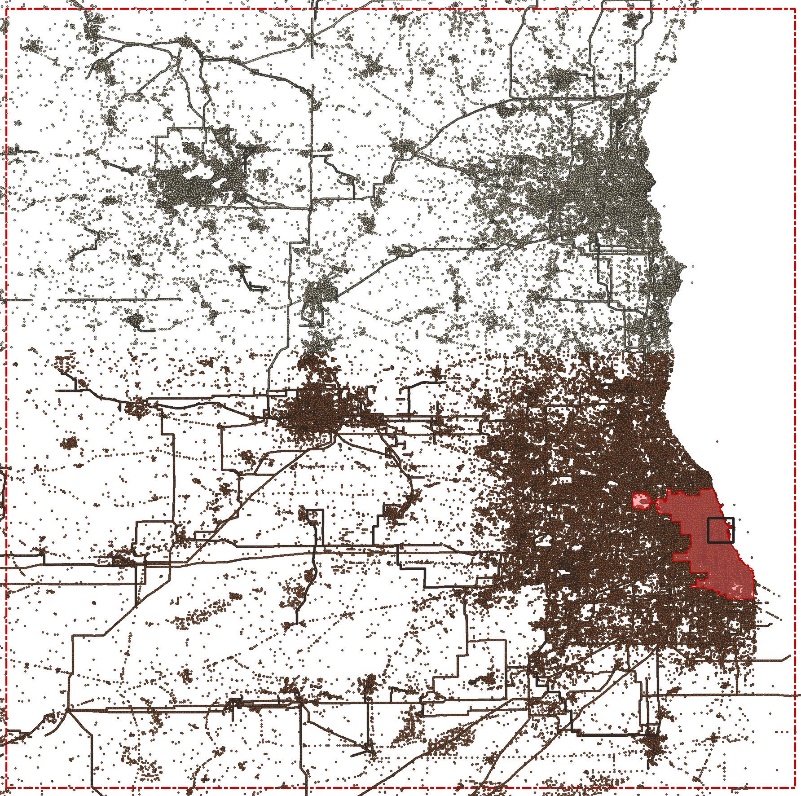
## 2.1.2 OSM数据与核密度估计

### 2.1.2.1 OpenStreetMap（OSM）数据处理

[OpenStreetMap（OSM）](https://www.openstreetmap.org/#map=13/41.8679/-87.6569)①。“欢迎访问OpenStreetMap，这是个为全世界创建和分发免费地理数据的项目。我们之所以这么做，是因为人们认为作为免费的地图，在使用方面实际上有法律或者技术上的限制，阻碍了人们以创作性的、多产的、或意想不到的方式使用它们”。OSM提供了世界各地的道路、小径、咖啡馆、火车站等诸多地理信息数据，是用于城市空间方面研究的宝贵数据财富。OSM数据的下载通过查看官网信息，一般来讲有两个途径：一个是直接下载显示窗口范围的数据，或输入坐标自定义范围。但这种方式下载的数据量有限制，如果范围过大则无法下载；另一种方式是直接从资源库下载，不同资源下载的方式也有所不同，需要根据研究目的来确定。同时OSM提供了多年历史数据，这为城市空间变化的研究提供了数据支持。本文使用的数据从[Geofabrik](https://download.geofabrik.de/north-america/us.html)②库中下载，因为分析的目标区域为芝加哥城，下载了[llinois-latest-free.shp.zip](https://download.geofabrik.de/north-america/us/illinois.html)③压缩文件340MB，解压后3.73GB。根据下载后点数据的分布情况，为了保持点连续的区域，增加下载[wisconsin-latest.osm.bz2](https://download.geofabrik.de/north-america/us/wisconsin.html)④数据文件324MB，解压后3.73GB。可以使用QGIS初步查看数据。包含的数据层有：lines，multilinestrings，multipolygons，other\_relations 和points。

下图同时打开了了Ilinois和wisconsin的点数据，内部红色半透明区域为[芝加哥城行政范围](https://data.cityofchicago.org/Facilities-Geographic-Boundaries/Boundaries-City/ewy2-6yfk)⑤，数据来源于[Chicago Data Portal](https://data.cityofchicago.org/)⑥。外部红色虚线为实验数据确定的初步边界。最小的黑色矩形是用于代码调试提取的小规模数据范围。



#### 1）OSM原始数据处理

OSM数据处理包括合并两个区域数据，及裁切，或者先分别裁切再合并。可以根据电脑内存需求和处理速度确定前后顺序。裁切OSM数据。这里选用的裁切方法是使用[osmosis](https://wiki.openstreetmap.org/wiki/Osmosis)⑦命令行工具，非常适合处理大数据文件，裁切和更新数据。同时可以参考[Manipulating Data with Osmosis](https://learnosm.org/en/osm-data/osmosis/)⑧。查询osmosis给出的案例，寻找应用Polygon提取数据的代码为：osmosis --read-xml file="planet-latest.osm" --bounding-polygon file="country2pts.txt" --write-xml file="germany.osm"，涉及到三个参数，原始OSM数据；裁切边界Polygon（TXT数据格式）。该处的Polygon为osmosis格式的Polygon格式数据，需要编写转换代码；及输出路径。

目视粗略判断点集聚的范围，在QGIS中绘制常规的Polygon边界，如上图外红色虚线。首先编写Polygon到osmosis格式的Polygon代码，查询其数据格式为：

australia\_v  
first\_area  
 0.1446693E+03 -0.3826255E+02  
 0.1446627E+03 -0.3825661E+02  
 0.1446763E+03 -0.3824465E+02  
 0.1446813E+03 -0.3824343E+02  
 0.1446824E+03 -0.3824484E+02  
 0.1446826E+03 -0.3825356E+02  
 0.1446876E+03 -0.3825210E+02  
 0.1446919E+03 -0.3824719E+02  
 0.1447006E+03 -0.3824723E+02  
 0.1447042E+03 -0.3825078E+02  
 0.1446758E+03 -0.3826229E+02  
 0.1446693E+03 -0.3826255E+02  
END  
second\_area  
 0.1422436E+03 -0.3839315E+02  
 0.1422496E+03 -0.3839070E+02  
 0.1422543E+03 -0.3839025E+02  
 0.1422574E+03 -0.3839155E+02  
 0.1422467E+03 -0.3840065E+02  
 0.1422433E+03 -0.3840048E+02  
 0.1422420E+03 -0.3839857E+02  
 0.1422436E+03 -0.3839315E+02  
END  
END

定义转换格式函数shpPolygon2OsmosisTxt()，调用了osgeo类，该类包含于[GDAL](https://pypi.org/project/GDAL/)⑨库中。GDAL是一个用于栅格（raster）和矢量（vector）地理空间数据格式的开源转换库，包含大量格式驱动，大多数Python的地理空间数据处理库通常基于GDAL编写，是最为基础的库。GeoPandas库基于Fiona库，方便用户对地理空间数据的处理，而Fiona包含链接到GDAL的扩展模块。通常在使用Python地理空间数据库时，并没有使用哪个库的限制，目前最常用处理地理空间数据的库是GeoPandas，但是有时这些库不能满足所有要求，因此需要调用GDAL来处理。查看GDAL帮助信息，可以浏览[GDAL/OGR Cookbook!](http://pcjericks.github.io/py-gdalogr-cookbook/index.html)⑩，及[GDAL documentation](https://gdal.org/)⑪。

def shpPolygon2OsmosisTxt(shape\_polygon\_fp,osmosis\_txt\_fp):  
 '''  
 function - 转换shape的polygon为osmium的polygon数据格式（.txt），用于.osm地图数据的裁切  
   
 Params:  
 shape\_polygon\_fp - 输入shape地理数据格式的polygon文件路径；string  
 osmosis\_txt\_fp - 输出为osmosis格式的polygon数据格式.txt文件路径；string  
   
 Returns:  
 None  
 '''  
 from osgeo import ogr # osgeo包含在GDAL库中  
   
 driver=ogr.GetDriverByName('ESRI Shapefile') # GDAL能够处理众多地理数据格式，此时调入了ESRI Shapefile数据格式驱动  
 infile=driver.Open(shape\_polygon\_fp) # 打开.shp文件  
 layer=infile.GetLayer() # 读取层  
 f=open(osmosis\_txt\_fp,"w")   
 f.write("osmosis polygon\nfirst\_area\n")  
   
 for feature in layer:   
 feature\_shape\_polygon=feature.GetGeometryRef()   
 print(feature\_shape\_polygon) # 为polygon  
 firsts\_area\_linearring=feature\_shape\_polygon.GetGeometryRef(0) # polygon不包含嵌套，为单独的形状  
 print(firsts\_area\_linearring) # 为linearRing  
 area\_vertices=firsts\_area\_linearring.GetPointCount() # 提取linearRing对象的点数量  
 for vertex in range(area\_vertices): # 循环点，并向文件中写入点坐标  
 lon, lat, z=firsts\_area\_linearring.GetPoint(vertex)   
 f.write("%s %s\n"%(lon,lat))  
 f.write("END\nEND")   
 f.close()   
   
# 转换初步实验边界  
shape\_polygon\_fp='./data/OSMBoundary/OSMBoundary.shp'  
osmosis\_txt\_fp='./data/OSMBoundary.txt'  
shpPolygon2OsmosisTxt(shape\_polygon\_fp,osmosis\_txt\_fp)  
  
# 转换代码调试小批量数据边界  
shape\_polygon\_small\_fp='./data/OSMBoundary\_small/OSMBoundary\_small.shp'  
osmosis\_txt\_small\_fp='./data/OSMBoundary\_small.txt'  
shpPolygon2OsmosisTxt(shape\_polygon\_small\_fp,osmosis\_txt\_small\_fp)

POLYGON ((-90.0850881031402 40.9968994947319,-90.0850881031402 43.6657936592248,-87.383039973871 43.6657936592248,-87.383039973871 40.9968994947319,-90.0850881031402 40.9968994947319))  
LINEARRING (-90.0850881031402 40.9968994947319,-90.0850881031402 43.6657936592248,-87.383039973871 43.6657936592248,-87.383039973871 40.9968994947319,-90.0850881031402 40.9968994947319)  
POLYGON ((-87.6807286451907 41.8373927809521,-87.6807286451907 41.9214101975252,-87.5941157249019 41.9214101975252,-87.5941157249019 41.8373927809521,-87.6807286451907 41.8373927809521))  
LINEARRING (-87.6807286451907 41.8373927809521,-87.6807286451907 41.9214101975252,-87.5941157249019 41.9214101975252,-87.5941157249019 41.8373927809521,-87.6807286451907 41.8373927809521)

