03-COVID-19Acc-Reanalysis

December 15, 2023

0.1 ## Reproduction of Spatial Accessibility of COVID-19 Healthcare Resources in Illinois

Reproduction of: Rapidly measuring spatial accessibility of COVID-19 healthcare resources: a case study of Illinois, USA

Original study *by* Kang, J. Y., A. Michels, F. Lyu, Shaohua Wang, N. Agbodo, V. L. Freeman, and Shaowen Wang. 2020. Rapidly measuring spatial accessibility of COVID-19 healthcare resources: a case study of Illinois, USA. International Journal of Health Geographics 19 (1):1–17. DOI:10.1186/s12942-020-00229-x.

Reproduction Authors: Joe Holler, Derrick Burt, and Kufre Udoh With contributions from Peter Kedron, Drew An-Pham, Benjamin Cordola, Tate Sutter, Ola Zalecki, and Grayson Shanley Barr and the Spring 2021 Open Source GIScience class at Middlebury

Reproduction Materials Available at: github.com/HEGSRR/RPr-Kang-2020

Created: `2021-06-01` Revised: `2021-11-30`

0.1.1 Original Data

To perform the ESFCA method, three types of data are required, as follows: (1) road network, (2) population, and (3) hospital information. The road network can be obtained from the OpenStreetMap Python Library, called OSMNX. The population data is available on the American Community Survey. Lastly, hospital information is also publically available on the Homelanad Infrastructure Foundation-Level Data.

0.1.2 Modules

Import necessary libraries to run this model. See `environment.yml` for the library versions used for this analysis.

[2]:
```
# Import modules
import numpy as np
import pandas as pd
import geopandas as gpd
import networkx as nx
import osmnx as ox
import re
from shapely.geometry import Point, LineString, Polygon
import matplotlib.pyplot as plt
```

```
from tqdm import tqdm
import multiprocessing as mp
import folium
import itertools
import os
import time
import warnings
import IPython
import requests
from IPython.display import display, clear_output
from shapely.ops import nearest_points   #for hospital_setting function

warnings.filterwarnings("ignore")
print('\n'.join(f'{m.__name__}=={m.__version__}' for m in globals().values() if␣
 ↪getattr(m, '__version__', None)))
```

```
numpy==1.22.0
pandas==1.3.5
geopandas==0.10.2
networkx==2.6.3
osmnx==1.1.2
re==2.2.1
folium==0.12.1.post1
IPython==8.3.0
requests==2.27.1
```

0.2 Check Directories

Because we have restructured the repository for replication, we need to check our working directory and make necessary adjustments.

[3]:
```
# Check working directory
os.getcwd()
```

[3]: '/home/jovyan/work/RPr-Kang-2020/procedure/code'

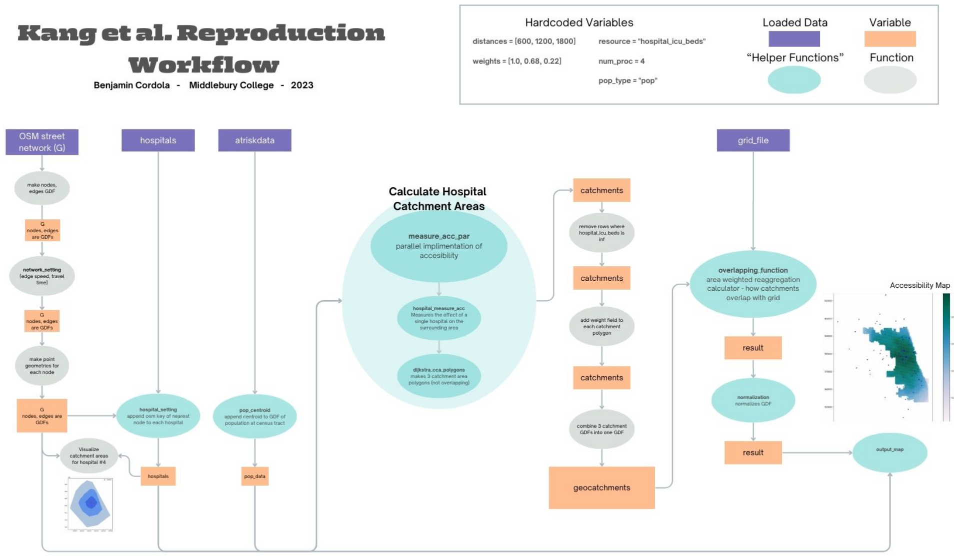
[4]:
```
# Use to set work directory properly
if os.path.basename(os.getcwd()) == 'code':
    os.chdir('../../')
os.getcwd()
```

[4]: '/home/jovyan/work/RPr-Kang-2020'

0.3 Display Workflow

This workflow explains the functions and all data manipulation done in the study. You can download a .pdf of the file in the main repository.

[39]:
```python
from PIL import Image

image = Image.open('./workflow.jpg')
image.show()
```



0.4 Load and Visualize Data

0.4.1 Population and COVID-19 Cases Data by County

'Cases' comes in as 'Unnamed_0'

If you would like to use the data generated from the pre-processing scripts, use the following code:

```python
covid_data = gpd.read_file('./data/raw/public/Pre-Processing/covid_pre-processed.shp')
atrisk_data = gpd.read_file('./data/raw/public/Pre-Processing/atrisk_pre-processed.shp')
```

[5]:
```python
# Read in at risk population data
atrisk_data = gpd.read_file('./data/raw/public/PopData/Illinois_Tract.shp')
atrisk_data.head()
```

[5]:
```
         GEOID STATEFP COUNTYFP TRACTCE                NAMELSAD   Pop  \
0  17091011700      17      091  011700       Census Tract 117  3688
1  17091011800      17      091  011800       Census Tract 118  2623
2  17119400951      17      119  400951   Census Tract 4009.51  5005
3  17119400952      17      119  400952   Census Tract 4009.52  3014
4  17135957500      17      135  957500      Census Tract 9575  2869

   Unnamed_ 0                                        NAME  OverFifty  \
```

```
0         588     Census Tract 117, Kankakee County, Illinois       1135
1         220     Census Tract 118, Kankakee County, Illinois        950
2        2285   Census Tract 4009.51, Madison County, Illinois       2481
3        2299   Census Tract 4009.52, Madison County, Illinois       1221
4        1026   Census Tract 9575, Montgomery County, Illinois       1171

   TotalPop                                          geometry
0      3688  POLYGON ((-87.88768 41.13594, -87.88764 41.136…
1      2623  POLYGON ((-87.89410 41.14388, -87.89400 41.143…
2      5005  POLYGON ((-90.11192 38.70281, -90.11128 38.703…
3      3014  POLYGON ((-90.09442 38.72031, -90.09360 38.720…
4      2869  POLYGON ((-89.70369 39.34803, -89.69928 39.348…
```

[6]:
```
# Read in covid case data - not using to simplify the study,
# but did not want to delete the path in case someone wants to bring this in␣
↪later.

# covid_data = gpd.read_file('./data/raw/public/PopData/Chicago_ZIPCODE.shp')
# covid_data['cases'] = covid_data['cases']
# covid_data.head()
```

0.4.2 Load Hospital Data

Note that 999 is treated as a "NULL"/"NA" so these hospitals are filtered out. This data contains the number of ICU beds and ventilators at each hospital.

