CS 432 Final Project

**Food Desert Detection**

**Sean Kunz**

I. PROBLEM

For the uninitiated, a food desert is a neighborhood that has limited access to healthy and nutritious food options. Locating and eliminating food deserts is of upmost importance to the stability and long-term health of a neighborhood and its residents. While there have been prior attempts to solve this issue, such as the USDA’s current map, food desert detection continues to be persistent problems. Despite these dire consequences and many attempts to solve the issue, most cities still contain food deserts. However, this project aims to properly detect food deserts in a given city so that government officials can best plan for the introduction of healthy food options in a neighborhood. This task is accomplished through the usage of various APIs, tools, and databases that can quickly and efficiently detect food desert neighborhoods through proper application and context.

II. SOFTWARE DESIGN AND IMPLEMENTATION

A. *Software Design, NoSQL Database Chosen, and Tools Used*

To first tackle this issue, proper technology had to be chosen. The project was written using Python, a versatile language that can be used both for scripting and application software design. Several APIs and libraries were also utilized to best tackle challenging issues through different perspectives. These APIs/libraries included Tkinter, Python Image Library (PIL), and the Google Places and Geocoding APIs. Additionally, MongoDB was selected as the NoSQL database in which to store the information contained in the map, so the Python PyMongo library was also utilized. MongoDB proved itself to be extremely efficient and easy to use, as its document storage format is similar to JSON, the format in which the Google APIs returned data in.

In terms of design, the project was split into two major pieces: the data ETL pipeline and the graphical user interface (GUI). The data ETL pipeline handled the extraction and transformation of data from the Google APIs and loaded the data into the MongoDB database. The GUI displayed information from the databases using several queries. A driver brought the two together to produce an effective product.

The data ETL pipeline is largely contained in the ‘fetchstores’ script. This script took in three command line arguments: a filename (used for initial debugging – a flag should be inputted otherwise), a city name, and a state abbreviation. Both city name and state abbreviation are necessary to avoid confusion when there exist several cities with the same name, such as Columbus, OH and Columbus, GA. Additionally, the Google Geocoding API needs both city and state information when performing a query. After parsing the input, the script initializes the extraction phase of the pipeline. The first step of this process is the calculation of latitude/longitude coordinates contained in the city limits. This step is unfortunately necessary because the Google Places API is designed to avoid the scraping of data. A call to the API functions like a search on Google Maps; if you simply type ‘grocery stores in [city name],’ not all stores will be returned. A grid-like approach to searching alleviates this issue. The ‘calcCoords’ method gets the southwestern and northeastern most coordinates of the city limit. It then calculates coordinates and search radii within the city using the Google Geocoding API to ensure that all stores are found. Lastly, the function returns this information as a list of tuples. From here, the ‘findPlaces’ function iterates through the coordinates and fetches information from the Google Places API. The program then simplifies the data, the brief transformation phase of the ETL process, so that only the necessary information gets inserted into the database. Finally, the load phase begins as various tuples (detailed in part B) insert and update the data.

While performing fewer computations, the GUI still plays an important role, as it handles the actual presentation of the data. The GUI’s code can be found in the Driver and Connection classes. The Driver class mainly handles graphical issues, while the Connection class mainly handles database interaction. Despite containing mostly tedious Tkinter-supporting methods, the Driver class has several non-trivial methods, notably ‘newPage’ and ‘getChoiceStores.’ These methods load and display grocery store information. However, some measures need to be taken so that all tuples are accounted for if requested. These methods output about twenty tuples per page, with the option of selecting a next page. Recursion and lambda functions are necessary to make this work. Additionally, these functions and more are supported by the Connection class, where the actual queries are performed. The Connection class contains several methods that perform the MongoDB equivalent of select statements to return values so that the Driver class can utilize them. These queries will be explained more in depth in part B.

B. *Supported Queries*

The Food Desert Detection project supports many queries, including inserts, updates, and selects. The main query utilized by the ETL pipeline is as follows: db.storeinfo.update({'\_id':p['ID']}, store, upsert=True). This query actually functions as both and update and an insert; if the ID is already in the storeinfo collection, its value will be updated (if there is any change) and a new document will be inserted into the collection if the ID is not found in the database. This transaction is seamless, as the dictionary data structure in Python matches perfectly with MongoDB’s document format, allowing for minimal cleaning of the data.

Most queries used by the project reside in the GUI implementation. These are mostly the MongoDB equivalent of select statements known as find statements. Some examples include: cursor = self.db.storeinfo.find({}, {"\_id": 0}), cursor = self.db.storeinfo.find({}, {"City": 1, "\_id": 0}).distinct("City") and cursor = self.db.mapinfo.find({"City": value}, {"\_id": 0}). The first query selects all documents in the database. The second query selects all distinct values for ‘City’ in the database, which is necessary to build the drop-down list for each option menu. The last query selects all maps for the specified city.

REFERENCES

All code was designed and implemented by myself, Sean Kunz. Some one-line snippets were inspired by tutorials and Stack Overflow posts, but are largely inconsequential.