执行转换后，写入.txt格式文件的实际实验边界Polygon如下：

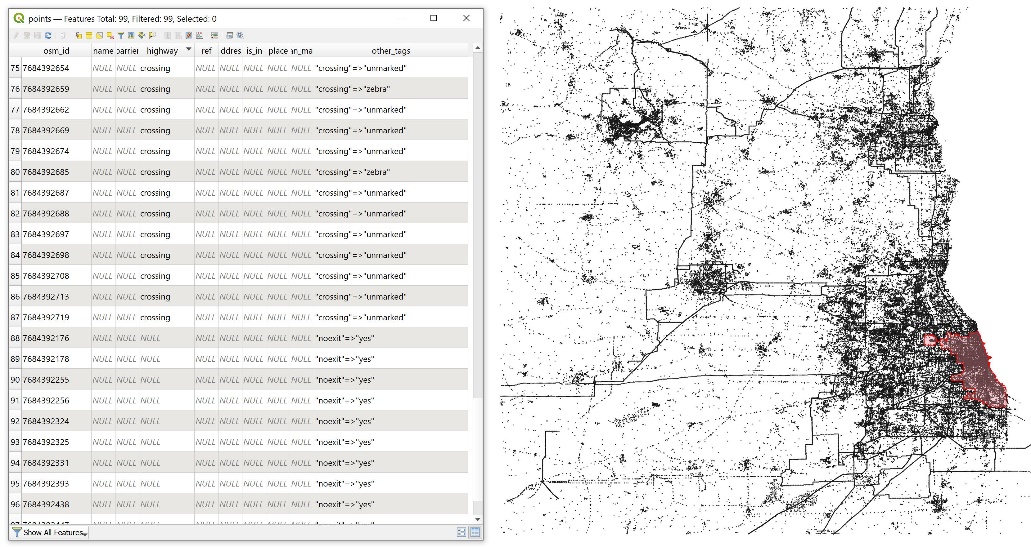
osmosis polygon  
first\_area  
-90.08508810314017 40.99689949473193  
-90.08508810314017 43.66579365922478  
-87.38303997387102 43.66579365922478  
-87.38303997387102 40.99689949473193  
-90.08508810314017 40.99689949473193  
END  
END

用于调试小批量数据提取的.txt边界：

osmosis polygon  
first\_area  
-87.68072864519071 41.83739278095207  
-87.68072864519071 41.92141019752525  
-87.59411572490187 41.92141019752525  
-87.59411572490187 41.83739278095207  
-87.68072864519071 41.83739278095207  
END  
END

osmosis也提供了多个.osm地理空间数据的合并方法，合并示例代码为osmosis --rx 1.osm --rx 2.osm --rx 3.osm --merge --wx merged.osm。首先执行合并操作，针对该实验的osmosis合并代码为osmosis --rx "F:/GitHubBigData/illinois-latest.osm" --rx "F:/GitHubBigData/wisconsin-latest.osm"--merge --wx "F:/GitHubBigData/illinois-wisconsin.osm"，合并后的文件大小为7.57GB。再执行裁切命令osmosis --read-xml file="F:\GitHubBigData\illinois-wisconsin.osm" --bounding-polygon file="C:\Users\richi\omen\_richiebao\omen\_github\USDA\_CH\_final\USDA\notebook\data\OSMBoundary.txt" --write-xml file="F:\GitHubBigData\osm\_clip.osm"，裁切后的文件大小为3.80GB。可以再通过QGIS查看数据是否已经按照预期合并裁切完毕。用于代码调试的小规模数据提取直接裁切osmosis --read-xml file="F:\GitHubBigData\illinois-wisconsin.osm" --bounding-polygon file="C:\Users\richi\omen\_richiebao\omen\_github\USDA\_CH\_final\USDA\notebook\data\OSMBoundary\_small.txt" --write-xml file="F:\GitHubBigData\osm\_small\_clip.osm"。

osmosis命令行，于Windows系统的命令行终端中执行，建议在[Windows PowerShell](https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7)⑫终端中执行代码。具体文件路径需根据个人情况调整。



OSM使用附加在基本数据结构上的标签（tag）来表示地面上的物理特征（feature），例如道路或者建筑物等。在QGIS中打开属性表，可以查看各要素属于的标签。对于具体标签的内容可以查看[Map Features](https://wiki.openstreetmap.org/wiki/Map_Features)⑬，下面仅列出主要标签的分类：

| 序号 | 一级标签 | 二级标签 |
| --- | --- | --- |
| 1 | Aerialway |  |
| 2 | Aeroway |  |
| 3 | Amenity | Sustenance,Education, Transportation, Financial, Healthcare, Entertainment, Arts & Culture, Others |
| 4 | Barrier | Linear barriers, Access control on highways |
| 5 | Boundary | Attributes |
| 6 | Building | Accommodation, Commercial, Religious,Civic/Amenity, Agricultural/Plant production, Sports,Storage,Cars, Power/Technical buildings,Other Buildings, Additional Attributes |
| 7 | Craft |  |
| 8 | Emergency | Medical Rescue,Firefighters, Lifeguards, Assembly point,Other Structure |
| 9 | Geological |  |
| 10 | Highway | Roads, Link roads, Special road types, Paths,Lifecycle, Attributes,Other highway features |
| 11 | Historic |  |
| 12 | Landuse | Common Landuse Key Values - Developed land, Common Landuse Key Values - Rural and agricultural land, Other Landuse Key Values |
| 13 | Leisure |  |
| 14 | Man\_made |  |
| 15 | Military |  |
| 16 | Natural | Vegetation or surface related, Water related, Landform related |
| 17 | Office |  |
| 18 | Place | Administratively declared places, Populated settlements, urban,Populated settlements, urban and rural, Other places |
| 19 | Power | Public Transport |
| 20 | Public Transport |  |
| 21 | Railway | Tracks, Additional features, Stations and Stops, Other railways |
| 22 | Route |  |
| 23 | Shop | Food, beverages, General store, department store, mall, Clothing, shoes, accessories, Discount store, charity, Health and beauty, Do-it-yourself, household, building materials, gardening, Furniture and interior, Electronics, Outdoors and sport, vehicles, Art, music, hobbies,Stationery, gifts, books, newspapers, Others |
| 24 | Sport |  |
| 25 | Telecom |  |
| 26 | Tourism |  |
| 27 | Waterway | Natural watercourses, Man-made waterways, Facilities,Barriers on waterways,Other features on waterways |
| Additional properties |  |  |
| 1 | Addresses | Tags for individual houses, For countries using hamlet, subdistrict, district, province, state, Tags for interpolation ways |
| 2 | Annotation |  |
| 3 | Name |  |
| 4 | Properties |  |
| 5 | References |  |
| 6 | Restrictions |  |

osmosis工具是由OSM及其开源社区成员建立的[osmcode.org](https://osmcode.org/)开发的工具。

#### 2）读取、转换OSM数据

用Python读取OSM数据仍然使用[osmcode.org](https://osmcode.org/)提供的工具[Pyosmium](https://docs.osmcode.org/pyosmium/latest/)⑭。Pyosmium用于处理不同格式的OSM文件，内核为c++的osmium库，能够有效快速的处理OSM数据。上述处理后的OSM数据文件osm\_clip.osm为3.80GB，如果一开始就使用较大的数据来编写程序，花费的时间成本可能较高，因此可以给更少的数据编写、调试，达到预期效果后再使用待要分析的大文件数据。

[Pyosmium GitHub仓库及安装](https://github.com/osmcode/pyosmium)详见：<https://github.com/osmcode/pyosmium>

编写读取.osm数据，需要对OSM的数据结构有所了解，从而能够清晰的提取所需要的值。[元素（elements）](https://wiki.openstreetmap.org/wiki/Elements)⑮是OSM物理世界概念数据模型的基本组成部分，包括节点nodes、路径或区域ways、关系relations，及标签tag。

| 元素（elements） | 图标 | 解释 | 对位shape地理空间数据（vector矢量） |
| --- | --- | --- | --- |
| node |  | 由经纬度坐标定义的地理空间点 | point |
| way |  | 由点(20-2000个)构成的路径以及闭合的区域，包含open way, closed way和area | polyline,polygon |
| relation |  | 记录两个或多个元素之间关系的多用途数据结构，关系可以有不同的含义，其意义由对应的标签定义 |  |
| tag |  | nodes、ways和relations都可以有描述其意义的标签，一个标签由键key:值value组成，Key必须是唯一的 | 字段 field |

* 示例

**Node**

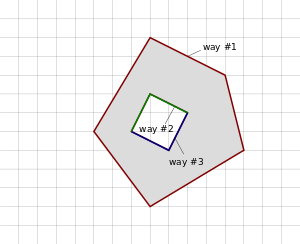
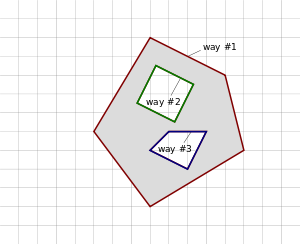
<node id="25496583" lat="51.5173639" lon="-0.140043" version="1" changeset="203496" user="80n" uid="1238" visible="true" timestamp="2007-01-28T11:40:26Z">  
 <tag k="highway" v="traffic\_signals"/>  
</node>

**way**

简单的路径或区域 simple way

<way id="5090250" visible="true" timestamp="2009-01-19T19:07:25Z" version="8" changeset="816806" user="Blumpsy" uid="64226">  
 <nd ref="822403"/>  
 <nd ref="21533912"/>  
 <nd ref="821601"/>  
 <nd ref="21533910"/>  
 <nd ref="135791608"/>  
 <nd ref="333725784"/>  
 <nd ref="333725781"/>  
 <nd ref="333725774"/>  
 <nd ref="333725776"/>  
 <nd ref="823771"/>  
 <tag k="highway" v="residential"/>  
 <tag k="name" v="Clipstone Street"/>  
 <tag k="oneway" v="yes"/>  
 </way>

多边形区域集合 multipolygon area



<relation id="12" timestamp="2008-12-21T19:31:43Z" user="kevjs1982" uid="84075">  
 <member type="way" ref="2878061" role="outer"/> <!-- picture ref="1" -->  
 <member type="way" ref="8125153" role="inner"/> <!-- picture ref="2" -->  
 <member type="way" ref="8125154" role="inner"/> <!-- picture ref="3" -->  
  
 <member type="way" ref="3811966" role=""/> <!-- empty role produces  
 a warning; avoid this; most software works around it by computing  
 a role, which is more expensive than having one set explicitly;  
 not shown in the sample pictures to the right -->  
  
 <tag k="type" v="multipolygon"/>  
 </relation>

* 元素的属性

| 名称 | 值类型 | 解释 |
| --- | --- | --- |
| id | integer (64-bit) | 用于表示元素 |
| user | character string | 最后修改对象的用户名 |
| uid | integer | 最后修改对象的用户ID |
| timestamp | W3C standard date and time | 最后修改时间 |
| visible | “true” or “false” | 数据库中的对象是否被删除 |
| version | integer | 版本控制 |
| changeset | integer | 创建或更新对象时使用的变更集编号 |