[7]:
```
# Read in hospital data
hospitals = gpd.read_file('./data/raw/public/HospitalData/Chicago_Hospital_Info.
↪shp')
hospitals.head()
```

[7]:
```
   FID                                        Hospital      City  ZIP_Code  \
0    2                     Methodist Hospital of Chicago   Chicago     60640
1    4                    Advocate Christ Medical Center  Oak Lawn     60453
2   13                                Evanston Hospital  Evanston     60201
3   24  AMITA Health Adventist Medical Center Hinsdale  Hinsdale     60521
4   25                              Holy Cross Hospital   Chicago     60629

          X          Y  Total_Bed  Adult ICU  Total Vent  \
0 -87.671079  41.972800        145         36          12
1 -87.732483  41.720281        785        196          64
2 -87.683288  42.065393        354         89          29
3 -87.920116  41.805613        261         65          21
4 -87.690841  41.770001        264         66          21

                      geometry
0  MULTIPOINT (-87.67108 41.97280)
```

```
1  MULTIPOINT (-87.73248 41.72028)
2  MULTIPOINT (-87.68329 42.06539)
3  MULTIPOINT (-87.92012 41.80561)
4  MULTIPOINT (-87.69084 41.77000)
```

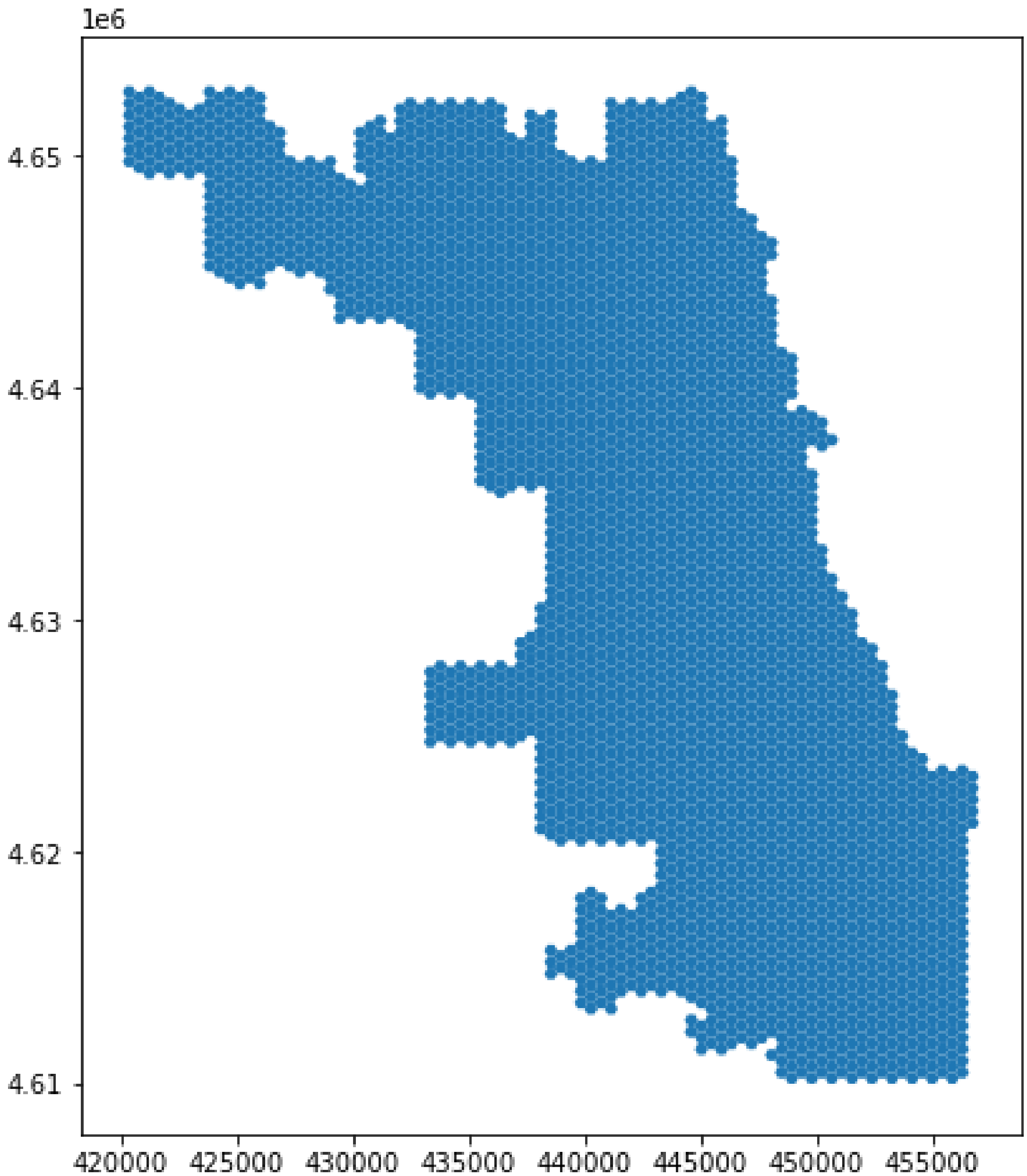
0.4.3 Generate and Plot Map of Hospitals

[ ]:
```
# Plot hospital data
m = folium.Map(location=[41.85, -87.65], tiles='cartodbpositron', zoom_start=10)
for i in range(0, len(hospitals)):
    folium.CircleMarker(
      location=[hospitals.iloc[i]['Y'], hospitals.iloc[i]['X']],
      popup="{}{}\n{}{}\n{}{}".format('Hospital Name: ',hospitals.
 ↪iloc[i]['Hospital'],
                                      'ICU Beds: ',hospitals.iloc[i]['Adult␣
 ↪ICU'],
                                      'Ventilators: ', hospitals.iloc[i]['Total␣
 ↪Vent']),
      radius=5,
      color='blue',
      fill=True,
      fill_opacity=0.6,
      legend_name = 'Hospitals'
    ).add_to(m)
legend_html =   '''<div style="position: fixed; width: 20%; heigh: auto;
                            bottom: 10px; left: 10px;
                            solid grey; z-index:9999; font-size:14px;
                            ">  Legend<br>'''


m
```

0.4.4 Load and Plot Hexagon Grids (500-meter resolution)

[8]:
```
# Read in and plot grid file for Chicago
grid_file = gpd.read_file('./data/raw/public/GridFile/Chicago_Grid.shp')
grid_file.plot(figsize=(8,8))
```

[8]: <AxesSubplot:>



0.4.5 Load the Road Network

If `Chicago_Network_Buffer.graphml` does not already exist, this cell will query the road network from OpenStreetMap.

