **more efficiently** in an **open-source** and **scalable** framework that runs **offline** for confidential data.

Generating large shortest path matrices for different travel modes is an important tool for spatial data science, but does not currently have a solution in Python that is open source, highly scalable and efficient. Several tools currently exist for similar purposes as this software package. OSRM, Valhalla, and OpenTripPlanner, among other services, offer matrix APIs to compute the shortest path distance for datasets but the open-source solutions break down when applied to very large datasets without dockerized solutions. On the other hand, both Graphhopper and GoogleMaps charge for the service, which becomes prohibitely expensive at scale.

Each of the above services caps the number of entries in a request at 25-50, meaning that generating a matrix with 500,000 rows requires breaking the original matrix into millions of submatrices and making millions of individual queries. This approach works well for small datasets, but includes substantial overhead which is prohibitive on a large scale. The point-to-point shortest path algorithm presented here (p2p) can generate matrices between a set of origin and destination points (or origins-origins) in 2 lines of code, efficiently and with a low memory footprint.

The example in this notebook generates a driving shortest path matrix for 46,251 blocks in Chicago in ~14 minutes (18 minutes for walking) whereas the same task took > 18 hours using Valhalla. For this particular dataset, the mean difference between time values for the driving shortest path matrix and Google Maps' Matrix API is 2 minutes.

Overview of Travel Time Matrices

Travel time matrices can be computed for walking, biking and driving times between origin and destination points. Instead, you can also choose to compute distances (in meters) between these points. We will refer to travel times by default in these notebooks (since this is the default setting and often of greater interest) but distances are implied and can easily be computed by changing one parameter (use_meters=True, as shown in the [TRAVEL TIME MATRIX DEMO (./3_Travel_Time_Matrix.ipynb)](./3_Travel_Time_Matrix.ipynb)).

There are two routes to compute these matrices: Creating **asymmetric** (nxm) or **symmetric** (nxn) matrices. Symmetric matrices are estimated origin to origin, while asymmetric matrices calculate origin to destination. You can generate a symmetric distance matrix and snap the points of interest to the matrix or create an asymmetric distance matrix that already incorporates origin and destination points. The symmetric approach is more appropriate when you need to calculate several metrics for the same area and different destinations.

Overview of Spatial Access Metrics

After obtaining the travel times from origins to destinations (in this case, from the centroids of tracts to the health facilities), you can then calculate:

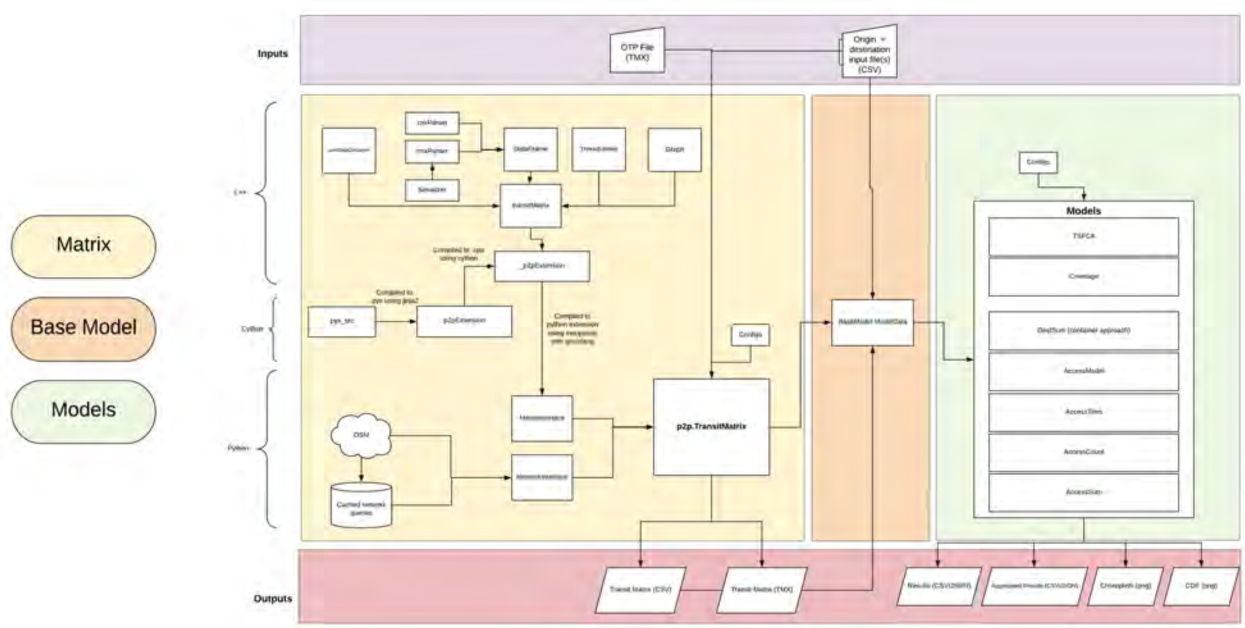
- **Access metrics (origin-based)**:
  - **Access Time**: Time to closest destinations (time to nearest neighbor)
  - **Access Count**: Count of destinations within a catchment area (e.g. how many destinations within a 30-minute walk?)
  - **Access Sum**: Sum of an attribute of destinations within a catchment area
  - **Access Model**: Score from each origin to destination (e.g. tract to health facilities) (gravity-model)
  - **Dest Sum**: Sum of destinations' attributes within areas (container approach)

- **Coverage metrics (destination-based)**:
  - **Coverage**: Supply-demand ratio for the extent of an area that a provider covers (e.g. for each hospital: number of physicians per patient within catchment area of hospital, also called physician-to-patient ratio)

- **Two Stage Floating Catchment Area (origin-based)**: Sum of coverage each point of origin has access to.
  For details on the TSFCA click [here (https://journals.sagepub.com/doi/10.1068/b29120)](https://journals.sagepub.com/doi/10.1068/b29120).

Package Structure and Workflow

This diagram shows how the package is structured:



The workflow first estimates a point-to-point shortest path **(p2p)** [(./spatial_access/p2p.py)](./spatial_access/p2p.py) algorithm for creating the travel-time matrix by travel mode (walking/driving/biking). The code takes the outermost values of the origins and creates a bounding box using their latitude and longitude (destinations need to be constrained to the spatial extent of the origins). Once it generates the bounding box, it queries the network data from OSM, retrieving information on different types of roads and building a graph. Based on this, p2p then creates the travel time matrix.

Then, it creates a base model infrastructure **(BaseModel** (./spatial_access/BaseModel.py) for creating the metrics, using the BaseModel class (parent of Models.py). Specifically, this class allows the user to generate any type of metric, suiting each user's needs. Finally, it creates the models **(Models** (./spatial_access/Models.py) for creating aggregate measures of the Access Model, AccessTime, AccessCount, AccessSum, DestinationSum, Coverage Score, and TSFCA.

This framework provides the user with the flexibility to start at different stages along the process:

1) Start by creating an asymmetric travel time matrix using the p2p algorithm.
2) Start by creating a symmetric travel time matrix using the p2p algorithm and then subsetting it to create an asymmetric travel time matrix.
3) Input an external travel time matrix and run the metrics.

How Travel Time Matrices Work

(Disregard this section if you already have a travel-time matrix.)

Input Requirements

In order to construct the travel time matrices, the csv table should contain **ID, latitude, longitude** variables for the origins and destinations.
Destinations need to be constrained to the spatial extent of the origins.

OpenStreetMap (OSM) structure

To better understand how the algorithm computes travel times, a brief description of OSM's structure follows. OSM's data structure is composed of four elements: nodes, ways, relations, and tags. Nodes are latitude and longitude coordinates (projected in WGS 84) that represent the map's features. Ways are a list of nodes that compose the geometry features (i.e. point, line, polygon) within a map, depicting streets, waterways, parks, etc. Relations express the relationship between nodes and ways. Lastly, tags are attached to nodes, ways or relations, storing metadata about the map objects.

We download the OpenStreetMap network using the area of the previously determined bounding box (i.e. the area of interest defined by the latitude and longitude coordinates). The complexity of the network depends on the number of nodes within this bounding box. In contrast, the number of observations should not affect the efficiency of the running times. In order to get the distances from OSM, OSM-Net calculates the distances of the relations, creating the edges that are queried for the travel time estimation. To estimate these distances, both origin and destination files should be using the same WGS 84 coordinate reference (EPSG:4326).

P2P (point to point) algorithm

In order to calculate the network distance matrix, first, the code extracts the outermost value of latitude/longitude from the origin input table to create a bounding box of the area of interest. The size of the bounding box is buffered, specifically it is increased by 'epsilon', to avoid cutting off the network of datapoints near the boundary of the bounding box. The user can tweak the value of epsilon in **Configs.py**.

P2P uses a k-d tree to match each point in the origin and destination data to its nearest neighbor node in the OSM network, and then finds the Vincenty distance between the two points. Vincenty's formulae estimate the geodesic distance between two points according to an ellipsoidal model of the Earth.

For the travel time computation between origin and destination, the classic Dijkstra's algorithm is then applied to consider every possible route and then select the fastest route. Therefore, P2P also uses an adjacency list representation for Dijkstra's algorithm to find the shortest path for every node to every other node in the underlying OSM network, but it can skip doing any processing for nodes that do not have an attached origin data point. The advantage of this approach is that it scales to very large datasets; as opposed to the adjacency matrix representation (which can easily exceed the memory of many systems for reasonably large datasets). P2P never loads the entire network into memory at one time, meaning the memory footprint is relatively small. This also means the multi-threaded performance of P2P greatly outperforms the single-threaded performance.

For every point in the origin dataset to every point in the destination dataset, the base impedence is the cost found using Dijkstra. To the base value we add the 'last mile' inferred impedence from the origin and destination points to their respective nearest nodes, determined by the Euclidean distance and a constant traversal speed. The 'last mile' is figurative; in the City of Chicago, for instance, 75 percent of block centroids were within 100 meters of the nearest OSM node and 95 percent of block centroids were within 200 meters.

Islands
Some of the units of analysis are classified as islands (disconnected nodes) by OSM. Therefore, Kosaraju's algorithm for directed graph strong connectedness is implemented in p2p (lines 713 - 805 of p2p.py under _request_network2 function). In graph theory, strong connectivity means that a path exists between any pair of nodes. Thus, we implement Kosaraju's algorithm to identify the disconnected nodes and we delete them from the network.

Script
The p2p.py script runs the point to point (p2p) algorithm and creates the class **TransitMatrix**. The output of p2p is the travel time matrix, which is computed in seconds. The **TransitMatrix** unified class run manages all aspects of computing a transit time matrix where matrices can be symmetric or asymmetric (as mentioned above). Therefore, load one input file if you want a symmetric distance graph, or two for an asymmetric matrix. Particularly, this class accounts for all the details that entail specifying the speed limits, creating the bounding box for the area of interest in order to run the OSM query, and calculating the shortest path matrix.

Specifics of P2P parameters
Several parameters should be taken into account when calculating the distance network matrix:

- The **network type** can be determined for walking, biking, or driving.
- Thresholds can be adjusted and are considered in the calculation of the distance matrix: the **average walking speed** is 5 km/h (3 mph) and the default **average driving speed** is 40 km/h (25 mph). You can adjust this parameter for different populations. For example, [Chicago (https://www.cityofchicago.org/dam/city/depts/cdot/StreetandSitePlanDesignStandards407.pdf)](https://www.cityofchicago.org/dam/city/depts/cdot/StreetandSitePlanDesignStandards407.pdf) estimates an average block dimension of 660 feet (200 m) by 330 feet (100 m). These dimensions might change across cities; therefore, the average walking speed of 3 mph

estimates that a person, on average, walks a block in 72 to 144 seconds (1.2 - 2.4 min). The default average speeds and speed limits for different OSM type of roads can be found in **Config.py** and specified when running the matrices.

- Also for walking and driving, you can specify a **node penalty** of X seconds for the number of intersections within the area of analysis. The logic is that having more intersections will increase the travel time due to crossings. However, by doing a time travel calibration between the p2p algorithm and GoogleMaps, there was no need for adding penalities for the city of Chicago for walking and biking, but we added 4 seconds for driving. It can be specified within the **Configs.py** file.
- For driving, the network is **directed**, meaning that one-way streets are respected and A->B and B->A can have different edge traversal speeds.
- **Epsilon**: Controls how large to make the network bounding box beyond your dataset. Larger epsilons result in longer computation times, but smaller epsilons result in slightly reduced accuracy at the very edges of the bounding box, especially for driving networks. The default is currently set at 0.05, which seems to balance the two reasonably well. (+/-) 0.02 will result in a large increase/decrease in computation time and accuracy. If too many values are defined as -1, it means that the epsilon is too small. Refer to the epsilon calibration to assess if this value must change and the matrix contains too many -1. The value of -1 is hardcoded in the tmat.h file and is considered as an NaN value of the origins when estimating the metrics.
- The package allows output of travel time matrices either in **seconds** or in **meters**. The user can specify the output in meters when running the matrix using `use_meters=True`.

GO TO [TRAVEL TIME MATRIX DEMO (./3_Travel_Time_Matrix.ipynb)](./3_Travel_Time_Matrix.ipynb)

Specification of Spatial Access Metrics

(Disregard this section if you only care about the travel-time matrix.)

Origin-based Metrics

The metrics covered in this section are attributes of the origin points, i.e. they considere spatial access from the perspective of someone accessing amenities. In contrast, the metrics in the next section are attributes of the destination, i.e. they consider spatial access from the perspective of the service provider. In addition, the 2-stage floating catchment area model is an origin-based metric that combined spatial access and coverage elements.

This spatial access package allows you to compute the following metrics that are attributes of the point of origin:

1. Access Time

Shortest time to the nearest facility/amenity.

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **category** for sub-setting
+ **larger areal ID** for aggregating.

2. Access Count

Total number of amenities/facilities within the catchment area

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **category** for sub-setting
+ **larger area ID** for aggregating.

3. Access Sum

Captures the sum of an attribute within a catchment area.
(e.g. number of doctors within a 30-minute walk from the origins)

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **capacity** for destinations
+ **category** for sub-setting
+ **larger area ID** for aggregating.

4. Dest Sum (container approach)

Captures the sum of the attributes of a destination, within an area.
(e.g. number of doctors within a community area - does not require travel time matrix)

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **capacity** for destinations
+ **category** for sub-setting
+ **larger area ID** for aggregating.

5. Access Model

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **category** for sub-setting
+ **larger areal ID** for aggregating.

How the Access Score Works

The Access Model generates an access score to measure how accessible a location is to multiple amenities within a given travel time (e.g. 20 minutes walking). In our example, tract centroids are points of origin and health facilities are destination points.

The score is a weighted sum. Every destination point receives a value that represents the product of the following weights, which are then summed across destinations within a travel time of the point of origin to obtain the final score:

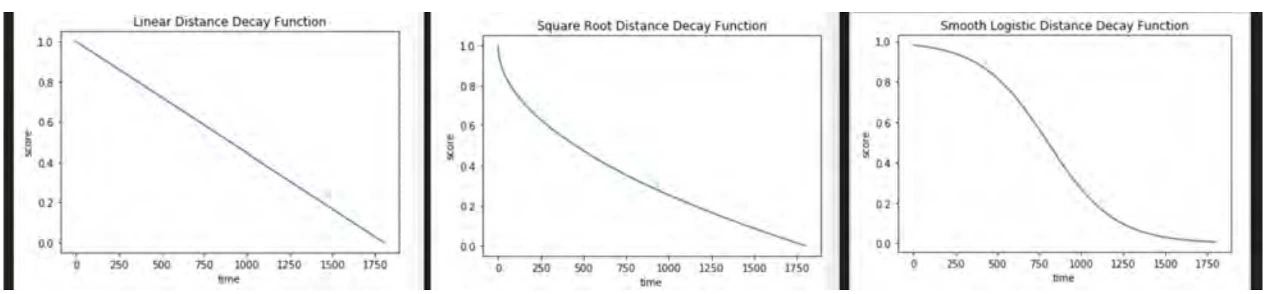
1) **distance decay** where closer amenities have more weight (default = linear)
2) **relative importance of an amenity type** (e.g. with a greater weight for supermarkets than museums)
3) **variety / penalty for same types** (where more of the same type of amenity has less weight).

This section explains how the three weights work and how the score is then constructed.

Distance Decay

Distance decay weights are applied to give closer amenities more weight and reduce the weight of more distant ones. Amenities beyond the specified travel time threshold (e.g. 30 min walk) are not considered in the score.

In more technical terms, the distance decay function describes the decreasing intensity of a value as the distance increases. You can add any function in the code, depending on your amenities' intensity behavior. Out of the box, this package provides the three functions shown below: linear, square root, and logit (default = linear):



Relative Importance and Variety

You can create the access score for one type of amenity (e.g. supermarkets) or a variety of types (e.g. supermarkets, museums and restaurants). In both cases, you have the option to manually assign the relative importance of amenities and give less weight to the same type of amenities. For instance, you can up-weight larger supermarkets or supermarkets vs restaurants vs. convenience stores and downweight any additional restaurant beyond the first few within a travel time. If you have a variety of types, you can compute the score for the pooled categories (supermarkets, museums and restaurants together) or for each category separately.

You can estimate the score with normalization (0-100) or without (and then compare intervals like quintiles across place or time).

