AIpiphany Final Report
Field Session Fall 2020

Client:
Michelle Archuleta

Team Members:
Aiden Sweezey, Gavin Castaneda, Isabel Johnson, Joseph Granmoe, Abigail Moore

Introduction

What is the goal of the project?

An automated, updatable database that maps complex medical terms or phrases to simple terms or phrases. The end goal is to be able to democratize medical language in order to give everyone the ability to navigate through the healthcare system.

How is "done" defined in this project?

A python project that can be deployed locally that is able to continually take in input data (Wikipedia articles, etc.) and update and add entries to a database that can associate complex phrases with simpler, more user-friendly, phrases. This database should be stored locally using a Postgres database and be able to interface with both our project and the company's other applications. An ambitious, but not required goal is to move the database online.

Specific Bullets:

- A thorough database that maps complex terms to simple ones
  - Thorough can be defined as 90% or more medical Wikipedia articles having a simple synonym for now
- A python script that is able to take Wikipedia data and choose complex terms out of that data and map each chosen complex term to a simple term
- The database must be updateable through the same script

Use Cases

The tool is a back end implementation to a front end tool to be used by doctors. The steps in the final tool look like:

1. Doctor submits medical note or record
2. The AI queries a database to create a simplified medical record
3. The database returns the simplified language from what the AI queried
4. The final note is reviewed by a doctor and sent to the patient

The client wanted us to specifically develop *only* the database mentioned in steps 2 and 3. A step-by-step use case for that looks like:

Setup (Done once, but must be set up to be able to be updated):

1. Import data from Wikipedia to a model that simplifies the phrases
   a. This model uses web scraping with set phrases that catch word associations.
2. Export the data into a Postgres database (locally stored, as per client request)
   a. Organize the database into two columns/tables (complex language -> simple language) and have a key or relationship between the two for each term.

Use (Done any time the front end tool is used):
A query of a complex term must return the simplified term in its place

Requirements

Functional Requirements:

- A python script that automatically generates simple synonyms for complex terms using web scraping and input those values into a Postgres database.
- The script is able to process multi-word groups (3-4 max).
- The program is able to take in data (e.g. from Wikipedia articles or other sources which have both the technical and common term) and create mappings between the technical and common terms.
- A Postgres database that has words sorted into complex and simple categories with relationships between them.

Non-Functional Requirements

- The program is a python script that takes file input and outputs to a database
- Data is be processed locally and pushed to a local Postgres database

