Lab Report

Title: Lab1

Notice: Dr. Bryan Runck

Author: Rob Hendrickson

Date: 10/9/2022

Repository: https://github.com/RwHendrickson/GIS5571/tree/main/Lab01

Time Spent: 15 hours

Abstract

In this lab, I explored 3 different spatial web API's and how to make them interoperable. The problem statement section explains why this is an important skill for a GIS professional as well as the general workflow of this assignment. The input data section will describe the different datasets acquired in this exercise. The methods section includes detailed visual and textual descriptions of the workflow conducted. The results section will present some visualizations showing the datasets that were created. This report concludes with a discussion on the different API conceptual models and their pros and cons.

Problem Statement

Web API's are a way for data scientists to programatically acquire information for their analyses. One challenge with this process is that there is not a standardized way to design an API. Furthermore, when spatial information is included, the format in which it is returned varies significantly. This lab compares three spatial web API's: Minnesota Geospatial Commons, Google Places, and North Dakota Agricultural Weather Newtowrk (NDAWN). After growing familiar with each interface, an ETL routine is developed which makes the datasets interoperable. We conclude by spatially joining information between each API and saving the integrated data to an ESRI geodatabase.

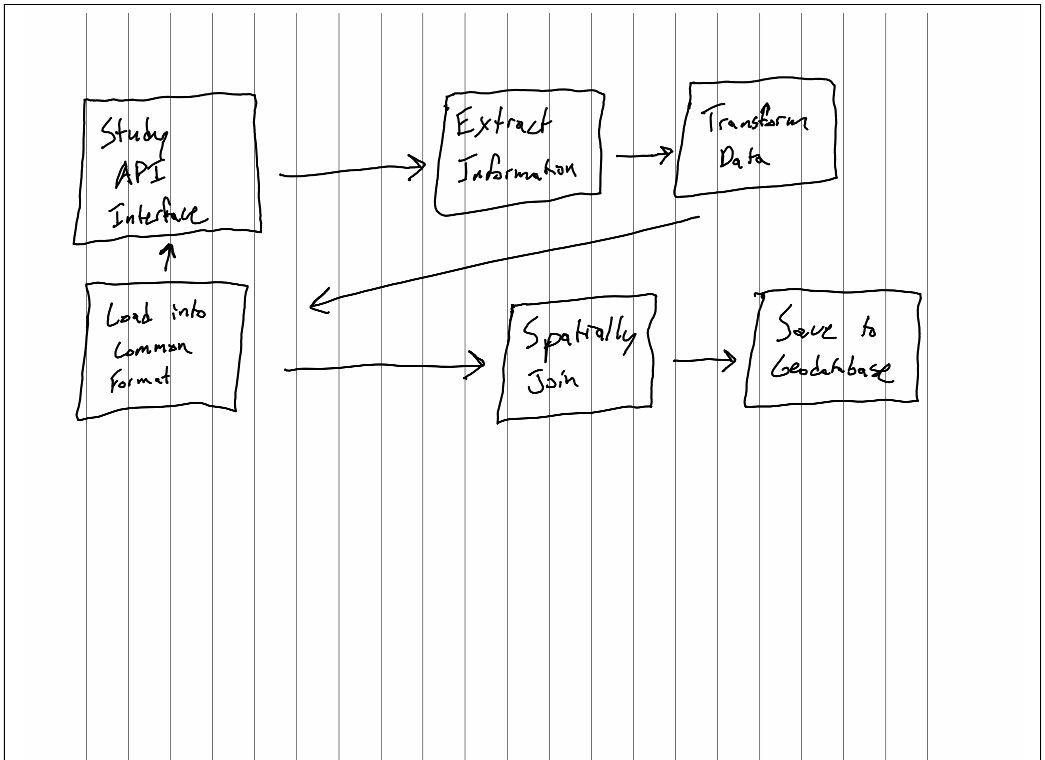


Figure 1: General Steps

	Requirement	Defined As	(Spatial) Data	Attribute Data	Dataset	Preparation
1	Study Minnesota Geospatial Commons API	Explore interface, extract data, transform	Road Geometry — Municipal Boundaries	Traffic Volume Counts — Boundary Names	Minnesota Dept. of Transportation — Metropolitan Council	Navigated the MN Geospatial Commons Website
2	Study Google Places API	Explore interface, extract data, transform	Pools Point Geometry — Schools Point Geometry	Address, names, business status, rating, etc.		Create Google API Key
3	Study NDAWN API	Explore interface, extract data, transform	Station Point Geometry	Month, Year, Min/-Max/Avg. Temperature		Navigated the NDAWN website
4	Synthesize Data	Ensure that datasets are in the same CRS and spatially join				
5	Save synthesized data	Save to a common geodatabase				

Table 1: Project Steps

Input Data

The data acquired from the Geospatial Commons was Minnesota Department of Transportation (MnDoT) current annual average daily traffic (AADT) of all the roads of Minnesota and the Metropolitan Council's (MetCouncil) city and township (CTU) boundaries. Information acquired from Google Places was the primary schools nearby the intersection of the 94 and 35W interstate highways and municipal pools in North Dakota. From NDAWN, monthly maximum, minimum, and average temperatures were acquired for all weather stations for the past year. Their API links can be found in the following table.