Each of the road network code blocks may take a few mintues to run.

[9]:
```
%%time
```

```python
# To create a new graph from OpenStreetMap, delete or rename data/raw/private/
↪Chicago_Network_Buffer.graphml
# (if it exists), and set OSM to True
OSM = True

# if buffered street network is not saved, and OSM is preferred, # generate a␣
↪new graph from OpenStreetMap and save it
if not os.path.exists("./data/raw/private/Chicago_Network_Buffer.graphml") and␣
↪OSM:
    print("Loading buffered Chicago road network from OpenStreetMap. Please␣
↪wait... runtime may exceed 9min...", flush=True)
    G = ox.graph_from_place('Chicago', network_type='drive', buffer_dist=24140.
↪2)
    print("Saving Chicago road network to raw/private/Chicago_Network_Buffer.
↪graphml. Please wait...", flush=True)
    ox.save_graphml(G, './data/raw/private/Chicago_Network_Buffer.graphml')
    print("Data saved.")

# otherwise, if buffered street network is not saved, download graph from the␣
↪OSF project
elif not os.path.exists("./data/raw/private/Chicago_Network_Buffer.graphml"):
    print("Downloading buffered Chicago road network from OSF...", flush=True)
    url = 'https://osf.io/download/z8ery/'
    r = requests.get(url, allow_redirects=True)
    print("Saving buffered Chicago road network to file...", flush=True)
    open('./data/raw/private/Chicago_Network_Buffer.graphml', 'wb').write(r.
↪content)

# if the buffered street network is already saved, load it
if os.path.exists("./data/raw/private/Chicago_Network_Buffer.graphml"):
    print("Loading buffered Chicago road network from raw/private/
↪Chicago_Network_Buffer.graphml. Please wait...", flush=True)
    G = ox.load_graphml('./data/raw/private/Chicago_Network_Buffer.graphml')
    print("Data loaded.")
else:
    print("Error: could not load the road network from file.")
```

```
Loading buffered Chicago road network from
raw/private/Chicago_Network_Buffer.graphml. Please wait…
Data loaded.
CPU times: user 35.5 s, sys: 1.74 s, total: 37.2 s
Wall time: 37.2 s
```

0.4.6 Plot the Road Network

[ ]:
```
%%time
ox.plot_graph(G, node_size = 1, bgcolor = 'white', node_color = 'black',␣
↪edge_color = "#333333", node_alpha = 0.5, edge_linewidth = 0.5)
```

Check speed limit values Display all the unique speed limit values and count how many network edges (road segments) have each value. We will compare this to our cleaned network later.

[10]:
```
%%time
# Turn nodes and edges into geodataframes
nodes, edges = ox.graph_to_gdfs(G, nodes=True, edges=True)

# Get unique counts of road segments for each speed limit
print(edges['maxspeed'].value_counts())
print(str(len(edges)) + " edges in graph")
```

```
25 mph                          4793
30 mph                          3555
35 mph                          3364
40 mph                          2093
45 mph                          1418
20 mph                          1155
55 mph                           614
60 mph                           279
50 mph                           191
40                                79
15 mph                            76
70 mph                            71
65 mph                            54
10 mph                            38
[40 mph, 45 mph]                  27
[30 mph, 35 mph]                  26
45,30                             24
[40 mph, 35 mph]                  22
70                                21
25                                20
[55 mph, 45 mph]                  16
25, east                          14
[45 mph, 35 mph]                  13
[30 mph, 25 mph]                  10
[45 mph, 50 mph]                   8
50                                 8
[40 mph, 30 mph]                   7
[35 mph, 25 mph]                   6
[55 mph, 60 mph]                   5
20                                 4
[70 mph, 60 mph]                   3
```

```
[65 mph, 60 mph]                3
[40 mph, 45 mph, 35 mph]        3
[70 mph, 65 mph]                2
[70 mph, 45 mph, 5 mph]         2
[40, 45 mph]                    2
[35 mph, 50 mph]                2
35                              2
[55 mph, 65 mph]                2
[40 mph, 50 mph]                2
[15 mph, 25 mph]                2
[40 mph, 35 mph, 25 mph]        2
[15 mph, 40 mph, 30 mph]        2
[20 mph, 25 mph]                2
[30 mph, 25, east]              2
[65 mph, 55 mph]                2
[20 mph, 35 mph]                2
[55 mph, 55]                    2
55                              2
[15 mph, 30 mph]                2
[45 mph, 30 mph]                2
[15 mph, 45 mph]                2
[55 mph, 45, east, 50 mph]      2
[20 mph, 30 mph]                1
[5 mph, 45 mph, 35 mph]         1
[55 mph, 35 mph]                1
[5 mph, 35 mph]                 1
[55 mph, 50 mph]                1
Name: maxspeed, dtype: int64
384240 edges in graph
CPU times: user 34.2 s, sys: 179 ms, total: 34.4 s
Wall time: 34.4 s
```

0.4.7 network_setting function

Cleans the OSMNX network to work better with drive-time analysis.

Calculates edge speeds using osmx function. This is a smart function, and populates any missing speed limits with averages of other edges of the same road type, ex resedential or highway. Then, calculates edge travel times using those speeds.

Important! Travel time is output in seconds.

Args:

- network: OSMNX network for the spatial extent of interest

Returns:

- OSMNX network: cleaned OSMNX network for the spatial extent

[ ]:
```python
# view all highway types
print(edges['highway'].value_counts())
```

[12]:
```python
def network_setting(network):
    ox.speed.add_edge_speeds(network)
    ox.speed.add_edge_travel_times(network)

    print("Number of nodes: {}".format(network.number_of_nodes()))
    print("Number of edges: {}".format(network.number_of_edges()))
    return(network)
```

0.4.8 Preprocess the Network using network_setting

[13]:
```python
%%time
G = network_setting(G)
# Create point geometries for each node in the graph, to make constructing␣
 ↪catchment area polygons easier
for node, data in G.nodes(data=True):
    data['geometry']=Point(data['x'], data['y'])
```

```
Number of nodes: 142318
Number of edges: 384240
CPU times: user 41.7 s, sys: 194 ms, total: 41.9 s
Wall time: 41.9 s
```

Re-check speed limit values Display all the unique speed limit values and count how many network edges (road segments) have each value. Compare to the previous results.

[ ]:
```python
# Get unique counts for each road network
print(edges['maxspeed'].value_counts())
print(str(len(edges)) + " edges in graph")
```

0.5 "Helper" Functions

These functions are called when the model is run.

0.5.1 hospital_setting

Finds the nearest network node for each hospital.