了解OSM基本的数据类型、结构和属性，通过继承osmium的类.SimpleHandler，用.apply\_file方法传入.osm文件，并定义所要提取的元素类型，给出该元素类型的属性提取对应的属性值。在地理空间数据分析中，通常比较关键的属性包括：元素修改的最后时间（.timestamp），标签（tags,<tag.k,tag.v>），生成的几何对象（geometry<point,linestring,multipolygon>）。下述函数分别提取了node, way(area)对象的属性和几何对象，同时将其转换为GeoDataFrame的数据格式，并存储为GPKG的格式数据，方便日后调用，尤其大批量数据。因为涉及到大批量数据，因此调入datatime时间模块，观察所用时间，帮助调试代码。

import osmium as osm  
  
class osmHandler(osm.SimpleHandler):   
 '''  
 class-通过继承osmium类 class osmium.SimpleHandler读取.osm数据.   
 '''  
   
 def \_\_init\_\_(self):  
 osm.SimpleHandler.\_\_init\_\_(self)  
 self.osm\_node=[]  
 self.osm\_way=[]  
 self.osm\_area=[]  
   
 def node(self,n):  
 import pandas as pd  
 import shapely.wkb as wkblib  
   
 wkbfab=osm.geom.WKBFactory()  
   
 wkb=wkbfab.create\_point(n)  
 point=wkblib.loads(wkb,hex=True)  
 self.osm\_node.append([  
 'node',  
 point,  
 n.id,  
 n.version,  
 n.visible,  
 pd.Timestamp(n.timestamp),  
 n.uid,  
 n.user,  
 n.changeset,  
 len(n.tags),  
 {tag.k:tag.v for tag in n.tags},  
 ])  
  
 def way(self,w):   
 import pandas as pd  
 import shapely.wkb as wkblib  
   
 wkbfab=osm.geom.WKBFactory()  
   
 try:  
 wkb=wkbfab.create\_linestring(w)  
 linestring=wkblib.loads(wkb, hex=True)  
 self.osm\_way.append([  
 'way',  
 linestring,  
 w.id,  
 w.version,  
 w.visible,  
 pd.Timestamp(w.timestamp),  
 w.uid,  
 w.user,  
 w.changeset,  
 len(w.tags),  
 {tag.k:tag.v for tag in w.tags},   
 ])  
 except:  
 pass  
   
 def area(self,a):   
 import pandas as pd  
 import shapely.wkb as wkblib  
   
 wkbfab=osm.geom.WKBFactory()   
   
 try:  
 wkb=wkbfab.create\_multipolygon(a)  
 multipolygon=wkblib.loads(wkb, hex=True)  
 self.osm\_area.append([  
 'area',  
 multipolygon,  
 a.id,  
 a.version,  
 a.visible,  
 pd.Timestamp(a.timestamp),  
 a.uid,  
 a.user,  
 a.changeset,  
 len(a.tags),  
 {tag.k:tag.v for tag in a.tags},   
 ])  
 except:  
 pass   
  
import datetime   
a\_T=datetime.datetime.now()  
print("start time:",a\_T)  
# osm\_Chicago\_fp=r"F:\data\osm\_small\_clip.osm" # 待读取的.osm数据路径, 用提取的小范围数据调试代码  
osm\_Chicago\_fp=r"F:\data\osm\_clip.osm" # 用小批量数据调试完之后，计算实际的实验数据  
  
osm\_handler=osmHandler() # 实例化类osmHandler()  
osm\_handler.apply\_file(osm\_Chicago\_fp,locations=True) # 调用 class osmium.SimpleHandler的apply\_file方法  
b\_T=datetime.datetime.now()  
print("end time:",b\_T)  
duration=(b\_T-a\_T).seconds/60  
print("Total time spend:%.2f minutes"%duration)

start time: 2021-12-24 11:00:18.357860  
end time: 2021-12-24 11:34:11.350215  
Total time spend:33.87 minutes

当读取全部OSM元素数据后，定义保存函数，如果是小批量数据，通常可以一起保存，但是本次实验数据有3.80GB，将读取后的数据转换为GeoDataFrame数据格式保存，将花费较多时间。因此将OSM元素逐个转换保存。同时，注意到在小批量调试时，保存node为GeoJSON格式文件其大小为104MB，而保存为GPKG仅有52.3MB，因此对于实验数据的保存，这里选择后者。

def save\_osm(osm\_handler,osm\_type,save\_path=r"./data/",fileType="GPKG"):   
 '''  
 function - 根据条件逐个保存读取的osm数据（node, way and area）  
   
 Params:  
 osm\_handler - osm返回的node,way和area数据，配套类osmHandler(osm.SimpleHandler)实现；Class  
 osm\_type - 要保存的osm元素类型，包括"node"，"way"和"area"；string  
 save\_path - 保存路径。The default is "./data/"；string  
 fileType - 保存的数据类型，包括"shp", "GeoJSON", "GPKG"。The default is "GPKG"；string  
   
 Returns:  
 osm\_node\_gdf - OSM的node类；GeoDataFrame(GeoPandas)  
 osm\_way\_gdf - OSM的way类；GeoDataFrame(GeoPandas)  
 osm\_area\_gdf - OSM的area类；GeoDataFrame(GeoPandas)  
 '''   
 import geopandas as gpd  
 import os  
 import datetime  
   
 a\_T=datetime.datetime.now()  
 print("start time:",a\_T)   
  
 def duration(a\_T):  
 b\_T=datetime.datetime.now()  
 print("end time:",b\_T)  
 duration=(b\_T-a\_T).seconds/60  
 print("Total time spend:%.2f minutes"%duration)  
   
 def save\_gdf(osm\_node\_gdf,fileType,osm\_type):  
 if fileType=="GeoJSON":  
 osm\_node\_gdf.to\_file(os.path.join(save\_path,"osm\_%s.geojson"%osm\_type),driver='GeoJSON')  
 elif fileType=="GPKG":  
 osm\_node\_gdf.to\_file(os.path.join(save\_path,"osm\_%s.gpkg"%osm\_type),driver='GPKG')  
 elif fileType=="shp":  
 osm\_node\_gdf.to\_file(os.path.join(save\_path,"osm\_%s.shp"%osm\_type))  
  
 epsg\_wgs84=4326 # 配置坐标系统，参考：https://spatialreference.org/   
 osm\_columns=['type','geometry','id','version','visible','ts','uid','user','changeet','tagLen','tags']  
 if osm\_type=="node":  
 osm\_node\_gdf=gpd.GeoDataFrame(osm\_handler.osm\_node,columns=osm\_columns,crs=epsg\_wgs84)  
 save\_gdf(osm\_node\_gdf,fileType,osm\_type)  
 duration(a\_T)  
 return osm\_node\_gdf  
  
 elif osm\_type=="way":  
 osm\_way\_gdf=gpd.GeoDataFrame(osm\_handler.osm\_way,columns=osm\_columns,crs=epsg\_wgs84)  
 save\_gdf(osm\_way\_gdf,fileType,osm\_type)  
 duration(a\_T)  
 return osm\_way\_gdf  
   
 elif osm\_type=="area":  
 osm\_area\_gdf=gpd.GeoDataFrame(osm\_handler.osm\_area,columns=osm\_columns,crs=epsg\_wgs84)  
 save\_gdf(osm\_area\_gdf,fileType,osm\_type)  
 duration(a\_T)  
 return osm\_area\_gdf

node\_gdf=save\_osm(osm\_handler,osm\_type="node",save\_path="./data/OSM\_processed",fileType="GPKG")

start time: 2021-12-24 11:35:34.746935  
end time: 2021-12-24 13:03:08.123318  
Total time spend:87.55 minutes

way\_gdf=save\_osm(osm\_handler,osm\_type="way",save\_path=r"./data/OSM\_processed",fileType="GPKG")

start time: 2021-12-24 14:16:00.736027  
end time: 2021-12-24 14:28:42.828275  
Total time spend:12.70 minutes

area\_gdf=save\_osm(osm\_handler,osm\_type="area",save\_path=r"./data/OSM\_processed",fileType="GPKG")

start time: 2021-12-24 14:29:30.864597  
end time: 2021-12-24 14:39:25.170465  
Total time spend:9.90 minutes

存储过程中，该部分实验数据node元素存储时间约87分钟，存储的GPKG文件大小为3.10GB，way和area元素则存储时间相对较短，存储文件较小。因为已经转换为GeoDataFrame格式地理空间数据，因此可以直接.plot()查看数据分布情况，初步判断是否读取和转换正确。下述代码用来测试读取的时间，其中way和area读取时间较短，而node元素读取时间较长。

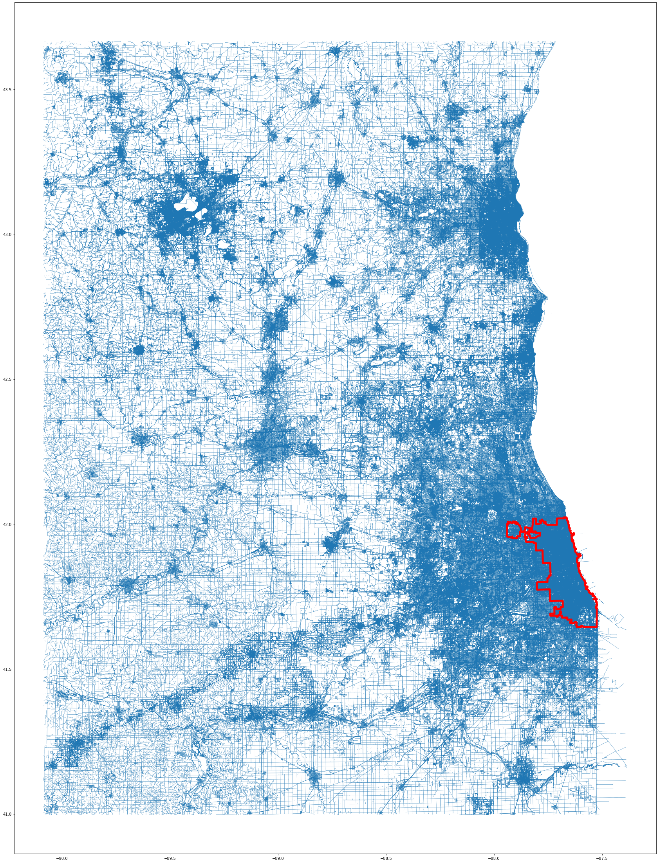
def start\_time():   
 '''  
 function-计算当前时间  
 '''  
 import datetime  
   
 start\_time=datetime.datetime.now()  
 print("start time:",start\_time)  
 return start\_time  
  
def duration(start\_time):   
 '''  
 function-计算持续时间  
   
 Params:  
 start\_time - 开始时间；datatime  
 '''  
 import datetime  
   
 end\_time=datetime.datetime.now()  
 print("end time:",end\_time)  
 duration=(end\_time-start\_time).seconds/60  
 print("Total time spend:%.2f minutes"%duration)

import geopandas as gpd  
  
start\_time=start\_time()  
read\_way\_gdf=gpd.read\_file("./data/OSM\_processed/osm\_way.gpkg")  
duration(start\_time)

start time: 2021-12-24 17:14:44.041892  
end time: 2021-12-24 17:17:46.256983  
Total time spend:3.03 minutes

因为数据量较大，直接gdf.plot()会花费较长时间

import matplotlib.pyplot as plt  
  
Chicago\_boundary\_city\_fp='./data/Chicago boundaries\_city/Chicago boundaries\_city.shp'  
Chicago\_boundary\_city=gpd.read\_file(Chicago\_boundary\_city\_fp)  
  
fig, ax=plt.subplots(figsize=(40,40))  
read\_way\_gdf.plot(ax=ax,linewidth=0.5,zorder=1)  
Chicago\_boundary\_city.plot(ax=ax,color='none', edgecolor='red',alpha=1,linewidth=5,zorder=2)  
plt.show()

