**Lab Report**

Title: GIS 5571 Lab 1

Notice: Dr. Bryan Runck

Author: Eric Gibson

Date: October 10, 2023

**Project Repository:** *<if applicable weblink to public repository>*

**Google Drive Link:** *<if applicable with data, notebooks, etc.>*

**Time Spent:** *<report to the nearest quarter hour>*

**Abstract**

This lab introduces how to connect to application programming interfaces (APIs) and create an extract, transform, and load (ETL) code within Python. A comparison will be made between the Google Places, Minnesota Geospatial Commons, and North Dakota Agricultural Weather Network (NDAWN) APIs. An ETL code will also be created which acquires the data, performs a spatial join on the data, and adds the data to a geodatabase. The data used includes a GeoJSON from the Google Places API containing point geometry of coffee shops within Minneapolis and Saint Paul and a shapefile from the Minnesota Geospatial Commons containing all counties within Minnesota. Utilizing a Jupyter Notebook, the “requests” library was initially used to acquire the data from these APIs, then the “Geopandas” library was primarily used to manipulate and display the data for visualizations of the results. The Google Places API was the most easy-to-use and provided the most functionality of the two other APIs within this lab. The Minnesota Geospatial Commons API was less simple, though still provided more functionality than the NDAWN API, which was unable to be connected with.

**Problem Statement**

Within this lab, I will first be creating a pipeline using a Jupyter Notebook that extracts two datasets (using only two of the three APIs), transforms both datasets to the same coordinate system, spatially joins them, displays the heading of the joined attribute table, and saves the dataset to a geodatabase. I believe starting with creating the code to acquire the data from the API will be more beneficial since it will give me a feel for each API before comparing. Next, I will be comparing and contrasting the conceptual models for the Minnesota Geospatial Commons, Google Places, and NDAWN application programming interfaces (APIs). The differences in the quantity and quality of each API will be compared to find which APIs work the best and provide the most features between the three mentioned.

|  |  |
| --- | --- |
| Part 1 | * Create a Jupyter Notebook that   + Extracts 2 datasets from 2 of the 3 APIs   + Transforms both datasets to the correct coordinate system   + Spatially joins both datasets   + Displays the heading (first 5 rows) of the attribute table   + Saves the datasets to a geodatabase |
| Part 2 | Compare and contrast the Minnesota Geospatial Commons, Google Places, and NDAWN API conceptual models by utilizing various features within each API as well as determining the ease of access to each API. |

*Table 1. Elements for initializing the solution to the problem*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **Requirement** | **Defined As** | **(Spatial) Data** | **Attribute Data** | **Dataset** | **Preparation** |
| 1 | Minnesota Geospatial Commons | An example of an API with only Minnesota GIS data | Contains GIS data for the entire state of Minnesota; Uses CKAN API | Provides population, land use, natural resources, and many other attributes | [Mn Geospatial Commons](https://gisdata.mn.gov/content/?q=help/api) | Compare with other APIs and create a pipeline to gather data from this API |
| 2 | Google Places | An example of a broadly used API for many different purposes | Can be used with many different datasets; API tends to seem more user friendly | Provides ways to view and enhance data from other sources as well as providing broad GIS data | [Google Places](https://developers.google.com/maps/documentation/places/web-service/overview) | Study and compare with other APIs before creating a pipeline to gather data from this API |
| 3 | NDAWN | An API used specifically for accessing weather data from North Dakota | Particularly focuses on climatic processes and data within North Dakota | Provides access to data describing temperature and precipitation patterns | [NDAWN](https://ndawn.ndsu.nodak.edu/) | Compare the functionality with other APIs and create a pipeline to gather data from this API |
| 4 | Scripting | Use a Jupyter Notebook to access, transform, join, present, and save GIS data from these APIs | This will include 2 datasets from the Minnesota Geospatial Commons, Google Places, and NDAWN | The attribute data could range a lot from census data in Minnesota to weather data in North Dakota because this is more for practice with APIs | Link to datasets I collect | Create a Jupyter Notebook with all libraries and linkages I need to access each API |