Args:

- hospital: GeoDataFrame of hospitals
- G: OSMNX network

Returns:

- GeoDataFrame of hospitals with info on nearest network node

[16]:
```
def hospital_setting(hospitals, nodes):

    join = gpd.sjoin_nearest(hospitals, nodes, distance_col="distances")

    #rename column from osmid to nearest_osm, so that it works with other code
    join = join.rename(columns={"osmid": "nearest_osm"})

    ## Some reformatting to get the GDF to look like it did before ##
    # Drop columns
    columns_to_drop = ['index_right', 'x', 'y', 'highway', 'ref', 'distances']
    join = join[join.columns[~join.columns.isin(columns_to_drop)]]

    return(join)
```

0.5.2 pop_centroid

Converts geodata (population at census tract level) to centroids

Args:

- pop_data: a GeodataFrame
- pop_type: a string, either "pop" for at-risk population over 50 years old, or "covid" for COVID-19 case data

Returns:

- GeoDataFrame of centroids with population data. Three columns: code, pop, and geometry. Geometry is the centroid of each population.

[17]:
```
def pop_centroid (pop_data, pop_type):

    pop_data = pop_data.to_crs({'init': 'epsg:4326'})

    #Select at risk pop where population is greater than 0
    pop_data=pop_data[pop_data['OverFifty']>=0]

    # replace the geometry with its centroid
    pop_data["geometry"] =  pop_data["geometry"].centroid

    # rename columns
    pop_data = pop_data.rename(columns={"GEOID": "code", "OverFifty": "pop"})

    # keep only code, pop, and geometry columns
    pop_data = pop_data[["code", "pop", "geometry"]]

    return(pop_data)
```

0.5.3 djikstra_cca_polygons

Function written by Joe Holler + Derrick Burt. A more efficient way to calculate distance-weighted catchment areas for each hospital. First, create a dictionary (with a node and its corresponding drive time from the hospital) of all nodes within a 30 minute drive time (using networkx single_cource_dijkstra_path_length function). From here, two more dictionaries are constructed by querying the original one. From these dictionaries, single part convex hulls are created for each drive time interval and appended into a single list (one list with 3 polygon geometries). Within the list, the polygons are differenced from each other to produce three catchment areas.

Args: * G: cleaned network graph *with node point geometries attached* * nearest_osm: A unique nearest node ID calculated for a single hospital * distances: 3 distances (in drive time) to calculate catchment areas from * distance_unit: unit to calculate (time)

Returns: * A list of 3 differenced (not-overlapping) catchment area polygons (10 min poly, 20 min poly, 30 min poly)

[18]:
```
def dijkstra_cca_polygons(G, nearest_osm, distances, distance_unit =
 ↪"travel_time"):

    ## Distance_unit is given in seconds ##

    ## CREATE DICTIONARIES ##
    # create dictionary of nearest nodes
    nearest_nodes_30 = nx.single_source_dijkstra_path_length(G, nearest_osm,
 ↪distances[2], distance_unit) # creating the largest graph from which 10 and
 ↪20 minute drive times can be extracted from

    # extract values within 20 and 10 (respectively) minutes drive times
    nearest_nodes_20 = dict()
    nearest_nodes_10 = dict()
    for key, value in nearest_nodes_30.items():
        if value <= distances[1]:
            nearest_nodes_20[key] = value
        if value <= distances[0]:
            nearest_nodes_10[key] = value

    ## CREATE POLYGONS FOR 3 DISTANCE CATEGORIES (10 min, 20 min, 30 min) ##

    # 30 MIN
    # If the graph already has a geometry attribute with point data,
    # this line will create a GeoPandas GeoDataFrame from the nearest_nodes_30
 ↪dictionary
    points_30 = gpd.GeoDataFrame(gpd.GeoSeries(nx.get_node_attributes(G.
 ↪subgraph(nearest_nodes_30), 'geometry')))

    # This line converts the nearest_nodes_30 dictionary into a Pandas data
 ↪frame and joins it to points
```

```
    # left_index=True and right_index=True are options for merge() to join on␣
↪the index values
    points_30 = points_30.merge(pd.Series(nearest_nodes_30).to_frame(),␣
↪left_index=True, right_index=True)

    # Re-name the columns and set the geodataframe geometry to the geometry␣
↪column
    points_30 = points_30.rename(columns={'0_x':'geometry','0_y':'z'}).
↪set_geometry('geometry')

    # Create a convex hull polygon from the points
    polygon_30 = gpd.GeoDataFrame(gpd.GeoSeries(points_30.unary_union.
↪convex_hull))
    polygon_30 = polygon_30.rename(columns={0:'geometry'}).
↪set_geometry('geometry')

    # 20 MIN # 1200 seconds!
    # Select nodes less than or equal to 20
    points_20 = points_30.query("z <= 1200")

    # Create a convex hull polygon from the points
    polygon_20 = gpd.GeoDataFrame(gpd.GeoSeries(points_20.unary_union.
↪convex_hull))
    polygon_20 = polygon_20.rename(columns={0:'geometry'}).
↪set_geometry('geometry')

    # 10 MIN # 600 seconds!
    # Select nodes less than or equal to 10
    points_10 = points_30.query("z <= 600")

    # Create a convex hull polygon from the points
    polygon_10 = gpd.GeoDataFrame(gpd.GeoSeries(points_10.unary_union.
↪convex_hull))
    polygon_10 = polygon_10.rename(columns={0:'geometry'}).
↪set_geometry('geometry')

    # Create empty list and append polygons
    polygons = []

    # Append
    polygons.append(polygon_10)
    polygons.append(polygon_20)
    polygons.append(polygon_30)

    # Clip the overlapping distance ploygons (create two donuts + hole)
    for i in reversed(range(1, len(distances))):
```

```
        polygons[i] = gpd.overlay(polygons[i], polygons[i-1], how="difference")

    return polygons
```

0.5.4 hospital_measure_acc (adjusted to incorporate dijkstra_cca_polygons)

Measures the effect of a single hospital on the surrounding area. (Uses `dijkstra_cca_polygons`)

Args:

- _thread_id: int used to keep track of which thread this is
- hospital: Geopandas dataframe with information on a hospital
- pop_data: Geopandas dataframe with population data
- distances: Distances in time to calculate accessibility for
- weights: how to weight the different travel distances

Returns:

- Tuple containing:
  - Int (_thread_id)
  - GeoDataFrame of catchment areas with key stats

[19]:
```
def hospital_measure_acc (_thread_id, hospital, pop_data, distances, weights):

    # Create polygons
    polygons = dijkstra_cca_polygons(G, hospital['nearest_osm'], distances)

    # iterate over pop_data and check if each point is within a polygon
    # if so, multiply the pop and weight for that polygon and appends it to␣
↪num_pops.
    num_pops = []
    for j in pop_data.index:
        point = pop_data['geometry'][j]
        # Multiply polygons by weights
        for k in range(len(polygons)):
            if len(polygons[k]) > 0: # To exclude the weirdo (convex hull is␣
↪not polygon)
                if (point.within(polygons[k].iloc[0]["geometry"])):
                    num_pops.append(pop_data['pop'][j]*weights[k])

    # sum all the weighted populations
    total_pop = sum(num_pops)

    # update polygons with time, total population, and ICU beds. Set CRS to␣
↪4326, then convert to 32616
    for i in range(len(distances)):
        polygons[i]['time']=distances[i]
        polygons[i]['total_pop']=total_pop
```

```
        polygons[i]['hospital_icu_beds'] = float(hospital['Adult ICU'])/
↪polygons[i]['total_pop'] # proportion of # of beds over pops in 10 mins
        polygons[i].crs = { 'init' : 'epsg:4326'}
        polygons[i] = polygons[i].to_crs({'init':'epsg:32616'})

    # print the thread ID
    print('{:.0f}'.format(_thread_id), end=" ", flush=True)

    # return a tuple containing the thread ID and a list of copied polygons
    return(_thread_id, [ polygon.copy(deep=True) for polygon in polygons ])
```

0.5.5 measure_acc_par

Parallel implementation of accessibility measurement.