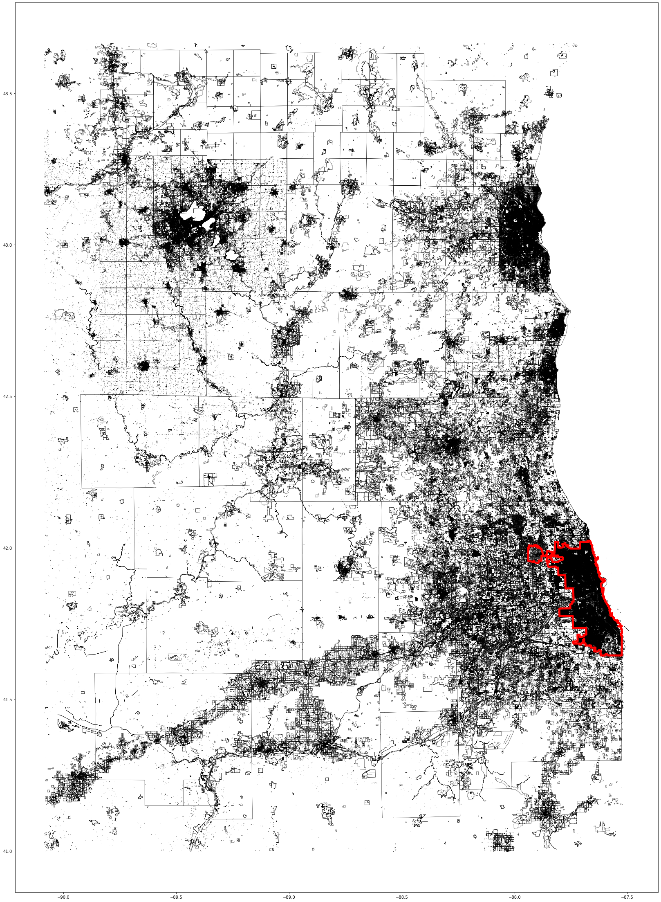


del read\_way\_gdf # 如果内存有限，可以使用del 删除不再使用的变量，节约内存

start\_time=start\_time()  
read\_area\_gdf=gpd.read\_file("./data/OSM\_processed/osm\_area.gpkg")  
duration(start\_time)

start time: 2021-12-24 17:10:05.802124  
end time: 2021-12-24 17:12:52.787002  
Total time spend:2.77 minutes

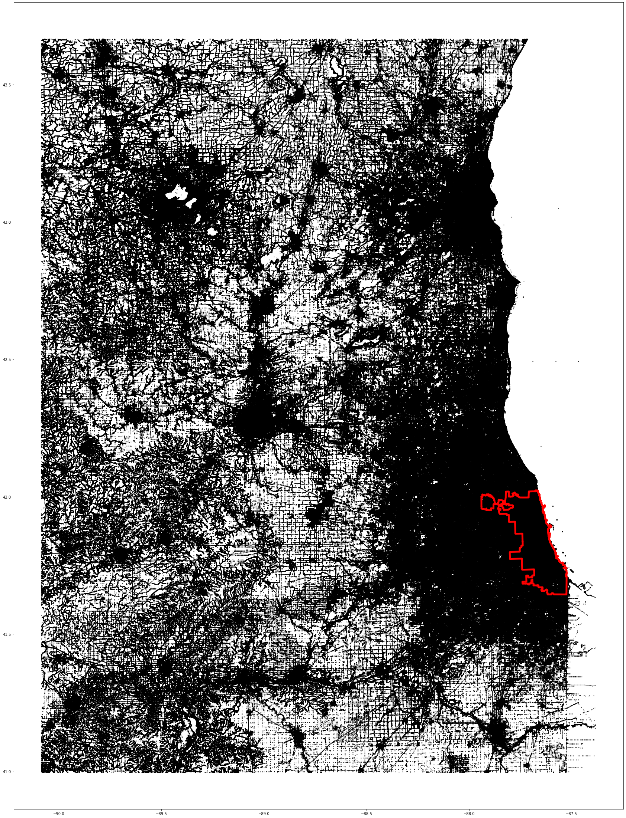
fig, ax=plt.subplots(figsize=(40,40))  
read\_area\_gdf.plot(ax=ax,color='none', edgecolor='black',linewidth=0.5,zorder=1)  
Chicago\_boundary\_city.plot(ax=ax,color='none', edgecolor='red',alpha=1,linewidth=5,zorder=2)  
plt.show()



start\_time=start\_time()  
read\_node\_gdf=gpd.read\_file("./data/OSM\_processed/osm\_node.gpkg")  
duration(start\_time)

start time: 2021-12-24 17:43:04.319218  
end time: 2021-12-24 18:13:09.240528  
Total time spend:30.07 minutes

fig, ax=plt.subplots(figsize=(40,40))  
read\_node\_gdf.plot(ax=ax,color='black',markersize=1)  
Chicago\_boundary\_city.plot(ax=ax,color='none', edgecolor='red',alpha=1,linewidth=5,zorder=2)  
plt.show()



### 2.1.2.2 核密度估计与地理空间点密度分布

#### 1）单变量（一维数组）的核密度估计

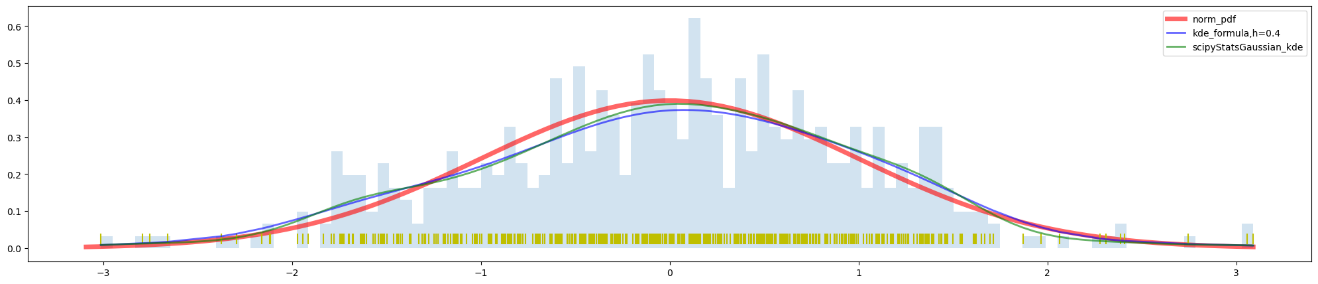
在[正态分布](https://richiebao.github.io/USDA_CH_final/#/./markdown/2_1_1_%E6%95%B0%E6%8D%AEPOI%E4%B8%8E%E6%8F%8F%E8%BF%B0%E6%80%A7%E7%BB%9F%E8%AE%A1%E5%92%8C%E6%AD%A3%E6%80%81%E5%88%86%E5%B8%83)一节中阐述有，当直方图的组距无限缩小至极限后，能够拟合出一条曲线，计算这个分布曲线的公式为概率密度函数（Probability Density Function，PDF）。用于估计概率密度函数的非参数方法就是核密度估计。核密度估计（Kernel density estimation，KDE）是一个基本的数据平滑问题（fundamental data smoothing problem），例如对不平滑的直方图平滑，给定一个核，并指定带宽（bandwidth），其值为正数，核密度估计定义为： $ *{n} (x)=* {i=1}^n {K( )} Kh e^{- x^{2} } *{n} (x)=* {i=1}^n { e^{ - } } $[1]。

在下述代码中绘制了三条曲线，红色粗线为概率密度函数；两条细线均为核密度估计（高斯核），只是蓝色线是依据核密度公式直接编写代码，并设置带宽h=0.4；绿色线则是使用SciPy库下的stats.gaussian\_kde()方法计算高斯核密度估计。

非参数统计（Nonparametric Statistics）是统计的一个分支，但不是完全基于参数化的概率分布，例如通过参数均值和方差（或标准差）定义一个正态分布，非参数统计基于自由分布（distribution-free）或指定的分布但未给分布参数，例如当处理PDF的一般情况时，不能像正态分布那样给定参数进行分类。其基本思想是在尽可能少的假定时，利用数据对一个未知量做出推断，通常意味着利用具有无穷维的统计模型[2]。

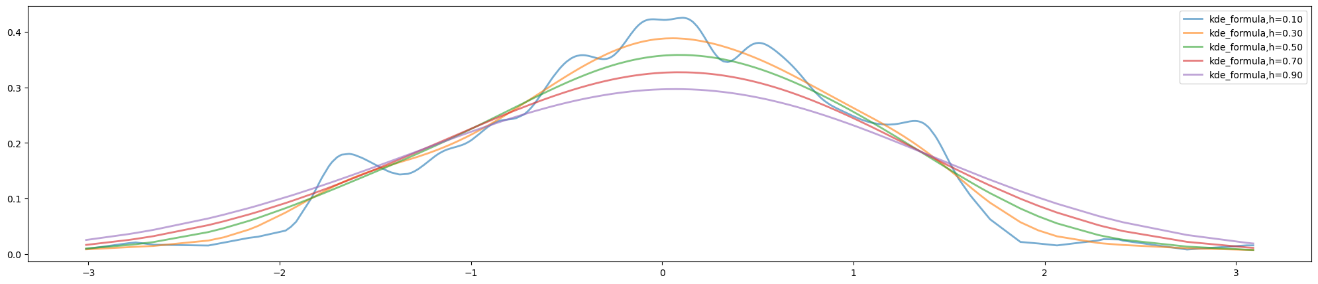
对于核密度估计名词中密度一词可以形象理解为下图中橄榄绿小竖线的分布密度。

import numpy as np  
from scipy import stats  
import matplotlib.pyplot as plt  
import math  
  
x=np.linspace(stats.norm.ppf(0.001,loc=0,scale=1),stats.norm.ppf(0.999,loc=0,scale=1), 100) # 等分概率为0.1%到99.9%之间的数值。如果不给参数loc和scale，则默认为标准正态分布，即loc=0, scale=1  
pdf=stats.norm.pdf(x)  
  
plt.figure(figsize=(25,5))  
plt.plot(x,pdf,'r-', lw=5, alpha=0.6, label='norm\_pdf')  
  
random\_variates=stats.norm.rvs(loc=0,scale=1,size=500)  
count, bins, ignored =plt.hist(random\_variates,bins=100,density=True,histtype='stepfilled',alpha=0.2)  
plt.eventplot(random\_variates,color='y',linelengths=0.03,lineoffsets=0.025) # 给定位置画出对应的短线  
  
rVar\_sort=np.sort(random\_variates)  
h=0.4 # 带宽（bandwidth,bw）  
n=len(rVar\_sort)  
kde\_Gaussian=[sum(math.exp(-1\*math.pow(vi-vj,2)/(2\*math.pow(h,2))) for vj in rVar\_sort)/(h\*n\*math.sqrt(2\*math.pi)) for vi in rVar\_sort] # 将上述高斯核密度估计公式转换为代码  
plt.plot(rVar\_sort,kde\_Gaussian,'b-', lw=2, alpha=0.6, label='kde\_formula,h=%s'%h)  
  
scipyStatsGaussian\_kde=stats.gaussian\_kde(random\_variates)  
plt.plot(bins,scipyStatsGaussian\_kde(bins),'g-', lw=2, alpha=0.6, label='scipyStatsGaussian\_kde')  
plt.legend()  
plt.show()