**Input Data**

*Describe the data in two paragraphs max. Fill out the table.*

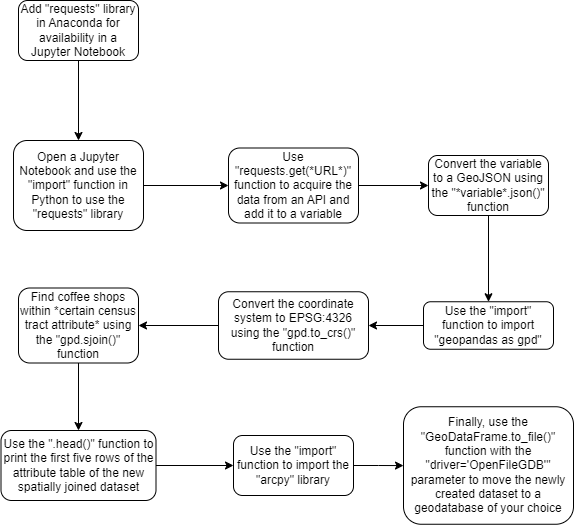
The first dataset was acquired using the Google Places API via searching for all coffee shops located in Minneapolis and Saint Paul, Minnesota. Using the link in the table below, a pipeline was built in Python to bring in the GeoJSON with the address, name, rating, opening hours, and geometry (in latitude and longitude coordinates). This allows for the data to be pulled in and consistently updated through Google Maps.

*Table 2. Data acquired from different APIs to use in analysis*

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Title** | **Purpose in Analysis** | **Link to Source** |
| 1 | Coffee Shops in Minneapolis | Provide a coffee shop dataset to determine the number of coffee shops located within a specific area in Minneapolis. | https://maps.googleapis.com/maps/api/place/textsearch/json?query=coffee%20shops%20Minneapolis%20and%20Saint%20Paul&inputtype=textquery&fields=fomatted\_address%2Cname%2Crating%2Copening\_hours%2Cgeometry&key=AIzaSyB1XJ\_8b3ZcM4ZSXtWhM-fmZwULZxC5s6c |
| 2 | Minnesota Counties | Provide the data to perform a spatial join with the coffee shops dataset (points), providing a dataset to be added to a newly created geodatabase. This spatial join will select all of the coffee shops within Hennepin County | https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us\_mn\_state\_dot/bdry\_counties/shp\_bdry\_counties.zip |

**Methods**

*Figure 1. Data flow diagram describing the process using Python code.*



This lab is meant to provide an introduction to requesting data from an API, manipulate the data, and save the data to a geodatabase. This allows datasets to be updated whenever the author of the data updates it, instead of simply downloading data locally. This allows for manipulation and processing of the most current and accurate data from the original source of the information. This lab will also be performed within Jupyter Notebooks, as they will provide the ability to save and comment for easy editing in the future.

Initially, it is important to use the “requests” library within Python to perform the “requests.get()” function which acquires data from a specific URL, in this case the API URL. For the purposes of this lab, the Google Places and Minnesota Geospatial Commons APIs will be chosen to connect to and acquire data from. Both of these APIs seemed to be more advanced than the NDAWN API, with the Google Places API being the most advanced with many ways of collecting and presenting the data. Once these datasets are collected, a GeoJSON will be created using the “geojson.dumps()” function, which creates a GeoJSON that can be read with Geopandas and formed into a spatial dataset. This GeoJSON, however, must first have the specific geometry specified by Geopandas, or a spatial join will not work on the data. To do that, a simple for loop will be created using the “geopandas.Point()”, “geopandas.Feature()”, and “geopandas.FeatureCollection()” functions. The dataset from the Minnesota Geospatial Commons, however, will instead be brought in via the shapefile’s URL using the same “requests.get()” function. Since geopandas can use shapefiles for presenting spatial data, manipulating a GeoJSON to have specific geometry will not be needed (I am worried I did this wrong and did not bring in data from the API).