Args:

- hospitals: Geodataframe of hospitals
- pop_data: Geodataframe containing population data
- network: OSMNX street network
- distances: list of distances to calculate catchments for
- weights: list of floats to apply to different catchments
- num_proc: number of processors to use.

Returns:

- Geodataframe of catchments with accessibility statistics calculated

[20]:
```
def measure_acc_par (hospitals, pop_data, network, distances, weights, num_proc␣
↪= 4):

    # initialize catchment list, 3 empty geodataframes
    catchments = []
    for distance in distances:
        catchments.append(gpd.GeoDataFrame())

    # pool = mp.Pool(processes = num_proc)

    # makes a list of all hospital info.  len = 66
    # looks like this, except with all info, and for all 66 hospitals
    # [[2, Methodist Hospital of Chicago, Chicago], [4, Advocate Christ Medical␣
↪Center, Oak Lawn]]
    hospital_list = [ hospitals.iloc[i] for i in range(len(hospitals)) ]

    print("Calculating", len(hospital_list), "hospital catchments...\ncompleted␣
↪number:", end=" ")

    # call hospital_acc_unpacker
    # returns a tuple containing the thread ID and a list of copied polygons
```

```
    #results = pool.map(hospital_acc_unpacker, zip(range(len(hospital_list)),␣
↪hospital_list, itertools.repeat(pop_data), itertools.repeat(distances),␣
↪itertools.repeat(weights)))

    results = []
    for i in range(len(hospital_list)): #do from 1 to 66
        result = hospital_measure_acc(i, hospital_list[i], pop_data, distances,␣
↪weights)
        results.append(result)

    # pool.close()

    # sort and extract the results
    results.sort()
    results = [ r[1] for r in results ]

    # combine catchment results into the respective GeoDataFrames in the␣
↪catchments list
    for i in range(len(results)):
        for j in range(len(distances)):
            catchments[j] = catchments[j].append(results[i][j], sort=False)

    return catchments
```

0.5.6 overlapping_function

Calculates how all catchment areas overlap with and affect the accessibility of each grid in our grid file.

Args:

- grid_file: GeoDataFrame of our grid
- catchments: GeoDataFrame of our catchments
- service_type: the kind of care being provided (ICU beds vs. ventilators)
- weights: the weight to apply to each service type
- num_proc: the number of processors

Returns:

- Geodataframe - grid_file with calculated stats

[21]:
```
def overlapping_function (grid_file, catchments, service_type, weights,␣
↪num_proc = 4):

    ## Area Weighted Reaggregation
        # set weighted to False for original 50% threshold method
        # switch to True for area-weighted overlay
    weighted = True

```

```python
    # if the value to be calculated is already in the hegaxon grid, delete it
    # otherwise, the field name gets a suffix _1 in the overlay step
    if resource in list(grid_file.columns.values):
        grid_file = grid_file.drop(resource, axis = 1)

    # calculate hexagon 'target' areas
    grid_file['area'] = grid_file.area

    # Intersection overlay of hospital catchments and hexagon grid
    print("Intersecting hospital catchments with hexagon grid...")
    fragments = gpd.overlay(grid_file, geocatchments, how='intersection')

    # Calculate percent coverage of the hexagon by the hospital catchment as
    # fragment area / target(hexagon) area
    fragments['percent'] = fragments.area / fragments['area']

    # if using weighted aggregation...
    if weighted:
        print("Calculating area-weighted value...")
        # multiply the service/population ratio by the distance weight and the␣
↪percent coverage
        fragments['value'] = fragments[resource] * fragments['weight'] *␣
↪fragments['percent']

    # if using the 50% coverage rule for unweighted aggregation...
    else:
        print("Calculating value for hexagons with >=50% overlap...")
        # filter for only the fragments with > 50% coverage by hospital␣
↪catchment
        fragments = fragments[fragments['percent']>=0.5]
        # multiply the service/population ration by the distance weight
        fragments['value'] = fragments[resource] * fragments['weight']

    # select just the hexagon id and value from the fragments,
    # group the fragments by the (hexagon) id,
    # and sum the values
    print("Summarizing results by hexagon id...")
    sum_results = fragments[['id', 'value']].groupby(by = ['id']).sum()

    # join the results to the hexagon grid_file based on hexagon id
    print("Joining results to hexagons...")
    results = pd.merge(grid_file, sum_results, how="left", on = "id")

    # rename value column name to the resource name
    return(results.rename(columns = {'value' : resource}))
```

0.5.7 normalization

Normalizes our result (Geodataframe).

[22]:
```
def normalization (result, resource):
    result[resource]=(result[resource]-min(result[resource]))/
↪(max(result[resource])-min(result[resource]))
    return result
```

0.5.8 file_import

Imports all files we need to run our code and pulls the Illinois network from OSMNX if it is not present (will take a while).

NOTE: even if we calculate accessibility for just Chicago, we want to use the Illinois network (or at least we should not use the Chicago network) because using the Chicago network will result in hospitals near but outside of Chicago having an infinite distance (unreachable because roads do not extend past Chicago).