The dictionary below shows an example of the weights assigned to each amenity:

```
In [ ]: #Example of importance and variety weights:

dict = {
"Hospitals": [10,10,10,10,10],
"Federally Qualified Health Centers": [8, 7, 6, 5, 4],
"School-Based Health Centers": [7, 7, 6, 6, 5],
"All Free Health Clinics": [5, 5, 5, 4, 4],
"Other Health Providers": [4,3,2,1,1]
}

#Make sure your categories in the dictionary match the spelling in the csv i
```

You can specify the weights based on your research needs. In this case, a hospital will be categorized as more important than a smaller free health clinic (10 vs 5 for the first of each facility). Moreover, the dictionary categorizes the second nearest FQHC as having less weight than the first one (8 vs 7). However, additional hospitals are not down-weighted since the demand for hospitals usually exceeds supply. In other words, the 5th hospital has the same weight as the 1st. If there is a sixth hospital within 30 minutes of a tract center, the score will neglect it since there are only 5 weights specified under 'Hospitals', so you want to make sure that your weight count equals or exceeds your destination count within the travel time of your point of origin. As mentioned before, destinations beyond the travel time threshold are ignored.

How the Access Score is Calculated

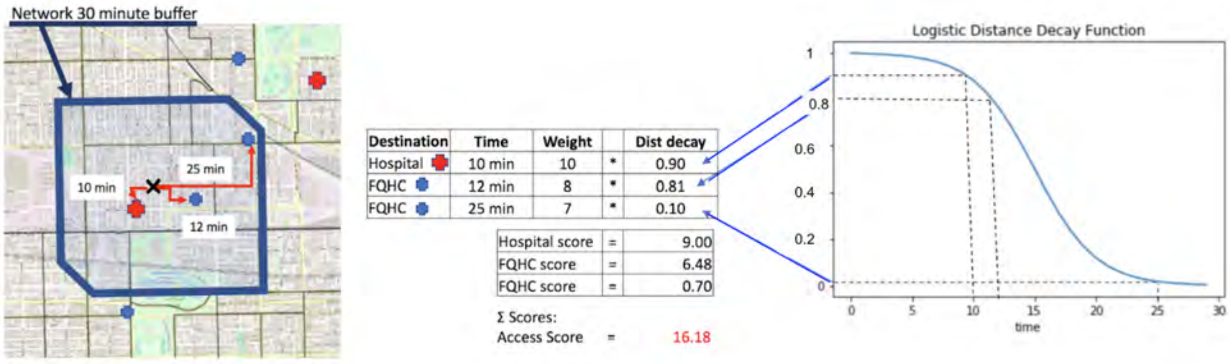
The figure below shows a point of origin (black x) and the three health facilities that can be reached within a 30 minute walk from there: two FQHCs (blue dots, 12 and 25 minutes away) and one hospital (red dot, 10 minutes away). The table next to the map lists these three facilities, the travel times to them, and their respective importance and variety weights, specified in the weights dictionary above. The closest facility, the hospital, is weighted as 10, followed by the next closest Federally Qualified Health Center, weighted as 8 and the third closest health center, weighted as 7,

The last column contains the weights from the distance decay function shown in the image to the right of the table. The distance decay function weighs each destination depending on its relative distance to the point of origin: closer destinations are weighed higher that more distant destinations.

This image shows how the distance decay function maps a given travel time (x-axis) to a score from 0-1 (y-axis). The 10-minute travel time to the hospital is weighted by the distance decay function with a score of 0.9 (and smaller weights of 0.81 and 0.1 for the other two facilities at the larger 12 and 25 min distances).

The score for each facility Is the product of the importance/variety weights and these distance decay weights: As shown in the table below, these scores are 9, 6.48 and 0.70 for the hospital and two FQHCs, respectively. The final score is the sum of these facility scores: In this case, it is 16.18.

Note that the more categories you have, the larger your score will be. By default, the score is not normalized to observe the overall distribution across places and time, but the results can also be standardized.



Specifications

In the **demo prompt**, you can specify parameters with two different commands:

name.AccessModel( ):

- network_type ('walk', 'bike', 'drive', 'otp')
- sources_filename (primary input data)
- destinations_filename (secondary input data)
- source_column_names (dictionary that maps column names to expected values)
- dest_column_names (dictionary that maps column names to expected values)
- transit_matrix_filename (sources-destination travel time matrix)
- decay_function ('linear', 'root', 'logit', default is 'linear')

name.calculate():

- upper_threshold (travel time threshold in seconds, default is 30 minutes; beyond the threshold, score will be zero)
- category_weight_dict (specifies the weights of each destination defined as dictionary, default dictionary will contain [1,1,1,1,1,1,1,1,1,1,1] weights.)
- normalize (accepts boolean, default is False and shows only non-normalized results, true shows normalized values.)
- normalize_type ('z_score' or 'minmax', default is 'minmax')

GO TO ACCESS SCORE DEMO (./4_Access_Metrics.ipynb)

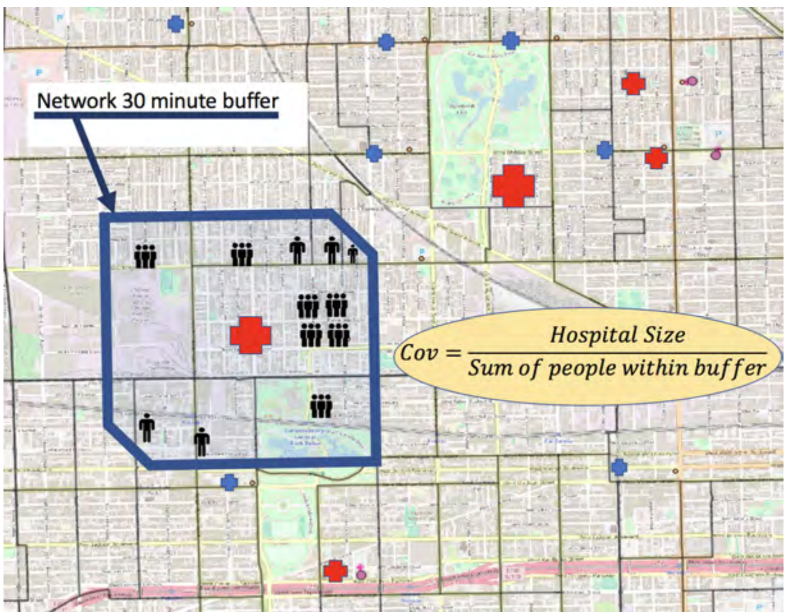
Destination-based Metrics

1. Coverage

The **Coverage** model generates a coverage access which shows the per capita spending available to a specific targeted population. The model focuses on the coverage of the destination, scrutinizing how many people are within a catchment area. Specifically, it takes the total spending of the facility/establishment and divides it by the total population it serves within a buffer (in this case, 30 minutes). In the specifications, the magnitude of the destination is denominated as target.

Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **population** (origins)
+ **capacity** (destinations)
+ **category** for sub-setting (destinations)
+ **larger area ID** for aggregating.



Specifications

name.Coverage( ):

- network_type ('walk', 'bike', 'drive', or 'otp')
- sources_filename (primary input data)
- destinations_filename (secondary input data)
- transit_matrix_filename (origin-destination transit matrix)

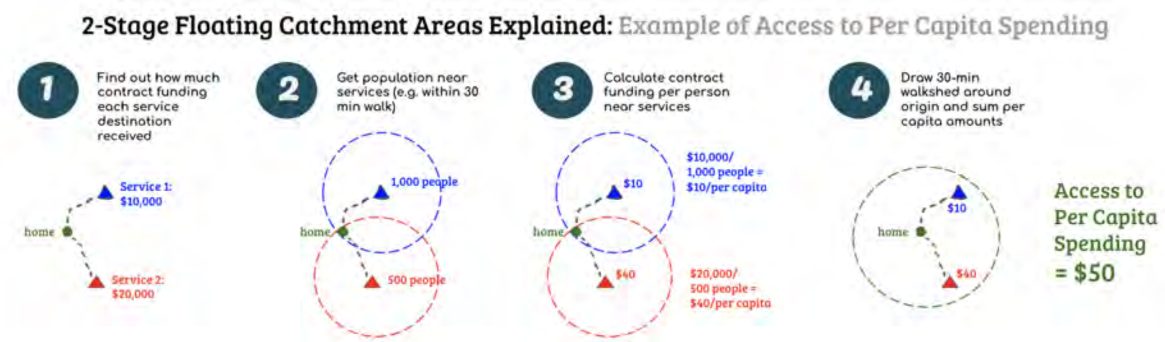
name.calculate():

- upper_threshold (travel time threshold in seconds; beyond the threshold, score will be zero)

GO TO [COVERAGE SCORE DEMO (./5_Coverage_Metrics.ipynb)](./5_Coverage_Metrics.ipynb)

Two-Stage Floating Catchment Area Model

The **TSFCA** model generates a coverage access which shows the per capita spending available to a specific targeted population. The model focuses on the coverage of the destination, scrutinizing how many people are within a catchment area. Specifically, it takes the total spending of the facility/establishment and divides it by the total population it serves within a buffer (in this case, 30 minutes). In the specifications, the magnitude of the destination is denominated as target.



Input Requirements:
csv file: **ID, latitude, longitude** for origins and destinations

+ **population** (origins)
+ **capacity** (destinations)
+ **category** for sub-setting (destinations)
+ **larger area ID** for aggregating.

GO TO [TSFCA SCORE DEMO (./6_TSFCA.ipynb)](./6_TSFCA.ipynb)

Subsetting, Aggregation and Plotting

The metrics can also be **subset** by categories. If you have many types of amenities, you can choose 1 to n categories to calculate the metrics for each category (e.g., health clinics and hospitals as opposed to all health facilities).

The scores can also be **aggregated** at a larger areal unit to show overall access patterns, as is shown in the demos.

The scripts also contain hard-coded empirical cumulative distribution function and choropleth **plots** to preview data patterns (see the notebook demos). However, these plots are not designed for presentation purposes. To create professional graphs and maps, the results saved in the csv files can be merged to the origin's or destination's shapefile for mapping and plotting in other software.

Chapter 2: Travel Time Matrix DEMO

Input Requirements

In order to construct a travel time matrix, the csv table should contain **ID, latitude, longitude** for the origins and destinations. Note that destinations need to be constrained to the spatial extent of the origins.

In [1]:
```
from spatial_access.p2p import *
```

In [ ]:
```
cd ../..
```

In [ ]:
```
%matplotlib inline
```

View structure of data example: Health Facilities in Chicago.
Health Facilities Data: [http://makosak.github.io/chihealthaccess/index.html (http://makosak.github.io/chihealthaccess/index.html)](http://makosak.github.io/chihealthaccess/index.html)

5 sources (tract centroids):

In [4]:
```
df_sources = pd.read_csv('./data/input_data/sources/tracts2010.csv')
df_sources.head()
```

Out[5]:

	geoid10	lon	lat	Pop2014	Pov14	community
0	17031842400	-87.630040	41.742475	5157	769	44
1	17031840300	-87.681882	41.832094	5881	1021	59
2	17031841100	-87.635098	41.851006	3363	2742	34
3	17031841200	-87.683342	41.855562	3710	1819	31
4	17031838200	-87.675079	41.870416	3296	361	28

5 destinations (health facilities):

In [6]:
```
df_dests = pd.read_csv('./data/input_data/destinations/health_chicago.csv')
df_dests.head()
```

Out[6]:

	ID	Facility	lat	lon	Type	target	category	community
0	1	American Indian Health Service of Chicago, Inc.	41.956676	-87.651879	5	127000	Other Health Providers	3
1	2	Hamdard Center for Health and Human Services	41.997852	-87.669535	5	190000	Other Health Providers	77
2	3	Infant Welfare Society of Chicago	41.924904	-87.717270	5	137000	Other Health Providers	22
3	4	Mercy Family - Henry Booth House Family Health...	41.841694	-87.624790	5	159000	Other Health Providers	35
4	6	Cook County - Dr. Jorge Prieto Health Center	41.847143	-87.724975	5	166000	Other Health Providers	30

Travel Time Matrices

Specifications for the asymmetric and symmetric distance matrices:

- **network_type**: can be walk, drive, bike, or otp (otp allows you to read in an external file from OpenTripPlanner)
- **primary_input**: sources file
- **secondary_input**: destinations file (omit to calculate an NxN matrix on the primary_input)
- **read_from_file**: tmx or csv filename (read in external matrix files)
- **primary_hints**: dictionary that contains column names (lat/lon/ID)
- **secondary_hints**: dictionary that contains column names (lat/lon/ID)
- **debug**: if set to `True` enables to see more detailed logging output
- **configs**: defaults to None, else pass in an instance of **Configs.py** to override default values. The following arguments in **configs** can be changed:
    - walk_speed: numeric (km/hr). Default is set to 5 km/hr.
    - bike_speed: numeric (km/hr). Default is set to 15.5 km/hr.
    - default_drive_speed: numeric (km/hr). Default is set to 40 km/hr.
    - walk_node_penalty: numeric (seconds). Default is set to 0.
    - bike_node_penalty: numeric (seconds). Default is set to 0.
    - drive_node_penalty: numeric (seconds). Default is set to 4.
    - speed_limit_dict: dictionary {edge type (string) : speed in km/hr}
    - use_meters: if `True` output will be in meters. If `False` , output will be in seconds.
    - disable_area_threshold: enables computation for areas exceeding the bounding box area constraint (set to 5,000 squared km in NetworkInterface.py).
    - require_extended_range: If true, use unsigned integers instead of unsigned shorts for value type to increase max range.
    - epsilon: factor by which to increase the requested bounding box. Increasing epsilon may result in increased accuracy for points at the edge of the bounding box, but will increase computation times. Default is set to 0.05.

Asymmetric Travel Time Matrix

You can create an asymmetric matrix from source to destination points (takes ~ 20 min for this example). This is useful when you only need to generate results once (as opposed to repeatedly for the same origins but different destinations).

Please map your latitude and longitude before reading them in to make sure they are correct. E.g. if incorrect lat-long values are far outside of your actual spatial extent, the results will take an excessively long time to compute or stall.

WALKING

In [19]:
```
# Calculate asymmetric distance matrix for walking (takes ~6 minutes to run,
w_asym_mat = TransitMatrix('walk',
                           primary_input='./data/input_data/sources/tracts2(
                           secondary_input='./data/input_data/destinations/h
                           primary_hints={'idx' : 'geoid10', 'population': '
                           secondary_hints={'idx': 'ID', 'capacity': 'skip',

w_asym_mat.process()
```

...

In [ ]:
```
#Saved as walk_asym_health_tracts.csv
w_asym_mat.write_csv(outfile = "./data/output_data/matrices/walk_asym_health
```

In [ ]:
```
# Saved as walk_asym_health_tracts.tmx
w_asym_mat.write_tmx(outfile = "./data/output_data/matrices/walk_asym_health
```

**Example of overriding default Configs

Here we are disabling the large bounding box constraint and lowering the drive speed. We are keeping the default output of the matrix set to travel times as opposed to distances (by setting meters to false). If you want to work with distances instead of travel times, set this parameter to True.

```
In [ ]: from spatial_access.Configs import Configs
        custom_config = Configs()
        custom_config.disable_area_threshold=True
        custom_config.default_drive_speed=35
        custom_config.use_meters=False

        # then run:
        w_asym_mat = TransitMatrix('walk',primary_input='./data/input_data/sources/t
                                   secondary_input='./data/input_data/destinations/h
                                   primary_hints={'idx' : 'geoid10', 'population': '
                                   secondary_hints={'idx': 'ID', 'capacity': 'skip',
                                   configs=custom_config)
        w_asym_mat.process()

        #make sure you add configs=custom_config in the last line or this won't run
```

DRIVING

```
In [ ]: # Calculate asymmetric distance matrix for driving (takes ~1.5 minutes to ru
        d_asym_mat = TransitMatrix('drive',
                                   primary_input='./data/input_data/sources/tracts2(
                                   secondary_input='./data/input_data/destinations/h
                                   primary_hints={'idx' : 'geoid10', 'population': '
                                   secondary_hints={'idx': 'ID', 'capacity': 'skip',

        d_asym_mat.process()
```

```
In [ ]: #Saved as drive_asym_health_tracts.csv
        d_asym_mat.write_csv(outfile = "./data/output_data/matrices/drive_asym_healt
```

```
In [ ]: # Saved as drive_asym_health_tracts.tmx
        d_asym_mat.write_tmx(outfile = "./data/output_data/matrices/drive_asym_healt
```

Symmetric Matrix

You can also create a symmetric travel time matrix, e.g. from each tract to all the other tracts (in this case, a 801 x 801 matrix). Then, you can merge destinations to this matrix using shared IDs or spatial joins in a GIS, GeoDa or R to create an asymmetric matrix as above. If you have several different destinations for the same spatial extent (or want to run simulations), the advantage of merging them with a symmetric matrix is that you only have to compute the travel times once.