Once that is completed, both datasets will be transformed to the same coordinate system: EPSG:4326 (WGS 84). This will be performed using Geopandas once again with the “geopandas.to\_crs()” function. The datasets cannot be accurately spatially joined if it was not for this transformation. Next, the spatial join will be done to filter all of the coffee shops located within Hennepin County using the “geopandas.sjoin()” function with the “how=’inner’” parameter. This parameter is essentially a “within” function that could be performed with geoprocessing tools in a GIS software.

**Results**

Utilizing the Google Places general API URL, which searched for coffee shops with Minneapolis, along with my Google Places API Key, I was able to create a URL which directly corresponded to a GeoJSON with all the coffee shops located within the cities of Minneapolis and Saint Paul. This URL can be viewed above within the **Input Data** section in the *Coffee Shops in Minneapolis* row. Once this original GeoJSON was converted to be used by Geopandas, the result was a feature class of points located at every coffee shop within Minneapolis. The next page displays a visual of the dataset using Geopandas:

*Figure 2. Geopandas plot of coffee shops within Minneapolis and Saint Paul using an OpenStreetMap basemap.*

A map of a city

Description automatically generated

Each point represents a coffee shop which would be displayed if one searched “coffee shops in Minneapolis and Saint Paul” into Google Maps. This query was specifically chosen to demonstrate the spatial join since some of these coffee shops are located outside of Hennepin County. When simply querying for coffee shops within Minneapolis or Minnesota, only coffee shops within Hennepin County were output (likely based on where I am located). This way, the spatial join will filter out any coffee shops outside of Hennepin County.

Hennepin County was also filtered out using a Geopandas geodataframe from the shapefile and, finally, a spatial join was performed, selecting only coffee shops within Hennepin County. Both plots can be visualized below.

A map of a large area with black dots

Description automatically generatedA map of the state of minnesota

Description automatically generated*Figures 3 & 4. Geopandas plot of Minnesota counties and Hennepin County coffee shops using a OpenStreetMap basemap.*

The map on the right simply depicts the counties within Minnesota directly from the shapefile acquired from the Minnesota Geospatial Commons. The map on the left depicts the coffee shops within Hennepin County after the spatial join had been performed with the black points representing the coffee shops and the blue polygon representing Hennepin County. These points also contain the data from Hennepin County as well, meaning the join worked with the geodataframes (This cannot be displayed here due to needing to scroll to see the number of columns. However, this can be viewed within a Jupyter Notebook utilizing the Python code from Github.).

Finally, a geodatabase was able to be created using the “.to\_file()” function along with the “driver=’OpenFileGDB’” parameter within Geopandas. This enabled the features to be displayed in ArcGIS Pro and organized into a map as depicted below.

*Figure 5. Screenshot of ArcGIS Pro showing the created file geodatabase and spatially joined coffee shops.*

A map of a country

Description automatically generated

*Figure 6. Map created in ArcGIS Pro depicting the Hennepin County coffee shops returned from the Google Places API.*

A map of a large area

Description automatically generated

**Results Verification**

This lab introduced gathering data from an API, manipulating it, and adding the manipulated dataset to a file geodatabase. The Geopandas library was primarily used to display and manipulate the data, so the verification of the results can be seen within the corresponding Python code in Github, as well as in the Geopandas plots within the **Results** section of this lab. In order to demonstrate how to perform a spatial join using Geopandas, a dataset utilizing coffee shops located within and outside Hennepin County was used. Using the Google Places API, a search query was successfully made for delivering the coffee shop information in GeoJSON form. While this GeoJSON did not store geometry data, a simple function was created within Python utilizing this initial GeoJSON to provide Geopandas with geometry data for displaying the points and eventually performing the spatial join. This new function seemed to work well as depicted above within **Figures 2 and 4**, which display points at each coffee shop location.