	Title	**Purpose in Analysis**	**Link to Source**
1	MetCouncil's CTU's	Joining with Road Network	https://gisdata.mn.gov/dataset/us-mn-state-metc-bdry-census2010counties-ctus
2	MnDoT's Current AADT Segments	Joining with CTU's and Schools	https://gisdata.mn.gov/dataset/trans-aadt-traffic-segments
3	Schools in Minneapolis	Joining with Roads	https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=44.965676%2C-93.259512&rankby=distance&type=primary_school&keyword=school
4	Pools in North Dakota	Joining with Weather Stations	https://maps.googleapis.com/maps/api/place/textsearch/json?input=Municipal%20Pool%20in%20North%20Dakota&inputtype=textquery&locationbias=rectangle:45.951407,-104.048971|49,-96.561788
5	Max/Min/Avg. Temps across North Dakota (Past year)	Joining with Pools	Really long URL

Table 2: Data Sources

Methods

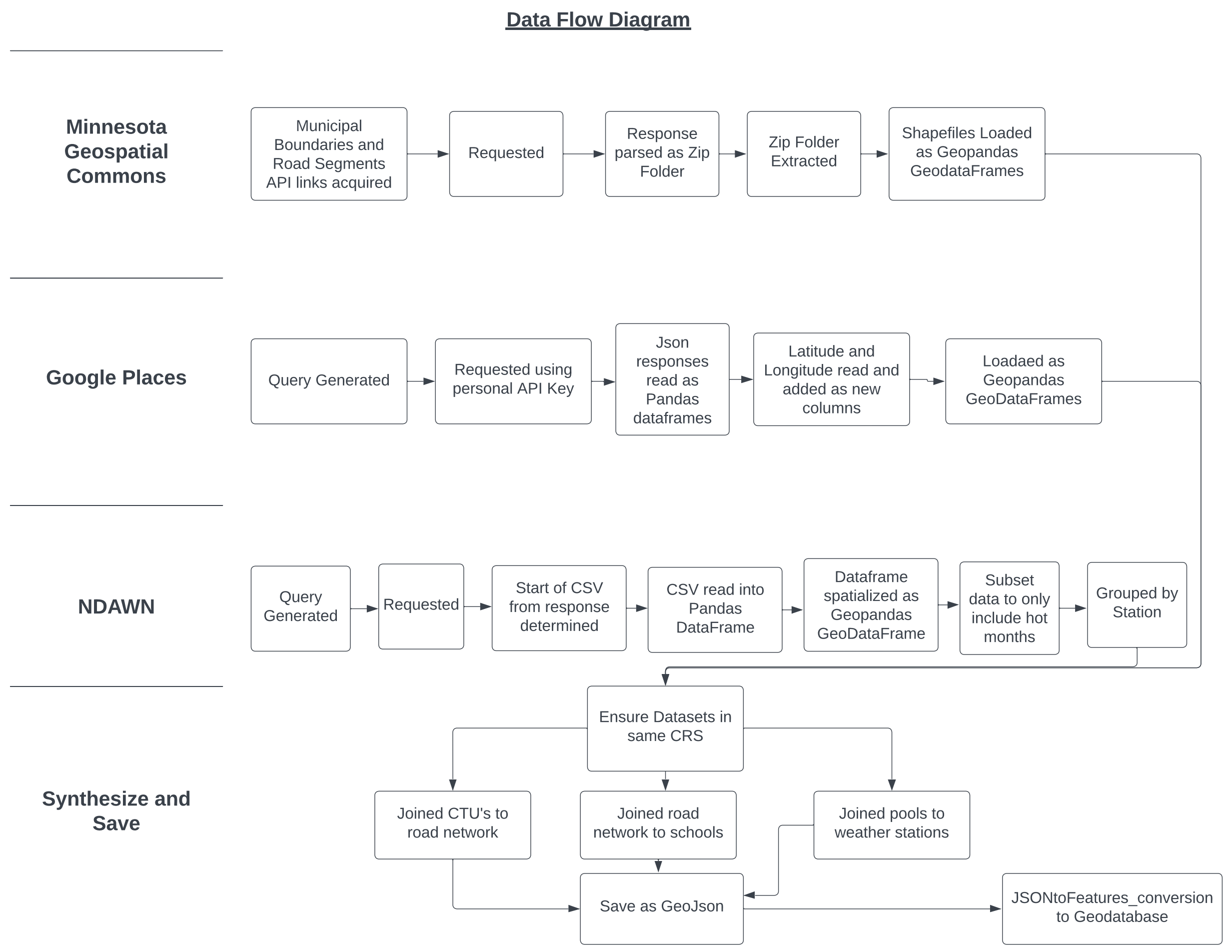


Figure 2: Detailed Workflow

Minnesota Geospatial Commons

Geospatial Commons' API links can be found through their public facing website. Right clicking on the desired download link yields the desired url. This url can then be passed to a request library's "get" function where it can be parsed as a zip file. After extracting the contents of the zip file into the current working directory, the shapefiles within can be loaded into Geopandas as a GeoDataFrame. Code to perform these steps can be found in the following figure.

```python
### Definitions

cwd = os.getcwd() # Current Working Directory

def extract_zip_from_url(url=None):
    '''Extract a zipfile from the internet and unpack it in to it's own folder within working directory.
    Takes a single url (string).'''

    if type(url) == str: # Single url
        # Create folder name for file
        folder_name = url.split('/')[-1][:-4]
        # Make folder for files
        path = os.path.join(cwd, folder_name)
        if folder_name not in os.listdir():
            os.mkdir(path)
        # Unload zip into the new folder
        response = urllib.request.urlopen(url) # Get a response
        zip_folder = zipfile.ZipFile(BytesIO(response.read())) # Read Response
        zip_folder.extractall(path=path) # Extract files
        zip_folder.close() # Close zip object
    else:
        print('Error Extracting: Invalid Input')
```

```python
# Download Data from Minnesota Geospatial Commons

## Twin Cities Metro Boundaries & AADT - Downloaded from MN GeospatialCommons gisdata.mn.gov  (~ 6mb)

boundary_url = "https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_metc/bdry_census2010counties_ctus/shp_bdry_census2010counties_ctus.zip"
aadt_url = 'https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dot/trans_aadt_traffic_segments/shp_trans_aadt_traffic_segments.zip'

extract_zip_from_url(boundary_url)
extract_zip_from_url(aadt_url)
```

```python
# Get Local Filepaths

boundary_folder = boundary_url.split('/')[-1][:-4] # Get folder name (last part of address minus .zip)
boundary_file = 'Census2010CountiesAndCTUs.shp'