WALKING

```
In [ ]: # Specify walking distance matrix (takes ~3 min to run)
        w_sym_mat = TransitMatrix('walk',
                                  primary_input='./data/output_data/matrices/walk_a
                                  primary_hints={'idx' : 'geoid10', 'population': '
        # Run process
        w_sym_mat.process()
```

```
In [ ]: # Saved as walk_sym_health_tracts.csv
        w_sym_mat.write_csv(outfile = "./data/output_data/matrices/walk_sym_health_t
```

```
In [ ]: # Saved as walk_sym_health_tracts.tmx
        w_sym_mat.write_tmx(outfile = "./data/output_data/matrices/walk_sym_health_t
```

DRIVING

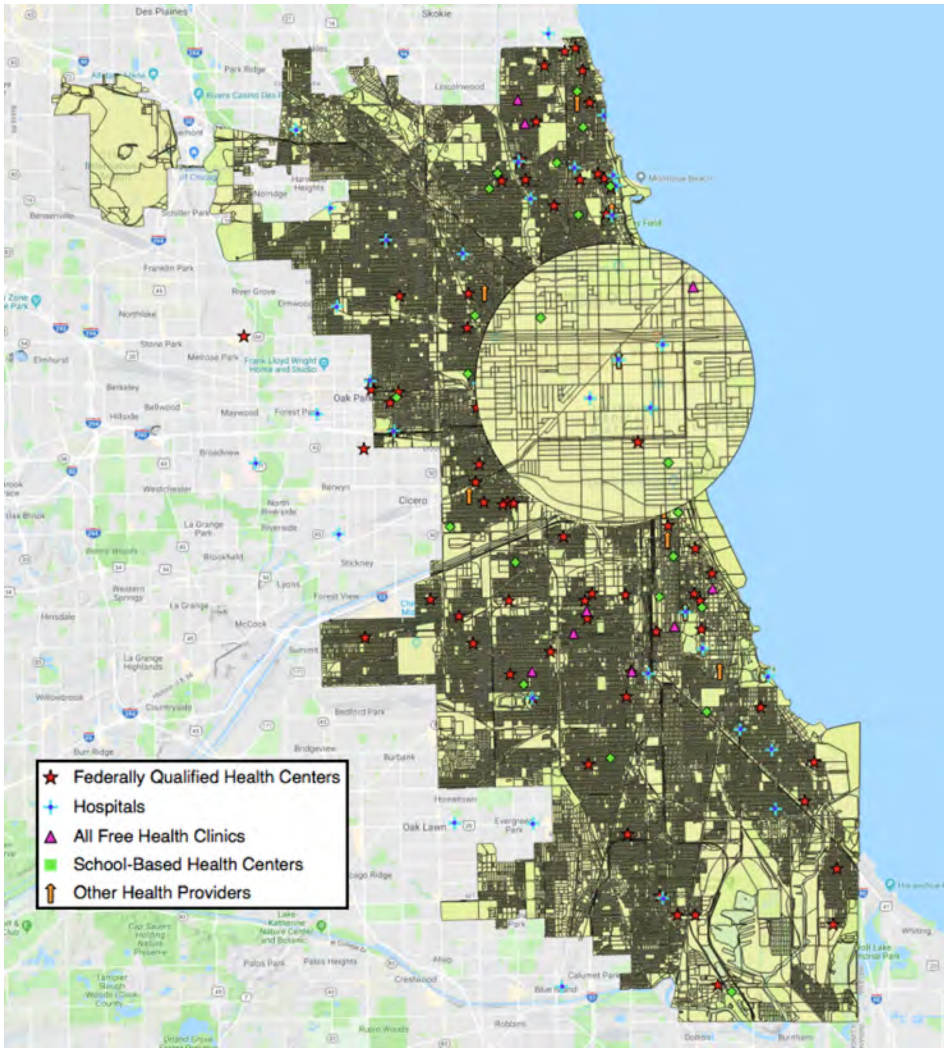
```
In [ ]: # Specify driving distance matrix (takes ~1.5 minute to run)
        d_sym_mat = TransitMatrix('drive',

        primary_input='/Users/whlu/spatial_access/data/in
        primary_hints={'idx' : 'geoid10', 'population': '

        # Run process. For driving, p2p queries OSM to fetch the street network and
        d_sym_mat.process()
```

```
In [ ]: # Saved as drive_sym_health_tracts.csv
        d_sym_mat.write_csv(outfile = "./data/output_data/matrices/
        drive_sym_health_
```

```
In [ ]: # Saved as drive_sym_health_tracts.tmx
        d_sym_mat.write_tmx(outfile = "./data/output_data/matrices/
        drive_sym_health_
```

Spatial Join (snap destinations to origins)

Now you can snap the destination points to the areas of origin. Before you do this, map origins and destinations to understand how the two layers are related: e.g., when points fall on the boundary of an area, which area they are assigned to can be arbitrary. If destinations fall within areas, you can use a within function that joins the destinations to area it falls into. If origins and destinations share a geoID, you can also merge the data that way. The following image shows that, in this case, we can safely run a function that assigns each destination point to the area that surrounds it.



Spatial join of health facilities and travel time matrix

We need to join the health facilities with the travel time matrix generated before. This will generate an asymmetric matrix with the travel times from all tracts in Chicago to the health facility destinations.

In [47]:
```
# Read destination files to join with boundaries
health_gdf = gpd.read_file('./data/input_data/destinations/health_chicago.sh
health_gdf.head()
#Use symmetric matrix calculated above or read your previously saved results
sym_walk=pd.read_csv('./data/matrices/walk_sym_health_tracts.csv')

# Read boundaries files
boundaries_gdf = gpd.read_file('./data/input_data/sources/tracts2010.shp')

# Rename the ID name in order to match both data frames.
sym_walk= sym_walk.rename(index=str, columns={"Unnamed: 0": "geoid10"})


# Spatial join of amenities within each area of analysis
#It drops values outside of the tracts shapefile. From 199 to 182 datapoints
s_join = gpd.sjoin(health_gdf, boundaries_gdf, how='inner', op='within')

# Convert geopanda dataframe to non-spatial dataframe to join
jb_df = pd.DataFrame(s_join)


# Make sure the id is of the same data type in both data frames.
# sym_walk.dtypes
# jb_df.dtypes
jb_df.geoid10=jb_df.geoid10.astype(int)
jb_df=pd.DataFrame(jb_df['geoid10'])

# Join the symmetric matrix with the spatially joined data (with geoid10 id,
j_asym=pd.merge(sym_walk, jb_df, left_on='geoid10', right_on='geoid10', how=

j_asym.to_csv('./data/output_data/matrices/walk_asym_health_tracts_join.csv'
```

In [48]:
```
#Check the output is correct
j_asym.head()
```

Out[48]:

	geoid10	1	2	3	4	5	6	7	8	9	...	793	794	795
0	1	0	9881	9106	11593	12167	8364	7089	27241	7104	...	9824	15701	16077
1	2	9881	0	3326	2115	3592	6092	14890	18531	16327	...	4472	9291	8947
2	3	9106	3326	0	3297	3777	9084	15494	18504	15926	...	7464	6881	7245
3	4	11593	2115	3297	0	1670	7905	16709	16992	18146	...	6285	7568	7205
4	5	12167	3592	3777	1670	0	9382	17433	15746	18870	...	7762	6141	5778

5 rows × 803 columns

In [49]:
```
j_asym.shape
```

Out[49]:

Now that you have a origin destination matrix, we can proceed to estimate spatial access metrics based on these matrices. For this demo's purpose, we will use drive_asym_health_tracts.csv and walk_asym_health_tracts.csv to run the metrics.

Chapter 3: Spatial Access Metrics DEMO

This notebook shows you how to calculate spatial access metrics that indicate how accessible points of origin are to destinations -- in this case, how spatially accessible home locations (centroids of Census tracts) are to health facilities like hospitals or health clinics. Using the travel time matrix from the p2p module, you can calculate the following spatial access metrics:

AccessModel: an access score
AccessTime: time to the closest destination
AccessCount: count of nearby destinations within a travel time threshold
AccessSum: sum of an attribute of destinations within a travel time threshold
DestSum: sum of destinations within an area (also called container approach).

Each model follows a similar procedure:

1. Define the model by providing the appropriate arguments
2. Calculate the model
3. Subset, aggregate, plot the results (optional)
4. Save the result as a csv or tmx file

Each of these steps are demonstrated below.

Standard Data Requirements

- Each model requires two csv files as inputs: sources and destinations.
- Destinations need to be constrained to the spatial extent of the origins.
- Field names with symbols will be replaced by underscores in the csv file.

The standard variables required for all models are listed below:

- Source File
  - Unique index identifier (**ID**) (integer or real)
  - **Latitude** and **longitude** coordinates (real)
  - To aggregate: **larger areal ID**
- Destination File
  - Unique index identifier (**ID**) (integer or real)
  - **Latitude** and **longitude** coordinates (real)
  - **Category** for each type of facility
  - To aggregate: **larger areal ID**

Additional variables are required for some models (specified above each model).

In [2]:
```
cd /Users/juliakoschinsky/spatial_access
```

```
/Users/juliakoschinsky/spatial_access
```

In [ ]:
```
# Import modules
from spatial_access.p2p import *
from spatial_access.Models import *
```

In [4]:
```
# View sources and destinations for Chicago health facilities
import pandas as pd
sources_df = pd.read_csv('./data/input_data/sources/tracts2010.csv')
dests_df = pd.read_csv('./data/input_data/destinations/health_chicago.csv')
```

Read in travel time matrix generated in [1_matrix.ipynb (./1_matrix.ipynb)](./1_matrix.ipynb):

In [ ]:
```
matrix_df = pd.read_csv('./data/output_data/matrices/walk_asym_health_tracts
```

View the first 5 sources (tract centroids):

In [3]:
```
sources_df.head()
```

Out[3]:

	geoid10	lon	lat	Pop2014	Pov14	community
0	17031842400	-87.630040	41.742475	5157	769	44
1	17031840300	-87.681882	41.832094	5881	1021	59
2	17031841100	-87.635098	41.851006	3363	2742	34
3	17031841200	-87.683342	41.855562	3710	1819	31
4	17031838200	-87.675079	41.870416	3296	361	28

View the first 5 destinations (health facilities):

In [4]:
```
dests_df.head()
```

Out[4]:

	ID	Facility	lat	lon	Type	capacity	category	community
0	1	American Indian Health Service of Chicago, Inc.	41.956676	-87.651879	5	127000	Other Health Providers	3
1	2	Hamdard Center for Health and Human Services	41.997852	-87.669535	5	190000	Other Health Providers	77
2	3	Infant Welfare Society of Chicago	41.924904	-87.717270	5	137000	Other Health Providers	22
3	4	Mercy Family - Henry Booth House Family Health...	41.841694	-87.624790	5	159000	Other Health Providers	35
4	6	Cook County - Dr. Jorge Prieto Health Center	41.847143	-87.724975	5	166000	Other Health Providers	30

View the first 5 records of the travel time matrix:

In [5]:
```
matrix_df.head()
```

Out[5]:

	Unnamed: 0	1	2	3	4	6	8	9	10	11	...	198	199
0	17031842400	17870	21397	17892	8483	13529	12425	5704	16935	7565	...	13236	16767
1	17031840300	11391	13937	8845	4120	3713	3939	6358	8448	3721	...	4671	7050
2	17031841100	9050	12577	9155	1374	5748	4035	5015	8198	2295	...	4799	8430
3	17031841200	9649	12195	7306	4588	3049	2017	8015	6846	5313	...	2958	5512
4	17031838200	8222	10768	6219	5068	3794	590	8589	5440	5887	...	1552	4789

5 rows × 201 columns

Access Model: Access Score for Multiple Destinations

The Access Model generates an access score to measure how accessible a location is to multiple amenities within a given travel time (e.g. 20 minutes walking). You can specify three types of weights for this score:

1) **distance decay** where closer amenities have more weight (default = linear)
2) **relative importance of an amenity type** (e.g. with a greater weight for supermarkets than museums)
3) **penalty for same types** (where more of the same type of amenity gets less weight).

You can estimate the score with or without normalization.
The AccessModel does not require population or target variables.

Specifications for the Access Model:

name = AccessModel( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (sources file)
- **destinations_filename** (destinations file)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.
- **decay_function** ('linear', 'root', 'logit', default is 'linear')

Note: Some access metrics do not need `population` or `capacity` columns but the prompt might still ask you for that column. If the population or capacity column are not needed, write as 'skip' in source_column_names/dest_column_names.

Column Inputs

- Standard data requirements (see above)

name.calculate()

- **upper_threshold** (maximum number of seconds between origins and destinations)
- **category_weight_dict** (specifies the weight (importance) of each destination as a dictionary; default weights = [1,1,1,1,1,1,1,1,1,1])
- **normalize** (Boolean: default is False and shows non-normalized results; True shows normalized values.)
- **normalize_type** ('z_score' or 'minmax', default = 'minmax')

Functions within the Access Model class (use as name.function())

- calculate ()
- set.focus.categories()
- aggregate()
- plot_cdf()
- plot_chloropleth ()

Each function is demonstrated below.

When specifying the Access Model, use the previously generated travel time matrix. Also specify the desired distance decay function. Here, source_column_names and dest_column_names are not specified so the model will ask you to map column names to expected values.

Specify travel mode, file names, variable names and the distance decay function:

```
In [ ]: accessM = AccessModel(network_type='walk',
                              sources_filename='./data/input_data/sources/tracts201(
                              destinations_filename='./data/input_data/destinations/
                              transit_matrix_filename='./data/matrices/walk_asym_hea
                              decay_function='linear')
```

Specify the weights for relative importance and same types:

```
In [68]: dict = {
         "Hospitals": [10,10,10,10,10],
         "Federally Qualified Health Centers": [8, 7, 6, 5, 4],
         "School-Based Health Centers": [7, 7, 6, 6, 5],
         "All Free Health Clinics": [5, 5, 5, 4, 4],
         "Other Health Providers": [4, 3, 2, 1, 1]
         }
```

Specify the travel time threshold in seconds (e.g. 1,800 seconds = 30 minutes), whether or not to normalize the score, and the importance/variety weights:

```
In [ ]: accessM.calculate(upper_threshold=1800,
                          category_weight_dict=dict,
                          normalize=False)
```

```
In [70]: #Preview the results
         accessM.model_results.head()
```

Out[70]:

	all_categories_score	All Free Health Clinics_score	School-Based Health Centers_score	Federally Qualified Health Centers_score	Other Health Providers_score	Ho
17031842400	0.318889	0.000000	0.318889	0.000000	0.000000	
17031840300	6.784444	0.000000	0.089444	6.695000	0.000000	
17031841100	14.783889	0.072222	6.148333	1.884444	4.323333	
17031841200	29.545556	0.000000	7.127778	14.178889	0.000000	
17031838200	47.820000	1.008333	6.667222	8.044444	2.688889	

After constructing the Access Model

Once the access model is built, we can do several things:

- Write out the data frame to a csv file
- Aggregate the results to a higher geographic level (in this example, from tracts to community areas)
- Subset data for specific categories of destinations (in this example, Federally Qualified Health Centers)
- Plot choropleth maps and cumulative distributive functions (CDF)

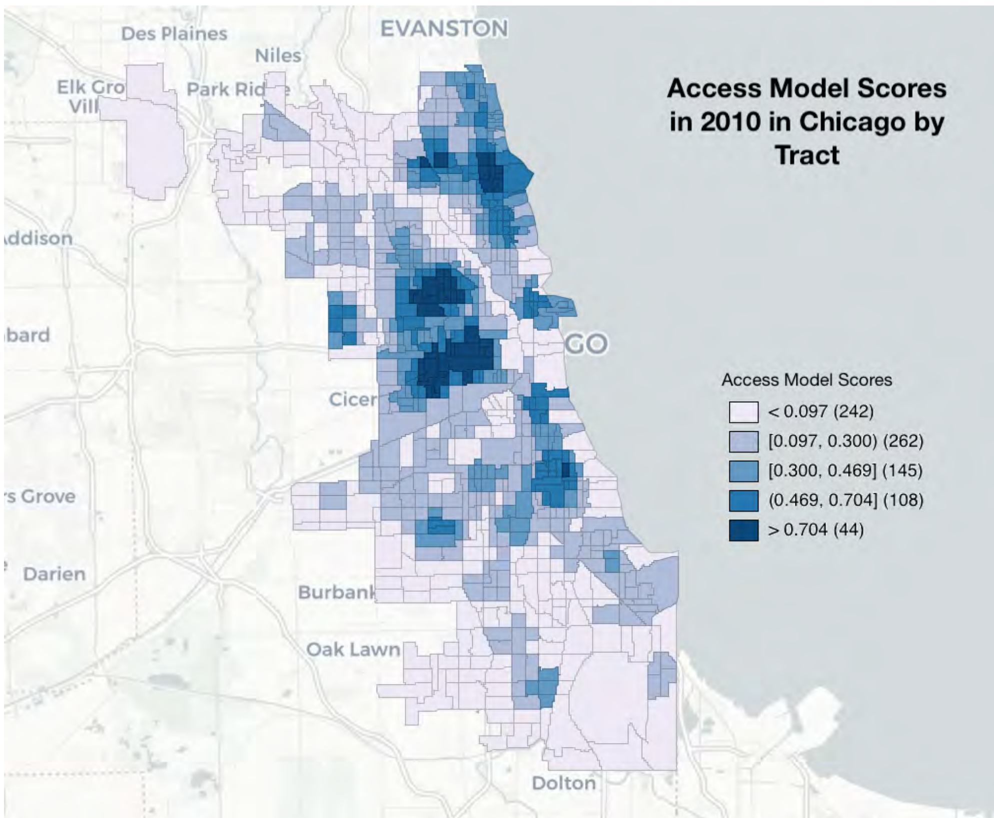
Write Output to CSV

Save the output as access_health_tracts_chicago.csv:

In [71]:
```
accessM.model_results.to_csv('./data/output_data/models/access_health_tracts
```

Visualize the data

Once the scores are in a csv, merge the scores to the origin's spatial file and map the scores to view the spatial distribution of the access scores. Here is example by tract:



Aggregate to a Larger Geographic Scale

The current results are displayed at the tract level. To view the results at a higher geographic level, we aggregate them at the community level. Then we write these results to a csv file.