The spatial join was successful in its execution, outputting only the coffee shops located within Hennepin County as depicted in **Figure 4**. As mentioned in the **Results** section, the attribute tables were also joined, adding the Hennepin County data to the coffee shops data, confirming the spatial join. This spatial join was relatively simple using the “.sjoin()” function within the Geopandas library, which turned out to be incredibly useful when creating this ETL pipeline, at least for acquiring, analyzing, and displaying the data.

Although some issues were run into when moving the data to a geodatabase and there still is an error when running the code, this error does not seem to affect the data being input into the geodatabase and can be successfully accessed in GIS software, like ArcGIS Pro as shown in **Figure 5**. The data within the geodatabase file contains all of the attributes depicted within the geodataframe and can be displayed with the correct geometry on a map, as depicted in **Figure 6**. Moving the data into a geodatabase was fairly simple with the “driver=’OpenFileGDB’” parameter, though it is important to note that the Fiona library was be updated alongside GDAL 3.6.0 or better in order to access that specific parameter. This can be done by simply updating GDAL to the latest version, then uninstalling and reinstalling the Fiona library in command prompt. The Arcpy library was avoided throughout all of these steps since it can only be accessed through ESRI framework and this lab specifically focused on creating a pipeline using Jupyter Notebooks.

**Discussion and Conclusion**

The three conceptual models for each API are completely different, with the Google Places API likely the simplest to use. The Google Places API allows for a direct pipeline to a GeoJSON, which can then be easily analyzed and added to a geodatabase. As shown above in the **Input Data** section, there is a specific URL which can be referenced and brought into a Jupyter Notebook using the “requests” library in Python. The URL is organized like this:

https://maps.googleapis.com/maps/api/\*service\*/\*function\*/\*response (how the data is delivered)\*”?”\*inputs for data search and API key\*

Due to the organization of each element within the URL, acquiring data from the Google Places API is fairly straightforward and the API model I found the most easy to use. On top of that, Google Places has many more functionalities when it comes to analyzing the data than the other two API conceptual models. For example, the Google Places API can monitor traffic via people using Google Maps, so it is possible to create maps depicting traffic during certain times of the day, month, year, etc. This is just one small example of what the Google Places API can do, which the others cannot. There was a limitation which was found for this specific lab, however, and that was the specific search query used. When searching for specific businesses, sites, parks, and other locations in Google Maps, a maximum of nineteen locations are output. This works well for navigation, as it does not overwhelm the user when searching for places of interest, but this does not work well when gathering data to be used in a GIS analysis. In order to provide an accurate and well-researched project, it is important to have as many data points as possible, so only outputting nineteen locations could allow for inaccurate information.

The Minnesota Geospatial Commons API conceptual model was a little more difficult to work with, as it did not provide a GeoJSON for a simple pipelining process. Instead, the Minnesota Geospatial Commons uses a CKAN API, which delivers JSON data with mostly metadata of a dataset. This made pipelining from the API directly into a Jupyter Notebook much more difficult and instead the URL to the specific shapefile within the dataset needed to be brought in using the “requests”module. This at least did allow for easy merging with the “Geopandas”library, since it can easily read shapefiles. However, I am unsure if this took the data from the API, or if it simply brought in a shapefile, leading me to believe I may have done this incorrectly.

The NDAWN API was unable to be accessed utilizing the “requests.get()” function with a URL to the spatial data. There was no mention of the specific API used within NDAWN and no way of requesting data from the API, so it is likely the NDAWN API is the least utilized and has the least functionality of both other APIs. This could be due to an error on my end and simply not finding any information about how to use the API, but thankfully the other two APIs were able to accessed and collected from.