boundary_path = os.path.join(boundary_folder, boundary_file)

aadt_folder = aadt_url.split('/')[-1][:-4]
aadt_file = 'Annual_Average_Daily_Traffic_Segments_in_Minnesota.shp'
aadt_path = os.path.join(aadt_folder, aadt_file)

# Load into geopandas

ctus = gpd.read_file(boundary_path) # Municipal boundaries
aadt = gpd.read_file(aadt_path) # Traffic Segments with Current Annual Average Daily Traffic
```

Figure 3: Code to interface with Geocommons API

Google Places

Google Places has extensive documentation on how to programatically query using their API. There are three options for Places querying: find place - for searching for a unique place, nearby search - for searching nearby a location, text search - for general queries based on a string. The format for their API url is:

`https://maps.googleapis.com/maps/api/place/*search type*/json?inputs\&fields\&key=*API key*`

After creating your own API key, you can develop your own queries. These queries can be requested and the responses can be read as dictionaries. Reading the dictionary as a Pandas DataFrame allows for easier manipulation. The latitude and longitude are within the dictionary in the geometry column under the location key. Once obtained, the DataFrame can be spatialized into a Geopandas GeoDataFrame using the points_from_xy function. Sample code is provided in the following figures. The first example is a nearby search of schools centered around the intersection of the 94 and 35W interstate highways. The second example searches for municipal pools in North Dakota with a location bias given as latitude, longitudes roughly correlating with the state of North Dakota.

```python
def google_places_to_gdf(url, api):

    ''' This function will take a url to the goole api and convert the response into a geodataframe.
        It does NOT work with a "find place" search

    # To Download Data from Google Places API
    # Must create a project on google API Console - https://console.developers.google.com/
    # Enable Google Places API
    # They need a credit card...

    # Base of the url = https://maps.googleapis.com/maps/api/place/details/output?parameters
    '''

    api_url = url + '&key=' + api

    response = requests.request("GET", api_url) # Get request

    results = response.json()['results'] # Read request as a dictionary
    df = pd.DataFrame(results) # Convert Dictionary to DataFrame (without correct "geometry" column)

    # Get lat/longs for geometry column

    df['x'] = None # Initialize column for Longitude
    df['y'] = None # Initialize column for Latitude

    for i, row in df.iterrows(): # Iterate through rows
        df.loc[i,'x'] = row.geometry['location']['lng'] # Get info
        df.loc[i,'y'] = row.geometry['location']['lat']

    # Convert to GeoDataFrame

    gdf = gpd.GeoDataFrame(df.drop(columns='geometry'),
                           geometry = gpd.points_from_xy(df['x'], df['y']),
                           crs = 'EPSG:4326')

    return gdf
```