In [ ]:
```
#specify a shapefile and projection
accessM.aggregate(aggregation_type=None,
                  shapefile='./data/chicago_boundaries/chicago_boundaries.sh
                  spatial_index='community',
                  projection='epsg:4326')
```

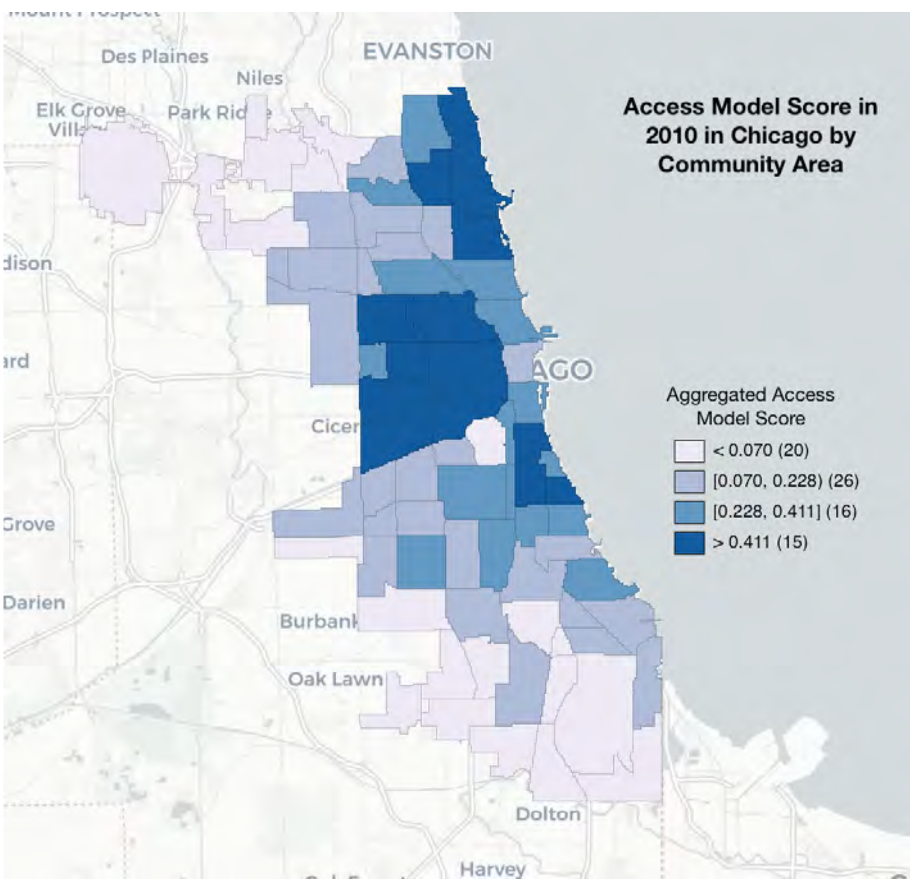
In [74]:
```
#Preview the output of the aggregated results by community area
accessM.aggregated_results.head()
```

Out[74]:

spatial_index	all_categories_score	All Free Health Clinics_score	School-Based Health Centers_score	Federally Qualified Health Centers_score	Other Health Providers_score	Hc
ALBANY PARK	15.033384	0.000000	6.700202	6.021566	0.000	
ARCHER HEIGHTS	6.844778	0.000000	0.000000	6.844778	0.000	
ARMOUR SQUARE	14.017778	0.014444	4.840333	3.961000	3.122	
ASHBURN	0.126389	0.000000	0.000000	0.000000	0.000	
AUBURN GRESHAM	5.466593	0.000000	2.484741	2.981852	0.000	

In [73]:
```
#For community areas: write to csv
accessM.write_aggregated_results(filename = "./data/output_data/models/acces
```

Visualize the Aggregated Data

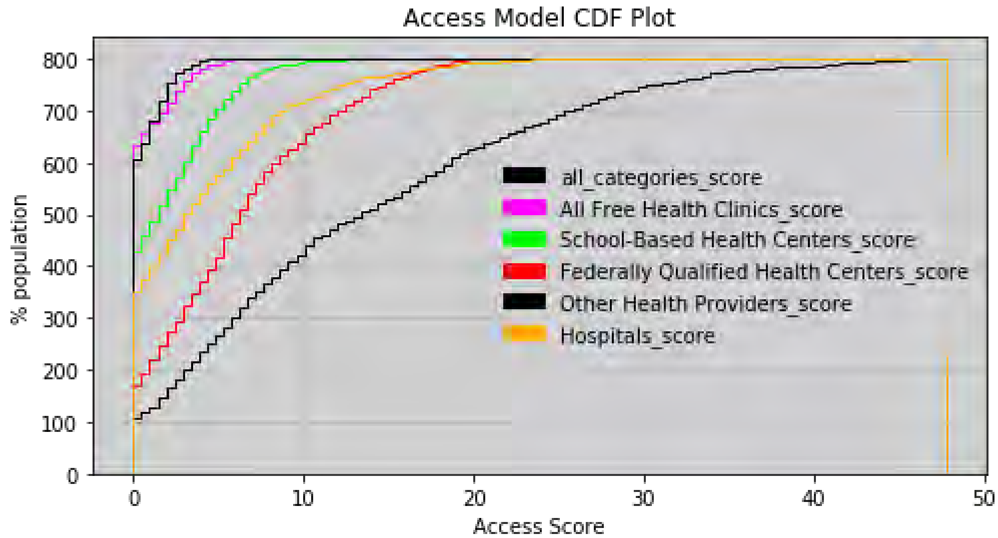


Plot Aggregated Data

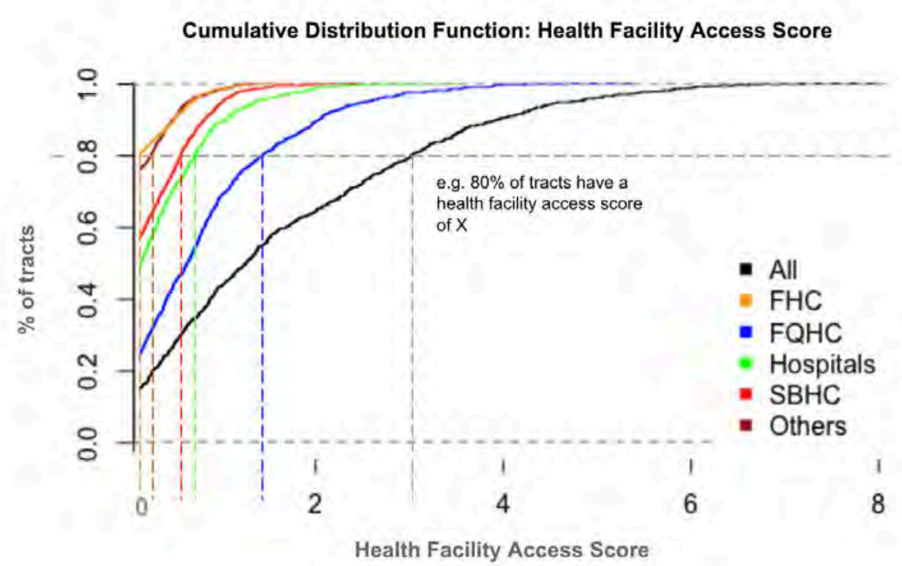
The following cumulative distribution function shows the percentage of the population by access score.

In [75]:
```
accessM.plot_cdf(filename = "./data/output_data/accessModel_CDFplot.png",
                xlabel = "Access Score", ylabel = "% population", title = "A
```

```
INFO:spatial_access.BaseModel:Plot was saved to: data/output_data/accessM
odel_CDFplot
```

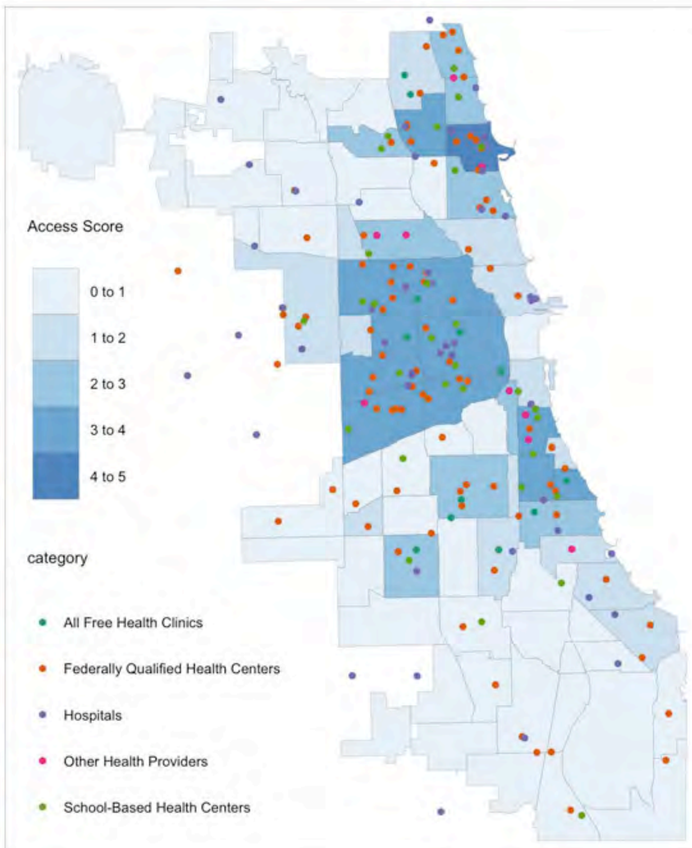


The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



```
In [ ]: accessM.plot_choropleth(column = "all_categories_score",
                                title = "Access Model Scores in Chicago",
                                filename = "./data/output_data/accessModel_choroplet
```

As with the plots above, in-built maps are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Subset the Data to Calculate the Access Score for Specific Categories

```
In [ ]: #Subset for Federally Qualified Health Centers (walking)
        accessM.set_focus_categories(['Federally Qualified Health Centers'])
```

```
In [66]: #Example of importance and variety weights:
         dict_FQ = {
         "Federally Qualified Health Centers": [10,10,10,10,10]
         }
```

```
In [ ]: accessM.calculate(upper_threshold=1800,
                          normalize=True,
                          category_weight_dict=dict_FQ)
```

In [70]:
```
#Preview the results
accessM.model_results.head()
```

Out[70]:

	all_categories_score	Federally Qualified Health Centers_score	Other Health Providers_score	School-Based Health Centers_score	All Free Health Clinics_score	Ho
17031842400	0.0	NaN	NaN	0.0	NaN	
17031840300	0.0	NaN	NaN	0.0	NaN	
17031841100	0.0	NaN	NaN	0.0	NaN	
17031841200	0.0	NaN	NaN	0.0	NaN	
17031838200	0.0	NaN	NaN	0.0	NaN	

In [72]:
```
accessM.model_results.to_csv('./data/output_data/models/FQHC.csv')
```

AccessTime: Time to the closest destination

AccessTime calculate the time it takes to reach the closest destination for each point of origin. AccessTime does not require population or target variables.

Specifications for AccessTime

name = AccessTime( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (primary input data)
- **destinations_filename** (secondary input data)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.

Column Inputs

- Standard data requirements (see above)

name.calculate()

- no specific input

Functions within the AccessTime class (use as name.function())

- calculate ()
- aggregate()
- set.focus.categories()
- plot_cdf()
- plot_choropleth

Note:

For the following models, the examples specify source_column_names and dest_column_names upfront to avoid having to specify the expected values every time.

In [ ]:
```
accessT = AccessTime(network_type='walk',
                     transit_matrix_filename = './data/matrices/walk_asym_he
                     sources_filename='./data/input_data/sources/tracts2010.
                     destinations_filename='./data/input_data/destinations/h
                     source_column_names={'idx' : 'geoid10', 'population': '
                     dest_column_names={'idx': 'ID', 'capacity': 'skip', 'ca
                    )
```

In [ ]:
```
#calculate Access Time
accessT.calculate()
```

In [80]:
```
#Preview the results
accessT.model_results.head()
```

Out[80]:

	time_to_nearest_All Free Health Clinics	time_to_nearest_School-Based Health Centers	time_to_nearest_Federally Qualified Health Centers	time_to_nearest Health Pro
17031842400	3580	1718	2472	
17031840300	3289	1777	515	
17031841100	1774	525	1376	
17031841200	2864	536	652	
17031838200	1437	853	562	

Write Data Frame to CSV

In [81]:
```
accessT.model_results.to_csv('./data/output_data/models/accessTime2010.csv')
```

Aggregate Data to the Community Area Level

In [ ]:
```
accessT.aggregate(aggregation_type = 'mean',
                  shapefile='./data/chicago_boundaries/chicago_boundaries.sh
                  spatial_index='community',
                  projection='epsg:4326')
```

In [87]:
```
accessT.aggregated_results.head()
```

Out[87]:

spatial_index	time_to_nearest_All Free Health Clinics	time_to_nearest_School-Based Health Centers	time_to_nearest_Federally Qualified Health Centers	time_to_nearest Health Pr
ALBANY PARK	3160.909091	759.545455	1020.545455	3787.
ARCHER HEIGHTS	4226.800000	2647.200000	886.800000	3563.
ARMOUR SQUARE	2600.600000	1041.600000	1108.800000	826.
ASHBURN	3996.875000	3354.375000	2955.875000	8258.
AUBURN GRESHAM	3638.533333	1179.400000	1132.466667	5745.

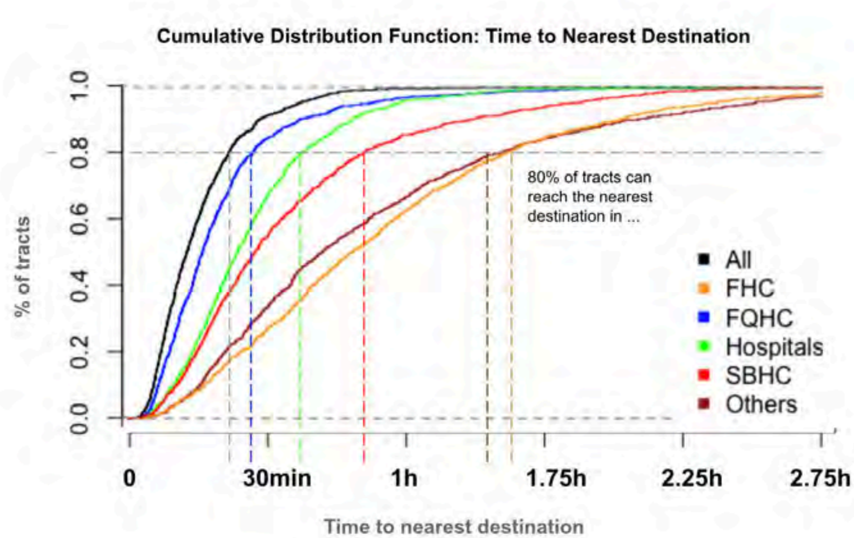
In [88]:
```
#write aggregated to csv
accessT.write_aggregated_results(filename = 'data/output_data/models/accessT
```

Plot Aggregated Data

CDF PLOT

In [ ]:
```
accessT.plot_cdf(plot_type = 'time_to_nearest_all_categories',
                 title = 'Time to Nearest',
                 xlabel = 'time (s)',
                 ylabel = 'population',
                 filename = 'data/output_data/accessTime_CDFplot_allcat.png'
                )
```

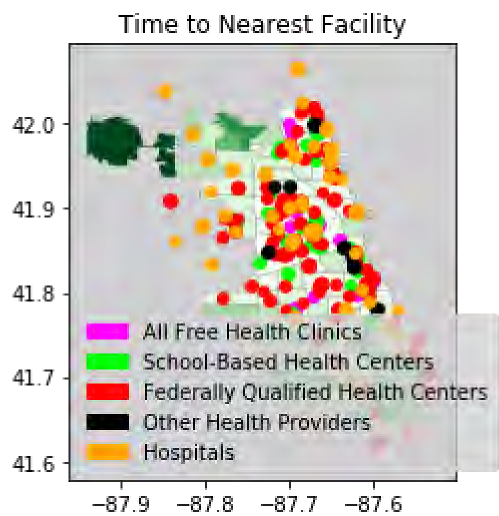
The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



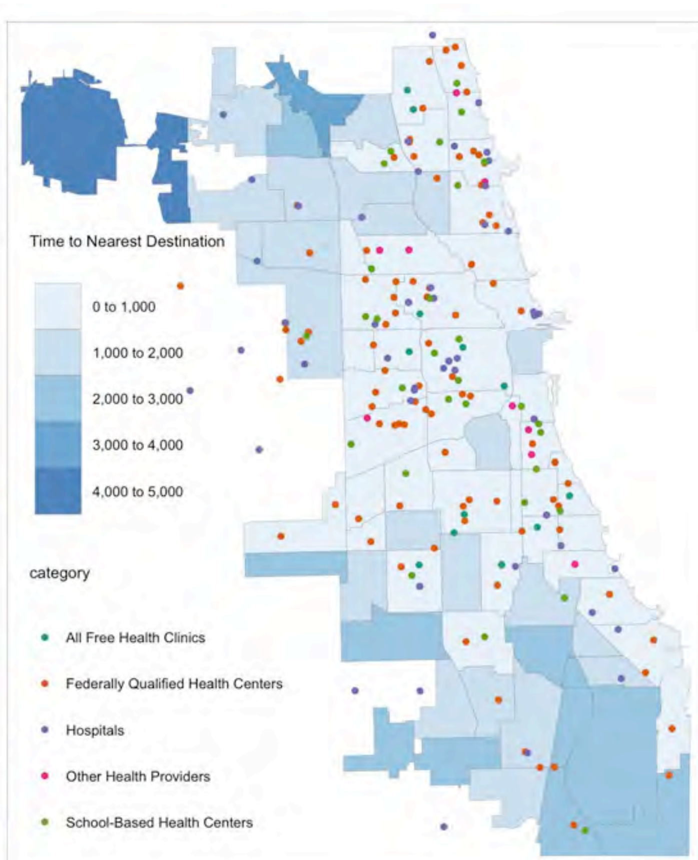
CHOROPLETH MAP

```
In [90]: accessT.plot_choropleth(column = 'time_to_nearest_all_categories',
                                 title = 'Time to Nearest Facility',
                                 filename = './data/output_data/accessTime_choropleth
```

```
INFO:spatial_access.BaseModel:Figure was saved to: data/output_data/acces
sTime_choropleth
```



As with the plots above, in-built maps are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Subset Data for Categories of Destinations

If you want to run the results for one or more provider type, you can subset the data by category.