Transitioning to creating the code itself, there were many issues and errors which occurred when learned to create an ETL pipeline. Initially, acquiring data from the Google Places and Minnesota Geospatial Commons’ APIs was difficult since, without experience, creating the URLs to pull from took some time. As mentioned previously, only a JSON with metadata information of the data was able to be accessed using the built-in CKAN API in the Minnesota Geospatial Commons. This made it extremely difficult to acquire actual spatial data with correct geometries from the Commons and, eventually, it was decided to pull the data from the actual link of the shapefile itself. Though it is not believed to be from the actual API, it should still update along with updates provided by the Minnesota Department of Natural Resources (MNDNR), allowing for up-to-date county information. The Google Places API was much easier to use since the “requests.get()” function could directly output a GeoJSON. It also provides much more functionality outside of the previously mentioned limitation of just outputting nineteen locations. Still, a for loop was needed to manipulate the gathered GeoJSON so it could be used with Geopandas, as the coordinates are not in the correct format when they are first acquired directly from the API.

Another large issue with the code which took awhile to find a solution for was adding the data to a geodatabase. This is due to the lack of information on how to add a geodataframe from Geopandas to a geodatabase as well as differences in versions of libraries and field type issues. There is plenty of information describing how to utilize Arcpy for moving features to a geodatabase, but little information regarding using Geopandas to perform the same action. On top of this, versions of GDAL and Fiona were incompatible with the “driver=’OpenFileGDB’” parameter and some of the fields within the geodataframe were a “list” type, which cannot be used with the “to\_file()” function in Geopandas. Once GDAL and Fiona were updated and a for loop changing all columns to a “string” type was completed, the features were finally added to the established geodatabase.

This lab was a large learning curve, as I had little experience acquiring data from an API, but through the problems and successes, I believe I was able to create a relatively viable pipeline for easy gathering of data. I also learned different types of APIs and how well they work for acquiring and manipulating data along with further functionalities for presenting it. To move the code I created further, I believe I should add some “input()” functions and change some lines so I can simply copy and paste API URLs for future projects and labs.

**References**

Minnesota Department of Natural Resources*.* “County Boundaries, Minnesota.” *Minnesota Geospatial Commons*, 15 July 2013, Accessed 27 Sept. 2023.

Chu, Justin. “Overview.” *Google*, developers.google.com/maps/documentation/places/web-service/overview. Accessed 09 Oct. 2023.

North Dakota Agricultural Weather Network. “NDAWN - North Dakota Agricultural Weather Network.” *NDAWN Center*, ndawn.ndsu.nodak.edu/.

Multiple prompts. ChatGPT, 13 Feb. version, OpenAI, Sept. - Oct. 2023, chat.openai.com.

Atwood, Jeff, and Joel Spolsky. “Where Developers Learn, Share, & Build Careers.” *Stack Overflow*, stackoverflow.com/.

**Self-score**

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Description** | **Points Possible** | **Score** |
| **Structural Elements** | All elements of a lab report are included **(2 points each)**:  Title, Notice: Dr. Bryan Runck, Author, Project Repository, Date, Abstract, Problem Statement, Input Data w/ tables, Methods w/ Data, Flow Diagrams, Results, Results Verification, Discussion and Conclusion, References in common format, Self-score | 28 |  |
| **Clarity of Content** | Each element above is executed at a professional level so that someone can understand the goal, data, methods, results, and their validity and implications in a 5 minute reading at a cursory-level, and in a 30 minute meeting at a deep level **(12 points)**. There is a clear connection from data to results to discussion and conclusion **(12 points)**. | 24 |  |
| **Reproducibility** | Results are completely reproducible by someone with basic GIS training. There is no ambiguity in data flow or rationale for data operations. Every step is documented and justified. | 28 |  |
| **Verification** | Results are correct in that they have been verified in comparison to some standard. The standard is clearly stated **(10 points)**, the method of comparison is clearly stated **(5 points)**, and the result of verification is clearly stated **(5 points)**. | 20 |  |
|  |  | 100 |  |