Capítulo 5

Introdução à Análise de Dados Geoespaciais com Python

Gesiel Rios Lopes, Alexandre C. B. Delbem, Joélcio Braga de Sousa

Resumo

A análise espacial, ou apenas análise geoespacial, é uma abordagem para aplicar a análise estatística e outras técnicas analíticas a dados que possuem um aspecto geográfico ou espacial. Essa análise normalmente é feita utilizando técnicas de renderização de mapas a partir do processamento de dados espaciais e a aplicação de métodos analíticos a conjuntos de dados terrestres ou geográficos. Este minicurso é uma introdução à análise de dados geoespaciais com Python, com foco em dados vetoriais tabulares usando GeoPandas. O conteúdo concentra-se em apresentar as diferentes bibliotecas para trabalhar com dados geoespaciais e as relações no espaço. Isso inclui a importação de dados em diferentes formatos (por exemplo, `shapefile`, `GeoJSON`*), visualizando, combinando e organizando-os para análise, fazendo o uso de bibliotecas como* `pandas`, `geopandas`, `shapely`, `pyproj`, `matplotlib`, `cartopy`, *dentre outras.*

5.1. Introdução

A compreensão da distribuição espacial a partir de dados originados de fenômenos ocorridos no espaço constitui um grande desafio para diversas áreas do conhecimento, seja em saúde, em geologia, em agronomia, computação entre tantas outras. Tais estudos vem se tornando cada vez mais comum, devido a crescente disponibilidade de dados espaciais, além do crescimento vertiginoso das tecnologias de Sistemas de Informação Geográficas - SIG (do inglês, *Geographic Information System – GIS*) aliada procedimentos computacionais e recursos humanos [Monteiro et al. 2004].

Entender a distribuição dos dados geoespaciais, ou seja, levando em conta a localização espacial do fenômeno em estudo de forma explícita, e traduzi-los em padrões considerando propriedades mensuráveis e relacionadas faz parte da Análise Espacial de Dados Geográficos [Monteiro et al. 2004].

A partir dos dados espaciais, é possível descobrir não apenas a localização, mas também o comprimento, tamanho, área ou forma de qualquer objeto. Os dados geo-

espaciais têm um grande número de aplicações em nossa vida cotidiana, indo desde o entendimento e modelagem do comportamento urbano de pessoas, veículos e outros objetos móveis e utilizando esse entendimento na construção e aperfeiçoamento de modelos de contágio, auxiliando no desenvolvimento de ações e medidas preventivas no controle de epidemias [Zheng et al. 2014, Domingues et al. 2020].

Neste minicurso será apresentada uma introdução à análise de dados geoespaciais com Python, com foco em dados vetoriais tabulares a partir de diferentes bibliotecas para trabalhar com dados geoespaciais e as relações no espaço, como `pandas`, `geopandas`, `shapely`, `pyproj`, `matplotlib`, dentre outras.

5.2. Introdução aos dados vetoriais

Como já mencionado, quando falamos sobre dados geoespaciais, estamos falando sobre os dados que representam objetos/características na superfície da Terra e sua localização específica no globo, sendo portanto disponibilizados em vários formatos. Um dos formatos mais utilizados em aplicações de análise geoespaciais é o de **Dados Vetoriais** [Monteiro et al. 2004, Domingues et al. 2020, Lawhead 2015].

O formato dos dados vetoriais baseia-se na utilização de pontos ou vértices sequenciais para definir a localização e os limites de um objeto, ou seja, conectar esses pontos forma uma linha e conectar essas linhas acabará por criar um polígono. Este polígono certamente envolverá uma área. Podemos facilmente usar isso para representar um objeto ou uma região na superfície superficial da terra. Portanto, podemos dizer que os vetores são mais bem usados para apresentar generalizações de objetos ou características na superfície da Terra [Domingues et al. 2020].

5.2.1. Formatos para criar e compartilhar o conjunto de dados espaciais

Shapefile: O `shapefile` é um formato não-topológico para bases de dados geoespaciais e vetoriais. É considerado um formato aberto, apesar de proprietário. Por ser aberto, o formato recebe suporte de diversos aplicativos de processamento de mapas gratuitos e de código livre. Apesar de ser um termo no singular, o formato `shapefile` consiste numa coleção de arquivos de mesmo nome e terminações diferentes, armazenados no mesmo diretório. Existem três arquivos obrigatórios para o funcionamento correto de um `shapefile`: `.shp`, `.shx` e `.dbf`. O arquivo `shapefile` propriamente dito é o `.shp`, mas se distribuído sozinho não será capaz de exibir os dados armazenados. A distribuição deve ser feita juntamente com os outros dois arquivos.

- `.shp` — formato shape; as características da geometria propriamente dita;
- `.shx` — formato índice de shape, ou seja, um índice com as características das geometrias para permitir buscas mais rápidas;
- `.dbf` — formato de atributos; atributos apresentados em colunas para cada "`shape`"

GeoJSONs: O `GeoJSON` é um formato de intercâmbio de dados geoespaciais de padrão aberto que representa características geográficas simples e seus atributos não

espaciais, ou seja, informações não espaciais sobre uma característica geográfica. A ideia central é fornecer uma especificação para codificação de dados geoespaciais, permanecendo decodificáveis por qualquer decodificador `JSON`. Sendo um subconjunto do `JSON` imensamente popular, o suporte de análise está em um nível diferente do !Shapefile!. Além disso, para obter suporte da maioria dos SIGs. Os recursos suportados pelo `GeoJSON` são pontos, `MultiPoint`, `LineString`, `Polygon`, `MultiPoint`, `MultiLineString` e `MultiPolygon`.

GeoPackage: O GeoPackage foi desenvolvido pela *Open Geospatial Consortium* (OGC), tornando-se a alternativa oficial para o `Shapefile`. É um subconjunto do SQLite, que por sua vez é uma implementação SQL leve projetada para bancos de dados autônomos. Semelhante ao `GeoJSON`, isso torna o `GeoPackage` altamente compatível por design e acessível a qualquer SIG.

5.2.2. Ferramentas e pacotes python para começar com os dados geoespaciais

Os pacotes python que podem ser utilizados para começar a explorar dados geoespaciais são os seguintes:

- **Pandas:** Pandas[1] é uma biblioteca licenciada com código aberto que oferece estruturas de dados de alto desempenho e de fácil utilização voltado a análise de dados para a linguagem de programação Python [Coelho 2017].
- **Matplotlib:** O Matplotlib[2] é uma biblioteca de plotagem 2D do Python que produz números de qualidade de publicação em vários formatos de cópia impressa e ambientes interativos entre plataformas. O Matplotlib pode ser usado em *scripts* Python, nos *shell* Python e IPython, *notebooks* Jupyter e em servidores *web*.
- **GDAL:** GDAL[3] é uma biblioteca de tradução para formatos de dados geoespaciais vetoriais e raster que é lançada sob uma Licença de Código Aberto do estilo X/MIT pela Open Source Geospatial Foundation. Como uma biblioteca, ela apresenta um único modelo de dados abstratos de rasterização e um único modelo de dados abstratos de vetor para o aplicativo de chamada para todos os formatos suportados.
- **Shapely:** Shapely[4] é um pacote Python para análise teórica de conjunto e manipulação de recursos planares usando (via módulo ctypes do Python) funções da biblioteca GEOS[5] bem conhecida e amplamente implantada.
- **GeoPandas:** GeoPandas[6] é um projeto de código aberto para facilitar o trabalho com dados geoespaciais em python. GeoPandas estende os tipos de dados usados pelos pandas para permitir operações espaciais em tipos geométricos. As operações geométricas são realizadas pelo shapely.

[1]Disponível em `https://pandas.pydata.org/`

[2]Disponível em `https://matplotlib.org/`

[3]Disponível em `https://gdal.org/`

[4]Disponível em `https://github.com/Toblerity/Shapely`

[5]GEOS, uma porta do Java Topology Suite (JTS), é o mecanismo de geometria da extensão espacial PostGIS para o PostgreSQL RDBMS

[6]Disponível em `https://geopandas.org/`

- **Seaborn:** Seaborn[7] é uma biblioteca de visualização de dados Python baseada no *matplotlib*. Ela fornece uma interface de alto nível para desenhar gráficos estatísticos atraentes e informativos.

