Creating a Healthcare Facility Locator

Challenges, Methodologies, and Outcomes

Nick Fitzpatrick, Yukari Ide

December 11, 2024

Contents

1 Abstract

Developing a global healthcare facility locator that integrates classification, mapping, various data points, and user interactivity involves overcoming numerous challenges. This report details the iterative process undertaken to address these challenges. Key themes include API utilization, data integration, web scraping, user-centered design, and the application's real-world impact.

2 Introduction

Access to healthcare facilities is a critical need worldwide. Geographical health disparities in the U.S. are prevalent amongst rural and urban communities, making this a nationwide public health issue. In rural populations, some of the healthcare barriers include lack of transportation to and from healthcare facilities, the hours of operation for healthcare facilities, and lack of certain healthcare specialties. For urban populations, while there may be an increased number of healthcare facilities, there are other barriers including increased population with high demand for healthcare providers and socioeconomic inequities. Furthermore, health literacy levels nationwide vary, with the health insurance system only adding more complications. It can cause some people to delay seeking healthcare and worsen their health outcomes. Despite technological advances, challenges remain in creating a user-friendly platform that serves diverse needs effectively. This paper outlines the development of a healthcare facility locator application that bridges these gaps through real-time location-based searches, intelligent issue classification, and interactive mapping features. The central thesis is that by leveraging modern APIs and libraries and refining the design iteratively, it is possible to create a tool that enhances healthcare accessibility for users of all health literacy levels.

2.1 Research Questions

The following research questions guided the development process:

- How can real-time API data be used effectively to locate nearby healthcare facilities?
- What methods best classify user needs based on unstructured input?
- How can interactive maps improve usability and user experience?
- What challenges arise in combining and processing datasets for accurate visualization and decision-making?

2.2 Background and Objectives

I aimed to provide users with an intuitive interface that locates healthcare facilities quickly by a given location and various parameters such as open, medicaid supported, wheelchair accessible and ranks them based off of reviews. I used a large language model to classify health concerns into categories: "All Healthcare", "Pharmacy", "Hospital", "Doctor", "Dentist", "Veterinary", and "Physiotherapy". The categories could be more expansive in a future iteration of the project but in this case are limited by Google's categories through their Places API. The vast amount that Google provides allows for a unique project that was further enhanced through utilizing another dataset that was found through web scraping.

The application addresses a few barriers to healthcare access, such as a lack of information, limited visualization and interactivity, as well as poor categorization of services. By considering geographical and health literacy disparities, we are able to prioritize the needs of health disparate populations in our application.