In [91]:
```
#set focus category to FQHC; you set multiple focus categories, separated by
accessT.set_focus_categories(['Federally Qualified Health Centers'])
```

In [ ]:
```
#calculate Access Time for focus categories
accessT.calculate()
```

In [93]:
```
#Preview results
accessT.model_results.head()
```

Out[93]:

	time_to_nearest_Federally Qualified Health Centers	time_to_nearest_all_categories
17031842400	2472	2472
17031840300	515	515
17031841100	1376	1376
17031841200	652	652
17031838200	562	562

In [95]:
```
accessT.write_results(filename = './data/output_data/models/accessTime_subse
```

AccessCount: The number of destinations within a catchment area

Access Count measures the number of destinations within a given travel time (e.g. number of providers within 30 min walk of housing blocks). It does not require population or target variables.

Specifications for AccessCount

name = AccessCount( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (primary input data)
- **destinations_filename** (secondary input data)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.

Column Inputs

- Standard data requirements (see above)

name.calculate()

- upper_threshold (max time travel in seconds)

Functions within the AccessTime class (use as name.function())

- calculate ()
- aggregate()
- set.focus.categories()
- plot_cdf()
- plot_chlorepleth

In [ ]:

```
accessC = AccessCount(network_type='walk',
                      transit_matrix_filename='./data/matrices/walk_asym_hea
                      sources_filename = './data/input_data/sources/tracts2(
                      destinations_filename='./data/input_data/destinations/
                      source_column_names={'idx' : 'geoid10', 'population':
                      dest_column_names={'idx': 'ID', 'capacity': 'skip', 'c
                     )
```

In [ ]:

```
#walking threshold of 30 minutes
accessC.calculate(upper_threshold=1800)
```

In [98]:
```
#Preview the results
accessC.model_results.head()
```

Out[98]:

	count_in_range_All Free Health Clinics	count_in_range_School-Based Health Centers	count_in_range_Federally Qualified Health Centers	count_in_range_O Health Provi
17031842400	0	1	0	
17031840300	0	1	2	
17031841100	1	2	1	
17031841200	0	4	7	
17031838200	1	4	4	

In [99]:
```
#writes output to csv file
accessC.model_results.to_csv('./data/output_data/models/accessCount2010.csv'
```

Aggregate Data to the Community Area Level

In [ ]:
```
#Aggregate Access Count Data to the Chicago Community Area Level
accessC.aggregate()
```

In [101]:
```
#Preview results
accessC.aggregated_results.head()
```

Out[101]:

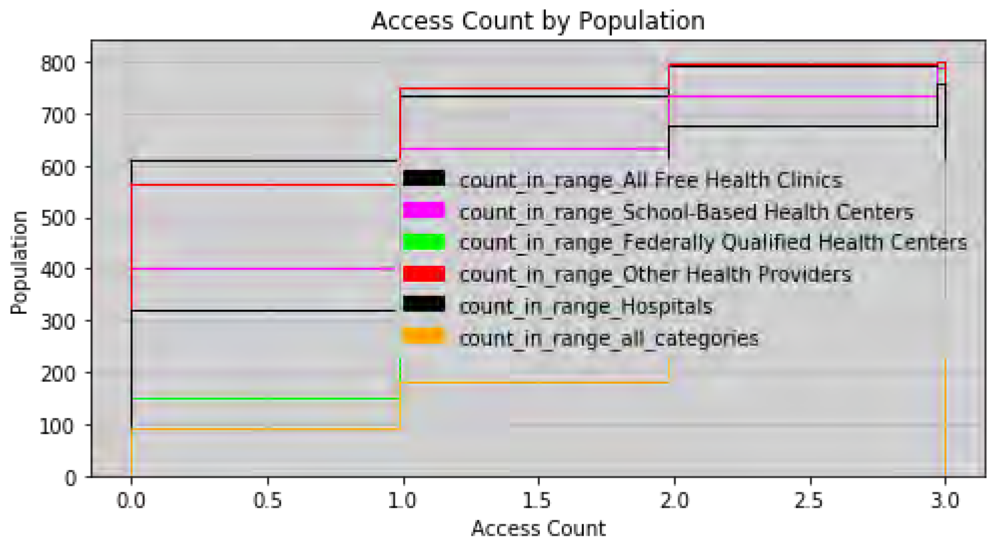
	count_in_range_All Free Health Clinics	count_in_range_School-Based Health Centers	count_in_range_Federally Qualified Health Centers	count_in_range_O Health Provi
spatial_index				
ALBANY PARK	0.0	1.818182	2.272727	
ARCHER HEIGHTS	0.0	0.000000	2.600000	
ARMOUR SQUARE	0.2	2.600000	1.600000	
ASHBURN	0.0	0.000000	0.000000	
AUBURN GRESHAM	0.0	0.866667	1.066667	

In [102]:
```
#Write results to a csv file
accessC.write_aggregated_results(filename='./data/output_data/models/accessC
```

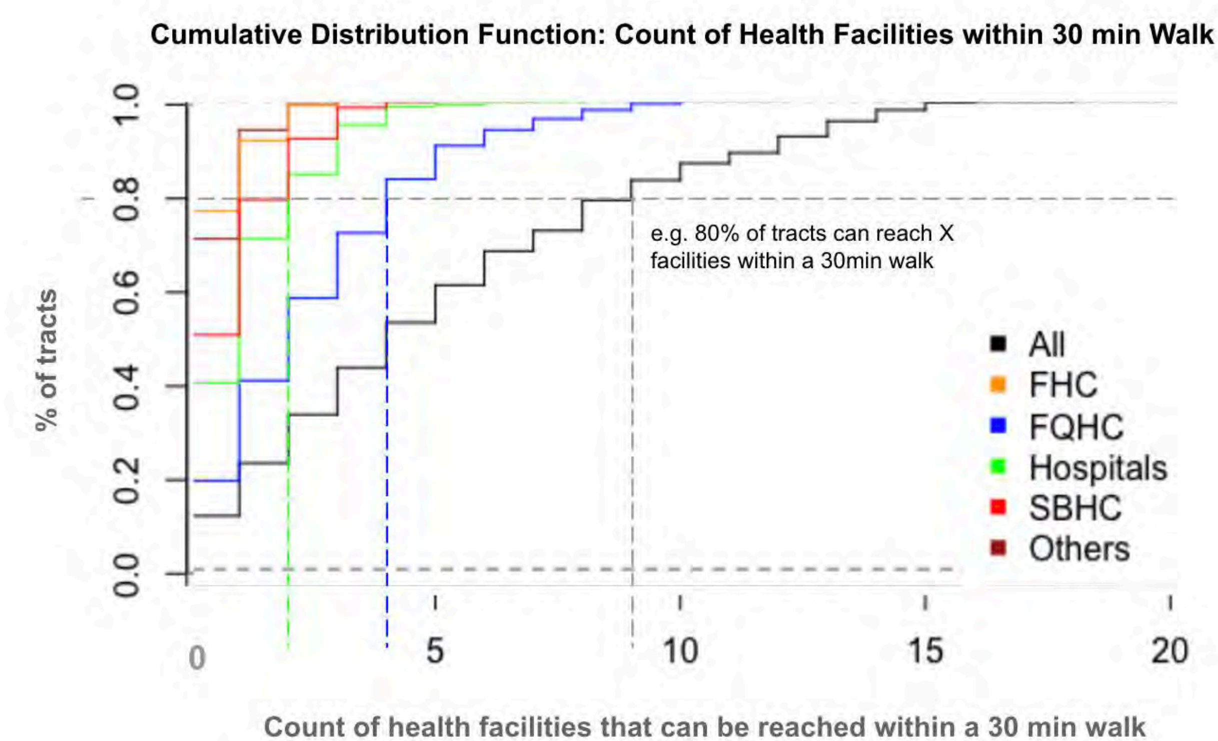
Plot Aggregated Data

In [105]:
```
accessC.plot_cdf(filename = './data/output_data/accessCount_CDFplot.png',
                 xlabel = 'Access Count',
                 ylabel = 'Population',
                 title = 'Access Count by Population')
```

```
INFO:spatial_access.BaseModel:Plot was saved to: data/output_data/accessC
ount_CDFplot
```

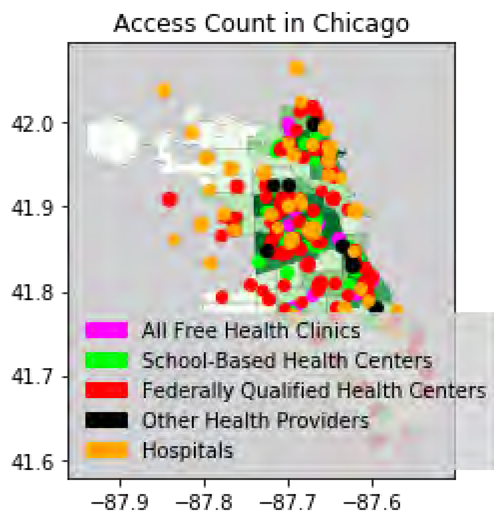


The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:

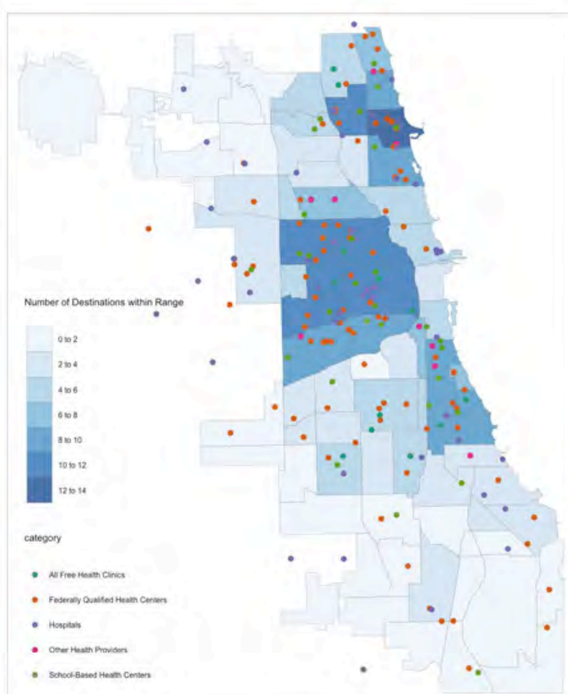


In [106]:
```
accessC.plot_choropleth(column='count_in_range_all_categories',
                        filename = './data/output_data/accessCount_choroplet
                        title= 'Access Count in Chicago')
```

```
INFO:spatial_access.BaseModel:Figure was saved to: data/output_data/acces
sCount_choropleth
```



As with the plots above, in-built maps are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Subset Data for Focus Categories

In [107]:
```
#Limit category to FQHC
accessC.set_focus_categories(['Federally Qualified Health Centers'])
```

In [ ]:
```
#Calculate Access Time for FQHC
accessC.calculate(upper_threshold = 1800)
```

In [109]:
```
#Preview subsetted results
accessC.model_results.head()
```

Out[109]:

	count_in_range_Federally Qualified Health Centers	count_in_range_all_categories
17031842400	0	0
17031840300	2	2
17031841100	1	1
17031841200	7	7
17031838200	4	4

In [110]:
```
#Write subsetted results to csv
accessC.write_results(filename = 'data/output_data/models/accessCount_subset
```

Access Sum: Captures the sum of an attribute within a catchment area

(e.g. number of doctors within a 30 min walk tracts' centroids)

Access Sum sums an attribute of a destination within a catchment area, e.g. the size of supermarkets within 30 minutes walking time from a point of origin. It requires a target variable.

Specifications for Access Sum

name = AccessSum( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (primary input data)
- **destinations_filename** (secondary input data)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.

Column Inputs

- Standard data requirements (see above) plus **capacity** for each facility

name.calculate()

- upper_threshold (max time travel in seconds)

Functions within the AccessSum class (use as name.function())

- calculate ()
- aggregate()

- set.focus.categories()
- plot_cdf()
- plot_chlorepleth

Specify travel mode, file names and variable names:

In [ ]:
```
accessS = AccessSum(network_type='walk',
                    transit_matrix_filename='./data/matrices/walk_asym_heal
                    sources_filename='./data/input_data/sources/tracts2010.
                    destinations_filename='./data/input_data/destinations/h
                    source_column_names={'idx' : 'geoid10', 'population': '
                    dest_column_names={'idx': 'ID', 'capacity': 'capacity',
                    )
```

In [ ]:
```
#Calculate results
accessS.calculate(upper_threshold=1800)
```

In [113]:
```
#Preview results
accessS.model_results.head()
```

Out[113]:

	sum_in_range_All Free Health Clinics	sum_in_range_School-Based Health Centers	sum_in_range_Federally Qualified Health Centers	sum_in_range_Other Health Providers
17031842400	0	120000	0	0
17031840300	0	163000	337000	0
17031841100	143000	268000	193000	329000
17031841200	0	654000	960000	0
17031838200	196000	721000	543000	192000

In [114]:
```
#Write model to csv
accessS.model_results.to_csv('./data/output_data/models/accessSum2010.csv')
```

Aggregate Data to the Community Area Level

In [ ]:
```
#Aggregate Access Sum data to the Chicago Community Area level
accessS.aggregate()
```

In [116]:

```
#Preview results
accessS.aggregated_results.head()
```

Out[116]:

spatial_index	sum_in_range_All Free Health Clinics	sum_in_range_School-Based Health Centers	sum_in_range_Federally Qualified Health Centers	sum_in_range_Other Health Providers
ALBANY PARK	0.0	299090.909091	326181.818182	0.0
ARCHER HEIGHTS	0.0	0.000000	333000.000000	0.0
ARMOUR SQUARE	28600.0	335400.000000	313600.000000	443800.0
ASHBURN	0.0	0.000000	0.000000	0.0
AUBURN GRESHAM	0.0	104000.000000	148200.000000	0.0

In [117]:

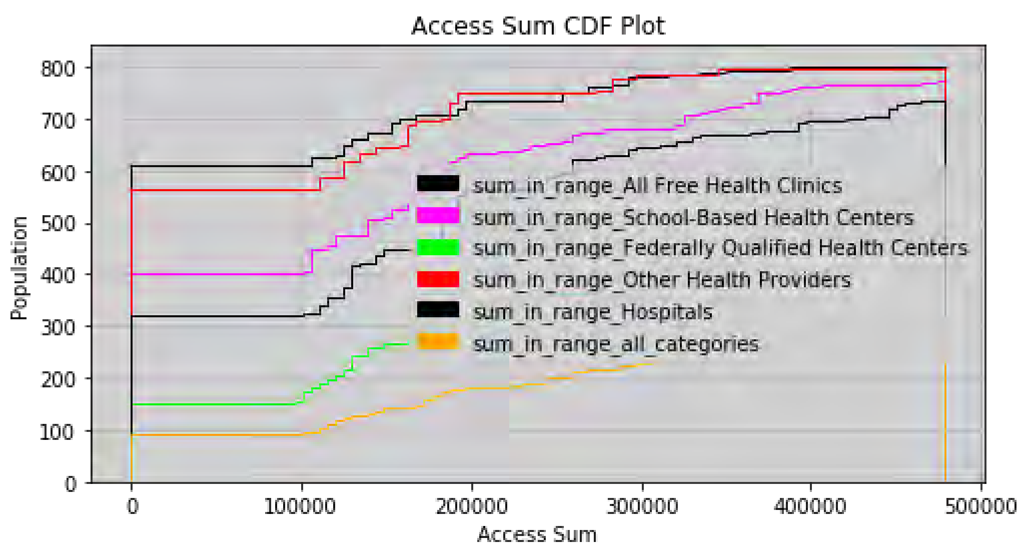
```
#Write results to a csv file
accessS.write_aggregated_results(filename='./data/output_data/models/accessS
```

Plot Aggregated Data

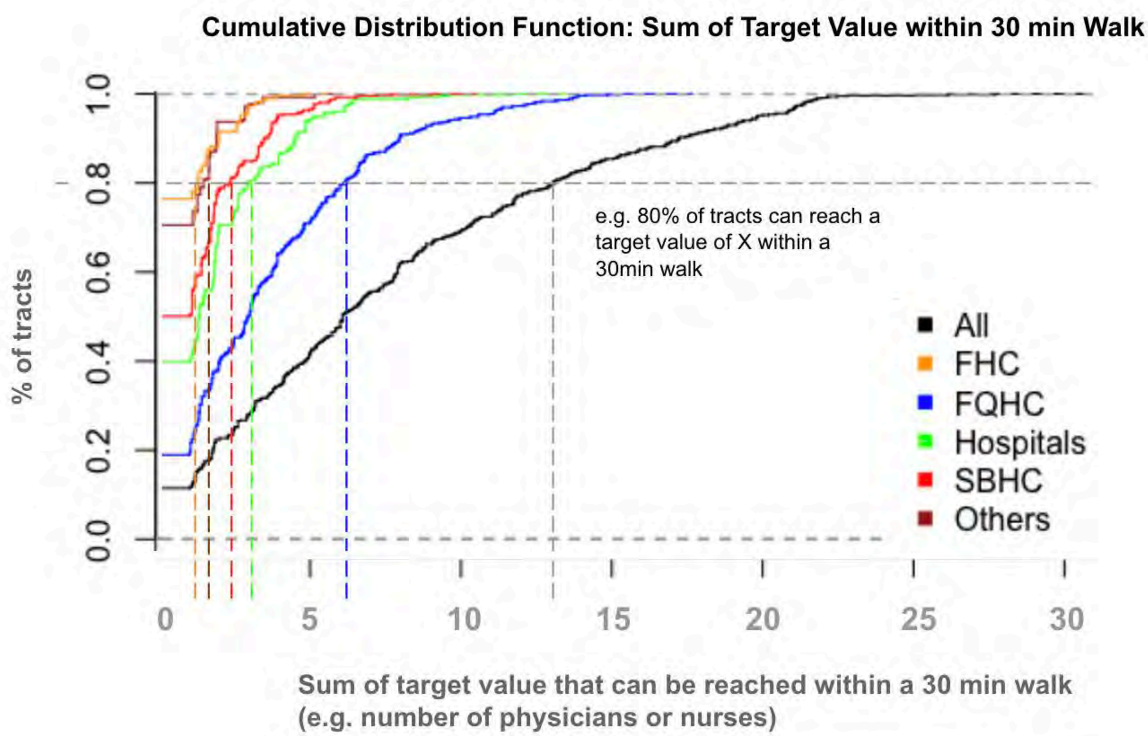
In [118]:

```
accessS.plot_cdf(filename= './data/output_data/accessSum_cdfplot.png',
                 title = 'Access Sum CDF Plot',
                 xlabel = 'Access Sum',
                 ylabel = 'Population')
```

```
INFO:spatial_access.BaseModel:Plot was saved to: data/output_data/accessS
um_cdfplot
```