Para o desenvolvimento deste minicurso, será utilizado o ambiente do Jupyter Notebook, uma interface de programação literária interativa[8] muito interessante para criar seus modelos e compartilhar com quem quiser, disponibilizado pelo *Anaconda*[9], conforme Figura 5.1 [Shen 2014, Pimentel et al. 2021]. Em [Pimentel et al. 2021] e [Lopes et al. 2019] pode ser encontrado um vasto material de como utilizar o Jupyter Notebook.

Figura 5.1. Diagrama das ferramentas do ambiente de trabalho utilizados.

Um alternativa similar ao Jupyter Notebook e que não requer configuração para ser usada é uma ferramenta desenvolida pelo Google chamada Colaboratory [10]. Para usar este ambiente, é necessário apenas ter uma conta google. No Google Colaboratory o código do seu notebook é executado em uma máquina virtual dedicada à sua conta. As máquinas virtuais são recicladas após um determinado tempo ocioso, ou caso a janela seja fechada. Ao restaurar um notebook com manipulação de arquivos, talvez seja necessário refazer o *upload* dos arquivos utilizados e executar as opções "*Runtime*" e "*Restart and run all*".

5.2.3. Introdução ao GeoPandas

GeoPandas, como já visto, estende as estruturas de dados do Pandas, uma das mais populares bibliotecas de ciência de dados, adicionando suporte para dados geoespaciais.

A estrutura de dados central do GeoPandas é `geopandas.GeoDataFrame`, uma subclasse do `pandas.DataFrame` capaz de armazenar colunas geométricas e realizar operações espaciais. As geometrias são tratadas como `geopandas.GeoSeries`, uma subclasse de `pandas.Series`. Portanto, seu GeoDataFrame é uma combinação de Series com seus dados (numéricos, booleanos, texto etc.) e GeoSeries com geometrias (pontos, polígonos e etc.), A Figura 5.2 é apresentada uma visão geral de um GeoDataFrame.

O primeiro passo, antes de ler alguns dados geoespaciais, é declarar o uso da biblioteca GeoPandas (Figira 5.3). Primeiramente, usaremos `\%matplotlib inline`, linha 1, uma configuração para permitir que os mapas apareçam diretamente no nosso no-

[7]Disponível em `https://seaborn.pydata.org/`

[8]O paradigma de programação literária busca ajudar na comunicação de programas através da alternância de texto em linguagem natural formatada, pedaços de código executáveis, e resultados de computações [Pimentel et al. 2021]

[9]Anaconda, disponível em `https://docs.anaconda.com/anaconda/install/`

[10]Disponível em: `https://colab.research.google.com/`

Figura 5.2. Estrutura de um GeoDataFrame do GeoPandas.

tebook, ao invés de serem exibidos em uma janela diferente. Além disso, importaremos o GeoPandas com o alias `gpd`, linha 2.

```
1 %matplotlib inline
2 import geopandas as gpd
```

Figura 5.3. Declaração do GeoPandas.

Assumindo que temos um arquivo contendo dados e geometria (por exemplo, GeoPackage, GeoJSON, Shapefile), podemos lê-lo facilmente usando a função *gpd.read_file*, que detecta automaticamente o tipo de arquivo e cria um GeoDataFrame. Para criar nosso primeiro mapa, utilizaremos a malha de setores censitários do estado do Piauí que provêm do Instituto Brasileiro de Geografia e Estatística (IBGE) e que pode ser baixado no seguinte endereço: `https://shorturl.at/klAF7`.

A Figura 5.4 apresenta os primeiros cinco registros do GeoDataFrame da malha de setores censitários do estado do Piauí.

```
1 setores_censitarios_pi = gpd.read_file(url_setores_censitarios_pi_ibge)
2 setores_censitarios_pi.head()
```

	CD_SETOR	CD_SIT	NM_SIT	CD_UF	NM_UF	SIGLA_UF	CD_MUN	NM_MUN	CD_DIST	NM_DIST	CD_SUBDIST	NM_SUBDIST	geometry
0	220005305000001	1	Área Urbana de Alta Densidade de Edificações	22	Piauí	PI	2200053	Acauã	220005305	Acauã	22000530500	None	POLYGON ((-41.08058 -8.21775, -41.08105 -8.218...
1	220005305000002	8	Área Rural (exclusive aglomerados)	22	Piauí	PI	2200053	Acauã	220005305	Acauã	22000530500	None	POLYGON ((-40.85118 -8.17031, -40.85080 -8.170...
2	220005305000004	8	Área Rural (exclusive aglomerados)	22	Piauí	PI	2200053	Acauã	220005305	Acauã	22000530500	None	POLYGON ((-40.78822 -8.22331, -40.78802 -8.223...
3	220005305000005	8	Área Rural (exclusive aglomerados)	22	Piauí	PI	2200053	Acauã	220005305	Acauã	22000530500	None	POLYGON ((-40.88414 -8.23197, -40.88462 -8.231...
4	220005305000006	8	Área Rural (exclusive aglomerados)	22	Piauí	PI	2200053	Acauã	220005305	Acauã	22000530500	None	POLYGON ((-40.96740 -8.31242, -40.96741 -8.312...

Figura 5.4. Malha de setores censitários do estado do Piauí.

Na Figura 5.5, é apresentado o mapa com a malha de setores censitários do Piauí, criada a partir da chamada da função `plot()`, uma das funcionalidades do Matplotlib embutidas no GeoPandas. Cada GeoDataFrame contém uma coluna especial de geometria, coluna "*geometry*", que contém todos os objetos geométricos que são exibidos quando chamamos a função `plot()`.

É possível manipular o GeoDataFrame da mesma forma que manipulamos um DataFrame no pandas e para exemplificar, iremos selecionar apenas a malha de setores censitários urbanos de Teresina, capital do Piauí, através da função *query* do pandas, linhas 4 e 5 da Figura 5.6.

```
1 setores_censitarios_pi.plot(figsize=(15,6));
```

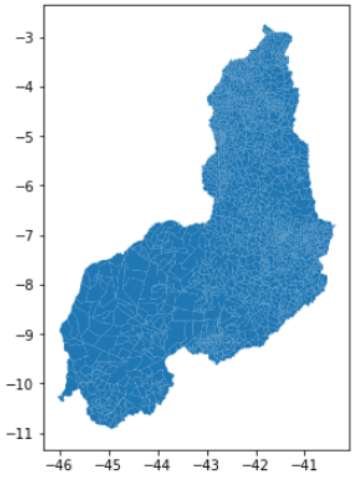


Figura 5.5. Mapa com a malha de setores censitários do Piauí.

5.2.4. Geometrias: pontos, linhas e polígonos

Como visto na seção **??**, os dados vetoriais espaciais podem consistir em diferentes tipos, e os 3 tipos fundamentais são Figura 5.7:

- **Point**: representa um único ponto no espaço.
- **Line** ("LineString"): representa uma sequência de pontos que formam uma linha.
- **Polygon**: representa uma área preenchida.

Na Figura 5.8 exemplos de criação de figuras básicas do shapely. Na linha 1 é feito o import do GeoPandas para plotar as geometrias que serão criadas, já na linha 2 é feita a importação da representação dos dados vetoriais do shapely, em seguida são criados três polígonos, linhas de 4 a 6, um polígono formados de LineString, linha 8 e um ponto, linha 10. Em seguida é criado uma GeoSeries para imprimir as geometrias criadas. Para que as linhas fiquem mais visíveis, alteraremos a paleta de cores para `tab10`.

5.2.5. Sistemas de Coordenadas

O campo de estudo que mede a forma e o tamanho da Terra é a geodésia. Segundo [Bolstad 2016], para que seja utilizado de maneira efetiva dados geoespaciais e os sistemas que derivam deles, é necessário estabelecer um entendimento claro de como os sistemas de coordenadas são definidos para a Terra, como essas coordenadas são medidas sobre a superfície curva da mesma, e por fim como são convertidas em diversas projeções para seu uso, seja manual ou digital.