带宽（bandwidth）影响光滑的程度，下述实验设置不同的值，观察核密度曲线的变化情况。关于最适宜的带宽推断，Bonus algorithm for large scale stochastic nonlinear programming problems第3章，Probability Density Function and Kernel Density Estimation[1]一节中提到一种方法，可以参考。

bws=np.arange(0.1,1,0.2)  
colors\_kde=['C{}'.format(i) for i in range(len(bws))] # maplotlib，指定颜色  
i=0  
plt.figure(figsize=(25,5))  
for h in bws:  
 kde\_Gaussian=[sum(math.exp(-1\*math.pow(vi-vj,2)/(2\*math.pow(h,2))) for vj in rVar\_sort)/(h\*n\*math.sqrt(2\*math.pi)) for vi in rVar\_sort] # 将上述高斯核密度估计公式转换为代码  
 plt.plot(rVar\_sort,kde\_Gaussian,color=colors\_kde[i], lw=2, alpha=0.6, label='kde\_formula,h=%.2f'%h)  
 i+=1  
plt.legend()  
plt.show()



#### 2）多变量（多维数组）的核密度估计

上文叙述了OSM数据处理的方法，并简要的概述了OSM地理空间数据集的结构。OSM的node标签分类丰富，针对不同的问题可以提取不同的标签用于分析。因为标签中的便利设施（amenity）与人们的日常生活栖息相关，包括生计、教育、交通、金融、医疗、娱乐、艺术和文化等，更进一步的分类则多达100多类。使用GeoPandas的gdf.clip(polygon)方法进一步裁切数据到目标区域保存，读取后再提取标签（tags）为amenity的空间点数据，计算核密度估计，查看其分布情况。

import util\_misc  
import geopandas as gpd  
  
start\_time=util\_misc.start\_time() # start\_time()和duration(start\_time)函数放置于util\_misc.py文件中，方便调用  
read\_node\_gdf=gpd.read\_file("./data/OSM\_processed/osm\_node.gpkg")  
util\_misc.duration(start\_time)

start time: 2021-12-24 21:11:29.190573  
end time: 2021-12-24 21:40:36.262947  
Total time spend:29.12 minutes

read\_node\_gdf.head()

|  | **type** | **id** | **version** | **visible** |  | **ts** | **uid** | **user** | **changeet** | **tagLen** | **tags** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | node | 219850 | 55 | True |  | 2018-02-20T05:50:28 | 0 |  | 0 | 2 | {"highway": "motorway\_junction", "ref": "276C"} | POINT (-87.91012 41.75859) |
| **1** | node | 219851 | 48 | True |  | 2018-02-20T05:50:29 | 0 |  | 0 | 2 | {"highway": "motorway\_junction", "ref": "277A"} | POINT (-87.90764 41.75931) |
| **2** | node | 219966 | 5 | True |  | 2009-04-04T22:47:50 | 0 |  | 0 | 0 | {} | POINT (-87.91596 43.01149) |
| **3** | node | 219968 | 12 | True |  | 2015-08-04T05:38:49 | 0 |  | 0 | 2 | {"ref": "73B", "highway": "motorway\_junction"} | POINT (-87.92464 43.05606) |
| **4** | node | 219969 | 6 | True |  | 2009-04-14T00:13:37 | 0 |  | 0 | 0 | {} | POINT (-87.92441 43.05684) |

* 给定边界（SHP格式，例如用于物种-鸟观测分析的边界），裁切OSM的node数据。

chicago\_species\_study\_area\_wgs84\_fp='./data/Chicago study area\_species/chicago\_species\_study\_area\_wgs84.shp'  
chicago\_species\_study\_area\_wgs84=gpd.read\_file(chicago\_species\_study\_area\_wgs84\_fp)  
start\_time=util\_misc.start\_time()   
chicago\_species\_node\_gdf=read\_node\_gdf.clip(chicago\_species\_study\_area\_wgs84)  
util\_misc.duration(start\_time)  
  
chicago\_species\_node\_gdf.to\_file('./data/OSM\_processed/osm\_node\_4speciesStudy.gpkg',driver='GPKG')

start time: 2021-12-24 21:49:56.113615  
end time: 2021-12-24 22:07:42.284118  
Total time spend:17.77 minutes

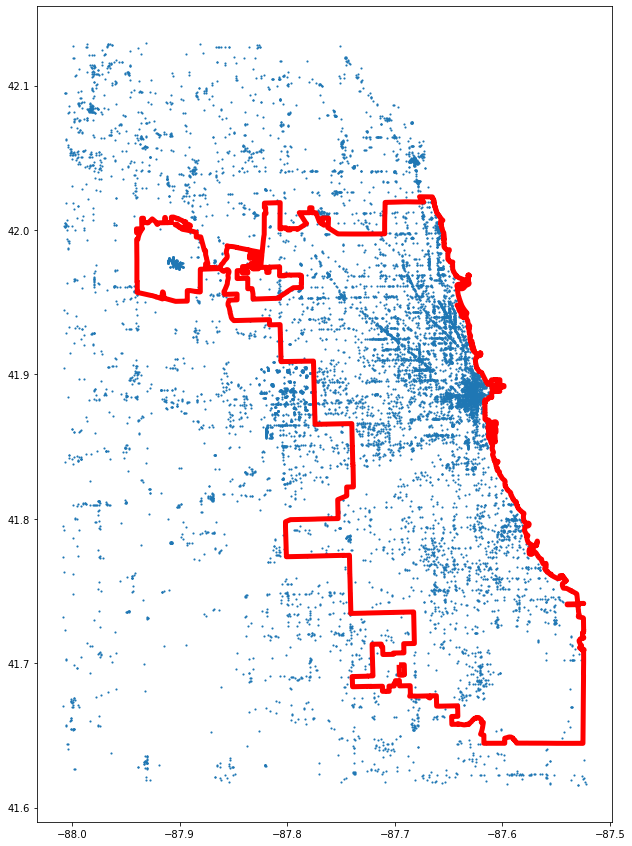
import geopandas as gpd  
   
osm\_node\_4speciesStudy=gpd.read\_file('G:/data/OSM\_processed/osm\_node\_4speciesStudy.gpkg') # './data/OSM\_processed/osm\_node\_4speciesStudy.gpkg'  
amenity\_poi=osm\_node\_4speciesStudy[osm\_node\_4speciesStudy.tags.apply(lambda row: "amenity" in eval(row).keys())] # 提取标tags列，含标签amenity的所有行  
print("Amenity extraction finished!")

Amenity extraction finished!

提取仅含amenity的行后，数据大幅度减少。

print(  
 "the overal data number:",osm\_node\_4speciesStudy.shape,'\n',  
 "the amenity data number:",amenity\_poi.shape,'\n',   
 )  
  
import matplotlib.pyplot as plt  
  
Chicago\_boundary\_city\_fp='./data/Chicago\_boundaries\_city/Chicago\_boundaries\_city.shp'  
Chicago\_boundary\_city=gpd.read\_file(Chicago\_boundary\_city\_fp)  
fig, ax=plt.subplots(figsize=(15,15))  
amenity\_poi.plot(ax=ax,marker=".",markersize=5,zorder=1)  
Chicago\_boundary\_city.plot(ax=ax,color='none', edgecolor='red',alpha=1,linewidth=5,zorder=2)  
plt.show()

the overal data number: (7124076, 11)   
 the amenity data number: (12606, 11)



##### 1. 核密度估计-scipy.stats.gaussian\_kde()方法

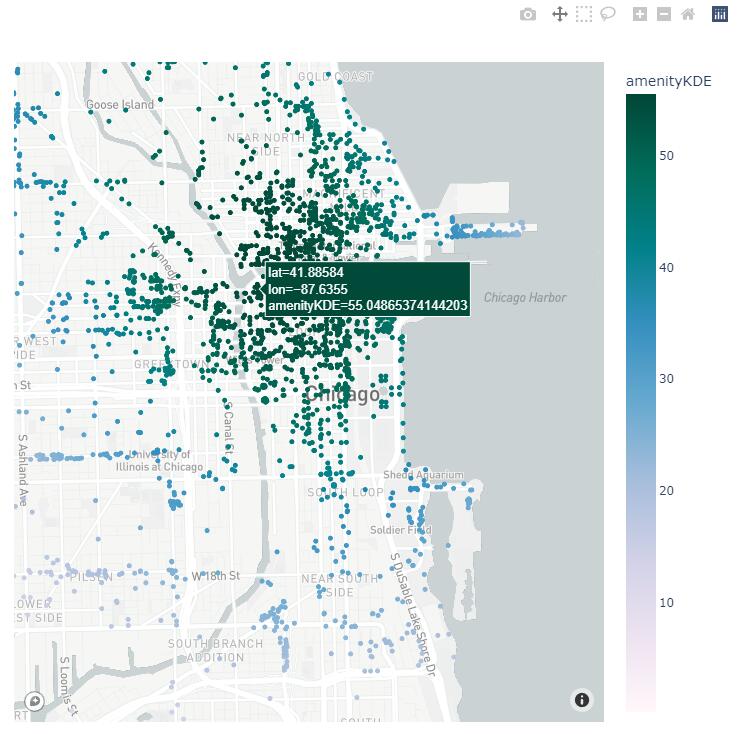
核密度估计可以平滑多维数据，例如热力图的制作是基于核密度估计的二维平滑，以OSM数据为例，直接使用scipy.stats.gaussian\_kde()计算点分布的核密度。

import pandas as pd  
import numpy as np  
from scipy import stats  
pd.options.mode.chained\_assignment=None  
  
poi\_coordinates=np.array([amenity\_poi.geometry.x,amenity\_poi.geometry.y])  
amenity\_kernel=stats.gaussian\_kde(poi\_coordinates) # 核密度估计  
amenity\_poi['amenityKDE']=amenity\_kernel(poi\_coordinates)

建立地图，打印显示核密度分布。

mapbox为连接至境外服务器，国内有可能无法显示地图

import plotly.express as px  
  
mapbox\_token='pk.eyJ1IjoicmljaGllYmFvIiwiYSI6ImNrYjB3N2NyMzBlMG8yc254dTRzNnMyeHMifQ.QT7MdjQKs9Y6OtaJaJAn0A'  
px.set\_mapbox\_access\_token(mapbox\_token)  
fig=px.scatter\_mapbox(amenity\_poi,lat=amenity\_poi.geometry.y, lon=amenity\_poi.geometry.x,color='amenityKDE',color\_continuous\_scale=px.colors.sequential.PuBuGn,size\_max=15,zoom=10)  
fig.update\_layout(autosize=False,width=800,height=800,)  
fig.show()  
amenity\_poi.head()