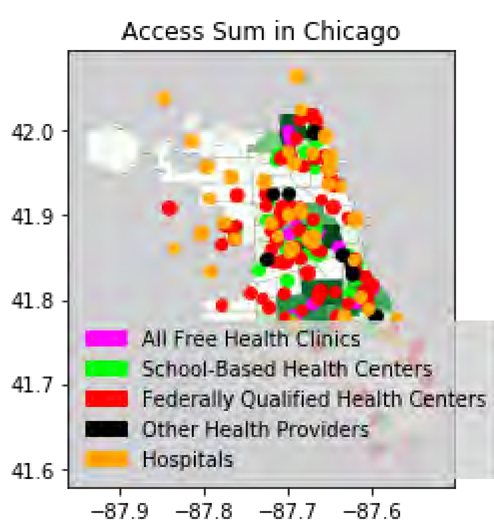


The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:

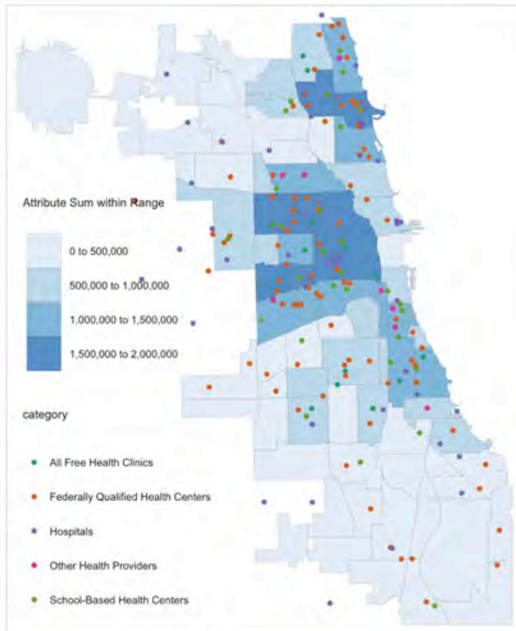


In [119]:
```
accessS.plot_choropleth(column= 'sum_in_range_All Free Health Clinics',
                        filename= 'data/output_data/accessSum_choropleth.png
                        title = 'Access Sum in Chicago')
```

INFO:spatial_access.BaseModel:Figure was saved to: data/output_data/accessSum_choropleth



As with the plots above, in-built maps are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Subset Data for Focus Categories

In [120]:
```
#Limit catgeory to for FQHC
accessS.set_focus_categories(['Federally Qualified Health Centers'])
```

In [ ]:
```
#calculate subset data
accessS.calculate(upper_threshold = 1800)
```

In [122]:
```
#preview results
accessS.model_results.head()
```

Out[122]:

	sum_in_range_Federally Qualified Health Centers	sum_in_range_all_categories
17031842400	0	0
17031840300	337000	337000
17031841100	193000	193000
17031841200	960000	960000
17031838200	543000	543000

In [123]:
```
#write subset results to csv
accessS.write_results(filename= './data/output_data/models/accessSum_subsetF
```

Destination Sum: The sum of a provider attribute within an area

(e.g. number of doctors within a community area - does not require travel time matrix)

Destination Sum sums an attribute of a destination within a geographic boundary. It also generates this result per capita within these boundaries.
This so-called container approach differs from Access Sum in that it sums point attributes within areas without relying on travel times. It requires population and target variables.

Specifications for Destination Sum

name = DestSum( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (primary input data)
- **destinations_filename** (secondary input data)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))

Column Inputs

- Standard data requirements (see above) as well as the **capacity** for each facility

name.calculate()

- shapefile (shape file of an area; here default = Chicago community areas)
- spatial_index (index of geospatial area in shapefile; here default = community)
- projection (default = 'epsg:4326')

Functions within the DestSum class (use as name.function())

- calculate ()
- set.focus.categories()
- plot_cdf()
- plot_choropleth

In [124]:
```
d_sum = DestSum(network_type='walk',
                sources_filename='./data/input_data/sources/tracts2010.csv',
                destinations_filename='data/input_data/destinations/health_c
                source_column_names={'idx' : 'geoid10', 'population': 'skip'
                dest_column_names={'idx': 'ID', 'capacity': 'capacity', 'cat
               )
```

In [ ]:
```
# calculates DestSum for Chicago
d_sum.calculate()
```

In [127]:
```
#Preview the results
d_sum.aggregated_results.head()
```

Out[127]:

	All Free Health Clinics	School-Based Health Centers	Federally Qualified Health Centers	Other Health Providers	Hospitals	all_categories	All Free Health Clinics_per_capita	C
spatial_index								
ALBANY PARK	0.0	329000.0	171000.0	0.0	0.0	500000.0	0.0	
ARCHER HEIGHTS	0.0	0.0	106000.0	0.0	0.0	106000.0	0.0	
ARMOUR SQUARE	0.0	0.0	0.0	170000.0	0.0	170000.0	0.0	
AUBURN GRESHAM	0.0	120000.0	141000.0	0.0	0.0	261000.0	0.0	
AUSTIN	0.0	190000.0	378000.0	0.0	125000.0	693000.0	0.0	

In [128]:
```
# writes result to csv
d_sum.write_aggregated_results('./data/output_data/models/destsum2010.csv')
```

In [ ]:
```
d_sum.set_focus_categories('Federally Qualified Health Centers')
```

In [ ]:
```
d_sum.head()
```

Chapter 4: Coverage Score DEMO

The metrics in 4_Access_Metrics (./4_Access_Metrics.ipynb) were attributes of the origin points, i.e. they considered spatial access from the perspective of someone accessing amenities. In contrast, the coverage metrics in this notebook are attributes of the destinations, i.e. they consider spatial access from the perspective of the service provider -- in this case for health facilities. Using the travel time matrix, you can calculate the coverage for each health facility (by type) within a catchment area. In addition to a capacity field, these metrics also require a population variable.

Coverage adds two variables to the destination file:
1) The number of people within the catchment area of a provider
2) a provider attribute divided by this nearby population count

E.g. you can use this to calculate the funding amount a service provider receives per people within the catchment area of the provider (such as 30 minutes walking time to the provider).

Each model follows the same procedure as the one presented in access models:

1. Define the model by providing the appropriate arguments
2. Calculate the model
3. Subset, aggregate, plot the results (optional)
4. Save the result as a csv or tmx file

Each of these steps are demonstrated below.

Standard Data Requirements

Each model requires two csv files as inputs: sources and destinations. Destinations need to be constrained to the spatial extent of the origins. The standard variables required for all models are listed below. Additional variables are required for some models (specified above each model).

- Source File
  - Unique index identifier (**ID**) (integer or real)
  - **Latitude** and **longitude** coordinates (real)
  - To aggregate: **ID for larger areas**
  - **Population** of the geographic unit
- Destination File
  - Unique index identifier (**ID**) (integer or real)
  - **Latitude** and **longitude** coordinates (real)
  - **Category** for each type of facility
  - To aggregate: **ID for larger areas**
  - **Capacity** for each facility

Field names with symbols will be replaced by underscores in the csv file.

In [1]:
```
cd ../..
```

In [ ]:
```
# Import modules
from spatial_access.p2p import *
from spatial_access.Models import *
```

In [ ]:
```
# View sources and destinations for Chicago health facilities
import pandas as pd
sources_df = pd.read_csv('./data/input_data/sources/tracts2010.csv')
dests_df = pd.read_csv('./data/input_data/destinations/health_chicago.csv')
```

Read in travel time matrix generated in [3_Travel_Time_Matrix (./3_Travel_Time_Matrix)](./3_Travel_Time_Matrix):

In [ ]:
```
matrix_df = pd.read_csv('./data/output_data/matrices/walk_asym_health_tracts
```

In [ ]:
```
sources_df.head()
```

In [ ]:
```
dests_df.head()
```

In [ ]:
```
matrix_df.head()
```

Specifications for the Coverage Model:

name = Coverage( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (sources file)
- **destinations_filename** (destinations file)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.

name.calculate():

- **upper_threshold** (the time (in seconds) in which the origin and destinations are considered to be out of range of each other)

Functions within the Coverage Model class (use as name.function()):

- calculate ()
- model_results (results of the Coverage calculations)
- write_csv (filename='name')
- set.focus.categories()
- aggregate ()
- write_aggregated_results()
- plot_cdf()
- plot_choropleth()

Each function is demonstrated below.

When defining the Coverage Model, use the previously generated travel time matrix. Also specify the desired distance decay function. Here, source_column_names and dest_column_names are not specified so the model will ask you to map column names to expected values.

In [ ]:
```
coverage = Coverage(network_type='walk',
                    transit_matrix_filename = './data/matrices/walk_asym_he
                    sources_filename='./data/input_data/sources/tracts2010.
                    destinations_filename='./data/input_data/destinations/h
```

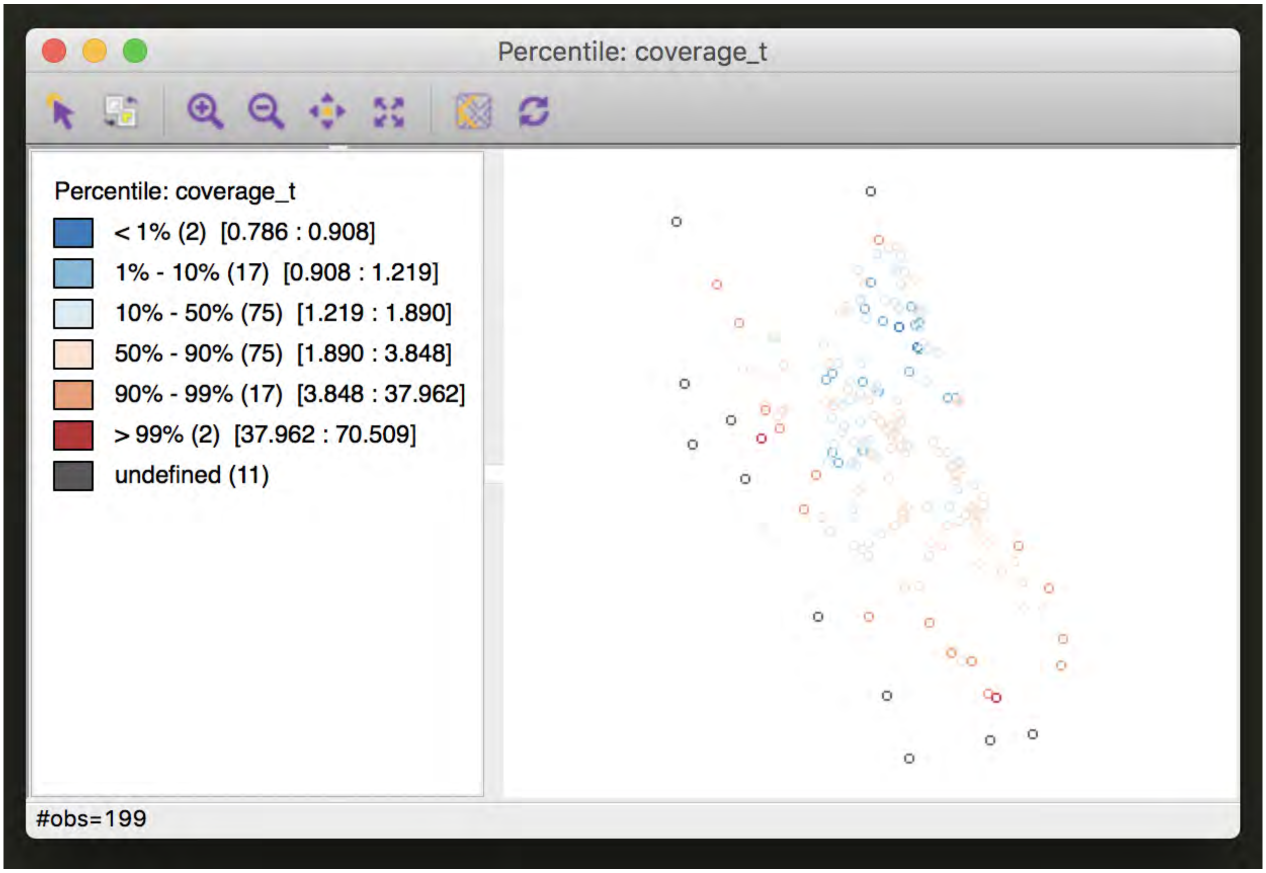
In [8]:
```
coverage.calculate(upper_threshold=1800)
```

Out[8]:

	service_pop	percap_spending	category
14	100892	1.001070	Federally Qualified Health Centers
15	49853	2.226546	Federally Qualified Health Centers
16	51802	2.721903	Federally Qualified Health Centers
17	74269	2.113937	Federally Qualified Health Centers
18	41958	4.004004	Federally Qualified Health Centers

In [ ]:
```
coverage.model_results.head()
```

In [9]:
```
#Writes output to csv
coverage.model_results.to_csv('./data/output_data/models/coverage_results.cs
```



Calculate the Coverage Score for a Subset of the Data

In [ ]:
```
#Set the Subset to Federally Qualified Health Centers
coverage.set_focus_categories(['Federally Qualified Health Centers'])
```

#Set the importance and variety weights:

dict = { "Federally Qualified Health Centers": [10,10,10,10,10] }

In [ ]:
```
coverage.calculate(upper_threshold=1800)
```

In [ ]:
```
#Preview the results
coverage.model_results.head()
```

In [ ]:
```
coverage.model_results.to_csv('FQHC_coverage.csv')
```

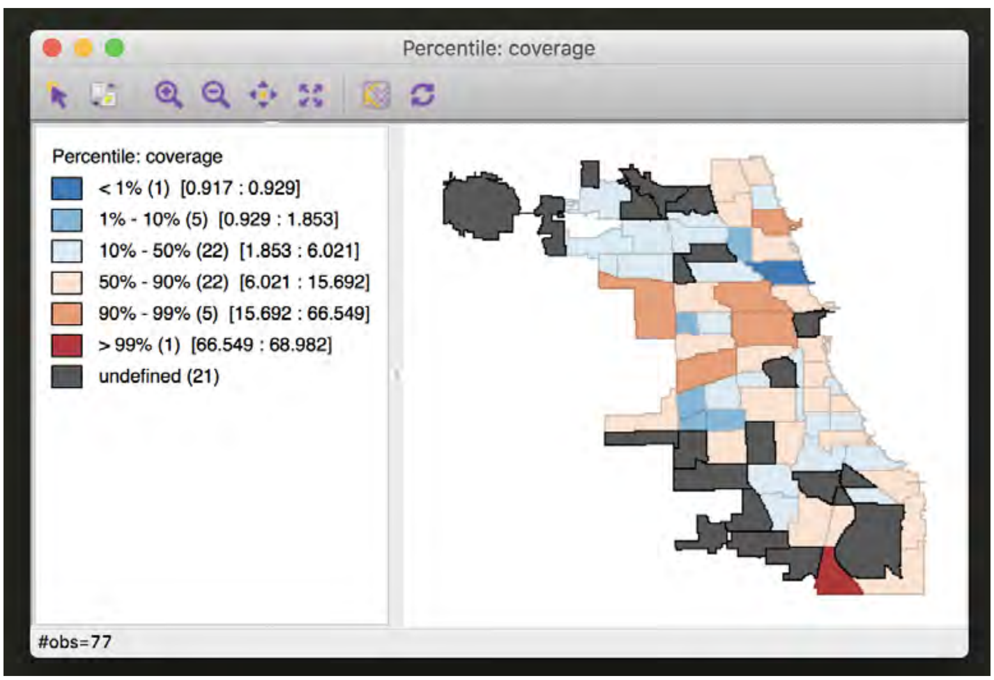
Aggregation by larger geographic units

In [10]:
```
coverage.aggregate(aggregation_type=None,
                   shapefile='./data/chicago_boundaries/chi_comm_boundaries.
                   spatial_index='community',
                   projection='epsg:4326').head()
```

Out[10]:

spatial_index	service_pop	percap_spending
ALBANY PARK	305008	1.638229
ARCHER HEIGHTS	59620	1.777927
ARMOUR SQUARE	58671	2.897513
AUBURN GRESHAM	106340	2.461102
AUSTIN	248120	3.094732

In [11]:
```
#For community areas write to csv
coverage.write_aggregated_results(filename = "./data/coverage score/coverage
                                  output_type = 'csv')
```