Um sistema de coordenadas geográficas que é definido por coordenadas bidimensionais com base na superfície da Terra tem coordenadas X, que é a longitude e tinha as

```
setores_censitarios_urbano_teresina = (
    setores_censitarios_pi
    .query(
        "NM_MUN == 'Teresina' and \
         NM_SIT.str.contains('Urban')",
         engine='python'
    )
)
setores_censitarios_urbano_teresina.plot(figsize=(15,6));
```

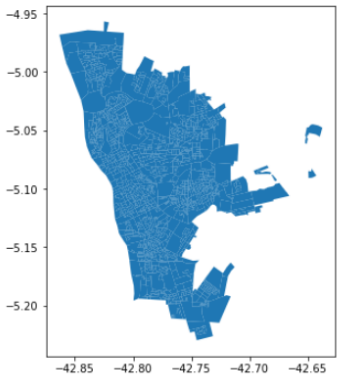


Figura 5.6. Mapa dos Setores Censitários Urbanos de Teresina - PI.

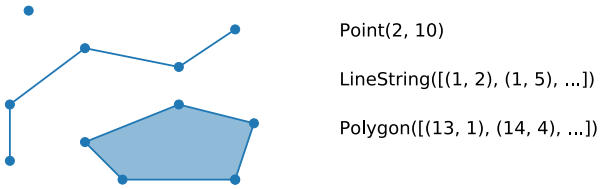


Figura 5.7. Exemplos de dados vetoriais espaciais da biblioteca shapely.

coordenadas Y, que é a latitude. Mas a terna (X,Y,Z) também contêm um valor de altura para elevação. O valor Z geralmente se refere à elevação naquele local do ponto.

As linhas de longitude têm coordenadas X entre -180 e +180 graus. O Meridiano de Greenwich (ou meridiano principal) é uma linha zero de longitude a partir da qual medimos o leste e o oeste. Longitudes positivas estão a leste do meridiano principal e as negativas estão a oeste. Na verdade, a linha zero passa pelo Observatório Real de Greenwich, na Inglaterra. Em um sistema de coordenadas geográficas, o meridiano principal é a linha que tem 0° de longitude. O Meridiano de Greenwich separa o leste do oeste da mesma forma que o Equador separa o norte do sul.

As linhas de latitudes têm valores Y que estão entre -90 e +90 graus. O equador é onde medimos o norte e o sul. Tudo ao norte do equador tem valores de latitude positivos. Considerando que, tudo ao sul do equador tem valores de latitude negativos. A Figura 5.2.5 a divisão do globo terrestre a partir do Meridiano de Greenwich e da Linha do Equador.

```
import geopandas as gpd
from shapely.geometry import Polygon, Point, LineString

p1 = Polygon([(0, 0),(1, 0),(1, 1),(0, 1)])
p2 = Polygon([(0, 0),(1, 0),(1, 1)])

p3 = Polygon([(2, 0),(3, 0),(3, 1),(2, 1)])

p4 = LineString([(0,1),(3,0),(1,1)])

p5 = Point(0.5, 0.5)

g = gpd.GeoSeries([p1, p2, p3, p4, p5])
g.plot(cmap="tab10");
```

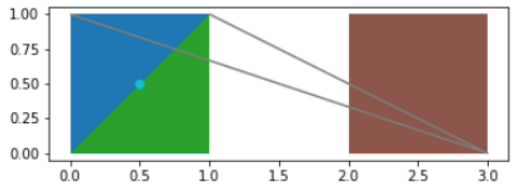


Figura 5.8. Código com exemplos de dados vetoriais espaciais da biblioteca shapely.

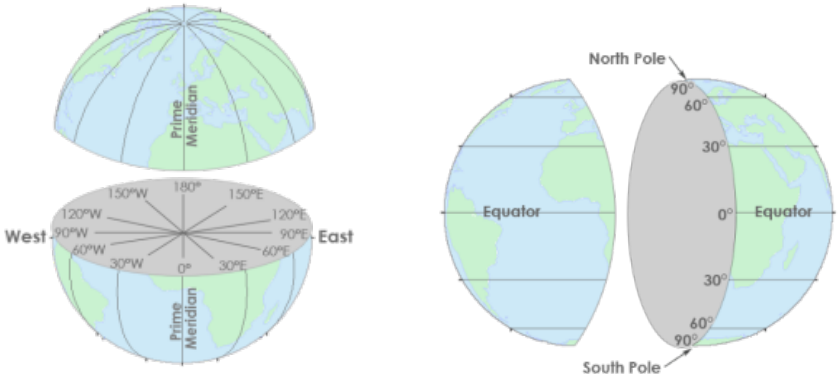
5.2.6. Projeções Espaciais

Devido à complexidade de se trabalhar com a forma real da Terra, é comumente feita uma aproximação da superfície para um modelo do do globo terrestre. Existem diversas projeções diferentes para o globo terrestre e, apesar de terem como fator comum a representação em uma superfície plana, elas variam quanto ao tipo e quanto às suas propriedades. O tipo da projeção se refere à forma geométrica utilizada para converter o globo em uma superfície plana. Comumente são utilizados três modelos para representa a superfície terrestre (Figura 5.10): projeção plana, formato esférico e formato elíptico [Bolstad 2016, Rosa and BRITO 2013, Domingues et al. 2020].

A escolha de uma projeção deve se basear na precisão desejada, no impacto sobre o que se pretende analisar e no tipo de dado disponível. A Tabela 5.1, adaptada de [Rosa and BRITO 2013] e [Domingues et al. 2020], apresenta uma análise comparativa de algumas projeções.

Para relacionar os pontos projetados a um local específico do globo terrestre é necessário adotar um Sistema de Referência de Coordenas (CRS do inglês, *Coordinate Reference Systems*). O CRS é uma forma padronizada de escrever as localizações no globo terrestre. Existe mais de um CRS, e sua escolha depende de um conjunto de fatores, como a abrangência geográfica ou mesmo em que época os dados foram coletados. Quando queremos comparar conjuntos de dados com CRSs diferentes, é importante estabelecer um CRS em comum entre eles para torná-los comparáveis.

Existem três parâmetros que, geralmente, aparecem na maioria das configurações de CRS, a projeção (`proj`), para definir a representação bidimensional do globo terrestre; elipses (`ellps`), utilizado para definir a forma da Terra; e DATUM (`datum`), que é responsável por ancorar as coordenadas da Terra, determinando as origens e a direção dos eixos.



(a) (b)

Figura 5.9. Sistema de coordenadas geográficas usado para localizar objetos sobre a Terra. (a) As linhas de longitude têm coordenadas X entre -180 e +180 graus. (b) As linhas de latitudes têm valores Y que estão entre -90 e +90 graus, adaptado de [Maling 2013].

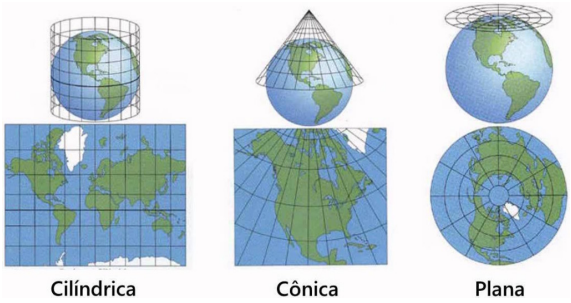


Figura 5.10. Tipos de projeções quanto a geometria de conversão [Domingues et al. 2020].

Para este minicurso, utilizaremos `latlong` e `utm` para projeções; `WGS84` e `GRS80` para elipses; e `WGS84`, utilizado pelos sistemas de GPS e `SIRGAS2000`, oficialmente utilizado pelo Brasil (pois é a melhor representação da América Latina) para o DATUM. Na Figura 5.11 temos um exemplo de configuração para CRS. Para obter uma explicação detalhada, consulte [crs].

```
crs = {
  'proj':'latlong',
  'ellps':'WGS84',
  'datum':'WGS84'
}
```

Figura 5.11. Exemplo de definição de um CRS.

Na Figura 5.12 é apresentado o CRS dos setores censitários urbanos de Teresina/PI (Figura 5.6) definidos originalmente pelo IBGE. Como é possível observar, o CRS definido é o SIRGAS 2000, pois como já mencionado, é o CRS oficialmente utilizado pelo Brasil e, portanto, é o utilizado pelo IBGE por padrão.