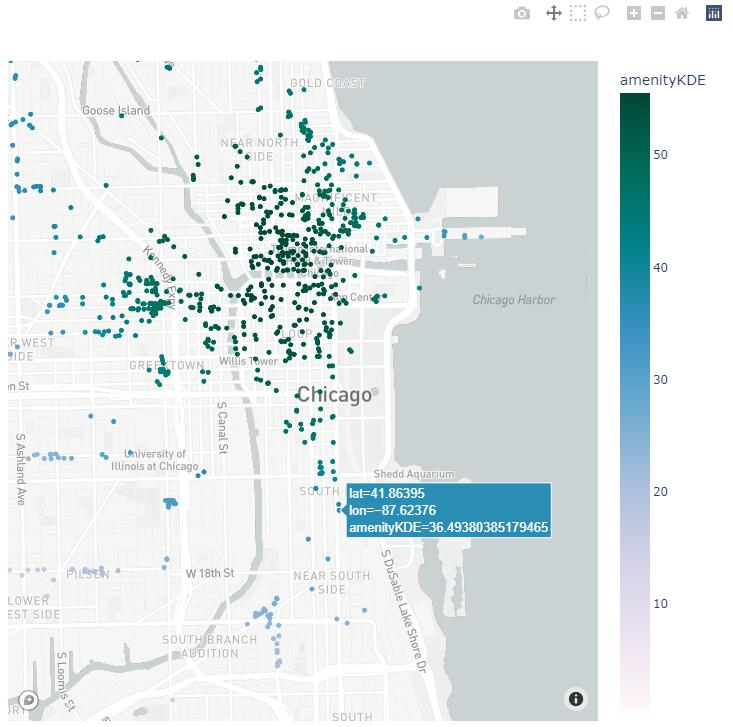
|  | **type** | **id** | **version** | **visible** | **ts** | **uid** | **user** | **changeet** | **tagLen** | **tags** | **geometry** | **amenityKDE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **51751** | node | 354150593 | 2 | True | 2009-08-09T20:12:06 | 0 |  | 0 | 7 | {"ele": "185", "name": "Wacker Elementary Scho... | POINT (-87.64847 41.71624) | 3.342937 |
| **73722** | node | 354128001 | 1 | True | 2009-03-01T01:48:33 | 0 |  | 0 | 7 | {"ele": "179", "name": "Our Lady School", "ame... | POINT (-87.56699 41.71559) | 1.879984 |
| **73753** | node | 354247611 | 3 | True | 2010-02-17T23:04:22 | 0 |  | 0 | 9 | {"ele": "179", "name": "Our Lady Gate of Heave... | POINT (-87.56671 41.71587) | 1.900003 |
| **75310** | node | 354146543 | 1 | True | 2009-03-01T02:23:05 | 0 |  | 0 | 7 | {"ele": "187", "name": "Saint Margaret School"... | POINT (-87.65449 41.71531) | 3.031726 |
| **75906** | node | 354273265 | 3 | True | 2010-02-17T23:04:13 | 0 |  | 0 | 9 | {"ele": "187", "name": "Saint Margaret of Scot... | POINT (-87.65505 41.71448) | 2.970502 |

提取amenity字段，查看分类，并单独计算restaurant的核密度。

amenity\_poi['amenity']=amenity\_poi.tags.apply(lambda row:eval(row)["amenity"])  
print("服务设施子类：\n",amenity\_poi.amenity.unique())

服务设施子类：  
 ['school' 'place\_of\_worship' 'fountain' 'fuel' 'bank' 'parking'  
 'restaurant' 'fire\_station' 'bar' 'bureau\_de\_change' 'ice\_cream'  
 'public\_building' 'fast\_food' 'courthouse' 'bicycle\_rental' 'pharmacy'  
 'atm' 'clinic' 'social\_facility' 'drinking\_water' 'vending\_machine'  
 'money\_transfer' 'dentist' 'cafe' 'bicycle\_parking' 'bench' 'theatre'  
 'waste\_basket' 'recycling' 'parking\_entrance' 'social\_centre'  
 'arts\_centre' 'car\_wash' 'post\_box' 'bicycle\_repair\_station' 'cinema'  
 'pub' 'car\_sharing' 'shower' 'community\_centre' 'car\_rental' 'toilets'  
 'veterinary' 'post\_office' 'marketplace' 'boat\_rental' 'photo\_booth'  
 'charging\_station' 'taxi' 'food\_court' 'police' 'loading\_dock' 'library'  
 'venue' 'nightclub' 'hospital' 'dojo' 'shelter' 'ferry\_terminal'  
 'doctors' 'college' 'studio' 'embassy' 'university' 'events\_venue'  
 'clock' 'retail' 'internet\_cafe' 'driving\_school' 'health\_center'  
 'parking\_space' 'research\_institute' 'bus\_station' 'kindergarten'  
 'townhall' 'community\_center' 'Auto Repair Shop' 'public\_bookcase'  
 'public\_bath' 'device\_charging\_station' 'book\_return' 'animal\_boarding'  
 'childcare' 'music\_school' 'dancing\_school' 'Community Center' 'dental'  
 'grave\_yard' 'telephone' 'language\_school' 'prep\_school' 'spa' 'gym'  
 'animal\_shelter' 'payment\_centre' 'salon' 'dance\_theater' 'brewery'  
 'water\_point' 'bbq' 'gymnasium' 'waste\_disposal' 'music\_venue' 'ATM'  
 'swimming\_pool' 'payment\_terminal' 'child\_care']

restaurant\_df=amenity\_poi.loc[amenity\_poi.amenity=='restaurant']  
restaurant\_coordinates=np.array([restaurant\_df.geometry.x,restaurant\_df.geometry.y])  
restaurant\_kernel=stats.gaussian\_kde(restaurant\_coordinates)   
restaurant\_df['restaurant\_kde']=restaurant\_kernel(restaurant\_coordinates)  
  
import plotly.express as px  
mapbox\_token='pk.eyJ1IjoicmljaGllYmFvIiwiYSI6ImNrYjB3N2NyMzBlMG8yc254dTRzNnMyeHMifQ.QT7MdjQKs9Y6OtaJaJAn0A'  
px.set\_mapbox\_access\_token(mapbox\_token)  
fig=px.scatter\_mapbox(restaurant\_df,lat=restaurant\_df.geometry.y, lon=restaurant\_df.geometry.x,color='amenityKDE',color\_continuous\_scale=px.colors.sequential.PuBuGn,size\_max=15,zoom=10)  
fig.update\_layout(autosize=False,width=800,height=800,)  
fig.show()



##### 2. 核密度估计结果转换为地理栅格数据

在地图表达上，上述二维地理空间点的核密度估计最终的计算值落在了点位置本身。如果希望能够以栅格的形式显示估计值，可以将其转换为栅格数据，即为地理空间点数据转栅格数据。在下述的方法中，提供了两种转换方式，一种方式是将存储有核密度估计值的GeoDataFrame格式数据先使用gdf.to\_file()存储为SHP格式的点数据，然后定义SHP转栅格的函数；第二种方式是定义一个函数，能够直接计算核密度估计并同时直接存储为栅格数据。

**EPSG编号体系**

在这两种方式中均需要定义GeoDataFrame格式数据的坐标投影，并提取信息用于使用GDAL库提取或定义坐标投影，因此必须在GeoPandas库和GDAL库之间有可以互相转换的共同坐标体系，EPSG编号体系是比较好的选择。EPSG（European Petroleum Survey Group）最早建立了该编码体系，其中最为重要的编码包括：

EPSG:4326 - 即WGS84，广泛应用于地图和导航系统，在地理空间数据分析中，这个地理坐标系统是最为基础表征位置数据的坐标系统，通常各类地理空间的数据信息都以WGS84为基本的坐标系统，而后可以在不同数据类型或者平台，及根据分析目的的不同，尤其实际地理位置的变化来配置投影。

EPSG:3857 - 伪墨卡托投影，也被称为球体墨卡托，用于显示许多基于网页的地图工具，包括Google地图和OpenStreetMap等。

关于EPSG编码可以查看：[Spatial Reference](https://spatialreference.org/)⑯，和[epsg.io](https://epsg.io/)⑰

本次实验输出栅格的坐标投影系统为EPSG:32616，即WGS 84 / UTM zone 16N，对应芝加哥城Landsat遥感影像所使用的坐标投影系统。通常在分析某一城市空间问题时，可能用到很多类型的，不同来源的数据，尽量以最基本的WGS84作为数据存储坐标系统之外，数据的显示往往需要对应区域的坐标系统，从而优化显示结果。

* SHP格式地理空间点数据转栅格数据

在定义字段名时，如果写入.shp格式数据后，字段名有可能被裁切，例如如果定义字段名为“amenity\_kde”，那么用GeoPandas存储为.shp后，字段名可能被裁切为”amenity\_kd”。如果读取该数据，不注意字段名的变化，可能带来不易察觉的错误。

import geopandas as gpd  
from pyproj import CRS  
import os  
  
print("original projection:",amenity\_poi.crs)  
amenity\_poi\_copy=amenity\_poi.copy(deep=True)  
amenity\_poi\_copy=amenity\_poi.to\_crs(CRS("EPSG:32616")) # EPSG:32616 - WGS 84 / UTM zone 16N - Projected  
print("re-projecting:",amenity\_poi\_copy.crs)  
  
amenity\_kde\_fn='./data\_processed/amenity\_kde/amenity\_kde.shp'  
amenity\_poi\_copy.to\_file(amenity\_kde\_fn)

original projection: epsg:4326  
re-projecting: EPSG:32616

定义pts2raster()函数的核心是GDAL库提供的gdal.RasterizeLayer()方法，可以将读取的.shp点层属性字段的值写入对应位置的栅格，避免编写对位栅格单元位置和字段值数组的代码。在栅格投影定义上直接提取.shp点数据的坐标投影系统，代码位置于向栅格单元写入数据之后，如果位于之前，会出现坐标投影错误。在栅格定义时，空值通常设置为-9999。GDAL提供栅格定义的方式为先获取栅格驱动gdal.GetDriverByName('GTiff')，再建立.Create(raster\_path, x\_res, y\_res, 1, gdal.GDT\_Float64)。并配置地理变换target\_ds.SetGeoTransform((x\_min, cellSize, 0, y\_max, 0, -cellSize))，通过读取栅格波段band=target\_ds.GetRasterBand(1)，向其写入值band.WriteArray()完成栅格的定义。地理变换中，因为地图通常向上为北向，因此第3和第5个参数通常配置为0。

def pts2raster(pts\_shp,raster\_path,cellSize,field\_name=False):   
 '''  
 function - 将.shp格式的点数据转换为.tif栅格(raster)  
 将点数据写入为raster数据。使用raster.SetGeoTransform，栅格化数据。参考GDAL官网代码   
   
 Params:  
 pts\_shp - .shp格式点数据文件路径；SHP点数据  
 raster\_path - 保存的栅格文件路径；string  
 cellSize - 栅格单元大小；int  
 field\_name - 写入栅格的.shp点数据属性字段；string  
   
 Returns:  
 返回读取已经保存的栅格数据；array  
 '''  
 from osgeo import gdal,ogr  
   
 # 定义空值（没有数据）的栅格数值 Define NoData value of new raster  
 NoData\_value=-9999  
   
 # 打开.shp点数据，并返回地理区域范围 Open the data source and read in the extent  
 source\_ds=ogr.Open(pts\_shp)  
 source\_layer=source\_ds.GetLayer()  
 x\_min, x\_max, y\_min, y\_max=source\_layer.GetExtent()  
   
 # 使用GDAL库建立栅格 Create the destination data source  
 x\_res=int((x\_max - x\_min) / cellSize)  
 y\_res=int((y\_max - y\_min) / cellSize)  
 target\_ds=gdal.GetDriverByName('GTiff').Create(raster\_path, x\_res, y\_res, 1, gdal.GDT\_Float64) #create(filename,x\_size,y\_size,band\_count,data\_type,creation\_options)。gdal的数据类型 gdal.GDT\_Float64,gdal.GDT\_Int32...  
 target\_ds.SetGeoTransform((x\_min, cellSize, 0, y\_max, 0, -cellSize))  
 outband=target\_ds.GetRasterBand(1)  
 outband.SetNoDataValue(NoData\_value)  
  
 # 向栅格层中写入数据  
 if field\_name:  
 gdal.RasterizeLayer(target\_ds,[1], source\_layer,options=["ATTRIBUTE={0}".format(field\_name)])  
 else:  
 gdal.RasterizeLayer(target\_ds,[1], source\_layer,burn\_values=[-1])   
   
 # 配置投影坐标系统  
 spatialRef=source\_layer.GetSpatialRef()  
 target\_ds.SetProjection(spatialRef.ExportToWkt())   
   
 outband.FlushCache()  
 return gdal.Open(raster\_path).ReadAsArray()  
  
amenity\_kde\_fn='./data\_processed/amenity\_kde/amenity\_kde.shp'  
raster\_path='./data\_processed/amenity\_epsg32616.tif'  
cellSize=300  
field\_name='amenityKDE'  
poiRaster\_array=pts2raster(amenity\_kde\_fn,raster\_path,cellSize,field\_name)  
print("conversion completed!")

conversion completed!