3 Data Acquisition and Processing

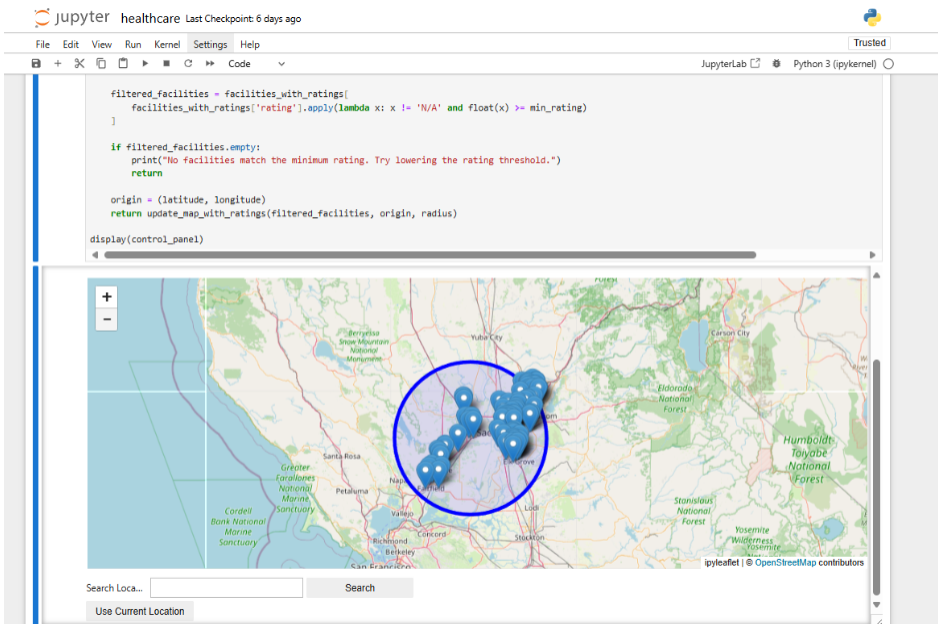
3.1 Healthcare Locations

3.1.1 Platform Selection

I first began development in Jupyter Notebook, where components like API requests and data handling were tested. Through testing, I decided to use `ipywidgets` and `ipyleaflet` for interactive map rendering (This is changed later). Jupyter Notebook is great for writing and testing code but I wanted the website to be more accessible. This motivated me to transition my code to interact with Streamlit and deploy the application onto a website. I should note that whilst utilizing streamlit during development I often went to jupyter notebook to write and test functions.

3.1.2 API Integration

I initially chose `Geoapify` for its superior categorization and filtering capabilities. However, it had a lack of detailed facility information that I deemed important such as user reviews and real-time open status. This led to the adoption of the Google Places API. I utilized the Geocoder API for user location data. I used OpenAI API to classify unstructured health-related input. I considered using Zocdoc's API for more insurance specific data but it was deemed impractical due to cost.



3.1.3 `Pandas` for Data Management

I utilized `Pandas` to process healthcare facility data that was fetched from the various APIs. The key tasks included storing the facility details in structured data frames, handling missing values and converting data types for smooth operations, and also sorting and filtering facilities by rating. None of this can be seen by the user but it seems important to mention as it helped me manage the data with ease whilst building the website.

3.1.4 Working with Unstructured Data Classification

I first utilized `Hugging Face Transformers` to classify user inputs into a set of predefined healthcare categories. However, after testing multiple of their models I deemed them not intelligent enough for my purposes. I switched to utilizing openAI models and with careful prompt engineering was able to create accurate outputs that aligned incredibly well with a vast variety of real world use cases.

3.2 Health Provider Information

3.2.1 Aggregator site for web scraping

To begin scraping data related to Medicaid coverage in the U.S., it is important to understand the scale of data being managed. The Medicaid program helps provide health insurance to adults and children of low socioeconomic status. While this program provides health insurance to count-

less individuals who would be unable to afford it otherwise, it is not accepted everywhere. For this project, we want to emphasize locating healthcare providers that accept Medicaid since we want our application to prioritize underserved communities. I decided to use an aggregator site, healthgrades.com, to obtain physician information. This website contains detailed information regarding provider addresses, accepted insurance, and their healthcare specialties. On this website, healthcare providers can be searched by state, city, or zip code.

3.2.2 Location searching

Searching only by state might give biased results toward metropolitan areas and overshadow rural healthcare results. Furthermore, the amount of healthcare provider data that may be scraped would be shallow in comparison to later search methods. This approach is too broad for our data-extracting purposes and defeats the purpose of our application. Geographic location is a social determinant of health that we want to prioritize in our data collection, so if we take an approach that undermines rural communities, that only further perpetuates the health disparities in rural areas.

Searching by city might be too involved of an approach for the scope of the project. It would require either finding or building a dataset with every city name already, and it could cause problems when extracting the health provider information because of overlap. A good middle ground is to search by zip codes because they often include more than one city if the cities are close enough and this decreases our number of search queries. Furthermore, there would be fewer overlaps that would need to be cleaned up later.