Já na Figura 5.13 temos um exemplo de realizar a reprojeção do CRS dos se-

Tabela 5.1. Análise comparativa das projeções.

Projeção	Classificação	Aplicações	Características da projeção
Albers	Cônica Equivalente	-cartas gerais e geográficas	-preserva áreas -garante precisão de escala -substitui com vantagens todas as outras cônicas equivalentes
Cilíndrica Conforme Equidistante Mercator	Cilíndrica Equidistante Cilíndrica Conforme	-mapas mundi -mapas em escala pequena -cartas náuticas -cartas geológicas e magnéticas -mapas mundi	-altera áreas -altera os ângulos -preserva os ângulos -mantém a forma de áreas pequenas celestes/meteorológicas
UTM	Cilíndrica Conforme	-mapeamento básico em escalas médias e grandes -cartas topográcas	-preserva ângulos -altera áreas (porém as distorções não ultrapassam 0.5%)
Gauss	Cilíndrica Conforme	-cartas topográcas -mapeamento básico em escala média e grande	-altera áreas (porém as distorções não ultrapassam 0.5%) -preserva os ângulos -similar à UTM com defasagem de 3^o de longitude entre os meridianos centrais
Estereográfica Polar	Plana Conforme	-mapeamento das regiões polares -mapeamento da Lua, Marte e Mercúrio	-preserva ângulos -preserva forma de pequenas áreas -oferece distorção de escalas

tores censitários urbanos de Teresina/PI para a projeção Mercator (`http://epsg.io/3395`), e isso é feito através da função to_crs do GeoDataFrame, linha 2. No site `http://www.spatialreference.org` é possível encontrar diversos CRSs e em vários formatos.

Uma das projeções mais utilizadas em pesquisas científicas é a projeção UTM (do inglês, *Universal Transverse Mercator*), pois possui alguns atributos que tornam a estimativa das distâncias mais precisa. A projeção UTM divide a Terra em 60 fusos ou zonas de 6^o de longitude. Para cada fuso, adota-se como superfície de projeção um cilindro transverso com eixo perpendicular ao seu meridiano central, que assume ainda o papel de longitude de origem.

Na Figura 5.14 temos uma projeção das zonas em que o Brasil está localizado (da 18 até a 25). Quando declararmos um CRS utilizando o parâmetro UTM, teremos que configurar alguns parâmetros adicionais, como a zona, o hemisfério e as unidades de

```
setores_censitarios_urbano_teresina.crs
```

```
<Geographic 2D CRS: EPSG:4674>
Name: SIRGAS 2000
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: Latin America - Central America and South America - onshore and offshore. Brazil - onshore and offshore.
- bounds: (-122.19, -59.87, -25.28, 32.72)
Datum: Sistema de Referencia Geocentrico para las AmericaS 2000
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

Figura 5.12. CRS dos setores censitários de Teresina no Piauí definidos pelo IBGE.

```
setores_censitarios_urbano_teresina = (
    setores_censitarios_urbano_teresina.to_crs("EPSG:3395")
)
ax = setores_censitarios_urbano_teresina.plot(figsize=(15,6))
ax.set_title(setores_censitarios_urbano_teresina.crs.name);
```

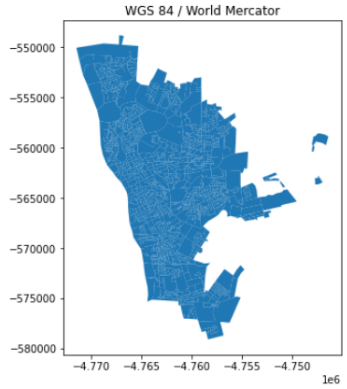


Figura 5.13. Reprojeção do CRS dos setores censitários de Teresina no Piauí para projeção Mercator.

medida que estão sendo trabalhadas. A Figura 5.15 uma possível configuração para o CRS com UTM para a cidade de Teresina/PI. Note que, para o parâmetro *zone*, escolhemos o valor 23 - que é exatamente a projeção em que está localizada a Teresina/PI. Também utilizamos o parâmetro *south*, definindo que estamos trabalhando com o hemisfério sul, e a unidade (*units*) como *m*, de "metros".

5.3. Relações e operações espaciais

Um aspecto importante dos dados geoespaciais é que podemos olhar para as relações espaciais: como dois objetos espaciais se relacionam entre si (se eles se sobrepõem, se

Figura 5.14. Representação das zonas em que o Brasil está localizado.

```
{
  'proj':'utm',
  'zone':23,
  'south':True,
  'ellps': 'GRS80',
  'units':'m',
  'no_defs': True
}
```

Figura 5.15. Exemplo, em forma de dicionário, de configuração UTM para a cidade de Teresina/PI.

cruzam, se contêm, e etc.).

As relações topológicas e teóricas de conjuntos em GIS são normalmente baseadas no modelo DE-9IM. O modelo DE-9IM expressa importantes relações espaciais que são invariantes às transformações de rotação, translação e escala. Para quaisquer dois objetos espaciais a e b, que podem ser pontos, linhas e/ou áreas poligonais, existem 9 relações derivadas de DE-9IM [Strobl 2008]:

CONTAINS: Verifica se uma representação contém completamente a outra. Inválida para a combinação ponto-linha, pois uma linha não pode estar completamente contida dentro de um ponto; porém a combinação inversa é válida.

CROSSES: Analisa se as representações se sobrepõem em algum lugar, ou seja, se as geometrias possuem pontos interiores em comum, mas não todos (uma não está contida na outra). Vale ressaltar que esta operação pode ser usada para representações com quantidade de dimensões diferentes, por exemplo uma linha e um polígono.

DISJOINT: Verifica se as representações utilizadas são disjuntas, ou seja, não compartilham nenhum ponto em comum.

EQUALS: Verifica se as duas geometrias são iguais.

INTERSECTS: Analisa se as geometrias se interceptam em algum ponto, ou seja, compartilham qualquer porção de espaço. Retorna FALSO se as geometrias forem disjuntas.

OVERLAPS: Analisa se representações de mesma dimensão se sobrepõem, mas uma não está contida na outra.

RELATE: Verifica de forma mais geral se duas representações se relacionam através de interseções nos limites, interiores ou exteriores desta, mas não são disjuntas. Esta operação é útil para verificar de uma só vez se há interseção ou se as geometrias se cruzam ou se tocam, por exemplo.

TOUCHES: Analisa se há interseção entre os limites das geometrias, mas seus interiores não se intersectam.

WITHIN: Verifica se uma geometria está dentro da outra. Representa a relação inversa de CONTAINS.

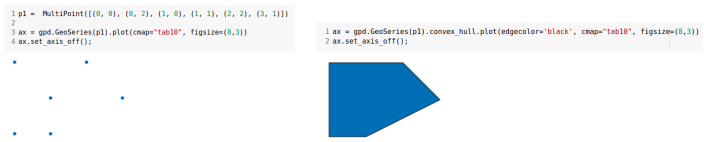
Existem também outras operações que não analisam apenas a relação entre duas geometrias, retornando VERDADEIRO ou FALSO, mas realizam operações espaciais, retornando valores ou novas geometrias como saída. Tais operações são:

BUFFER: Dada uma distância especificada pelo usuário, a operação irá gerar e retornar uma nova geometria resultante da adição de uma silhueta à geometria original (Figura 5.3).

(a) (b)

Figura 5.16. Operação BUFFER. (a) Geometria Base. (b) Geometria Resultante.

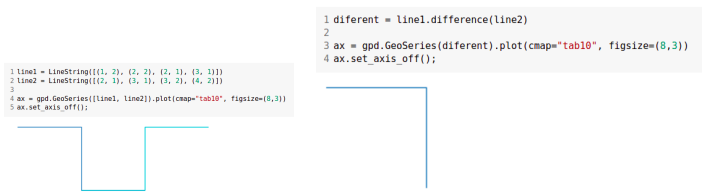
CONVEXHULL: Retorna o envoltório convexo da geometria especificada (Figura 5.3).



(a) (b)

Figura 5.17. Operação CONVEXHULL. (a) Geometria Base. (b) Geometria Resultante.

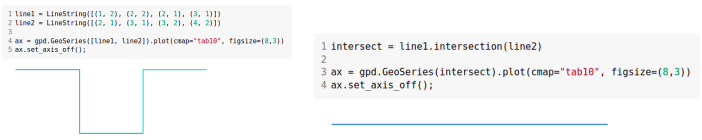
DIFFERENCE: Retorna uma geometria que contém todos os pontos que estão na representação de base mas não na geometria de comparação (Figura 5.3).



(a) (b)

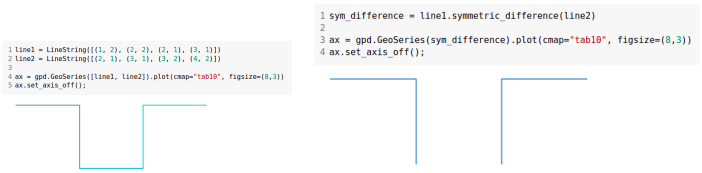
Figura 5.18. Operação DIFFERENCE. (a) Geometria Base. (b) Geometria Resultante.

INTERSECTION: Retorna a geometria que pode ser observada em ambas as representações utilizadas (Figura 5.3).



(a) (b)

Figura 5.19. Operação INTERSECTION. (a) Geometria Base. (b) Geometria Resultante.

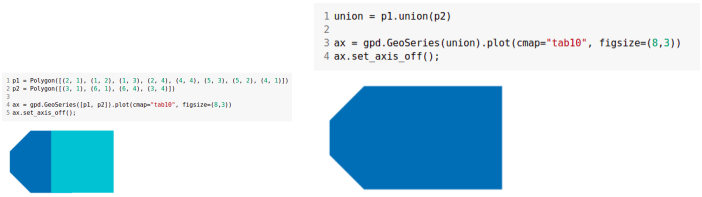


(a) (b)

Figura 5.20. Operação SYMDIFFERENCE. (a) Geometria Base. (b) Geometria Resultante.

SYMDIFFERENCE: Retorna a geometria que contém todas aquelas que não se intersectam nas representações utilizadas (Figura 5.3).

UNION: Retorna a geometria obtida com a união de todas aquelas presentes nas duas representações (Figura 5.3).



(a) (b)

Figura 5.21. Operação UNION. (a) Geometria Base. (b) Geometria Resultante.

5.4. Visualização de dados geoespaciais

A visualização geoespacial está se tornando uma maneira progressiva e sofisticada para analistas de dados ou cientistas transmitirem suas análises de forma eficiente e como já visto na Figura 5.4, o GeoPandas também pode traçar mapas, para que possamos verificar como nossas geometrias se parecem no espaço. O método principal responsável por isso é o `plot()`, uma interface de alto nível para a biblioteca *matplotlib* para construir mapas. Vale ressaltar que, em geral, todas as opções que podem ser passadas para `pyplot` do *matplotlib*, como opções de estilo por exemplo, podem ser passadas para o método `plot()`.

Vamos agora determinar os centroides dos setores censitários urbanos de Teresina/PI, criado na Figura 5.6, linhas de 1 a 3 da Figura 5.22 e em seguida criar duas

camadas, uma para os setores censitários, linha 4 da Figura 5.22, e outra para os centroides, plotando-os de preto para destacar, linha 5 da Figura 5.22. O resultado dessa operação pode ser observado na Figura 5.23