Figure 4: A function to interface with Google Places API

```
api = getpass.getpass('Please enter your Google API key:')
```

```
Please enter your Google API key: ·····································
```

```
# Search for primary schools nearby intersection of 94 and 35W

url = 'https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=44.965676%2C-93.259512&rankby=distance&type=primary_school&keyword=school'
schools = google_places_to_gdf(url, api)

# Search for Municipal Pools in North Dakota

url = 'https://maps.googleapis.com/maps/api/place/textsearch/json?input=Municipal%20Pool%20in%20North%20Dakota&inputtype=textquery&locationbias=rectangle:45.951407,-104.048971|49,-96.561788'
pools_nd = google_places_to_gdf(url, api)
```

Figure 5: Example queries with the Google Places API

NDAWN

The NDAWN API was easiest to interface with using their front facing website. After conducting the desired query, there is an 'Export CSV File' link at the top of the page that is the link to the API. An example of their API url is:

```
https://ndawn.ndsu.nodak.edu/table.csv?station=40&variable=mdmxt
&variable=mdmnt&variable=mdavt&year=2022&ttype=monthly&quick_pick=1_y
&begin_date=2021-09&count=12
```

Which queries for the monthly maximum, minimum, and average temperatures of Minot, ND for the past year. The response from this is a CSV with a header describing the data and NDAWN. The first row of the CSV states the units of each column. This response can be decoded and read into a DataFrame. Latitude and longitude columns are provided so this DataFrame can be spatialized using the Geopandas points_from_xy function.

Sample code that interfaces with this API is provided in the following figure. It gets the monthly minimum, maximum, and average temperatures across all stations in their system for the past year. Because each station must be explicitly stated, the url for this example is quite long, but it can be found in table 2. After getting the data into a GeoDataFrame, it was further subset for "pool-worthy months" defined as being above 80 degrees Fahrenheit, grouped by station, and pool-worthy months were saved as lists within a new GeoDataFrame.

```
# This one gets max/min/avg temp for all stations for the past year
url = 'https://ndawn.ndsu.nodak.edu/table.csv?station=78&station=111&station=98&station=174&station=142&sta
response = requests.request('GET', url)
```

Figure 6: Getting a response from the NDAWN API

```python
# Find where CSV Starts in the response
# The beginning is a header describing what NDAWN is and such

start = response.text.find('Station Name')

# Decoding string

decoding = StringIO(response.text[start:])

# Read into Pandas

temps = pd.read_csv(decoding).iloc[1:,:] # Skipping first entry, it just gives the units of each column

# Spatialize

temps_gdf = gpd.GeoDataFrame(temps,
                             geometry = gpd.points_from_xy(x = temps.Longitude, y = temps.Latitude),
                             crs = 'EPSG:4326')

# Find the stations/months that had pool-worthy days

pool_days = temps_gdf[pd.to_numeric(temps_gdf['Max Temp']) > 80] # Months/stations that had > 80 degree days

# Group by unique stations

pool_days_gp = pool_days.groupby('Station Name').agg({'geometry':['unique'],
                                        'Month':['unique']})

# Get a new geodataframe with station name and months they were pool-worthy

pool_days_by_sta = gpd.GeoDataFrame(pool_days_gp.Month,
                                    geometry = pool_days_gp.geometry.unique.apply(lambda x:x[0]),
                                    crs = temps_gdf.crs).rename(columns = {'unique':'Months'})

# Convert np.arrays in Months into lists for saving in the future

pool_days_by_sta['Months'] = pool_days_by_sta.Months.apply(lambda x: list(x))
```

Figure 7: Processing a response from the NDAWN API

Synthesizing

To synthesize the data each dataset was ensured to be in the same coordinate reference system (CRS). The roads and CTU's were in EPSG:26915 (UTM 15N) and the others were in EPSG:4326 (WGS84). The roads and CTU's were spatially joined so each road had information on the city or township it was in. The schools were joined to their nearest road segment so we could see the traffic volume nearby. The pool-worthy weather stations were joined to the nearest municipal pool to include in weather reports for hot days. Code that accomplishes this is provided in the following figures.

```python
# Spatially Join Municipal Boundaries to Roads

# Check CRS

print('The CTU dataset is in the ', ctus.crs, ' CRS.')
print('The AADT dataset is in the ', aadt.crs, ' CRS.')
if ctus.crs == aadt.crs:
    print('They are in the same CRS, UTM 15N')
else:
    print('Transforming...')
    ctus = ctus.to_crs(aadt.crs)

# Clip AADT to CTU boundary

aadt_clipped = gpd.clip(aadt, ctus).reset_index()

# Spatially Join (Road segments keep their geometry and get Municipality information)

aadt_w_ctus = gpd.sjoin(left_df = aadt_clipped, right_df = ctus, how = 'left')

print(aadt_w_ctus.head())
```