Args:

- pop_type: population type, either "pop" for general population or "covid" for COVID-19 cases
- region: the region to use for our hospital and grid file ("Chicago" or "Illinois")

Returns:

- G: OSMNX network
- hospitals: Geodataframe of hospitals
- grid_file: Geodataframe of grids
- pop_data: Geodataframe of population

[23]:
```
def output_map(output_grid, base_map, hospitals, resource):
    ax=output_grid.plot(column=resource, cmap='PuBuGn',figsize=(18,12),␣
↪legend=True, zorder=1)
    # Next two lines set bounds for our x- and y-axes because it looks like␣
↪there's a weird
    # Point at the bottom left of the map that's messing up our frame (Maja)
    ax.set_xlim([314000, 370000])
    ax.set_ylim([540000, 616000])
    base_map.plot(ax=ax, facecolor="none", edgecolor='gray', lw=0.1)
    hospitals.plot(ax=ax, markersize=10, zorder=1, c='blue')
```

0.6 Run the model

Below you can customize the input of the model:

- Processor - the number of processors to use
- Population - the population to calculate the measure for
- Resource - the hospital resource of interest
- Hospital - all hospitals or subset to check code

0.6.1 Process population data

[24]:
```
'''
To simplify the reanalysis, in variables I will hardcode the use of
    4 processors
    Population: Population at Risk
    Resource: ICU Beds
    Hospital: All hospitals
'''

resource = "hospital_icu_beds"
num_proc = 4
pop_type = "pop"

## Create centroids for atrisk population at the census tract level
pop_data = pop_centroid(atrisk_data, pop_type)

distances = [600, 1200, 1800] # Distances in travel time (seconds!)
weights = [1.0, 0.68, 0.22] # Weights where weights[0] is applied to␣
 ↪distances[0]
```

[25]:
```
pop_data
```

[25]:
```
             code   pop                     geometry
0     17091011700  1135  POINT (-87.87355 41.12949)
1     17091011800   950  POINT (-87.87646 41.13978)
2     17119400951  2481  POINT (-90.09829 38.72763)
3     17119400952  1221  POINT (-90.08180 38.72984)
4     17135957500  1171  POINT (-89.60390 39.38915)
…             …     …                          …
3116  17037000100  2331  POINT (-88.65253 42.10661)
3117  17037001500  1360  POINT (-88.73721 41.88417)
3118  17037000400  2698  POINT (-88.68023 42.02216)
3119  17037000300  1020  POINT (-88.86924 41.96281)
3120  17037000200  1739  POINT (-88.82573 42.11145)

[3121 rows x 3 columns]
```

0.6.2 Process hospital data

[26]:
```
hospitals
```

[26]:
```
   FID                                       Hospital        City  \
0    2                   Methodist Hospital of Chicago     Chicago
1    4                   Advocate Christ Medical Center   Oak Lawn
2   13                               Evanston Hospital    Evanston
3   24  AMITA Health Adventist Medical Center Hinsdale    Hinsdale
```

```
4    25                          Holy Cross Hospital      Chicago
..   …                                             …            …
61  202               Presence Saint Elizabeth Hospital      Chicago
62  203             Presence Holy Family Medical Center  Des Plaines
63  204                     Resurrection Medical Center      Chicago
64  206                         Shirley Ryan AbilityLab      Chicago
65  211                                MacNeal Hospital       Berwyn

    ZIP_Code          X          Y  Total_Bed  Adult ICU  Total Vent  \
0      60640 -87.671079  41.972800        145         36          12
1      60453 -87.732483  41.720281        785        196          64
2      60201 -87.683288  42.065393        354         89          29
3      60521 -87.920116  41.805613        261         65          21
4      60629 -87.690841  41.770001        264         66          21
..       …         …          …        …          …           …
61     60622 -87.685883  41.907521        108         27           9
62     60016 -87.869807  42.055750        178         45          14
63     60631 -87.813134  41.988756        337         84          27
64     60611 -87.618897  41.894197        242         61          20
65     60402 -87.792752  41.832261        374         94          30

                          geometry
0   MULTIPOINT (-87.67108 41.97280)
1   MULTIPOINT (-87.73248 41.72028)
2   MULTIPOINT (-87.68329 42.06539)
3   MULTIPOINT (-87.92012 41.80561)
4   MULTIPOINT (-87.69084 41.77000)
..                              …
61  MULTIPOINT (-87.68588 41.90752)
62  MULTIPOINT (-87.86981 42.05575)
63  MULTIPOINT (-87.81313 41.98876)
64  MULTIPOINT (-87.61890 41.89420)
65  MULTIPOINT (-87.79275 41.83226)

[66 rows x 10 columns]
```

[27]:
```
#Finds the nearest network node for each hospital
hospitals = hospital_setting(hospitals, nodes)
```

[28]:
```
hospitals
```

[28]:
```
    FID                                   Hospital         City  \
0     2                 Methodist Hospital of Chicago      Chicago
1     4                  Advocate Christ Medical Center     Oak Lawn
2    13                              Evanston Hospital     Evanston
3    24  AMITA Health Adventist Medical Center Hinsdale     Hinsdale
4    25                            Holy Cross Hospital      Chicago
```