3.2.3 Obtaining and batching zip codes

The first attempt to get a dataset of zip codes was to use the `uszipcode` package. This package includes a function `SearchEngine` which can provide a list of zip codes with search filters, such as by state. Unfortunately, there was an update to a module within `uszipcode` that is no longer compatible. While trying to load in the package, there is an error `AttributeError: module sqlalchemy_mate has no attribute 'ExtendedBase'`. After searching on forums, the only solution is to pin a previous version of `sqlalchemy_mate` before loading `uszipcode`. This adds an unnecessary step that may make it more difficult for these results to be reproducible. Instead of using `uszipcode`, I imported a free zip code dataset from "United States Zip Codes" as a CSV file. The CSV file contains standard, P.O. box, military, and unique zip codes, but for the purpose of our research, only the standard zip codes would be necessary.

After filtering, the dataset included approximately 30,000 zip codes that would need to be

searched. I wrote a function to split the zip codes into groups of 50 for iterative querying. At first, I thought this would be helpful in not overloading the healthgrades website, but batching ended up being redundant for most of the project.

3.2.4 `BeautifulSoup` for HTML parsing

To understand the website, I tested my data scraping on the zip code "95610" before applying it to all the zip codes. On the website, you can see that every provider listed has a "card" with all their quick and important information listed. I used `BeautifulSoup` to help with parsing the HTML data because it was quite nested. While it was helpful, the difficulty came from the classes being named seemingly randomly. This made debugging and double-checking my code difficult because there was no semblance of a pattern. For example, the class name to find the health provider specialty was "ciXWR_F96QIbVTnf." Regardless, I wrote a try block for every provider card that would extract the provider's name, specialty, and full address. This information would be added to a unique dictionary, one for every provider. The dictionary would also include the zip code that was searched initially. Then every provider dictionary would be added to a list of all provider information.

An issue that came up was that some of the providers that were in the search results for "95610" had different zip codes in their office addresses. To avoid the same provider showing up in the final data frame multiple times, I wrote a condition that the provider's office zip code has to match the zip code that was searched for in order to be added to the list of providers.

3.2.5 `Selenium` for pagination

Now that I had one page of results running as expected, I decided to expand the data collection to include all the result pages. For the zip code "95610" there were 22 pages of results. I used the `Selenium` package to control the clicking of the next button. There were a couple of difficulties using `Selenium` but many were caused by my impatience. Once I added some `time.sleep(1)` lines to give the page time to load, the code worked well. I converted all of this into general terms to write the function `scrape_healthgrades_zip()`. I then wrote a function, `scrape_healthgrades_batch()`, to scrap through the batches of zip codes, which at this point of the project was redundant code.

3.2.6 Scraping the data, issues, and solutions

I greatly underestimated the runtime of my first attempt at scraping the data. My first mistake was letting my laptop fall asleep. This made my `ChromeDriver` quit and my three hours of running

had effectively gone to waste. I changed some of my laptop settings so it wouldn't shut off unless I explicitly asked it to. I'm not sure how long I expected 30,000 zip codes with an unknown number of pages each to take, but after 48 hours I got concerned and decided to change my approach. Keeping my laptop turned on and connected to power for over two days wouldn't have been so bad if I didn't need my laptop for other school work and class. Furthermore, if we expect this code to be reproducible, we may not want the runtime to be this long in the first place. I switched to scraping only the first two pages for each zip code to decrease the runtime. I ran a couple of sample zip codes to get an estimate of how long each zip code takes, and after doing some math, the number of hours long surpassed the time I had to collect the data. I tested it with only the first page of results and it was still too long. From here I decided to greatly decrease the scope of the data collection and focus on zip codes by state. I was able to extract health provider information for California, Oregon, and Washington in a timely matter. I moved this data into a CSV file for my group partner to work it into our healthcare facility locator application.