Figure 8: Spatially joining CTU's to roads

```python
# Spatially join schools to the roads from above

# Schools keep their geometry and get road information

schools_utm = schools.to_crs(aadt.crs) # Change to correct CRS
schools_w_roads = gpd.sjoin_nearest(schools_utm, aadt) # Join

print(schools_w_roads.head())
```

Figure 9: Spatially joining roads to schools

```python
# Spatially join NDAWN stations to pools in ND (they're in the same CRS)
# I know they should be in a UTM CRS for these calculations... But accuracy isn't as important here
# Stations keep their geometry and gain nearest pool info

stations_w_pools = gpd.sjoin_nearest(pool_days_by_sta, pools_nd) # Join

print(stations_w_pools.sample(5))
```

Figure 10: Spatially joining pools to weather stations

Saving

The results from the spatial joins were saved as GeoJson which were converted into the geodatabase of this project for visualization in ArcPro. These steps are given in the following figures.

```
# The spatially joined datasets were:
# aadt_w_ctus, schools_w_roads, and stations_w_pools
# Now to save them as geojsons add them into a arcpro geodatabase

# Save GeoDataFrames as geojsons

datasets = [aadt_w_ctus, schools_w_roads, stations_w_pools]
names = ['roads_w_ctus.geojson', 'schools_w_roads.geojson', 'stations_w_pools.geojson']

# Iterate through datasets

for i, data in enumerate(datasets):

    path = os.path.join('Results', names[i]) # Save Path

    if 'photos' in data.columns: # Remove photos column (lists within dictionary - tough to save...)
        data = data.drop(columns=['photos'])

    # Make all other lists into dictionaries

    for column in data.columns:
        if column != 'geometry':
            if (type([]) in data[column].apply(lambda x: type(x)).values): # If a list is in the series
                for i, row in data.iterrows(): # Iterate through elements
                    if (type(row[column]) == list): # If a list
                        l = data.loc[i, column] # Get the list
                        new_l = dict(zip(range(len(l)), l)) # Convert to dictionary
                        data.loc[[i], [column]] = new_l # Replace as dictionary

    data.to_file(path) # Save File
```

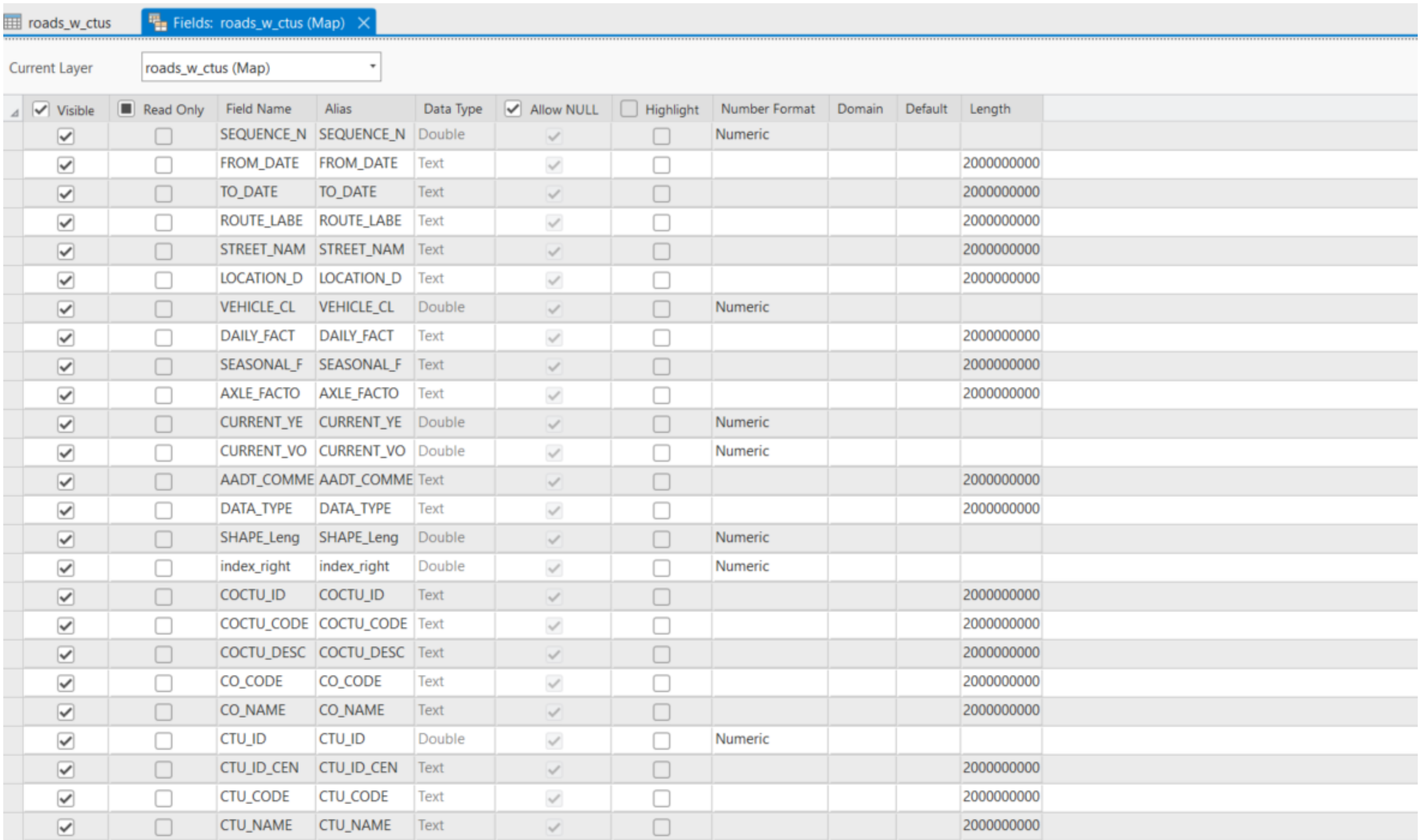
Figure 11: Code to save the joined datasets to GeoJsons.