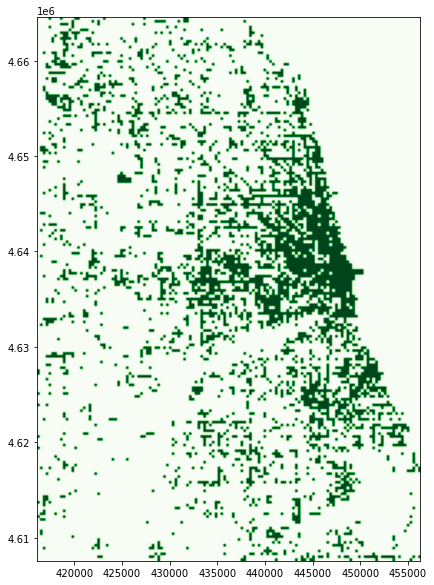
Python中处理栅格数据的库可以使用[rasterio](https://rasterio.readthedocs.io/en/latest/quickstart.html" \l "reading-raster-data)，相比GDAL能够大幅度减少代码书写量，方法相较也更便捷。用该库读取栅格数据的相关信息，并读取数据为数组（array），打印栅格查看数据。

如果环境中已经安装了GeoPandas，再安装rasterio（conda install -c conda-forge rasterio）后运行代码如果提示错误（库之间可能存在冲突），可以新建环境安装rasterio，在新建环境下执行下述代码。

import rasterio  
  
raster\_path='./data\_processed/amenity\_epsg32616.tif'  
dataset=rasterio.open(raster\_path)  
print(  
 "band count:",dataset.count,'\n', # 查看栅格波段数量  
 "columns wide:",dataset.width,'\n', # 查看栅格宽度  
 "rows hight:",dataset.height,'\n', # 查看栅格高度  
 "dataset's index and data type:",{i: dtype for i, dtype in zip(dataset.indexes, dataset.dtypes)},'\n',# 查看波段及其数据类型  
 "bounds:",dataset.bounds,'\n', # 查看外接矩形边界左下角与右上角坐标  
 "geospatial transform:",dataset.transform,'\n', # 数据集的地理空间变换  
 "lower right corner:",dataset.transform\*(dataset.width,dataset.height),'\n', # 计算外接矩形边界右下角坐标  
 "crs:",dataset.crs,'\n', # 地理坐标投影系统  
 "band's index number:",dataset.indexes,'\n' # 栅格层（波段）索引  
 )  
band1=dataset.read(1) # 读取栅格波段数据为数组  
print(band1)

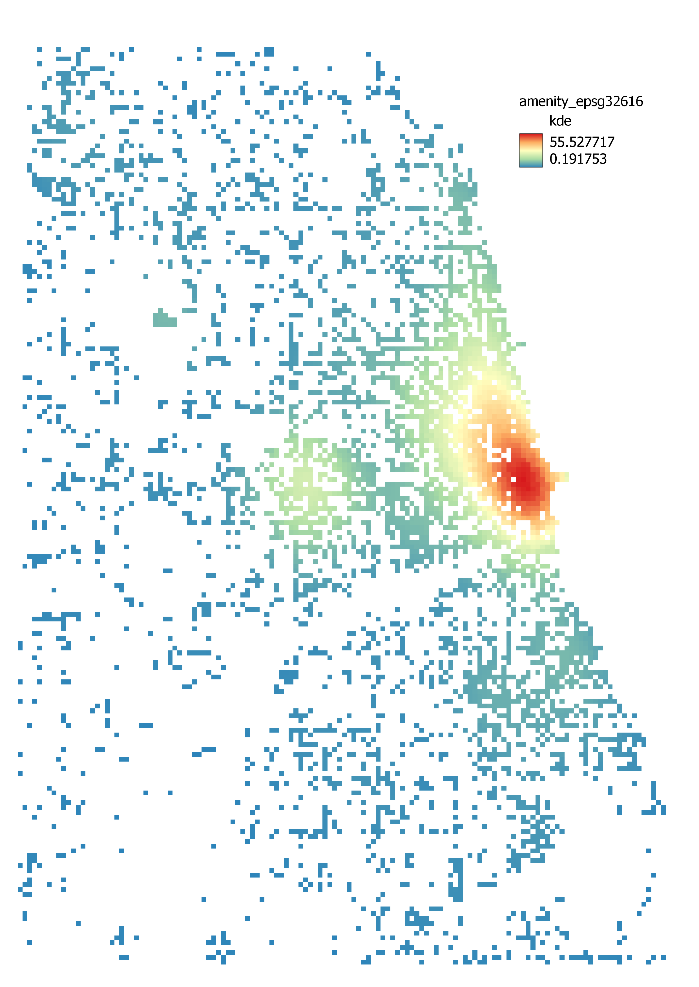
band count: 1   
 columns wide: 134   
 rows hight: 190   
 dataset's index and data type: {1: 'float64'}   
 bounds: BoundingBox(left=416035.55653533165, bottom=4607564.55138286, right=456235.55653533165, top=4664564.55138286)   
 geospatial transform: | 300.00, 0.00, 416035.56|  
| 0.00,-300.00, 4664564.55|  
| 0.00, 0.00, 1.00|   
 lower right corner: (456235.55653533165, 4607564.55138286)   
 crs: PROJCS["WGS 84 / UTM zone 16N",GEOGCS["WGS 84",DATUM["WGS\_1984",SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4326"]],PROJECTION["Transverse\_Mercator"],PARAMETER["latitude\_of\_origin",0],PARAMETER["central\_meridian",-87],PARAMETER["scale\_factor",0.9996],PARAMETER["false\_easting",500000],PARAMETER["false\_northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]],AXIS["Easting",EAST],AXIS["Northing",NORTH],AUTHORITY["EPSG","32616"]]   
 band's index number: (1,)   
  
[[-9999. -9999. -9999. ... -9999. -9999. -9999.]  
 [-9999. -9999. -9999. ... -9999. -9999. -9999.]  
 [-9999. -9999. -9999. ... -9999. -9999. -9999.]  
 ...  
 [-9999. -9999. -9999. ... -9999. -9999. -9999.]  
 [-9999. -9999. -9999. ... -9999. -9999. -9999.]  
 [-9999. -9999. -9999. ... -9999. -9999. -9999.]]

from rasterio.plot import show  
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(15,10))  
show((dataset,1),cmap='Greens')   
plt.show()



rasterio库支持的颜色（基本同matplotlib库，可以在该库查看具体名称对应的颜色带）： ‘Accent’, ‘Accent\_r’, ‘Blues’, ‘Blues\_r’, ‘BrBG’, ‘BrBG\_r’, ‘BuGn’, ‘BuGn\_r’, ‘BuPu’, ‘BuPu\_r’, ‘CMRmap’, ‘CMRmap\_r’, ‘Dark2’, ‘Dark2\_r’, ‘GnBu’, ‘GnBu\_r’, ‘Greens’, ‘Greens\_r’, ‘Greys’, ‘Greys\_r’, ‘OrRd’, ‘OrRd\_r’, ‘Oranges’, ‘Oranges\_r’, ‘PRGn’, ‘PRGn\_r’, ‘Paired’, ‘Paired\_r’, ‘Pastel1’, ‘Pastel1\_r’, ‘Pastel2’, ‘Pastel2\_r’, ‘PiYG’, ‘PiYG\_r’, ‘PuBu’, ‘PuBuGn’, ‘PuBuGn\_r’, ‘PuBu\_r’, ‘PuOr’, ‘PuOr\_r’, ‘PuRd’, ‘PuRd\_r’, ‘Purples’, ‘Purples\_r’, ‘RdBu’, ‘RdBu\_r’, ‘RdGy’, ‘RdGy\_r’, ‘RdPu’, ‘RdPu\_r’, ‘RdYlBu’, ‘RdYlBu\_r’, ‘RdYlGn’, ‘RdYlGn\_r’, ‘Reds’, ‘Reds\_r’, ‘Set1’, ‘Set1\_r’, ‘Set2’, ‘Set2\_r’, ‘Set3’, ‘Set3\_r’, ‘Spectral’, ‘Spectral\_r’, ‘Wistia’, ‘Wistia\_r’, ‘YlGn’, ‘YlGnBu’, ‘YlGnBu\_r’, ‘YlGn\_r’, ‘YlOrBr’, ‘YlOrBr\_r’, ‘YlOrRd’, ‘YlOrRd\_r’, ‘afmhot’, ‘afmhot\_r’, ‘autumn’, ‘autumn\_r’, ‘binary’, ‘binary\_r’, ‘bone’, ‘bone\_r’, ‘brg’, ‘brg\_r’, ‘bwr’, ‘bwr\_r’, ‘cividis’, ‘cividis\_r’, ‘cool’, ‘cool\_r’, ‘coolwarm’, ‘coolwarm\_r’, ‘copper’, ‘copper\_r’, ‘cubehelix’, ‘cubehelix\_r’, ‘flag’, ‘flag\_r’, ‘gist\_earth’, ‘gist\_earth\_r’, ‘gist\_gray’, ‘gist\_gray\_r’, ‘gist\_heat’, ‘gist\_heat\_r’, ‘gist\_ncar’, ‘gist\_ncar\_r’, ‘gist\_rainbow’, ‘gist\_rainbow\_r’, ‘gist\_stern’, ‘gist\_stern\_r’, ‘gist\_yarg’, ‘gist\_yarg\_r’, ‘gnuplot’, ‘gnuplot2’, ‘gnuplot2\_r’, ‘gnuplot\_r’, ‘gray’, ‘gray\_r’, ‘hot’, ‘hot\_r’, ‘hsv’, ‘hsv\_r’, ‘inferno’, ‘inferno\_r’, ‘jet’, ‘jet\_r’, ‘magma’, ‘magma\_r’, ‘nipy\_spectral’, ‘nipy\_spectral\_r’, ‘ocean’, ‘ocean\_r’, ‘pink’, ‘pink\_r’, ‘plasma’, ‘plasma\_r’, ‘prism’, ‘prism\_r’, ‘rainbow’, ‘rainbow\_r’, ‘seismic’, ‘seismic\_r’, ‘spring’, ‘spring\_r’, ‘summer’, ‘summer\_r’, ‘tab10’, ‘tab10\_r’, ‘tab20’, ‘tab20\_r’, ‘tab20b’, ‘tab20b\_r’, ‘tab20c’, ‘tab20c\_r’, ‘terrain’, ‘terrain\_r’, ‘twilight’, ‘twilight\_r’, ‘twilight\_shifted’, ‘twilight\_shifted\_r’, ‘viridis’, ‘viridis\_r’, ‘winter’, ‘winter\_r’