System Architecture

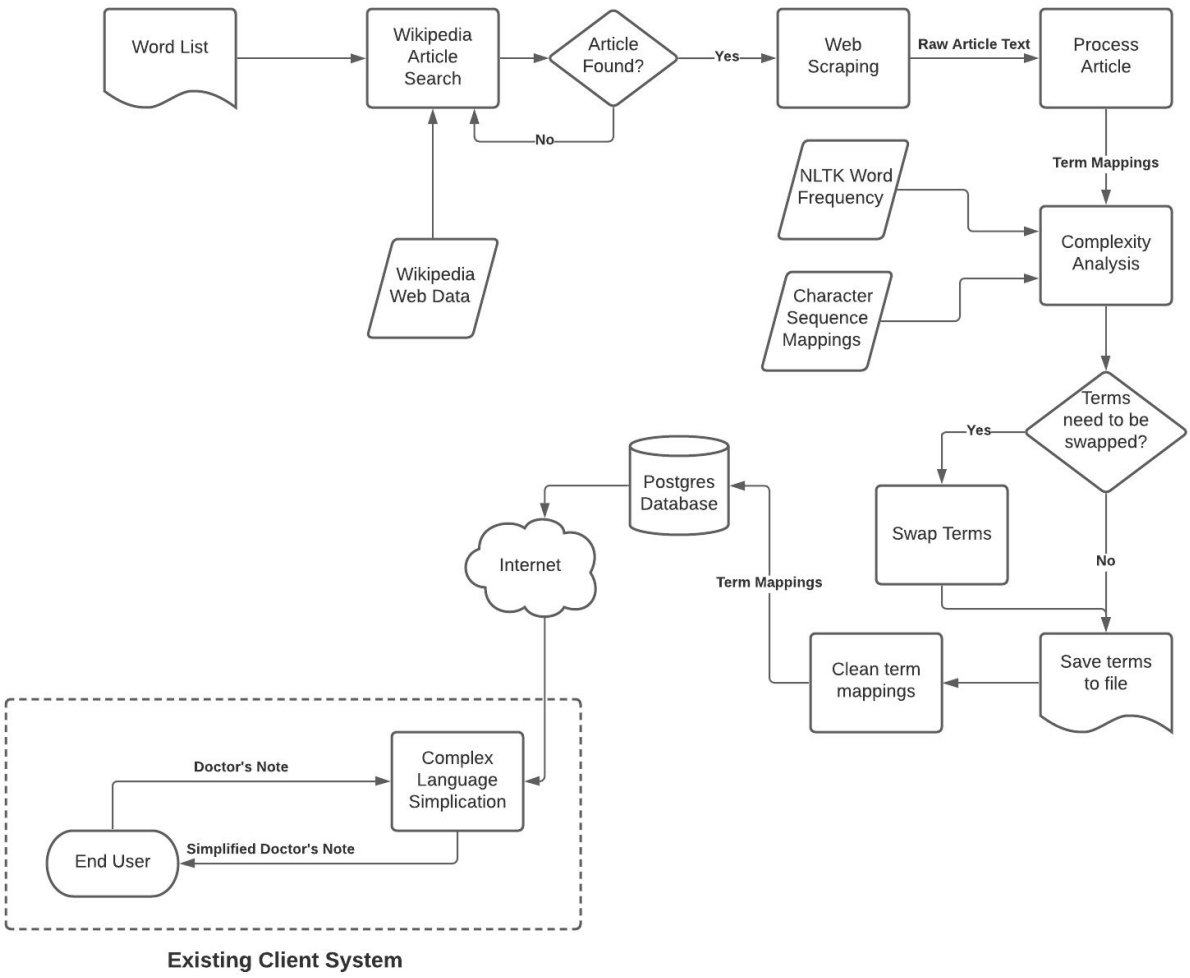

Figure 1. System Architecture Diagram

As shown in Figure 1, there are several components to our overall system:

1. The Wikipedia Article Search: this component is responsible for finding a Wikipedia article from the term supplied from the word list. The word list is a compilation of thousands of medical terms so that we can more accurately and efficiently search for medical articles. This component uses the Wikipedia Python API to be able to search based on a word or phrase. The word list is provided by the user of the program.
2. The web scraping component also uses the Wikipedia Python API to extract the raw article text based on the search results. This goes through the article and tries to match the text provided to key phrases like “also known as” or “referred to as”

to find the term mappings.

3. The Complexity Analysis component is able to reorder the term mappings. The output mappings should be standardized, with the complex term listed first. The web scraping data is often mixed up or swapped, so we need a process to determine the complex term out of the two. This is done using NLTK data and the wordfreq Python library. These two libraries are used to determine which word would be more complex to an end-user, and put that word first.
4. Finally, the mappings must be cleaned. This means removing extraneous quotation marks, parentheses, and references. This is in order to standardize the terms that are being outputted and, eventually, given to the client.
5. There is also a database, set up with Postgres, that the client will use to host these term mappings. The client wants the data in this format, so our group set up a temporary database to be able to format and structure the data properly for end-use. The database can be viewed/modified on the web using a custom Django web application.

Technical Design

Web scraping and phrase matching

The primary component of this project was the Web Scraper, the work flow of this component can be seen in Figure 2.