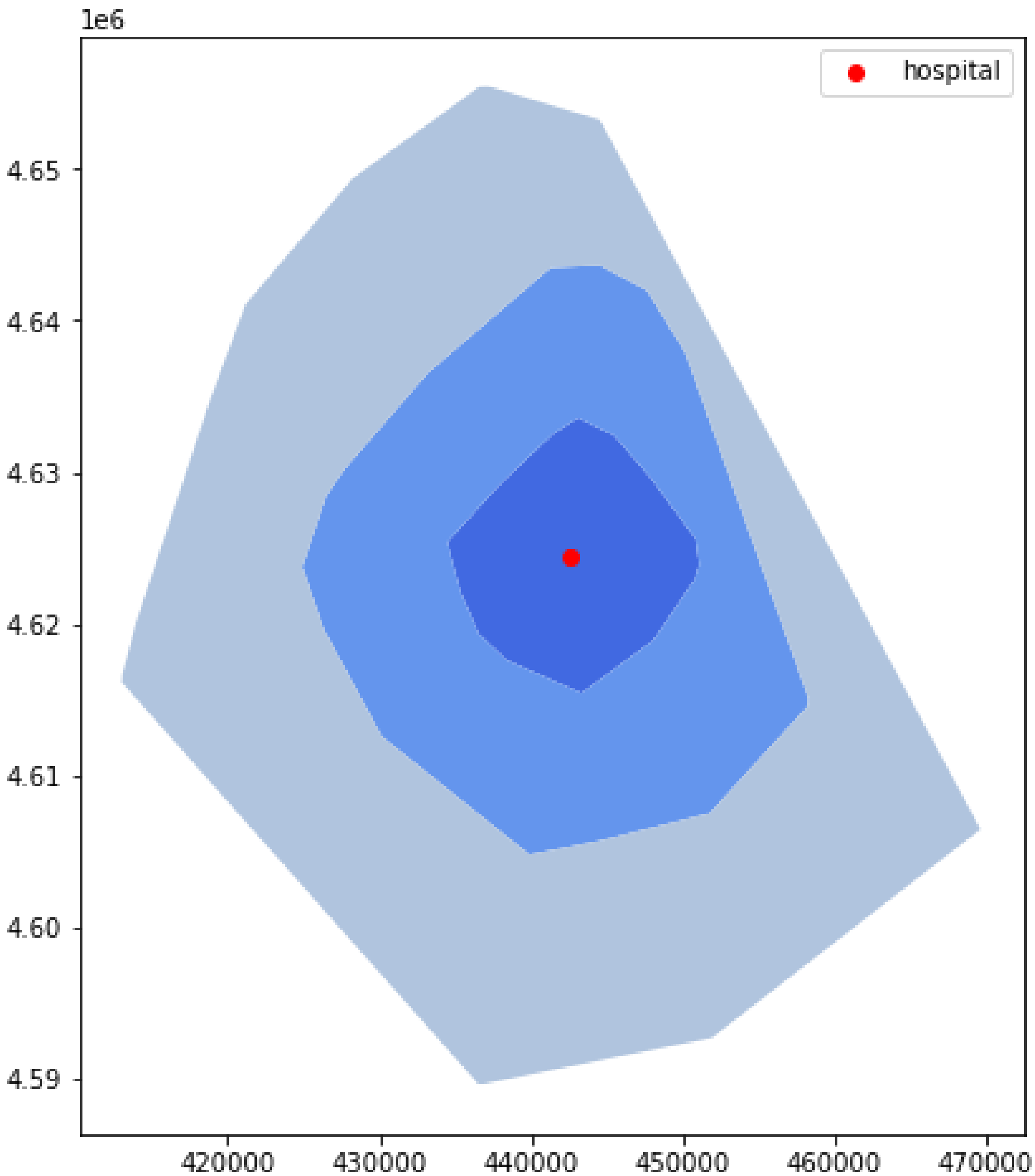
```
..  …                                            …         …
61  202             Presence Saint Elizabeth Hospital      Chicago
62  203           Presence Holy Family Medical Center  Des Plaines
63  204                   Resurrection Medical Center      Chicago
64  206                       Shirley Ryan AbilityLab      Chicago
65  211                              MacNeal Hospital       Berwyn

    ZIP_Code          X          Y  Total_Bed  Adult ICU  Total Vent  \
0      60640 -87.671079  41.972800        145         36          12
1      60453 -87.732483  41.720281        785        196          64
2      60201 -87.683288  42.065393        354         89          29
3      60521 -87.920116  41.805613        261         65          21
4      60629 -87.690841  41.770001        264         66          21
..       …          …          …        …          …           …
61     60622 -87.685883  41.907521        108         27           9
62     60016 -87.869807  42.055750        178         45          14
63     60631 -87.813134  41.988756        337         84          27
64     60611 -87.618897  41.894197        242         61          20
65     60402 -87.792752  41.832261        374         94          30

                       geometry  nearest_osm
0   MULTIPOINT (-87.67108 41.97280)   257157489
1   MULTIPOINT (-87.73248 41.72028)   261189594
2   MULTIPOINT (-87.68329 42.06539)  1842027877
3   MULTIPOINT (-87.92012 41.80561)   237694440
4   MULTIPOINT (-87.69084 41.77000)   261122131
..                              …           …
61  MULTIPOINT (-87.68588 41.90752)   261129958
62  MULTIPOINT (-87.86981 42.05575)  2394200372
63  MULTIPOINT (-87.81313 41.98876)  1343383340
64  MULTIPOINT (-87.61890 41.89420)   261151125
65  MULTIPOINT (-87.79275 41.83226)   261196704

[66 rows x 11 columns]
```

0.6.3 Visualize catchment areas for hospital #4

[29]:
```
# Create point geometries for entire graph

# which hospital to visualize?
fighosp = 4

# Create catchment for hospital 4
poly = dijkstra_cca_polygons(G, hospitals['nearest_osm'][fighosp], distances)

# Reproject polygons
for i in range(len(poly)):
```

```
    poly[i].crs = { 'init' : 'epsg:4326'}
    poly[i] = poly[i].to_crs({'init':'epsg:32616'})

# Reproject hospitals
hospital_subset = hospitals.iloc[[fighosp]].to_crs(epsg=32616)

fig, ax = plt.subplots(figsize=(12,8))

min_10 = poly[0].plot(ax=ax, color="royalblue", label="10 min drive")
min_20 = poly[1].plot(ax=ax, color="cornflowerblue", label="20 min drive")
min_30 = poly[2].plot(ax=ax, color="lightsteelblue", label="30 min drive")

hospital_subset.plot(ax=ax, color="red", legend=True, label = "hospital")

# Add legend
ax.legend()
```

```
[29]: <matplotlib.legend.Legend at 0x7fca2a8ba220>
```



0.6.4 Calculate hospital catchment areas

[31]:
```
%%time

catchments = measure_acc_par(hospitals, pop_data, G, distances, weights,␣
    ↪num_proc)
```

```
Calculating 66 hospital catchments…
completed number: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

```
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 CPU times: user 6min 2s, sys: 1.21
s, total: 6min 3s
Wall time: 6min 3s
```

0.6.5 Calculate accessibility

0.6.6 Post-process the catchments (for area weighted reaggregation)

[32]:
```python
# add weight field to each catchment polygon
for i in range(len(weights)):
    catchments[i]['weight'] = weights[i]
# combine the three sets of catchment polygons into one geodataframe
geocatchments = pd.concat([catchments[0], catchments[1], catchments[2]])
geocatchments
```

[32]:
```
                                             geometry  time   total_pop  \
0  POLYGON ((446359.955 4637144.048, 444654.345 4…   600   789023.74
0  POLYGON ((438353.601 4609853.779, 432065.727 4…   600   718489.92
0  POLYGON ((442878.135 4648745.067, 441056.875 4…   600   469346.52
0  POLYGON ((423900.989 4621140.151, 421031.920 4…   600   735110.64
0  POLYGON ((443322.063 4615428.578, 438387.446 4…   600   716375.12
..                                                …     …           …
0  POLYGON ((439884.526 4604782.264, 415910.447 4…  1800  1018558.48
0  MULTIPOLYGON (((418680.569 4620247.323, 411754…  1800   757050.08
0  POLYGON ((421589.871 4617483.974, 415910.447 4…  1800   975802.04
0  POLYGON ((415910.447 4618609.875, 410587.177 4…  1800   940777.78
0  POLYGON ((428248.191 4600502.152, 416051.040 4…  1800   824398.14

   hospital_icu_beds  weight
0           0.000046    1.00
0           0.000273    1.00
0           0.000190    1.00
0           0.000088    1.00
0           0.000092    1.00
..               …       …
0           0.000027    0.22
0           0.000059    0.22
0           0.000086    0.22
0           0.000065    0.22
0           0.000114    0.22

[198 rows x 5 columns]
```

0.6.7 Area Weighted Reaagregation

[33]:
```
%%time
result = overlapping_function(grid_file, catchments, resource, weights,␣
 ↪num_proc)
```

```
Intersecting hospital catchments with hexagon grid…
Calculating area-weighted value…
Summarizing results by hexagon id…
Joining results to hexagons…
CPU times: user 13 s, sys: 66 ms, total: 13.1 s
Wall time: 13.1 s
```

[34]:
```
%%time
result = normalization (result, resource)
```