```
1 setores_censitarios_urbano_teresina['centroid'] = (
2     setores_censitarios_urbano_teresina['geometry'].centroid
3 )
4 ax = setores_censitarios_urbano_teresina['geometry'].plot(figsize=(23,15))
5 setores_censitarios_urbano_teresina['centroid'].plot(ax=ax, color="black");
```

Figura 5.22. Cálculo dos centroides dos setores censitários urbanos de Teresina/PI.

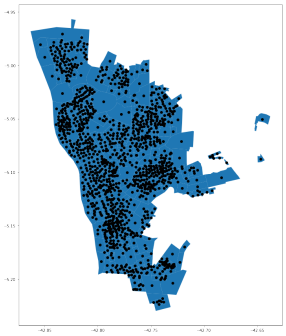


Figura 5.23. Mapa com os setores censitários urbanos de Teresina/PI com seus respectivos centroides.

Agora, vamos calcular a área de cada setor censitário urbano de Teresina/PI, linhas de 1 a 3 da Figura 5.24, e em seguida plotar essa área calculado com uma legenda, linha 4 da Figura 5.24. O resultado dessa operação pode ser observado na Figura 5.25

```
1 setores_censitarios_urbano_teresina['area'] = (
2     setores_censitarios_urbano_teresina['geometry'].area
3 )
4 setores_censitarios_urbano_teresina.plot('area', legend=True, figsize=(23,15))
```

Figura 5.24. Cálculo da área dos dos setores censitários urbanos de Teresina/PI.

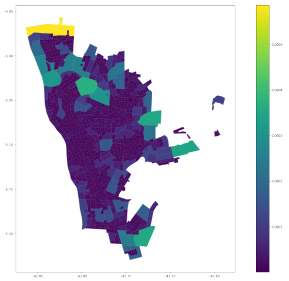


Figura 5.25. Mapa dos setores censitários urbanos de Teresina/PI graduados pela área.

Outra forma de visualização de dados geoespaciais é através de bibliotecas de visualizações de mapas interativos baseadas na web. Existem várias bibliotecas com essa finalidade, segue alguns pacotes com um exemplo para cada um:

- **Bokeh:** `https://bokeh.pydata.org/en/latest/docs/gallery/texas.html`
- **GeoViews** (outra interface para Bokeh/Matplotlib): `http://geo.holoviews.org`
- **Altair:** `https://altair-viz.github.io/gallery/choropleth.html`
- **Plotly:** `https://plot.ly/python/#maps`

Para criação dos nossos mapas interativos, utilizaremos a biblioteca Folium[11], uma biblioteca Python que possibilita a visualização geográfica interativa de dados espaciais através do *Leaflet.js*[12].

Para criar um mapa interativo com o Folium, basta importá-lo, linha 1 da Figura 5.26, e em seguida chamar o método `Map()` passando a localização em termos de latitude e longitude como parâmetro, linha 3 da Figura 5.26. No exemplo da Figura 5.26, foi utilizado às coordenadas de Teresina/PI[13] e o resultado pode ser observado na Figura 5.27.

```
1 import folium
2
3 mapa_teresina = folium.Map(location=[-5.088889, -42.801944], zoom_start = 12)
4 mapa_teresina
```

Figura 5.26. Código para criação de um mapa interativo com o Folium em Python.

Figura 5.27. Mapa interativo de Teresina/PI, plotado com o Folium em Python.

Um parâmetro interessante de mudar é o tipo de gráfico, através do parâmetro *tiles*. Vale ressaltar que um *tileset* é uma coleção de dados *raster* e vetoriais divididos em uma grade uniforme de ladrilhos quadrados. Cada *tileset* tem uma maneira diferente de

[11] Disponível em `http://python-visualization.github.io/folium/`

[12] Leaflet.js é uma biblioteca JavaScript de código aberto usada para construir mapas interativos e *mobile-friendly*. Ela utiliza dados do *OpenStreetMaps* para construir a projeção de mapas detalhados contendo informações de vias e demarcações de locais e transportes públicos [Domingues et al. 2020, Agafonkin 2014].

[13] Coordenadas de Teresina/PI disponível em `https://geohack.toolforge.org/geohack.php?pagename=Teresina&params=5_05_20_S_42_48_07_W_type:city_region:BR`

representar dados no mapa. O Folium nos permite criar mapas com diferentes *tiles* como *Stamen Terrain*, *Stamen Toner*, *Stamen Water Color*, *CartoDB Positron*. Por padrão, Folium define o *OpenStreetMap* como *tile* padrão. Na Figura 5.28 temos o mapa de Teresina/PI com o *tiles Stamen Terrain* que mostra o relevo.

```
1 mapa_teresina = folium.Map(
2     location=[-5.088889, -42.801944],
3     tiles= 'Stamen Terrain'
4 )
5 mapa_teresina
```

Figura 5.28. Mapa interativo de Teresina/PI com o *tiles Stamen Terrain*.

Como agora sabemos que cada *tileset* fornece informações de uma maneira diferente e serve a um propósito diferente, podemos sobrepô-los para obter mais informações apenas traçando um único mapa. Podemos fazer isso adicionando diferentes camadas de blocos a um único mapa. Na Figura 5.29 temos o mapa de Teresina/PI com os *tiles Stamen Terrain*, *Stamen Toner*, *Stamen Water Color*, *CartoDB Positron*, *Carto Dark Matter*, além do *OpenStreetMap*, adicionado através do método `TitleLayer()`, da linha 1 até a linha 5.