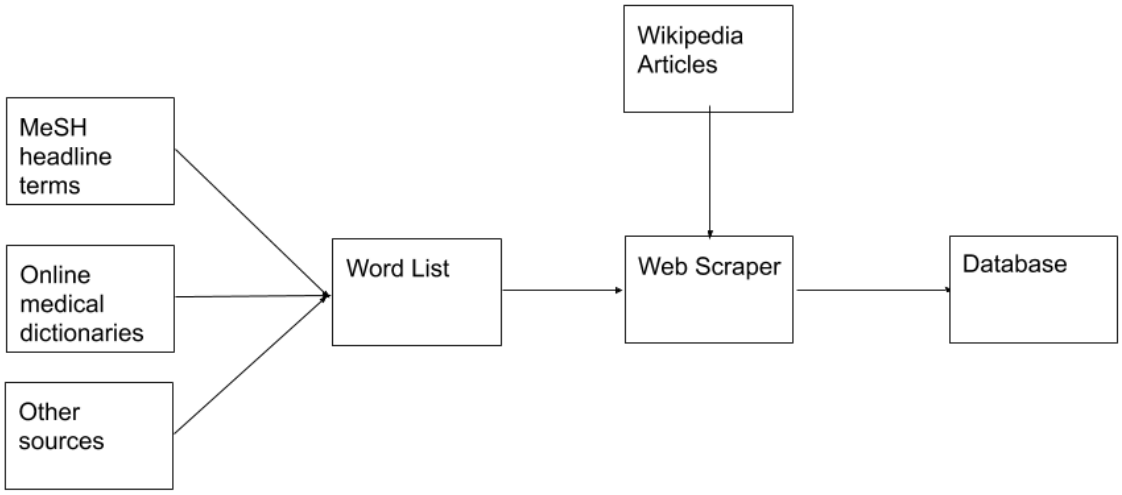

Figure 2. Web Scraper Diagram

As seen in Figure 2, the first input into the Web Scraper program was a word list. A variety of sources were used to compile a list of words that would then be fed into the Web Scraper. Some of these sources included the MeSH database, a database that would return medical journals based on keywords. This database was available in an XML format that contained a variety of keywords that were contained in the medical journals. These keywords were extracted from the XML file using a simple XML parser and then the words would be put into a text file. Another resource for compiling these word lists was online medical dictionaries, which offered similar downloads to the MeSH database and allowed for similar extraction of the medical words. These keywords were assembled into text files, this allowed for a uniform input to the Web Scraper, instead of passing it different file types with different formatting.

After these word lists were inputted to the Web Scraper, the Web Scraper would use Wikipedia's API to search a given term and return the Wikipedia article relevant to that term. If a Wikipedia article is found, it would then analyze the text of the article to find synonyms for the medical term. Often, a medical term's

Wikipedia article will mention the simplified term following a phrase such as "also known as". If we look at the Wikipedia article for Myocardial Infarction, we see the medical term "myocardial infarction", the phrase "commonly known as", and the simplified term, "heart attack".

A **myocardial infarction** (**MI**), commonly known as a **heart attack**, occurs when blood flow decreases or stops to a part of the heart, causing damage to the heart muscle.[1] The most common symptom is chest

Using this format and a variety of template phrases like "also known as" and "referred to", the Web Scraper can pick out the simplified term from the article's text and match it to the medical term. After the term was returned, it would be cleaned up (as it may have had extraneous words and symbols) so it has a standard format that can be entered into the Postgres database used in this project.