3.2.7 Threading for web-scraping

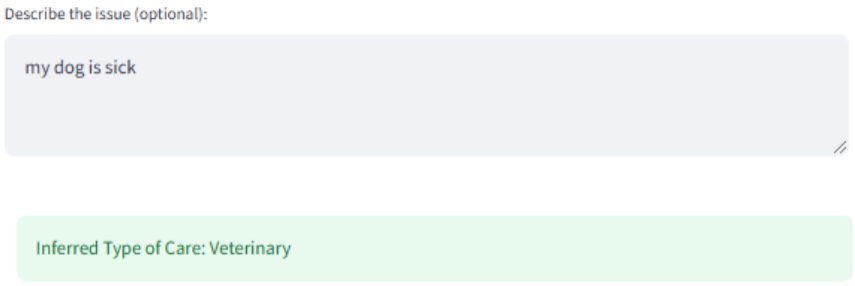
After learning about threading, I used `ThreadPoolExecutor` to implement it into my data scraping function `scrape_healthgrades_batch()`. The goal was to run multiple batches of zip codes at once to speed up the process. I also used the package `tqdm` to make a progress bar so I could estimate how long the data scraping would take. While this was working initially with five ongoing threads, my laptop updated overnight and lost over a day's worth of progress. At this point, it was too late for me to retry it again, so we proceeded with the West Coast healthcare provider data I had initially extracted.

In the future, more time and computationally efficient strategies should be explored for scraping this data. Our application is meant to be used in real-time and daily, so it is important to have reliable and up-to-date information regarding providers. Furthermore, one page of healthcare provider results is not enough to get a comprehensive view of which kinds of specialties are accepting Medicaid insurance and are in that zip code. Scraping aggregator sites daily to keep the application updated would be too computationally heavy, and scraping more than one page per zip code only adds to the workload. For the scope of this project, it was acceptable, but in future work, we would like to push these computer limitations and seek alternative data scraping methods.

4 Methodology

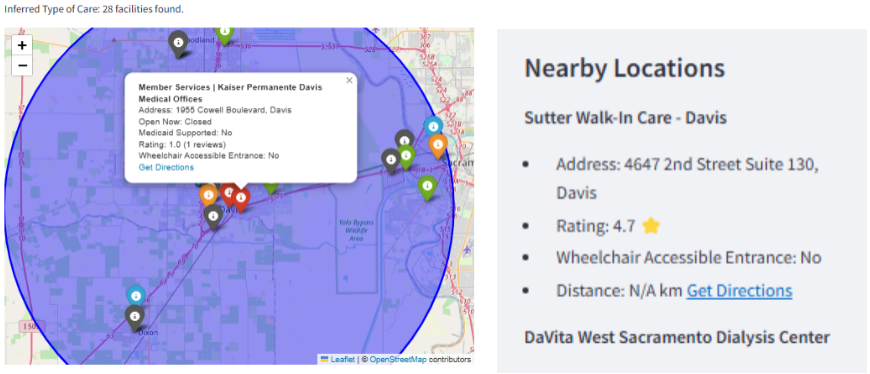
4.1 Classification of Healthcare Needs

Using OpenAI's models, the application inferred user needs from their text inputs. For example the user would input something along the lines of "I need medication for my cold." and the large language model would return "Pharmacy". I refined the prompt utilized to query the LLM based on misclassification to create a robust system. I also decided to cache outputs to optimize the performance and API usage.



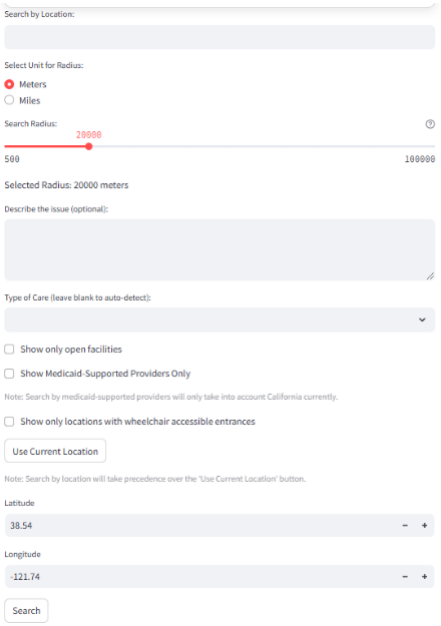
4.2 Interactive Map Visualization

As mentioned previously I utilized interactive maps through `ipywidgets` and `ipyleaflet`. I switched this to utilize `streamlit_folium` and `folium` because it better interacted with the streamlit website deployment process. Utilizing these packages I was able to create a dynamic map in which users can interact with and select various map markers to display pop ups that will display the name of facility, address, its open-status, whether or not Medicaid is supported, its rating based off of google reviews, wheelchair accessibility, and a hyperlink to get directions to the location. An interactive side-bar also provides these locations in descending order from best rated nearby.