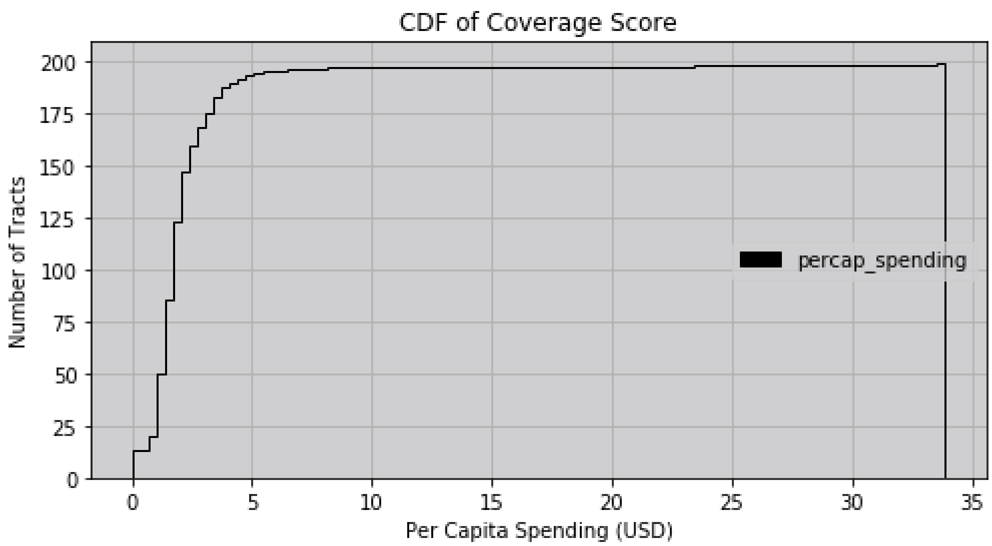


CDF Plot

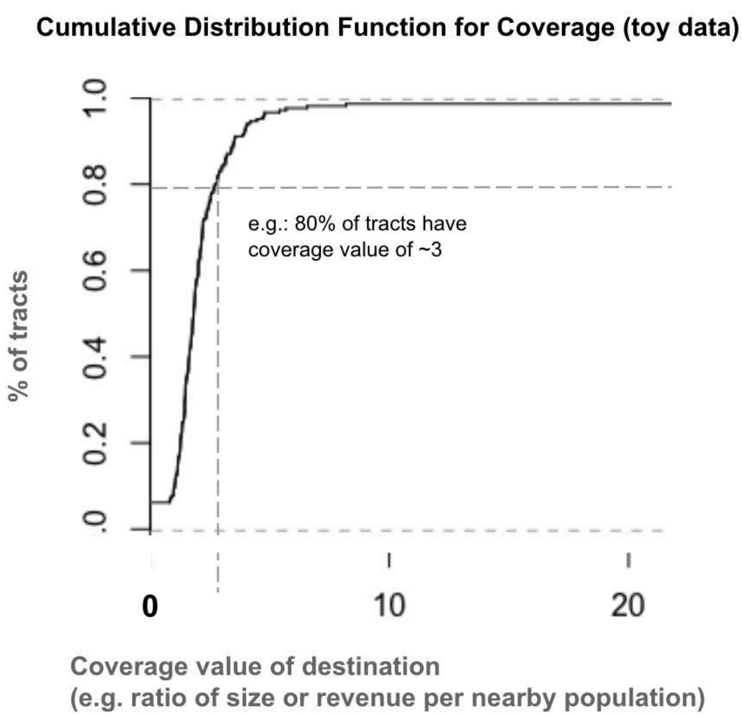
The following cumulative distribution function shows the number of tracts that fall below a certain level of per capita spending.

In [13]:
```
coverage.plot_cdf(filename = './data/coverage score/coverage_cdf_plot.png',
                  plot_type = "percap",
                  title = 'CDF of Coverage Score',
                  xlabel = 'Per Capita Spending (USD)',
                  ylabel = 'Number of Tracts')
```

```
INFO:spatial_access.BaseModel:Plot was saved to: /Users/whlu/spatial_acce
ss/data/coverage score/coverage_cdf_plot.png
```



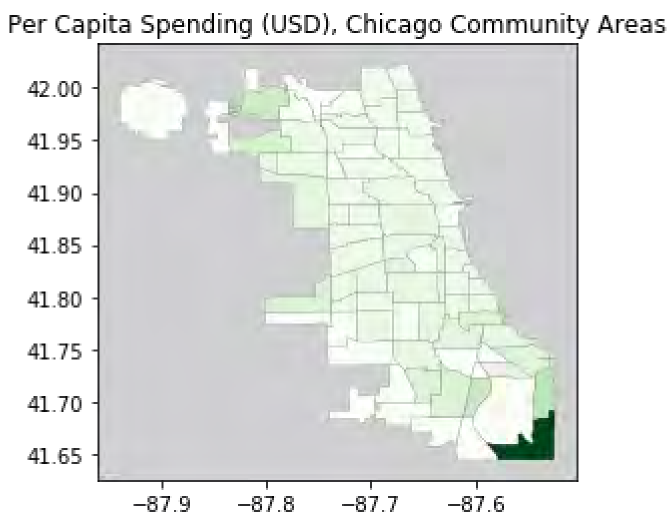
The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Choropleth Mapping

In [15]:
```
coverage.plot_choropleth(column = "percap_spending",
                         shapefile='./data/chicago_boundaries/chi_comm_bound
                         title = 'Per Capita Spending (USD), Chicago Communi
                         include_destinations = False,
                         filename = './data/coverage score/coverage_chorople
```

```
INFO:spatial_access.BaseModel:Figure was saved to: /Users/whlu/spatial_ac
cess/data/coverage score/coverage_choropleth.png
```



Chapter 5: Two Stage Floating Catchment Area DEMO

This notebook calculates the Two Stage Floating Catchment Area (TSFCA) model, using the travel time matrix as an input -- in this case, access to per capita spending for different types of health services.

TSFCA Models are a type of gravity model popularized by Luo and Wang in 2003 to estimate spatial access gaps to primary care. They are calculated in two stages (using the primary care example): In a first stage, the ratio of doctors to the nearby population is calculated for every provider. In the 2nd stage, these ratios are summed for every point of origin (such as a tract centroid) within a travel threshold. In other words, the ratio of doctors to people is first calculated for the catchment areas of doctors (1st stage) and then summed for the catchment areas around people's point of origins, like their home (2nd stage).

Each model follows the same procedure as the one presented in access models:

1. Define the model by providing the appropriate arguments
2. Calculate the model
3. Subset, aggregate, plot the results (optional)
4. Save the result as a csv or tmx file

Each of these steps are demonstrated below.

Standard Data Requirements
Each model uses inputs from both the sources and destination csv files. Destinations need to be constrained to the spatial extent of the origins.

- Source File
    - Unique index identifier (**ID**) (integer or real)
    - **Latitude** and **longitude** coordinates (real)
    - To aggregate: **larger areal ID**
    - **Population** within the areal unit
- Destination File
    - Unique index identifier (**ID**) (integer or real)
    - **Latitude** and **longitude** coordinates (real)
    - **Category** for each type of facility
    - **Capacity** for each facility
    - To aggregate: **larger areal ID**

In [2]:
```
cd ../..
```

```
/Users/irenefarah/Documents/GitHub/spatial_access
```

In [ ]:
```
# Import modules
from spatial_access.p2p import *
from spatial_access.Models import *
```

In [3]:
```
# View sources and destinations for Chicago health facilities
import pandas as pd
sources_df = pd.read_csv('./data/input_data/sources/tracts2010.csv')
dests_df = pd.read_csv('./data/input_data/destinations/health_chicago.csv')
```

Read in travel time matrix generated in [1_matrix.ipynb (./1_matrix.ipynb)](./1_matrix.ipynb):

In [4]:
```
matrix_df = pd.read_csv('./data/output_data/matrices/walk_asym_health_tracts
```

In [5]:
```
sources_df.head()
```

Out[5]:

	geoid10	lon	lat	Pop2014	Pov14	community
0	17031842400	-87.630040	41.742475	5157	769	44
1	17031840300	-87.681882	41.832094	5881	1021	59
2	17031841100	-87.635098	41.851006	3363	2742	34
3	17031841200	-87.683342	41.855562	3710	1819	31
4	17031838200	-87.675079	41.870416	3296	361	28

In [6]:
```
dests_df.head()
```

Out[6]:

	ID	Facility	lat	lon	Type	capacity	category	community
0	1	American Indian Health Service of Chicago, Inc.	41.956676	-87.651879	5	127000	Other Health Providers	3
1	2	Hamdard Center for Health and Human Services	41.997852	-87.669535	5	190000	Other Health Providers	77
2	3	Infant Welfare Society of Chicago	41.924904	-87.717270	5	137000	Other Health Providers	22
3	4	Mercy Family - Henry Booth House Family Health...	41.841694	-87.624790	5	159000	Other Health Providers	35
4	6	Cook County - Dr. Jorge Prieto Health Center	41.847143	-87.724975	5	166000	Other Health Providers	30

In [7]:
```
matrix_df.head()
```

Out[7]:

	Unnamed: 0	1	2	3	4	6	8	9	10	11	...	198	199
0	17031842400	17870	21397	17892	8483	13529	12425	5704	16935	7565	...	13236	16767
1	17031840300	11391	13937	8845	4120	3713	3939	6358	8448	3721	...	4671	7050
2	17031841100	9050	12577	9155	1374	5748	4035	5015	8198	2295	...	4799	8430
3	17031841200	9649	12195	7306	4588	3049	2017	8015	6846	5313	...	2958	5512
4	17031838200	8222	10768	6219	5068	3794	590	8589	5440	5887	...	1552	4789

5 rows × 201 columns

Specifications: Coverage Model:

name = tsfca( )

- **network_type** ('walk', 'bike', 'drive', 'otp')
- **sources_filename** (sources file)
- **destinations_filename** (destinations file)
- **source_column_names** (dictionary that contains column names (lat/lon/ID))
- **dest_column_names** (dictionary that contains column names (lat/lon/ID/category))
- **transit_matrix_filename** (sources-destination travel time matrix). If None, matrix estimated 'on the fly'.

name.calculate():

- **upper_threshold** (the time (in seconds) in which the origin and destinations are considered to be out of range of each other)

Functions within the TSFCA Model class (use as name.function()):

- calculate ()
- model_results (results of the TSFCA calculations)
- write_csv ()
- set.focus.categories()
- aggregate ()
- write_aggregated_results()
- plot_cdf()
- plot_choropleth()

Each function is demonstrated below

When defining the TSFCA Model, use the previously generated shortest-path matrix. Also specify the desired distance decay function. Here, source_column_names and dest_column_names are not specified so the model will ask the user to map column names to expected values.

When defining the TSFCA Model, use the previously generated travel time matrix. Also specify the desired distance decay function. Here, source_column_names and dest_column_names are not specified so the model will ask you to map column names to expected values.

In [ ]:
```
#Specify the network type here
tsfca = TSFCA(network_type='walk',
              sources_filename='./data/input_data/tracts2010.csv',
              destinations_filename='./data/input_data/health_chicago.csv')
```

In [ ]:
```
tsfca.calculate(upper_threshold=1800)
```

In [17]:
```
#Output
tsfca.model_results.head()
```

Out[17]:

	percap_spend_Other Health Providers	percap_spend_Hospitals	percap_spend_All Free Health Clinics	percap_spend_School-Based Health Centers	percap_s Qualified
1	0.000000	0.000000	0.000000	2.200301	
2	0.000000	0.000000	0.000000	3.192260	
3	5.050584	1.728574	3.377900	3.496114	
4	0.000000	5.871718	0.000000	8.771760	
5	2.542609	11.854331	2.499107	10.075997	

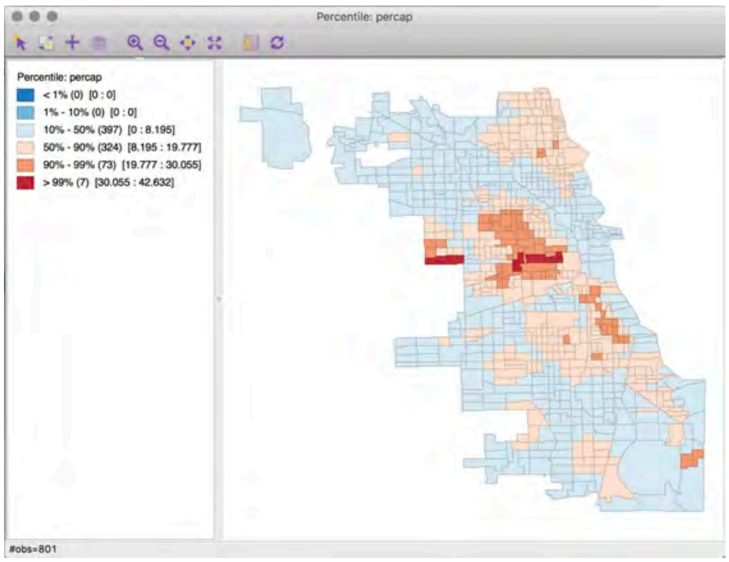
Scores

- tsfca_Federally Qualified Health Centers is the access to per capita spending by Federally Qualified Health Centers
- tsfca_School-Based Health Centers is the access to per capita spending by School-based Health Centers
- tsfca_All Free Health Clinics is the access to per capita spending by free health clinics
- tsfca_Hospitals is the access to per capita spending by hospitals

- tsfca_Other Health Providers is the access to per capita spending by all other healthcare providers
- tsfca_all_categories is the sum of the above five categories

Merge the per capita spending data to the origin's shapefile and map them out in order to view the distribution of the access to per capita spending by tract:

```
<img src="./figures/tsfca_results.png" width = 600>
```



In [18]:
```
#Writes output to csv
tsfca.model_results.to_csv('./data/tsfca/tsfca_results.csv')
```

Calculate the TSFCA Score for a Subset of the Data

In [ ]:
```
#Set the Subset to Federally Qualified Health Centers
tsfca.set_focus_categories(['Federally Qualified Health Centers'])
```

In [ ]:
```
#Set the importance and variety weights:

dict = {
"Federally Qualified Health Centers": [10,10,10,10,10]
}
```

In [ ]:
```
#Calculate the score for 30 minutes travel time
tsfca.calculate(upper_threshold=1800)
```

In [ ]:
```
#Preview the first 5 rows of the results
tsfca.model_results.head()
```

In [ ]:
```
#Save the results to csv
tsfca.model_results.to_csv('FQHC_tsfca.csv')
```

Aggregation to larger areas

```
#Gets the output of the aggregation by access to per capita spending by comm
In [19]: tsfca.aggregate(aggregation_type=None,

                    shapefile='./data/chicago_boundaries/chi_comm_boundaries.shp
                    spatial_index='community',
                    projection='epsg:4326').head()
```

Out[19]:

spatial_index	percap_spend_Other Health Providers	percap_spend_Hospitals	percap_spend_All Free Health Clinics	percap_spend_School-Based Health Centers
ALBANY PARK	0.000000	1.284000	0.00000	3.121864
ARCHER HEIGHTS	0.000000	0.000000	0.00000	0.000000
ARMOUR SQUARE	6.248138	1.382859	0.67558	4.595124
ASHBURN	0.000000	1.024378	0.00000	0.000000
AUBURN GRESHAM	0.000000	0.000000	0.00000	1.906927

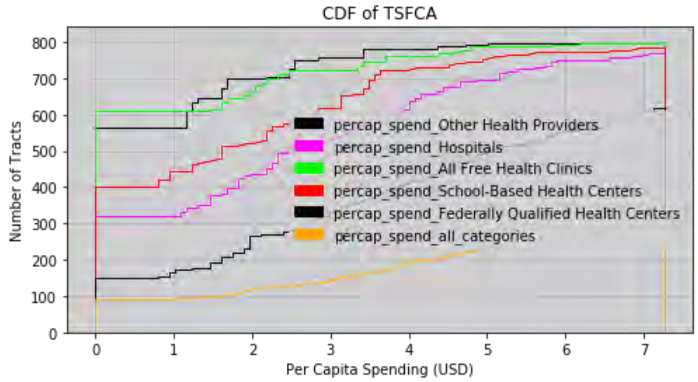
```
In [10]: #For community areas write to csv
         tsfca.write_aggregated_results(filename = "./data/tsfca/tsfca_aggregated.csv
```

CDF Plot

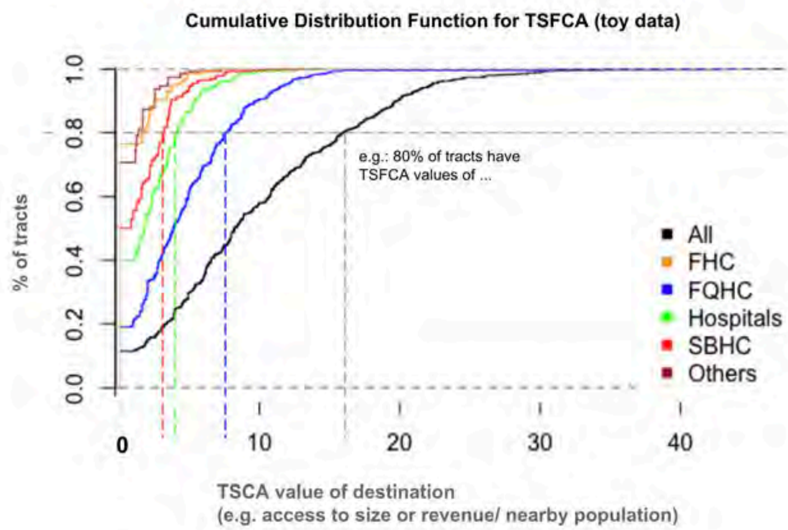
The following cumulative distribution function shows the number of tracts that fall below a certain level of per capita spending.

```
In [11]: tsfca.plot_cdf(filename = './data/tsfca/tsfca_cdf_plot.png',
                        plot_type = "percap", title = 'CDF of TSFCA',
                        xlabel = 'Per Capita Spending (USD)',
                        ylabel = 'Number of Tracts')
```

```
INFO:spatial_access.BaseModel:Plot was saved to: /Users/whlu/spatial_acce
ss/data/tsfca/tsfca_cdf_plot.png
```



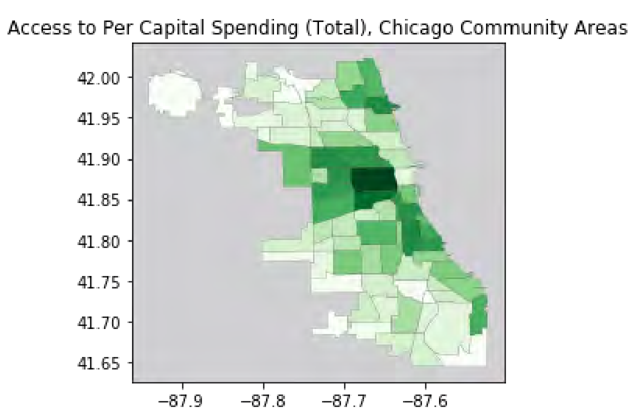
The in-built charts are not designed for presentation purposes but you can save the results and graph them in another program. Here is an example:



Choropleth Mapping

In [12]:
```
tsfca.plot_choropleth(column = "percap_spend_all_categories",
                      shapefile='/./data/chicago_boundaries/chi_comm_boundar
                      title = 'Access to Per Capital Spending (Total), Chica
                      include_destinations = False,
                      filename = './data/tsfca/tsfca_choropleth.png')
```

```
INFO:spatial_access.BaseModel:Figure was saved to: /Users/whlu/spatial_ac
cess/data/tsfca/tsfca_choropleth.png
```



APPENDIX

Appendix: Simple Demo for Testing

This notebook lets you test the core spatial access metrics with a toy dataset before running your own data. You will input two stored csv files:

1) **hyde_park_tracts.csv** contains 12 points of origins (tract centroids for Hyde Park, Chicago + population field)
2) **hyde_park_dests.csv** contains 7 amenities in three categories: museums, restaurants and supermarkets and one target field (this is an attribute of the amenity like number of employees, size or revenue).