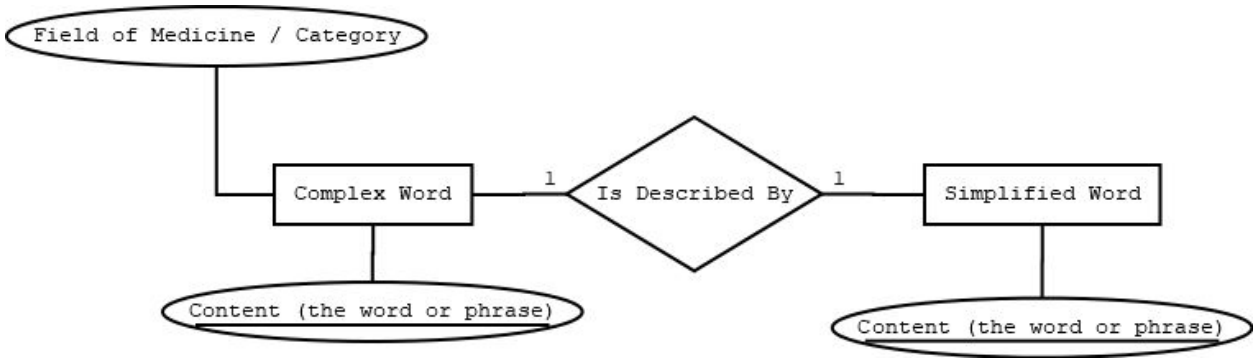

Figure 3. Database UML

The database is very simple. As seen in figure 3 it stores the complex word and the simple word. These are primary keys in their table, so only unique mappings can exist.

Word Complexity Analysis

The web scraping part of the program could generate word mappings, but relying on the order in which words are found in a sentence is not enough to make a statement about the complexity of a word.

To do this, some kind of complexity analysis must be included in the program. There are many ways to analyze and define the "complexity" of a word, and the team discussed several different approaches. The approach that was ultimately chosen was to check how often a word is used in the English language. The more the word has been historically used, the "simpler" the word is. The python

wordfreq library by the creators of the PyPi project does this. If the word isn't in the English language, or it isn't recognized by the wordfreq library, another approach is used to generate this score.

Score = $1 \cdot p_1 \cdot \ldots \cdot p_k$
where p_k = probability of character k appearing after substring [0:k - 1]
and k = the number of characters in a word

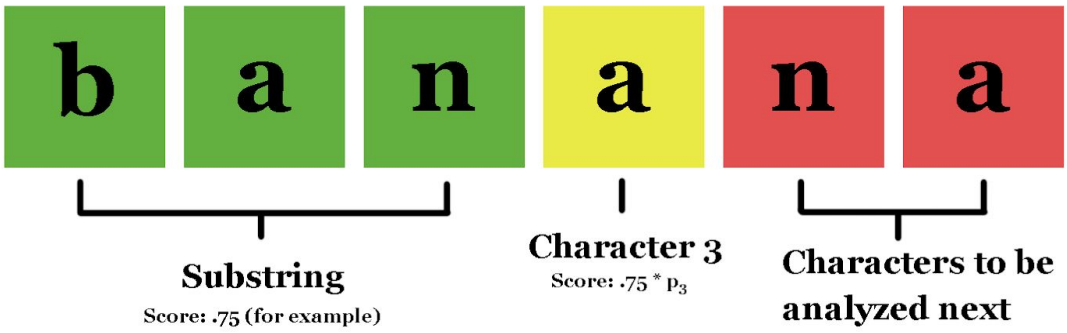

Figure 4

The approach that is taken is to analyze substrings and characters of words with probability maps. Essentially, the word is built on a rolling basis, and with each past substring, a map of the characters that could follow and their probabilities is built. Then, the score up to that point is multiplied by the probability of the letter that follows the substring appearing. For example, if the word "banana" were being analyzed in this way, and the substring "ban" had already been scored up to this point with a score of 0.75, the score would then be multiplied by the probability of the letter 'a' following "ban." The same follows for the substring "bana" and 'n' and "banan" and 'a' until the word is done being scored. Figure 4 helps to visualize this. It is important to note that every word being scored this way starts with a score of 1 and can only decrease in score as letters are being added. This leads to a bias against long terms being considered simple which is intentional.

Quality Assurance

Code Quality

In order to ensure code quality, we have performed regular code reviews with each other to go over the functionality of each new piece of code, as well as to ensure that it meets the coding standards. Along with that, we have regularly tested the code both through automated testing and manual verification to ensure that new functionality works as intended and does not break any previous functionality.

Code Metrics

The client did not specify any specific code syntax rules or style guidelines to follow, so we have followed general programming best practices: clear variable and function names, simple and readable code layout wherever possible, minimal monolithic functions, and clear documentation.