```
CPU times: user 3.97 ms, sys: 7 µs, total: 3.97 ms
Wall time: 3.65 ms
```

0.7 Results & Discussion

Extensive cleaning of unneccesary variables and lines of code that were never called.

0.7.1 Making code more efficient and easier to read with GeoPandas

1. Made the **pop_centroid** function much faster - previously took 3:30 to run, now less than a second. Instead of creating an empty GDF and iterating over all of the population geometries, adding data to this new GDF, I just used the native GeoPandas .centroid method, replacing the population geometries with centroids, and then dropping other unnecessary columns from atrisk_data.
2. Rewrote the **hospital_setting** function to find each hospital's nearest node using GeoPandas nearest join method. What took 1:20 to run now runs in less than a second. I also cleaned the GDF so that it matched what we were working with before.

0.7.2 Removed parallel processing from two functions.

1. **overlapping_function**
2. **measure_acc_par**

0.7.3 Theoretical Changes to the methodology

Area weighted reaggregation - assigned speeds to the road network using osnmx.

0.7.4 Simplifying Code for future students

My greatest contribution to this replication has been the simplification of code and adding documentation to functions. This has made the code much easier for future students to read through and understand, and has not sacrificed processing times. I also made a visual workflow, visualizing the replication study from start to finish, including all data and functions used to manipulate them.

Simplifications include:

I removed the dropdown menu that allows you to choose between population groups and hospital data. The benefits of this dropdown options were minimal, and it just made the code more confusing to follow and modify. In the form of a dropdown selection, it prevents the study from being one script, and introduces potential error as groups try to replicate eachother, if they are not clear about which choices they made with their mouse in the dropdown.

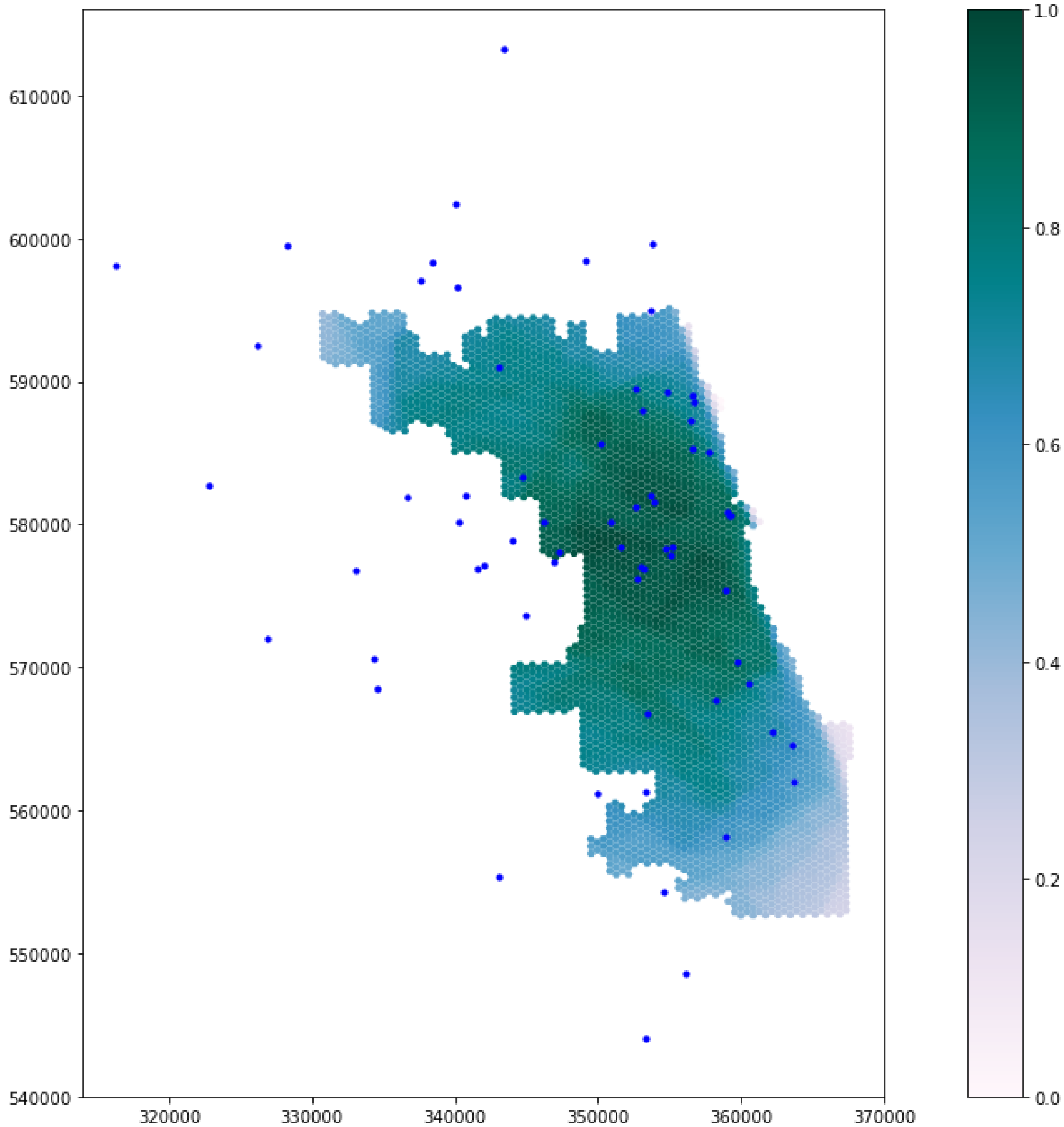
I was able to delete the function **overlap_calc**, after implementing its function into **overlapping_function** which was implements the area weighted reaggregation.

I removed a code block that filtered rows where the "hospital_icu_beds" value is infinity, which did not do anything.

0.7.5 Accessibility Map

[35]:
```
%%time
hospitals = hospitals.to_crs({'init': 'epsg:26971'})
result = result.to_crs({'init': 'epsg:26971'})
output_map(result, pop_data, hospitals, resource)
```

```
CPU times: user 1.48 s, sys: 168 ms, total: 1.65 s
Wall time: 1.45 s
```



Classified Accessibility Outputs

0.7.6 Conclusion

Reproduction confirms the original studies results, while highlighting some limitations of the data and theoretical methods. In this reanalysis, we populated in missing speed limit data and used an area weighted reaggregation to assign weights to catchments. Code was extensivily cleaned and simplified, both making the code faster to run but also simpler to read. Finally, the use of GeoPandas more efficiently transforms our spatial datasets.

It is hard to say how much quantifiable change our theoretical adjustments contributed to the code, as the final output map looks very similar to the resulting figure from the original study. However, handling of data as GeoPandas instead of dataframes in two functions reduced processing time by 5 minutes combined. Most notably, the code is much more clearly commented and simpler to understand. There is no morre parallel processing, which was more unnecessarily complicated than it was helpful, and there is no dropdown options for toggling between data sources. As the code is now, students and replicators will be able to spend more time critiquing the methodology and workflow, rather than getting lost in the syntax or confused by unnecessary functions.

0.7.7 References

Luo, W., & Qi, Y. (2009). An enhanced two-step floating catchment area (E2SFCA) method for measuring spatial accessibility to primary care physicians. Health & place, 15(4), 1100-1107.