Add to GeoDataBase

```python
# import Arcpy

import arcpy

# Set Working Directory

arcpy.env.workspace = os.getcwd() + 'Arc1_Lab1.gdb'

# Add Geojsons to GeoDataBase

files = os.listdir('Results')

for file in files:
    path = os.path.join('Results', file)
    feature_name = file.split('.')[0]

    if feature_name == 'roads_w_ctus':
        geom_type = 'Polyline'
    else:
        geom_type = 'Point'

    arcpy.JSONToFeatures_conversion(path, os.path.join("Arc1_Lab1.gdb", feature_name), geom_type)
```

Figure 12: Code to save the GeoJsons to the project geodatabase

Results



roads_w_ctus | Fields: roads_w_ctus (Map)

Current Layer: roads_w_ctus (Map)

Visible	Read Only	Field Name	Alias	Data Type	Allow NULL	Highlight	Number Format	Domain	Default	Length
☑	☐	SEQUENCE_N	SEQUENCE_N	Double	☑	☐	Numeric			
☑	☐	FROM_DATE	FROM_DATE	Text	☑	☐				2000000000
☑	☐	TO_DATE	TO_DATE	Text	☑	☐				2000000000
☑	☐	ROUTE_LABE	ROUTE_LABE	Text	☑	☐				2000000000
☑	☐	STREET_NAM	STREET_NAM	Text	☑	☐				2000000000
☑	☐	LOCATION_D	LOCATION_D	Text	☑	☐				2000000000
☑	☐	VEHICLE_CL	VEHICLE_CL	Double	☑	☐	Numeric			
☑	☐	DAILY_FACT	DAILY_FACT	Text	☑	☐				2000000000
☑	☐	SEASONAL_F	SEASONAL_F	Text	☑	☐				2000000000
☑	☐	AXLE_FACTO	AXLE_FACTO	Text	☑	☐				2000000000
☑	☐	CURRENT_YE	CURRENT_YE	Double	☑	☐	Numeric			
☑	☐	CURRENT_VO	CURRENT_VO	Double	☑	☐	Numeric			
☑	☐	AADT_COMME	AADT_COMME	Text	☑	☐				2000000000
☑	☐	DATA_TYPE	DATA_TYPE	Text	☑	☐				2000000000
☑	☐	SHAPE_Leng	SHAPE_Leng	Double	☑	☐	Numeric			
☑	☐	index_right	index_right	Double	☑	☐	Numeric			
☑	☐	COCTU_ID	COCTU_ID	Text	☑	☐				2000000000
☑	☐	COCTU_CODE	COCTU_CODE	Text	☑	☐				2000000000
☑	☐	COCTU_DESC	COCTU_DESC	Text	☑	☐				2000000000
☑	☐	CO_CODE	CO_CODE	Text	☑	☐				2000000000
☑	☐	CO_NAME	CO_NAME	Text	☑	☐				2000000000
☑	☐	CTU_ID	CTU_ID	Double	☑	☐	Numeric			
☑	☐	CTU_ID_CEN	CTU_ID_CEN	Text	☑	☐				2000000000
☑	☐	CTU_CODE	CTU_CODE	Text	☑	☐				2000000000
☑	☐	CTU_NAME	CTU_NAME	Text	☑	☐				2000000000