You will first create a matrix of walking times (in seconds) from these points of origin to the 7 destinations (the matrix will have 12 rows and 7 columns).

Then, the demo runs through a basic version of each spatial access and coverage metric for illustration purposes. The functionality of each spatial access metric is explained in more detail in the following notebooks.

In [ ]:
```
# Check to see what version of spatial access you are using
! pip3 show spatial-access
```

In [ ]:
```
cd ../..
```

Creating the Travel Time Matrix

This generates a matrix of walking times (in seconds) from the 12 origins to the 7 destinations (12 rows x 7 columns).

In [ ]:
```
from spatial_access.p2p import *
```

Read in the stored source and destination csv files:

In [4]:
```
import pandas as pd
sources_df = pd.read_csv('./data/input_data/sources/hyde_park_tracts.csv')
dests_df = pd.read_csv('./data/input_data/destinations/hyde_park_dests.csv')
```

View the source data (12 tract centroids):

In [5]:
```
sources_df
```

Out[5]:

	geoid10	lon	lat	Pop2014	Pov14	community
0	17031836300	-87.601757	41.801532	6465	234	41
1	17031836200	-87.601284	41.790469	1329	47	41
2	17031410100	-87.579323	41.801497	1956	551	41
3	17031410200	-87.594269	41.801668	1248	362	41
4	17031410500	-87.603745	41.797827	2630	717	41
5	17031410600	-87.598946	41.797971	2365	703	41
6	17031411100	-87.589702	41.790449	2246	154	41
7	17031410700	-87.594198	41.798040	1959	453	41
8	17031410800	-87.589626	41.797960	3201	741	41
9	17031410900	-87.576659	41.797874	2923	607	41
10	17031411000	-87.576873	41.790716	3313	465	41
11	17031411200	-87.594017	41.790556	1691	289	41

View the destination data (7 amenities):

In [6]:
```
dests_df
```

Out[6]:

	name	lon	lat	category	target
0	Museum of Science and Industry	-87.583131	41.790883	Museum	400
1	Medici	-87.593738	41.791438	Restaurant	50
2	Valois	-87.588328	41.799663	Restaurant	30
3	DuSable Museum	-87.607132	41.791985	Museum	100
4	Whole Foods	-87.587949	41.801978	Supermarket	50
5	Hyde Park Produce	-87.595524	41.799942	Supermarket	35
6	Jewel Osco	-87.607225	41.784580	Supermarket	70

Specify travel mode, variable names, and file locations:

```
In [ ]: #asymmetric matrix, different source and destination files
        matrix = TransitMatrix(network_type='walk',
                               primary_hints={'idx' : 'geoid10', 'population': 'skip
                               secondary_hints={'idx': 'name', 'capacity': 'skip', '
                               primary_input='./data/input_data/sources/hyde_park_tr
                               secondary_input='./data/input_data/destinations/hyde_
```

Get the travel times by querying OpenStreetMap data for the spatial extent of your source and destination coordinates:

```
In [ ]: matrix.process()
```

Save the travel time matrix in csv and/or tmx format (running access metrics with tmx is faster):

```
In [ ]: matrix.write_csv('./data/output_data/matrices/simple_demo_matrix.csv')
```

```
In [ ]: matrix.write_tmx('./data/output_data/matrices/simple_demo_matrix.tmx')
```

Access Metrics (Attributes of the Origin File)

Next, the travel time matrix serves as the input for the calculation of several spatial access metrics. We first calculate spatial access measures that are attributes of the point of origin (12 tract centroids). After that, we calculate so-called coverage metrics that are attributes of the destination points (7 amenities).

```
In [10]: from spatial_access.Models import *
```

Access Model

The first line of code defines the Access Model using the previously generated matrix of travel times from above.
If you specify **`transit_matrix_filename=None`** , the matrix will be estimated on the fly.

The Access Model generates an access score to measure how accessible a location is to multiple amenities within a given travel time (e.g. 20 minutes walking). You can specify three types of weights for this score:

1) **distance decay** where closer amenities have more weight (default = linear)
2) **relative importance of an amenity type** (e.g. with a greater weight for supermarkets than museums)
3) **penalty for same types** (where more of the same type of amenity gets less weight).

You can estimate the score with or without normalization.
The AccessModel does not require population or target variables.

Specify travel mode, file names, variable names and the distance decay function:

In [ ]:
```
access = AccessModel(network_type='walk',
                     transit_matrix_filename='./data/output_data/matrices/si
                     sources_filename='./data/input_data/sources/hyde_park_t
                     destinations_filename='./data/input_data/destinations/h
                     source_column_names={'idx' : 'geoid10', 'population': '
                     dest_column_names={'idx': 'name', 'capacity': 'skip', '
                     decay_function = 'linear')
```

Specify the weights for relative importance and same types:

In [12]:
```
category_dict = {
    "Museum": [5, 5, 3],
    "Restaurant": [10, 10],
    "Supermarket": [10, 7, 5]
}
```

Specify the travel time threshold in seconds (e.g. 1,800 seconds = 30 minutes), whether or not to normalize the score, and the importance/variety weights:

In [ ]:
```
access.calculate(upper_threshold=1800,
                 normalize=False,
                 category_weight_dict=category_dict)
```

View the first 5 records of the access score results by category:

In [17]:
```
access.model_results.head()
```

Out[17]:

	all_categories_score	Museum_score	Supermarket_score	Restaurant_score
17031836300	19.318333	2.108333	9.826667	7.383333
17031836200	22.553333	4.983333	8.342222	9.227778
17031410100	18.303889	2.011111	8.398333	7.894444
17031410200	25.826667	1.647222	12.301667	11.877778
17031410500	21.282778	3.519444	9.713333	8.050000

In [18]:
```
access.model_results.to_csv('./data/output_data/models/simple_demo_accessMod
```

Access Time: Time to closest destination

Next, you will calculate the time it takes to reach the closest destination for each point of origin. As before, you define the Access Time model using the sources and destinations csv. AccessTime does not require population or target variables.

```
In [ ]: accessT = AccessTime(network_type='walk',
                         transit_matrix_filename='./data/output_data/matrices/si
                         sources_filename='./data/input_data/sources/hyde_park_t
                         destinations_filename='./data/input_data/destinations/h
                         source_column_names={'idx' : 'geoid10', 'population': '
                         dest_column_names={'idx': 'name', 'capacity': 'skip', '
                        )
```

```
In [ ]: accessT.calculate()
```

```
In [ ]: accessT.model_results.head()
```

```
In [21]: accessT.model_results.to_csv('data/output_data/models/simple_demo_accessT.cs
```

Access Count: Number of Destinations within a Catchment Area

Access Count measures the number of destinations within a given travel time.
In this case, the catchment area is 1,800 seconds (30 minutes) of walking from a point of origin.
It does not require population or target variables.

```
In [ ]: accessC = AccessCount(network_type='walk',
                         transit_matrix_filename='./data/output_data/matrices/si
                         sources_filename='./data/input_data/sources/hyde_park_t
                         destinations_filename='./data/input_data/destinations/h
                         source_column_names={'idx' : 'geoid10', 'population': '
                         dest_column_names={'idx': 'name', 'capacity': 'skip', '
                         )
```

```
In [ ]: accessC.calculate(upper_threshold=1800)
```

```
In [34]: accessC.model_results.head()
```

Out[34]:

	count_in_range_Museum	count_in_range_Supermarket	count_in_range_Restaurant	cour
17031836300	1	3	2	
17031836200	2	3	2	
17031410100	1	2	2	
17031410200	2	2	2	
17031410500	2	3	2	

```
In [25]: accessC.model_results.to_csv('data/output_data/models/simple_demo_accessC.cs
```

Access Sum: The sum of an attribute of a destination within a given travel time

Access Sum sums an attribute of a destination within a catchment area, e.g. the size of supermarkets within 30 minutes walking time from a point of origin. It requires a target variable.

In [ ]:
```
accessS = AccessSum(network_type='walk',
                    transit_matrix_filename='data/output_data/matrices/simp
                    sources_filename='data/input_data/sources/hyde_park_tra
                    destinations_filename='data/input_data/destinations/hyd
                    source_column_names={'idx' : 'geoid10', 'population': '
                    dest_column_names={'idx': 'name', 'capacity': 'target',
                  )
```

In [ ]:
```
accessS.calculate(upper_threshold=1800)
```

In [42]:
```
accessS.model_results.head()
```

Out[42]:

	sum_in_range_Museum	sum_in_range_Supermarket	sum_in_range_Restaurant	sum_in_r
17031836300	100	155	80	
17031836200	500	155	80	
17031410100	400	85	80	
17031410200	500	85	80	
17031410500	500	155	80	

In [40]:
```
accessS.model_results.to_csv('./data/output_data/simple_demo_accessS.csv')
```

Destination Sum: Sum of a provider charactistic by area

Destination Sum sums an attribute of a destination within a geographic boundary. It also generates this result per capita within these boundaries.
This so-called container approach differs from Access Sum in that it sums point attributes within areas without relying on travel times. It requires population and target variables.

In [ ]:
```
d_sum = DestSum(network_type='walk',
                sources_filename='d./ata/input_data/sources/hyde_park_tracts
                destinations_filename='./data/input_data/destinations/hyde_p
                source_column_names={'idx' : 'geoid10', 'population': 'skip'
                dest_column_names={'idx': 'name', 'capacity': 'target', 'cat
               )
```

In [95]:
```
d_sum.calculate()
```
Out[95]:

	Museum	Supermarket	Restaurant	all_categories	Museum_per_capita	Supermarket
spatial_index						
HYDE PARK	400.0	85.0	80.0	565.0	44.444444	
WASHINGTON PARK	100.0	0.0	0.0	100.0	NaN	
WOODLAWN	0.0	70.0	0.0	70.0	NaN	

In [45]:
```
d_sum.aggregated_results.head()
```
Out[45]:

	Museum	Supermarket	Restaurant	all_categories	Museum_per_capita	Supermarket
spatial_index						
HYDE PARK	400.0	85.0	80.0	565.0	0.017291	
WASHINGTON PARK	100.0	0.0	0.0	100.0	NaN	
WOODLAWN	0.0	70.0	0.0	70.0	NaN	

In [37]:
```
d_sum.aggregated_results.to_csv('./data/output_data/simple_demo_destsum.csv'
```

Coverage Metrics (Attributes of Destinations)

The metrics above were attributes of the origin points, i.e. they considered spatial access from the perspective of someone accessing amenities. In contrast, the following metrics are attributes of the destination, i.e. they consider spatial access from the perspective of the service provider. In addition to a capacity field, these metrics also require a population variable.

Coverage

Coverage adds two variables to the destination file: The number of people within the catchment area of a provider and a provider attribute divided by this nearby population count. E.g. you can use this to calculate the funding amount a service provider receives per people within the catchment area of the provider (such as 30 minutes walking time to the provider).

In [ ]:
```
cov = Coverage(network_type='walk',
               transit_matrix_filename='./data/output_data/matrices/simple_d
               sources_filename='./data/input_data/sources/hyde_park_tracts.
               destinations_filename='./data/input_data/destinations/hyde_pa
               source_column_names={'idx' : 'geoid10', 'population': 'Pop201
               dest_column_names={'idx': 'name', 'capacity': 'target', 'cate
              )
```

In [47]:
```
cov.calculate(upper_threshold=1800)
```

Out[47]:

	service_pop	percap_spending	category
Museum of Science and Industry	24861	0.016089	Museum
DuSable Museum	23134	0.004323	Museum
Whole Foods	31326	0.001596	Supermarket
Hyde Park Produce	31326	0.001117	Supermarket
Jewel Osco	16726	0.004185	Supermarket
Medici	31326	0.001596	Restaurant
Valois	31326	0.000958	Restaurant

In [16]:
```
cov.model_results.to_csv('./data/output_data/models/simple_demo_cov.csv')
```

Two-Stage Floating Catchment Area (TSFCA)

TSFCA Models are a type of gravity model popularized by Luo and Wang in 2003 to estimate spatial access gaps to primary care. They are calculated in two stages (using the primary care example): In a first stage, the ratio of doctors to the nearby population is calculated for every provider. In the 2nd stage, these ratios are summed for every point of origin (such as a tract centroid) within a travel threshold. In other words, the ratio of doctors to people is first calculated for the catchment areas of doctors (1st stage) and then summed for the catchment areas around a home or work location (2nd stage). The field names below are for a case that calculates per capita spending.

In [ ]:
```
tsfca = TSFCA(network_type='walk',
              transit_matrix_filename='./data/output_data/matrices/simple_de
              sources_filename='./data/input_data/sources/hyde_park_tracts.c
              destinations_filename='./data/input_data/destinations/hyde_par
              source_column_names={'idx' : 'geoid10', 'population': 'Pop2014
              dest_column_names={'idx': 'name', 'capacity': 'target', 'categ
             )
```

In [49]:
```
tsfca.calculate(upper_threshold=1800)
```

Out[49]:

	percap_spend_Museum	percap_spend_Supermarket	percap_spend_Restaurant	percap_
17031836300	0.004323	0.006899	0.002554	
17031836200	0.020412	0.006899	0.002554	
17031410100	0.016089	0.002713	0.002554	
17031410200	0.020412	0.002713	0.002554	
17031410500	0.020412	0.006899	0.002554	
17031410600	0.020412	0.006899	0.002554	
17031411100	0.020412	0.006899	0.002554	
17031410700	0.020412	0.002713	0.002554	
17031410800	0.020412	0.002713	0.002554	
17031410900	0.016089	0.002713	0.002554	
17031411000	0.016089	0.002713	0.002554	
17031411200	0.020412	0.006899	0.002554	

In [11]:
```
tsfca.model_results.to_csv('./data/output_data/models/simple_demo_tsfca.csv'
```

Appendix: Installation Setup

The package is written in Python 3.6, C++ 11 and Cython by Logan Noel (https://www.linkedin.com/in/lmnoel/). (Minimum Python version 3.5)
Currently, the only supported operating systems are MacOS and Ubuntu (if you don't have either, a guide for installing Ubuntu 16.04 LTS is in README.)

We recommend setting up a separate anaconda environment for this package to prevent version conflicts between dependencies of this and other packages.

Note: Experienced users can download installation requirements directly in the terminal.

For MacOS:

In [ ]:
```
# Install Python3
! brew install python3
```

In [ ]:
```
# Install homebrew
! /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
```

In [ ]:
```
# Install pip3
! brew install pip3
```

In [ ]:
```
# Install jupyter, jupyterlab, and jupyter hub
! brew install jupyter
! brew install jupyterlab
```

In [ ]:
```
# Clone the repository:
! git clone https://github.com/jupyterhub/jupyterhub
```

In [ ]:
```
# Install spatial index package
! brew install spatialindex
```

In [ ]:
```
# Install spatial access package
! pip3 install spatial_access
```

In [ ]:
```
# Install scipy package
! brew install scipy
```

In [ ]:
```
# Install geopy package
! pip install geopy
```

In [ ]:
```
# Install rtree package
! pip install rtree
```

In [ ]:

```
# Install geopandas package
! conda install geopandas
```

In [ ]:

```
# Run setup.py to install all the packages
! sudo python setup.py install
```

In [ ]:

```
# Install scipy package
! brew install btree
```

In Ubuntu add:

In [ ]:

```
! sudo apt-get install libspatialindex-dev
```

In [ ]:

```
! sudo apt-get install python-tk
```