在QGIS中打开栅格数据查看结果。该部分的栅格化过程，开始配置空值outband.SetNoDataValue(-9999)。后续向栅格写入数据时，仅含有值的位置替换原空值，因此，所看到的栅格空值部分透明。



* 给定GeoDataFrame格式的地理空间点数据，计算核密度估计并存储为栅格数据

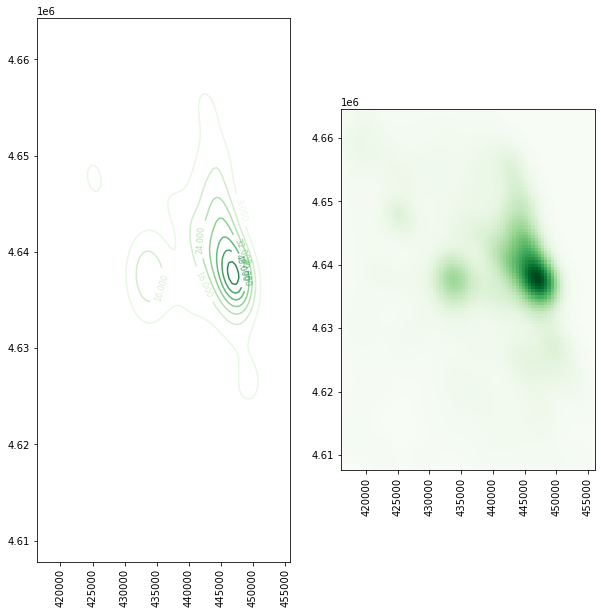
下述函数ptsKDE\_geoDF2raster()的定义将核密度估计置于函数之内，可以传入GeoDataFrame格式地理空间点数据（pts\_gdf），给定保存位置（raster\_path），栅格单元大小（cellSize），及调整核密度估计值比例缩放因子（scale）直接获取最后的核密度估计值栅格图（热力图）。这种计算方法能够减少中间步骤，不需先转换为SHP格式的点数据后再存储为栅格数据。但是这种将多个步骤置于一个函数中的计算方式，会将“单步”的计算时间拉长，因此最好在函数内通过print()函数打印相关完成信息，避免大批量数据计算时，不知道程序完成的进度，无法确定程序是否仍在正常运行，或者已经完成，甚至已经中断。

GDAL库提供建立栅格，向栅格单元写入数值的方法是outband.WriteArray()，传入的参数为数组，这个数组对应着栅格的位置，因此计算完核密度估计之后，定义提取估计值的位置坐标需要重新进行定义，而不能直接用点坐标。在位置（positions）定义上借助np.meshgrid()实现，同时需要注意上述定义的positions位置提取估计值，其顺序是逆反的，由下往上逐行读取，这个通常符合对图片像素的定义顺序；在地理栅格数据中，通常是由上往下写入，因此用np.flip(Z,0)翻转数组，将最后一行提为正数第一行，倒数第二行为正数第二行，以此类推。

def ptsKDE\_geoDF2raster(pts\_geoDF,raster\_path,cellSize,scale):  
 '''  
 function - 计算GeoDaraFrame格式的点数据核密度估计，并转换为栅格数据  
   
 Params:  
 pts\_geoDF - GeoDaraFrame格式的点数据；GeoDataFrame(GeoPandas)  
 raster\_path - 保存的栅格文件路径；string  
 cellSize - 栅格单元大小；int  
 scale - 缩放核密度估计值；int/float  
   
 Returns:  
 返回读取已经保存的核密度估计栅格数据；array  
 '''  
 from osgeo import gdal,ogr,osr  
 import numpy as np  
 from scipy import stats  
   
 # 定义空值（没有数据）的栅格数值 Define NoData value of new raster  
 NoData\_value=-9999  
 x\_min, y\_min,x\_max, y\_max=pts\_geoDF.geometry.total\_bounds  
  
 # 使用GDAL库建立栅格 Create the destination data source  
 x\_res=int((x\_max - x\_min) / cellSize)  
 y\_res=int((y\_max - y\_min) / cellSize)  
 target\_ds=gdal.GetDriverByName('GTiff').Create(raster\_path, x\_res, y\_res, 1, gdal.GDT\_Float64 )  
 target\_ds.SetGeoTransform((x\_min, cellSize, 0, y\_max, 0, -cellSize))  
 outband=target\_ds.GetRasterBand(1)  
 outband.SetNoDataValue(NoData\_value)   
   
 # 配置投影坐标系统  
 spatialRef = osr.SpatialReference()  
 epsg=int(pts\_geoDF.crs.srs.split(":")[-1])  
 spatialRef.ImportFromEPSG(epsg)   
 target\_ds.SetProjection(spatialRef.ExportToWkt())  
   
 # 向栅格层中写入数据  
 X, Y = np.meshgrid(np.linspace(x\_min,x\_max,x\_res), np.linspace(y\_min,y\_max,y\_res)) # 用于定义提取核密度估计值的栅格单元坐标数组  
 positions=np.vstack([X.ravel(), Y.ravel()])  
 values=np.vstack([pts\_geoDF.geometry.x, pts\_geoDF.geometry.y])   
 print("Start calculating kde...")  
 kernel=stats.gaussian\_kde(values)  
 Z=np.reshape(kernel(positions).T, X.shape)  
 print("Finish calculating kde!")  
 # print(values)  
   
 outband.WriteArray(np.flip(Z,0)\*scale) # 需要翻转数组，写栅格单元   
 outband.FlushCache()  
 print("conversion completed!")  
 return gdal.Open(raster\_path).ReadAsArray()  
   
raster\_path\_gpd='./data\_processed/amenity\_kde\_gdf.tif'  
cellSize=500 # cellSize值越小，需要计算的时间越长，开始调试时，可以尝试将其调大以节约计算时间  
scale=10\*\*10 # 相当于math.pow(10,10)  
poiRasterGeoDF\_array=ptsKDE\_geoDF2raster(amenity\_poi\_copy,raster\_path\_gpd,cellSize,scale)

Start calculating kde...  
Finish calculating kde!  
conversion completed!

import rasterio  
from rasterio.plot import show  
import matplotlib.pyplot as plt  
  
dataset\_gpd=rasterio.open(raster\_path\_gpd)  
fig, axs=plt.subplots(1,2,figsize=(10, 10))  
  
show((dataset\_gpd,1),contour=True,cmap='Greens',ax=axs[0]) # 开启等高线模式  
show((dataset\_gpd,1),cmap='Greens',ax=axs[1])  
axs[0].tick\_params(axis='x', rotation=90)  
axs[1].tick\_params(axis='x', rotation=90)  
plt.show()



注释（Notes）：

① OpenStreetMap（OSM），为开源世界地图，可以免费下载。地图数据托管服务由 UCL, Fastly, Bytemark Hosting和其他合作伙伴支持（<https://www.openstreetmap.org/#map=13/41.8679/-87.6569>）。

② Geofabrik，Geofabrik是一家位于德国Karlsruhe的咨询和软件开发公司，专门从事OpenStreetMap服务（<https://download.geofabrik.de/north-america/us.html>）。

③ llinois-latest-free.shp.zip，Illinois区域OpenStreetMap数据（<https://download.geofabrik.de/north-america/us/illinois.html>）。

④ wisconsin-latest.osm.bz2，wisconsin区域OpenStreetMap数据（<https://download.geofabrik.de/north-america/us/wisconsin.html>）。

⑤ 芝加哥城行政范围，（<https://data.cityofchicago.org/Facilities-Geographic-Boundaries/Boundaries-City/ewy2-6yfk>）。

⑥ Chicago Data Portal，为芝加哥城开放数据门户，可免费下载数据用于相关分析，其中许多数据集每天至少更新一次或数次（<https://data.cityofchicago.org/>）。

⑦ osmosis，用于处理OSM数据的命令行工具。该工具包含的组件（例如，读写数据库和文件，获取和应用数据源的变化，进行数据分类等）可以自由组合以执行更大的操作。（<https://wiki.openstreetmap.org/wiki/Osmosis>）。

⑧ Manipulating Data with Osmosis，用于操作和处理原始.osm数据，并通常处理大的数据文件，将OSM文件分割成较小的部分，及应用变化集更新现有文件（<https://learnosm.org/en/osm-data/osmosis/>）。

⑨ GDAL，一些用于编程和操作GDAL地理空间数据抽象库的工具，实际上包括两个库GDAL库和OGR库，分布用于处理地理空间的栅格数据和矢量数据（<https://pypi.org/project/GDAL/>）。

⑩ GDAL/OGR Cookbook，关于如何使用Python GDAL/OGR API的简单代码片断（<http://pcjericks.github.io/py-gdalogr-cookbook/index.html>）。

⑪ GDAL documentation，GDAL文档（<https://gdal.org/>）。

⑫ Windows PowerShell，是一个跨平台的任务自动化解决方案，由一个命令行命令行外壳（shell）、一种脚本语言和一个配置管理框架（framework）组成。（<https://learn.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7>）。

⑬ Map Features，使用附加在基本数据结构（nodes，ways，relations）上的标签来表示地面上的物理特征（例如，道路和建筑）。每个标签都描述了该node，way和relation所表示特征的一个地理属性（<https://wiki.openstreetmap.org/wiki/Map_features>）。

⑭ Pyosmium, 是一个处理不同OSM文件格式的库，为一个C++库的osmium封装器，允许快速有效连续的处理OpenStreetMap数据（<https://docs.osmcode.org/pyosmium/latest/>）。

⑮ 元素（elements），是OpenStreetMap物理世界概念数据模型的基本组成部分，包括nodes、ways和relations（<https://wiki.openstreetmap.org/wiki/Elements>）。

⑯ Spatial Reference，可以用于查找EPSG编号和CRS坐标系（<https://spatialreference.org/>）。

⑰ epsg.io， 世界各地坐标系（Coordinate Systems Worldwide），用于查找EPSG编号（<https://epsg.io/>）。

参考文献（References）:

[1] Urmila Diwekar, Amy David. Bonus algorithm for large scale stochastic nonlinear programming problems[M]. New York:Springer, 2015.

[2] [美]Larry Wasserman, 吴喜之译. 现代非参数统计[M].科学出版社,北京.2008.5. (Larry Wasserman. All of nonparametric statistics[M]. New York:Springer, 2006.）