```
1 folium.TileLayer('Stamen Terrain').add_to(mapa_teresina)
2 folium.TileLayer('Stamen Toner').add_to(mapa_teresina)
3 folium.TileLayer('Stamen Water Color').add_to(mapa_teresina)
4 folium.TileLayer('cartodbpositron').add_to(mapa_teresina)
5 folium.TileLayer('cartodbdark_matter').add_to(mapa_teresina)
6 folium.LayerControl(collapsed=False).add_to(mapa_teresina)
7
8 mapa_teresina
```

Figura 5.29. Mapa interativo de Teresina/PI com vários *tiles*.

É possível observar na Figura 5.29 que foi adicionado cinco camadas de *tiles* diferentes a um único mapa e agora se tem 6 camadas diferentes de conjuntos de *tiles*. Também foi adicionado ao mapa o `LayerControl()`, linha 6 da Figura 5.29, que fornece um ícone no canto superior direito do mapa para alternar entre as diferentes camadas.

Criando Marcadores

Marcadores são um dos itens mais utilizados para marcar uma localização em um mapa. Por exemplo, quando se usa o Google Maps para navegação, é marcada a localização de origem por um marcador e o destino é marcado por outro marcador. Vale ressaltar que os marcadores estão entre as coisas mais importantes e úteis em um mapa interativo.

Folium fornece uma classe `folium.Marker()` para criar marcadores em um mapa interativo. Basta passar a latitude e longitude do local, mencionar um *pop-up* e um *tooltip* e adicioná-lo ao mapa. A plotagem de marcadores é um processo de duas etapas. Primeiro, você precisa criar um mapa básico no qual seus marcadores serão colocados e, em seguida, adicionar seus marcadores a ele. Na Figura 5.30 é definido um *array* com as coordenadas da Ponte Estaiada João Isidoro França[14], um dos mais importantes pontos turísticos da capital piauiense, linha 1, e em seguida é criado um *marker* com essa coordenada e adicionado ao mapa de Teresina, linhas de 3 a 5. O resultado do código da Figura 5.30 pode ser observado na Figura 5.31.

```
1 coordenadas_ponte_estaiada = [-5.069861, -42.802472]
2
3 folium.Marker(
4   coordenadas_ponte_estaiada, popup="<i>Ponte Estaiada</i>"
5 ).add_to(mapa_teresina)
6
7 mapa_teresina
```

Figura 5.30. Código para criação do *marker* nas coordenadas da Ponte Estaiada em Teresina/PI.

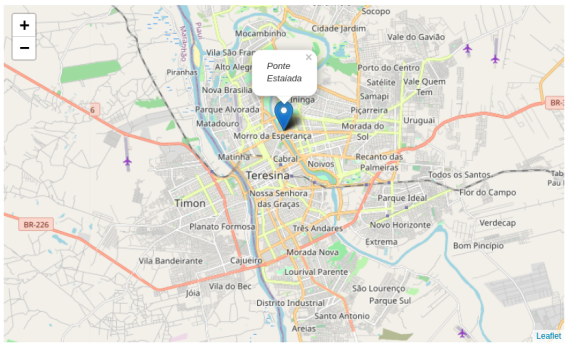


Figura 5.31. Mapa interativo de Teresina/PI com o *marker* nas coordenadas da Ponte Estaiada.

É possível também personalizar a aparência de um *marker*. O Folium fornece a classe `folium.Icon()` que pode ser usada para criar ícones personalizados para *markers*. O construtor da classe `icon()` recebe três parâmetros - `color`, `prefix` e `icon`, a cor, prefixo e ícone respectivamente. O parâmetro `color` é usado para alterar a cor do *marker*, `prefix` é usado para selecionar a origem do ícone (**fa** para *Fontawesome* e **glificon** para *Glyphicons*) e o `icon` é usado para selecionar o nome do ícone. Na Figura 5.32 temos a criação de dois *markers* personalizados para a Universidade Estadual do Piauí, da linha 1 até a linha 5, e para a Universidade Federal do Piauí, da linha 7 até a

[14]Coordenadas disponível em `https://shorturl.at/dryAF`

linha 11, e ambos adicionado ao mapa de Teresina/PI, criado anteriormente. O resultado do código da Figura 5.32 pode ser observado na Figura 5.33.

```
1 folium.Marker(
2     location=[-5.0778331015812, -42.82593947402195],
3     popup="UESPI",
4     icon=folium.Icon(color="green", prefix='fa',icon='university'),
5 ).add_to(mapa_teresina)
6
7 folium.Marker(
8     location=[-5.06143163033608, -42.79473533169222],
9     popup="UFPI",
10    icon=folium.Icon(color="red", prefix='glyphicon',icon='home'),
11 ).add_to(mapa_teresina)
12
13 mapa_teresina
```

Figura 5.32. Código com a criação de *markers* personalizados para as universidades de Teresina/PI.

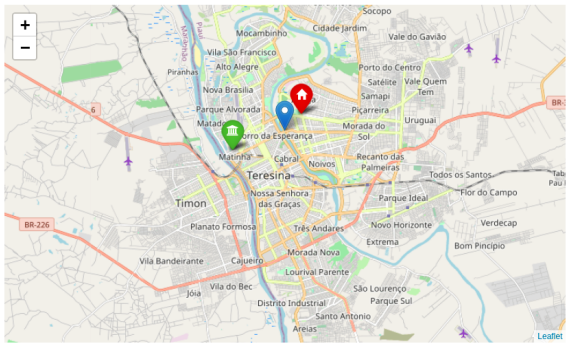


Figura 5.33. Mapa interativo com os *markers* personalizados para as universidades de Teresina/PI.

Antes de continuar a explorar algumas das funcionalidades mais comuns do Folium, vamos gerar um GeoDataFrame com 100 pontos aleatório contidos nos setores censitários urbanos de Teresina/PI, criados na Figura 5.6. Na Figura 5.34 temos um possível código para geração desse GeoDataFrame.

```
1  import random
2  from shapely.geometry import Point
3
4  teresina = setores_censitarios_urbano_teresina.dissolve()
5  minx, miny, maxx, maxy = teresina.bounds.iloc[0]
6
7  cont = 0
8  colecao_de_pontos = []
9
10 while cont < 100:
11     p = Point(random.uniform(minx, maxx), random.uniform(miny, maxy))
12     if teresina.contains(p).iloc[0]:
13         colecao_de_pontos.append({
14             'latlong': (p.y, p.x),
15             'geometry': p
16         })
17         cont += 1
18
19 pontos_aleatorios_teresina = gpd.GeoDataFrame(colecao_de_pontos)
20 pontos_aleatorios_teresina.crs = setores_censitarios_urbano_teresina.crs
```

Figura 5.34. Exemplo de código para geração de um GeoDataFrame com 100 pontos aleatório contidos nos setores censitários urbanos de Teresina/PI.

Na linha 1 da Figura 5.34 temos a importação da função *built-in* Python para geração de números aleatório segundo uma função uniforme de probabilidade. Já linha 2

é feito a importação da classe *Point* do *shapely* para criar as geometrias para o GeoDataFrame. Na linha 4 é feita a agregação das geometrias do setores censitários através da função `dissolve()` do GeoPandas, em seguida extraímos os limites da agregação resultante dos setores censitários através da função `bounds`. Nas linhas de 10 até 17 temos laço de repetição responsável por gerar um ponto aleatório a partir dos limites da agregação dos setores censitários e verificar se ele está contido dentro dos limites dos setores censitários urbanos de Teresina/PI. Após é criado um novo GeoDataFrane com esse conjuntos de pontos aleatórios, linha 19, com o mesmo sistema de coordenadas dos setores censitários, linha 20.

De posse do GeoDataFrame da Figura 5.34, podemos adicioná-lo ao mapa interativa de Teresina/PI criado na Figura 5.27, através da classe `GeoJson` do Folium (Figura 5.35).

