CSE 515 Project

Fall 2018 phase 1

Zackary Crosley

**Abstract.** The first phase of the project had every member of the group set up, individually, the basic functionality which will be necessary for later phases. The requirements included parsing the data from the files and using similarity and distance measures to find users, images, and locations that are similar based on term vectors. For my implementation, I produced a SQL Database to read the data into for efficient retrieval. Loader and Reader class operates on the data set folder to fill the SQL database. I also generated a database interface in python to provide abstraction from the direct database structure. Separate classes allow for creating vectors from the database, measuring distance between vectors, and evaluating similarity between vectors. A Neighbors class finds the k nearest vectors to one provided by using the aforementioned vector creation, distance, and similarity classes. Finally, an interface class provides a means for users to enter commands to load the database, retrieve entries, and get nearest neighbors to a vector.

# Introduction

## Terminology

There is no terminology to report in this phase.

## Goal Descriptions

The objective of this project phase is to create a program which loads the Flickr database provided and using that data calculate the most similar photos, users, and locations to a given item. Program should run within the 15 minute allotted time frame.

## Assumptions

There are no assumptions to report in this phase.

## Implementation

The developed program has several main components that abstract features to encapsulated classes and structures. These components include a database, a database interface, a loader and readers, a vectorizer, similarity measures, a nearest neighbor calculator, and an interface.

The database is developed in SQL Lite and stores all the textual descriptors, visual descriptors, user information, location information, user – photo map, photo – location map, tags, etc. In order to develop a good database structure data beyond the requirements for phase 1 were loaded into the database. This helped create efficient tables that will sufficiently expand to the needs in later phases, but introduced some issues in this phase. Namely, the loaded in ids not present in the textual and visual descriptor database. As a result distance measures had to be chosen for which empty vectors (ids with no textual or visual descriptors) were not selected as the nearest neighbor, a problem which occurred with many distance measures.

A database abstraction layer was built to allow for easier interfacing with the database in python, the language of the rest of the program. Rather than writing SQL throughout the program, SQL queries were limited to a few functions that were called as needed by the program. This has the benefit of allowing the database structure to change without requiring refactoring of the whole program, as queries can be changed in a single location. For instance, towards the end of the development time the visual descriptor tables were changed so each model had its own table, to reduce read time. This change took less than a minute and the database was rebuilt appropriately thanks to this abstraction layer.

The loader and reader classes in file loader.py read information from the dataset into the SQL database using the abstraction layer. Each reader is designed to read one file and store their data in the database, while the loader runs each of these readers to perform all database creation code automatically. This separation allows readers to be added easily as new files need to be loaded without requiring any major code changes. The readers also have python threading built in, however SQL Lite doesn’t support (at least not easily) thread safe reads and writes, so it is currently unused.

The vectorizer reads data from the database and creates a vector representing a desired user, photo, or location. This controls both the vector creation from textual data and visual data. Vectors are represented as dictionaries, since python dictionaries are very efficient and enable easy comparison by common terms. Using a default dict, which returns 0 if a key isn’t found, calculating distance is as simple as iterating over the combined set of keys.

The distance.py file contains all distance and similarity measures, though not all have been fully implemented. Currently the files implement dot product similarity, l\_p distance, and quadratic distance (given identify matrix for A). The program currently uses L3 distance as it was the distance measure found that prevents returning empty vectors as the nearest neighbors, since the larger differences play a larger role. Similarity measures use dot product similarity since it is easy to implement and fulfilled the needs of the program.

Nearest neighbor is calculated by the Neighbor class. This implements functions to calculate the nearest neighbor for both textual and visual descriptors. It also contains the methods to get the similarity contributions. This class uses the vectorizer, the distance measurements, and the database abstraction.

Finally, the interface provides commands for a user to specify a computation they want to perform on the data. This includes a help option to see the possible inputs, a get to retrieve data from the database, and the various nearest neighbor calculations. The interface uses minimal commands by selecting vectors to compare by input. Rather than implementing separate commands to get the nearest neighbors for users, photos, and locations they are all implemented as one command where user specifies if the type of id. This makes a simpler interface for users.

## Interface

The interface for the program is as follows. Parameters are ordered and are separated by spaces.

|  |  |  |
| --- | --- | --- |
| Command | Params | Function |
| help | N/A | Prints the list of commands that can be run, what they do, and their parameters. |
| load | Database Location (File) | Load the database at the file location specified. If it doesn’t exist, the user will be prompted to create and load the database. |
| get | Type ∈ [‘user’, ‘photo’, ‘location’]  Id | Retrieve the entry specified from the database. |
| nearest-text | K – number of items to return.  Type ∈ [‘user’, ‘photo’, ‘location’]  Id – Id of specified type.  Model ∈ [‘tf’, ‘idf’, ‘tf-idf’] | Retrieves the k items that are nearest to the item identified using the textual descriptors. Model indicates the type of textual descriptors to use in comparison. |
| nearest-visual | K -Number of items to return.  Location Id – Id of location to compare.  Model ∈ ['CM', 'CM3x3', 'CN', 'CN3x3', 'CSD', 'GLRLM', 'GLRLM3x3', 'HOG', 'LBP', 'LBP3x3', 'ALL'] | Retrieves the k locations that are most similar to the location provided. Model indicates the values to use in the comparison. ‘ALL’ runs part V which uses all the models. |
| quit | N/A | Exit the program. |

## Instructions

This program uses Python 3, specifically Python 3.7. Users must have installed the dateutil, numpy, opencv, and sqlite libraries for python to use. Installation instructions for these libraries for Linux systems can be found below. Note that both SQLite program and python library must be installed.

To run, execute the main.py file. This will display the user interface in the terminal. Users must load the SQLite database before any other commands can be executed. Decompressing the dataset and providing the folder path the program will load all the data into the database. Warning – this operation takes about 10 minutes. Once the database is loaded, mounting it will be instantaneous, making this a one-time cost.

|  |  |
| --- | --- |
| Package | Installation Command |
| Opencv | pip install opencv-contrib-python |
| dateutil | pip install dateutil |
| Numpy | pip install numpy |
| Sqlite | sudo apt install sqlite ./configure --enable-loadable-sqlite-extensions && make && sudo make install |

## Related Work

No related work to report in this phase.

## Conclusion

All functionality was successfully implemented. Nearest neighbors can be calculated both off of the textual and visual descriptors. Code is written to be extensible for future phases of the project.

While all functionality is working there are several aspects of the program that can be improved.

First, and most significantly, the time it takes to perform the calculations are excessive. Creating a vector for a location in part 4 and 5 involves getting the average of all visual descriptor entries for each photo in the location. This operation takes 15 seconds in total, far longer than ideal. Part 5 takes so long to run it likely won’t be doable in the 15 minute time period required. The code has been optimized in every way visible but it still takes an eternity to do anything with images, seemingly because there are a lot of them.

Secondly, some of the program structure can be improved. I believe there may be a better way to handle vectorizing using terms and positions that requires less repeated code. Additionally, the database abstraction layer can be cleaned up to be more efficient for several lookups.

## Bibliography

There are no cited works to report in this phase.

## Appendix

No group roles in an individual project.