4.3 User-Centered Design

The UI has an input for longitude and latitude as well as a location search bar for direct queries and also a "use current location" button. There is also an adjustable radius slider for tailored searches based off of distance that can be switched by the user from meters to miles. There is an optional chatbox to describe their issue that automatically determines the type of care that they need. A simple to use selector to manually determine the type of care is also an option. There are also various buttons to sort the map by only open facilities, medicated-supported providers, and locations with wheelchair accessibility.



4.4 API Limitations

There were challenges when dealing with APIs. Some of the solutions involved paginating through results for the entire completion of certain API calls, implementing caching to reduce redundant API calls, and also providing error messages to inform users of limitations. In terms of performance optimization, I decided to only display results that were limited based on the user defined criteria. I also optimized API requests by utilizing selective parameters

4.5 Combining Datasets

Various challenges were presented when combining datasets. I decided to create a script that utilized threading (for speed of computation) that added the latitude and longitude of the places

listed in the medicaid data set. At first when checking if all results from the Google Places API contained this location, none of them returned that they supported Medicaid. I learned that latitude and longitude can be incredibly preside and thus decided to round them off. This ended up producing more issues so I decided to create a function that would determine if all Google Place API locations were in a 50m radius of the latitude and longitude returned from the Medicaid dataset. This worked, however it increased the time to search by around 5-10 seconds. I deemed this to be unacceptable. I instead decided to utilize a bounding-box that essentially serves the same purpose but is slightly less accurate. This cut down the time each query took significantly and I deem the tradeoff of accuracy to be worth it. This did create an issue where certain locations nearby to healthcare facilities were also selected as having medicaid support, but this issue was simply solved utilizing filtering of healthcare categories.

5 Conclusion

The application met its objectives by integrating natural language processing classification using an LLM, efficient mapping, and multiple data sources while delivering real-time, accurate healthcare facility information and offering an intuitive user interface. From user feedback and testing, I added the "Show Only Open Facilities" option, refined tooltips for clarity, enhanced visual elements of the website, and completed various bug fixes.

Through this project, we were able to demonstrate how real-time API and web-scraped data could be used to locate healthcare facilities with certain filters. There were limitations with API availability and accuracy, as well as performance limitations on larger-scale data searches.

Our usage of a large language model to classify user needs from unstructured input is crucial for our application. It increases accessibility for users who may not know what kind of health services they need for their ailments. In future work, expanding the classifications to include provider specialties would improve response accuracy and assistance.

The interactive map includes features that significantly improve the application's usability. Visual indicators for healthcare facility ratings, responsive zooming in and out, and distance radius shading all contribute to a positive user experience.

Processing and combining datasets to be visually included in our application posed challenges such as computationally heavy data extraction, inconsistent data organization, and reliance on APIs. These challenges are part of using website data to build applications, but in the future, additional APIs and data sets could be utilized to provide greater coverage of insurance, support filtering by user insurance preferences, and add multilingual support.

The project demonstrates the value of iterative design. By addressing various challenges like data integration and API limitations the application achieved its goals by demonstrating a meaningful impact in improving healthcare access and equity.

6 Contributions

Nick Fitzpatrick: Abstract, Introduction, Data Acquisition and Processing (Healthcare Locations), Methodology, Conclusion, organizing GitHub repository

Yukari Ide: Introduction (initial paragraph), Data Acquisition and Processing (Health Provider Information), Conclusion, formatting report

GitHub Repository: https://github.com/FitzDS/Healthcare-app

Project Website: https://sta141bhealthcare.streamlit.app/