```
1 folium.GeoJson(
2     pontos_aleatorios_teresina,
3     marker= folium.Marker(
4         icon=folium.Icon(
5             color="black",
6             prefix='fa',
7             icon='bug'
8         )
9     )
10 ).add_to(mapa_teresina)
11
12 mapa_teresina
```

Figura 5.35. Adição do GeoDataFrame com 100 pontos aleatório ao mapa interativo de Teresina/PI.

O primeiro parâmetro, obrigatório, do construtor da classe `GeoJson` do Folium é os dados que queremos visualizar, no nosso exemplos, o conjunto de pontos aleatórios criados na Figura 5.34, além de um *marker* customizado para eles. O resultado do código da Figura 5.35 pode ser observado na Figura 5.36.

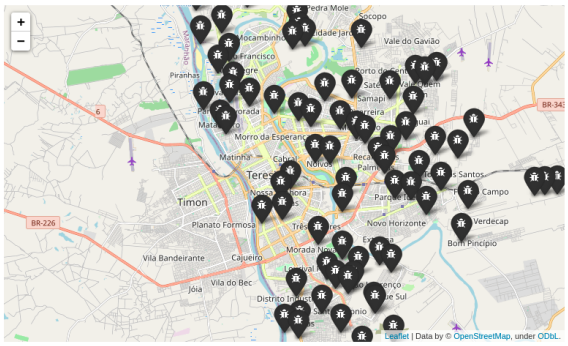


Figura 5.36. Mapa interativo de Teresina/PI com 100 *markers* criados aleatoriamente.

Como pode ser observado na Figura 5.36, os marcadores parecem estar empilhados e um pouco bagunçados. Uma forma de organizar os *markers* é através de *clusters* de *markers*. O Folium disponibiliza um *plugin* com essa finalidade, o **MarkerCluster**. A Figura 5.37 é apresentado como utilizar o *plugin* **MasterCluster** com o nosso GeoDataFrame criado na Figura 5.34.

```
1  from folium.plugins import MarkerCluster
2
3  marker_cluster = MarkerCluster().add_to(mapa_teresina)
4
5  folium.GeoJson(
6      pontos_aleatorios_teresina,
7      marker= folium.Marker(
8          icon=folium.Icon(
9              color="black",
10             prefix='fa',
11             icon='bug'
12         )
13     )
14 ).add_to(marker_cluster)
15
16 mapa_teresina
```

Figura 5.37. Exemplo de utilização do *plugin* MasterCluster do Folium.

Na linha 1 da Figura 5.37 é feita a importação do *plugin* **MasterCluster**, na linha 3 é criado um objeto **MasterCluster** e adicionamos ele ao mapa interativo de Teresina/PI. Ao invés de adicionar o nosso GeoDataFrame de pontos aleatórios direto no mapa, como é feito na Figura 5.35, adicionamos o nosso GeoDataFrame de pontos ao objeto **MasterCluster** criado na linha 3. O resultado do código da Figura 5.37 pode ser observado na Figura 5.38.

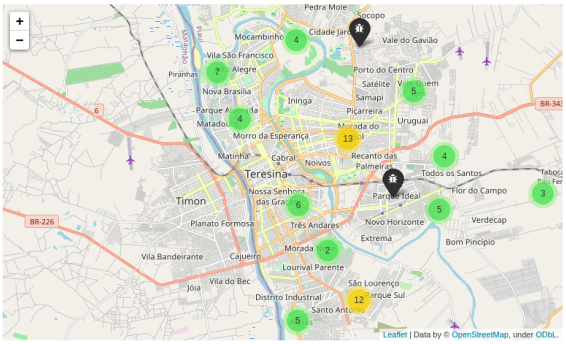


Figura 5.38. Mapa interativo de Teresina/PI com 100 *markers* criados aleatoriamente agrupados.

Criando *HeatMaps*

É possível também implementar *HeatMaps*, ou mapas de calor, usando Folium. Um *HeatMap* é uma representação gráfica de dados que usa um sistema de codificação de cores para representar diferentes valores. Isso é útil para monitorar a intensidade das estatísticas regionais com mais facilidade em uma determinada região por exemplo.

Para criação de um *HeatMap* o Folium disponibiliza um *plugin* chama justamente *HeatMap* e para exemplificar, vamos criar um *HeatMap* com o GeoDataFrame criado na Figura 5.34. Na Figura 5.39 temos o código para essa finalidade. Na linha 1 é feita a importação do *plugin* do Folium *HeatMap* responsável por criar o mapa de calor. Já na linha 3 é feita a criação do *HeatMap* passando como parâmetro uma lista de pontos na forma `[lat, lng]` que se deseja plotar, também é possível passar uma lista na [lat, lng, weight]) ou fornecer um `numpy.array (n,2)` ou `(n,3)`. O resultado do código da Figura 5.39 pode ser observado na Figura 5.40.

```
from folium.plugins import HeatMap

HeatMap(pontos_aleatorios_teresina['latlong'].tolist()).add_to(mapa_teresina)
mapa_teresina
```

Figura 5.39. Código para criação de um *HeatMap*.

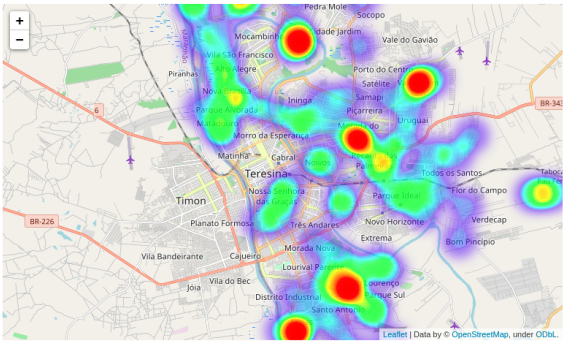


Figura 5.40. Mapa de calor de Teresina/PI com 100 *markers* criados aleatoriamente.

Criando Mapas Coropléticos

Muitas vezes lidamos com dados espaciais cuja a localização está associada a áreas delimitadas por polígonos. Isso ocorre na maioria das vezes quando não se dispõe da localização exata dos eventos, sendo portanto agregados por municípios, bairros ou setores censitários. A forma usual de apresentação de dados agregados por áreas é através do **mapas coropléticos** ou coloridos com o padrão espacial do fenômeno [Monteiro et al. 2004].

É possível criar um mapa coroplético através do parâmetro `style_function` da classe `GeoJson` do Folium, fornecendo uma função que possa especificar um estilo dependendo do recurso que se queira mapear. Para exemplificar a criação de mapas coropléticos, utilizaremos a coluna `area` do GeoDataFrame de setores censitários urbanos de Teresina/PI criada na Figura 5.24. A Figura 5.41 é apresentado uma possível solução.

```
from branca.colormap import linear

colormap = linear.YlOrRd_04.scale(
    setores_censitarios_urbano_teresina['area'].min(), setores_censitarios_urbano_teresina['area'].max()
)

colormap.caption = "Escala de Cor para Área dos Setores Censitários"
colormap.add_to(mapa_teresina)

area_dict = setores_censitarios_urbano_teresina.set_index("CD_SETOR")["area"]

folium.GeoJson(
    setores_censitarios_urbano_teresina,
    name="area",
    style_function=lambda feature: {
        "fillColor": colormap(area_dict[feature['properties']["CD_SETOR"]]),
        "color": "black",
        "weight": 1,
        "dashArray": "5, 5",
        "fillOpacity": 0.3,
    },
).add_to(mapa_teresina)