Figure 13: Fields view of CTU's joined to roads

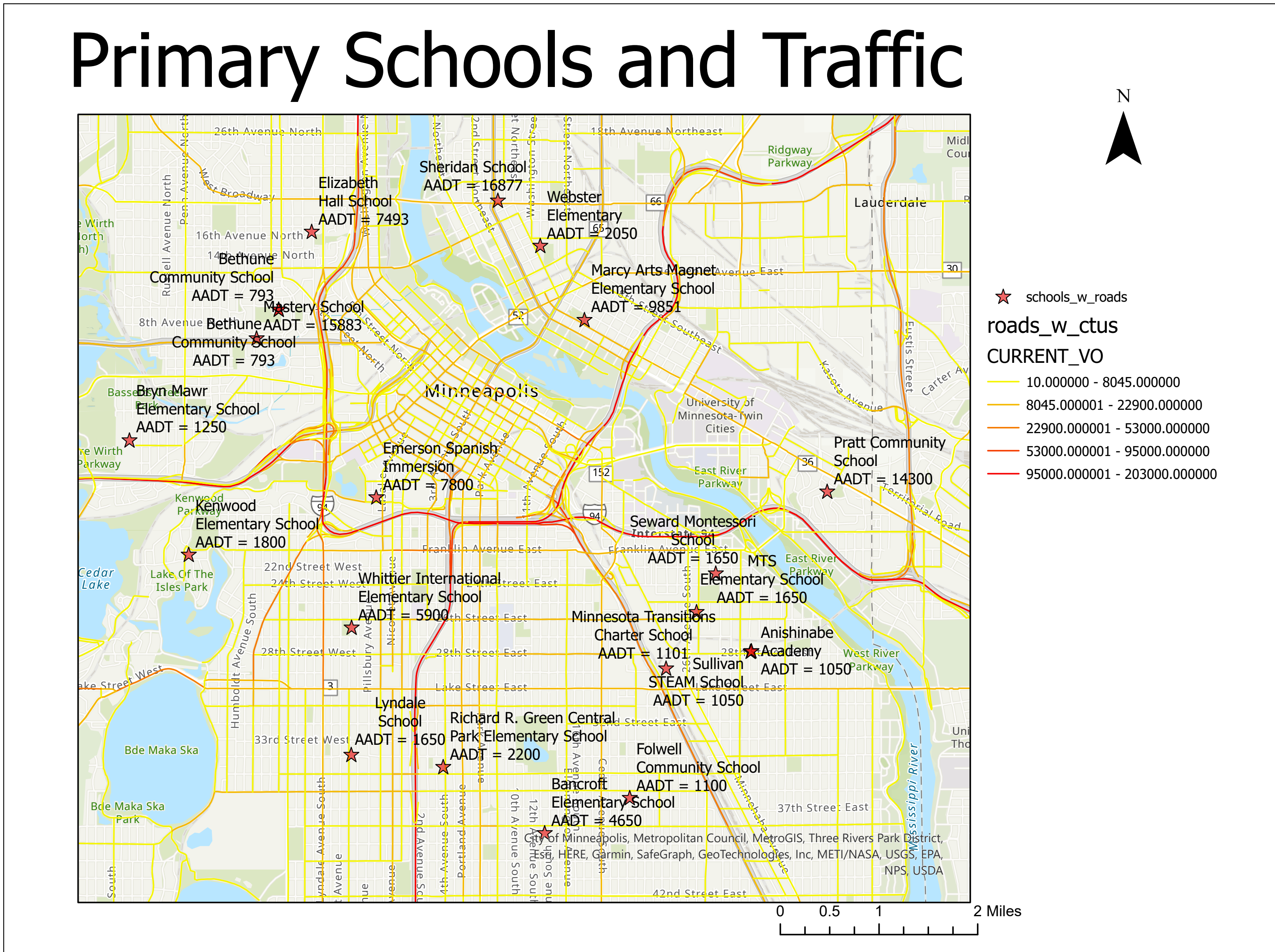


Figure 14: Visualization of roads and schools. School labels include AADT of their nearest road.

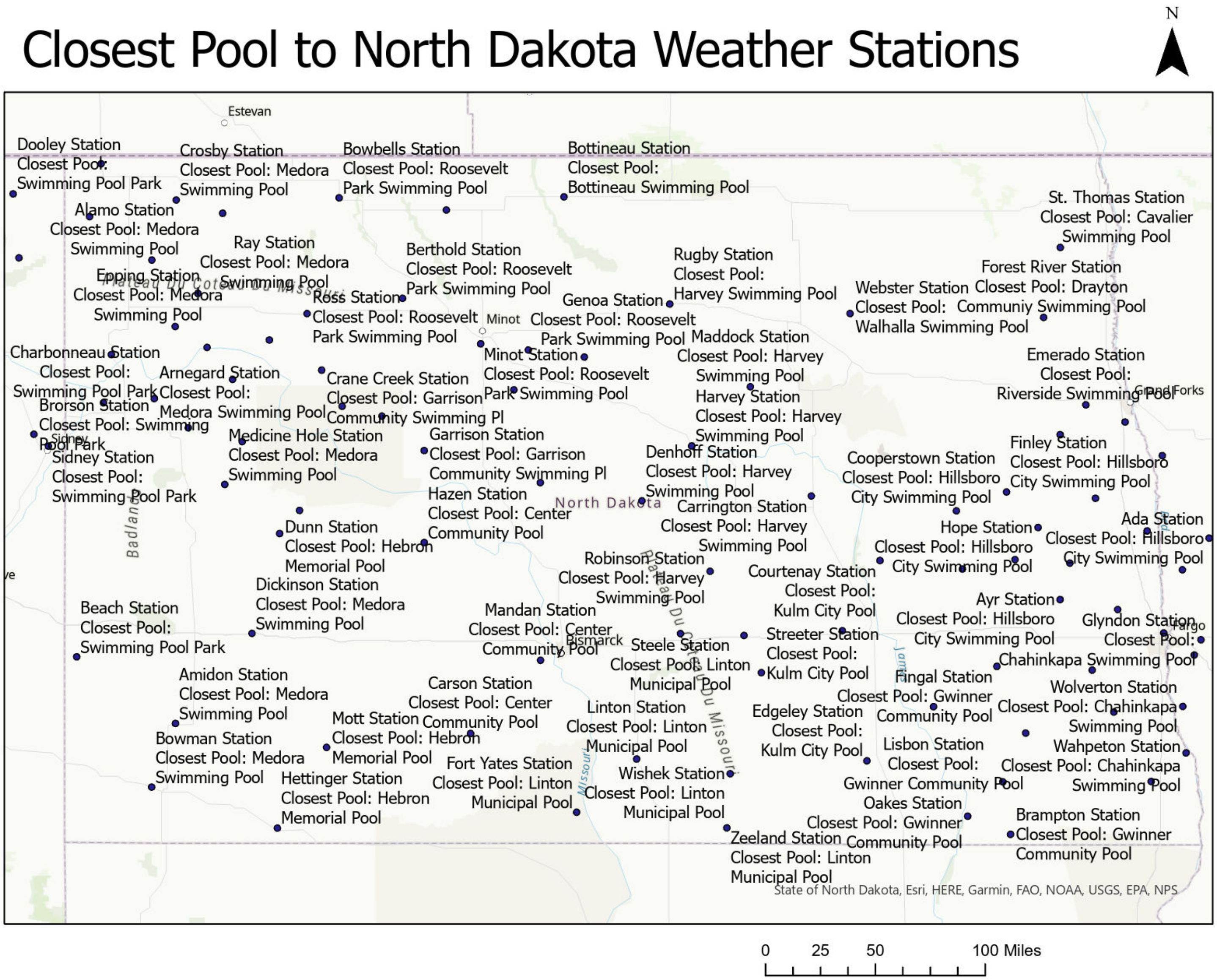


Figure 15: Visualization of North Dakota weather stations. Labels include nearest pool to each station.

Visually, we can see that the results across all datasets are in the correct locations and have the appropriate fields.

Discussion and Conclusion

Through this lab I was able to explore how to programatically interact with three very different API's.

The Minnesota Geospatial Commons API provides data in a familiar spatial format (shapefiles).

The naming conventions across the API, however, do not appear very standardized. This would make each request rather unique and someone needing to access multiple datasets would likely need to navigate their website first to develop a list of their API urls.

Google Places has pretty good documentation and standardized querying conventions. It would be quite easy to develop a flexible ETL workflow that can handle a wide variety of requests. The trouble is that you need to have an API key to perform queries which involves supplying a credit card and, potentially, a bill!

NDAWN's API was similar to Google Places in how it was structured and an ETL workflow handling a variety of requests could be developed. There was no real documentation that I could find, however, and they used a few abbreviations that were incomprehensible without performing a few requests manually.

Upon completing this lab, I am much more confident interfacing with API's and feel ready to begin to develop some ETL's for my project!

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	22
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	20
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	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	10
		100	80