mapa_teresina
```

Figura 5.41. Código de exemplo para criação de um mapa coroplético.

Inicialmente é feito a importação da função que utilizaremos para mapear um valor para uma cor RGB (da forma `#RRGGBB`), Nas linhas de 3 a 5 criamos a nossa paleta de cores de amarelo até vermelho, com o limite inferior como sendo a menor área do e o limite superior como sendo a maior área, em seguida é definido uma legenda para a nossa

paleta, linha 7 e depois ela é adicionada ao nosso mapa interativo de Teresina/PI, linha 8. Já na linha 10 é criado um dicionário para mapear o setor censitário à sua respectiva área. Entre as linhas 15 e 21, é definida uma função anônima que mapeia para cada setor censitário (*feature* do GeoJSON) um dicionário com a cor RBB e outros parâmetros de definição do nosso mapa coroplético. O resultado do código da Figura 5.41 pode ser observado na Figura 5.42.

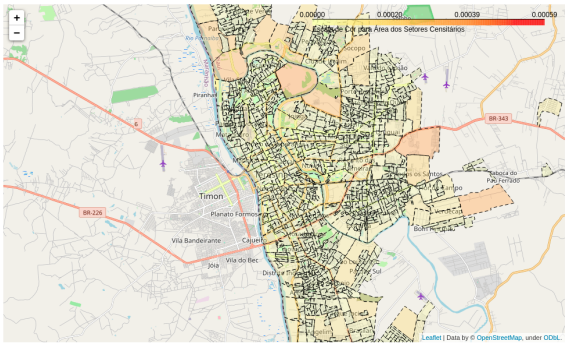


Figura 5.42. Mapa coroplético dos setores censitários urbanos de Teresina/PI a partir da sua respectiva área.

O Folium também disponibiliza uma classe `Choropleth` para criação de mapas corropléticos de forma mais rápida. Assim como na classe `GeoJson`, é possível fornecer a ela um nome de arquivo, um dicionário ou um GeoDataFrame. A Figura 5.43 apresenta um exemplo de como usá-la, utilizando a coluna `area` do GeoDataFrame de setores censitários urbanos de Teresina/PI do exemplo anterior.

```
1  folium.Choropleth(
2      geo_data = setores_censitarios_urbano_teresina,
3      data = setores_censitarios_urbano_teresina[["CD_SETOR", "area"]],
4      columns = ["CD_SETOR", "area"],
5      key_on = "feature.properties.CD_SETOR",
6      fill_color = "YlOrRd",
7      fill_opacity = 0.3,
8      legend_name = "Escala de Cor para Área dos Setores Censitários",
9  ).add_to(mapa_teresina)
10
11 mapa_teresina
```

Figura 5.43. Código de exemplo para criação de um mapa coroplético por meio da classe Choropleth.

Na linha 2 da Figura 5.43 o parâmetro `geo_data` define a origem das geometrias que serão utilizadas, no exemplo utilizamos GeoDataFrame com os setores censitários urbanos de Teresina/PI. Já o parâmetro `data` é definido um novo DataFrame com o código do setor censitário, que será a chave para vinculação ao GeoJSON, e o valor da área, como dado para o mapa coroplético, linha 3 da Figura 5.43. O parâmetro `colmns` é especificado a chave de vinculação dos dados e a coluna de dados que no neste exemplo é a área, linha 4 da Figura 5.43. O parâmetro `key_on` é definida a variável do `geo_data` do GeoJSON para vincular os dados, linha 5 da Figura 5.43. Note que esse parâmetro deve começar com '*feature*' e estar em notação de objeção *JavaScript*, como por exemplo '*feature.id*' ou '*feature.properties.statename*'. Os demais parâmetros utilizados no construtor da classe define a cor da área a ser preenchida, sua opacidade e a legenda da paleta de co-

res utilizada, respectivamente. O resultado do código da Figura 5.43 pode ser observado na Figura 5.44.

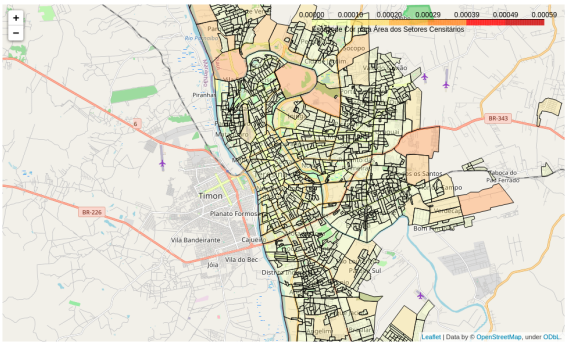


Figura 5.44. Mapa coroplético dos setores censitários urbanos de Teresina/PI a partir da sua respectiva área criado com a classe Choropleth.

O Folium disponibiliza diversos outros recursos e *plugins* e demonstrar todos eles está fora do escopo deste minicurso, no entanto, no seguinte link `https://shorturl.at/mvEIP` é possível encontrar uma lista de vários *notebooks* de exemplos com todos eles.

5.5. Considerações finais

Este capítulo apresentou os conceitos gerais sobre a Análise de Dados Geoespaciais através da linguagem de programação Python e alguns exemplos de como extrair conhecimento e valor por meio de bibliotecas como `pandas`, `geopandas`, `shapely`, `pyproj`, `matplotlib`, `cartopy`, dentre outras.

A Seção 5.2, foi apresentado alguns conceitos sobre um dos formatos mais utilizados em aplicações de análise geoespaciais, Dados Vetoriais, as ferramentas e pacotes Python necessários para começar a análise de dados geoespaciais, além dos conceitos centrais da utilização de GeoDataFrames. Já a Seção 5.3 apresentou as relações e operações para dois objetos espaciais. Por fim, a Seção 5.4 discutiu as formas de visualização que podem ser utilizadas no estudo e na aplicação de dados geoespaciais. Muitas outros aspectos, ferramentas e técnicas de análise de dados geoespaciais poderiam ter sido abordados, porém, como é apenas uma introdução esses e outros pontos ficará para trabalhos futuros.

Agradecimentos

Este trabalho foi realizado com apoio financeiro da CAPES, CNPq, Fundação de Amparo à Pesquisa e ao Desenvolvimento Científico e Tecnológico do Maranhão (FAPEMA) e Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo 2020/16578-5. Agradecemos também ao ICMC-USP e ao LCR por oferecerem a infraestrutura necessária para este estudo.

Referências

[crs] Coordinate reference systems. https://docs.qgis.org/2.8/en/docs/gentle_gis_introduction/coordinate_reference_systems.html. Acessado em: 10-09-2021.

[Agafonkin 2014] Agafonkin, V. (2014). Leaflet: an open-source javascript library for mobile-friendly interactive maps. *Accessed September*, 21:2020.

[Bolstad 2016] Bolstad, P. (2016). *GIS fundamentals: A first text on geographic information systems*. Eider (PressMinnesota).

[Coelho 2017] Coelho, A. S. (2017). Introdução a análise de dados com python e pandas. *Anais Eletrônicos ENUCOMP*, pages 862–876.

[Domingues et al. 2020] Domingues, A., Silva, F., Santos, L., Souza, R., Coimbra, G., and Loureiro, A. A. F. (2020). Dados geoespaciais: Conceitos e técnicas para coleta, armazenamento, tratamento e visualização. *Sociedade Brasileira de Computação*.

[Lawhead 2015] Lawhead, J. (2015). *Learning geospatial analysis with Python*. Packt Publishing Ltd.

[Lopes et al. 2019] Lopes, G. R., Almeida, A. W. S., Delbem, A. C., and Toledo, C. F. M. (2019). Introdução à análise exploratória de dados com python. In *Minicursos ERCAS ENUCMPI 2019*, pages 160–176, Porto Alegre, RS, Brasil. SBC.

[Maling 2013] Maling, D. H. (2013). *Coordinate systems and map projections*. Elsevier.

[Monteiro et al. 2004] Monteiro, A. M. V., Câmara, G., Carvalho, M., and Druck, S. (2004). Análise espacial de dados geográficos. *Brasília: Embrapa*.

[Pimentel et al. 2021] Pimentel, J. F., Oliveira, G. P., Silva, M. O., Seufitelli, D. B., and Moro, M. M. (2021). Ciência de dados com reprodutibilidade usando jupyter. *Sociedade Brasileira de Computação*.

[Rosa and BRITO 2013] Rosa, R. and BRITO, J. L. S. (2013). Introdução ao geoprocessamento. *UFU: Apostila. Uberlândia*.

[Shen 2014] Shen, H. (2014). Interactive notebooks: Sharing the code. *Nature News*, 515(7525):151.

[Strobl 2008] Strobl, C. (2008). Dimensionally extended nine-intersection model (de-9im).

[Zheng et al. 2014] Zheng, Y., Capra, L., Wolfson, O., and Yang, H. (2014). Urban computing: concepts, methodologies, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(3):1–55.