Data Verification

All entries generated by our program are uploaded to a database, which can then be viewed and verified from a web interface. Any errors can be manually corrected from this interface as well.

Unit tests

We have implemented unit tests which verify the following criteria:

- Articles are retrieved properly from article name
- Articles are properly encoded
- Articles handle errors correctly if names cannot be found
- Terms are correctly pulled from articles
- Scoring of words and phrases for english words is handled correctly
- Character-based markov chains are built correctly from articles
- Markov chain mappings are used correctly to score words which do not appear in the reference database for word frequency

User Acceptance Testing

Our client is the only user of our product as it is to be used as an internal tool to supplement their other software. We engaged in frequent discussions with our client regarding their vision for the product, as well as any modifications we needed to make to the final product, to ensure everything was kept to the client's standards.

Results

General Results

From our program, we have been able to extract about 2000 terms so far. We have been able to web scrape the simplified terms for the medical terminology lists we have and find several matches. We have also implemented several programs and processes to refine these terms and have been able to determine the relative complexity of two terms, in order to order them properly.

When given a sample article, our web scraping program is easily able to find the terms stated in the article. For example, given the article on Edema (Figure 5),

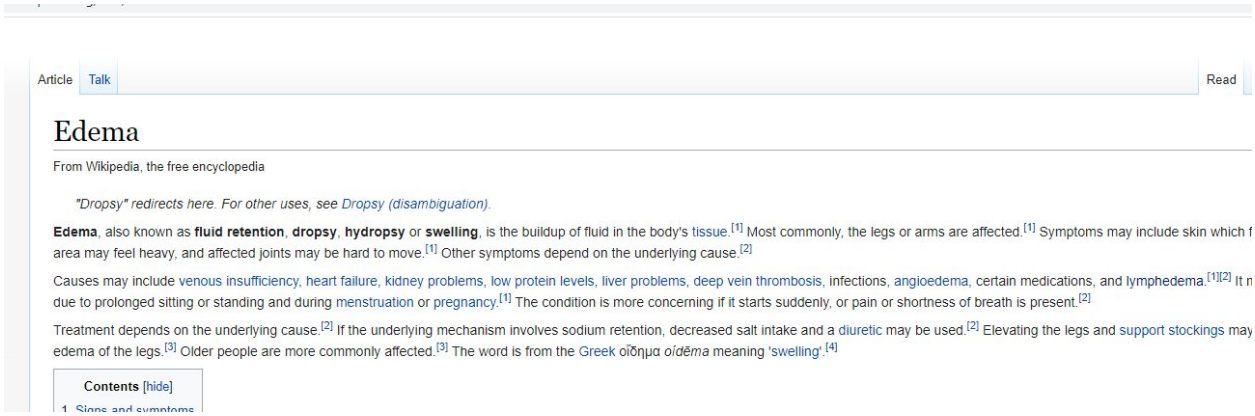

Figure 5. Wikipedia Edema Article

Our software extracts the terms fluid retention, dropsy, hydropsy, and swelling, and associates them with edema.

We cannot build a program that can grade the medical correctness of these terms because it takes a significant medical understanding to confirm or deny these mappings. Because we do not want to provide the end-user with inaccurate information, we built a website that allows the doctors from AIpiphany to accept, reject, or modify the term mappings presented to them.

What We Did Implement

- A Python script that generates word mappings of complex words to simple words
- Approximately 2,000 word mappings
- Postgres database with word mappings in relational tables
- Local Django website that the client can use to view/modify mappings

Features We Did Not Have Time to Implement

- Implementing an absolute complexity measurement in terms of reading level

- Automatically recursively looking up terms. We want to be able to simplify the simple term, to recursively simplify a complex term to different reading levels. We did not have time to implement the logic in order to do this.
- Using other websites and medical databases (largely due to licensing and legal issues as well)

Future Work/Stretch Goals

Much of the suggested future work coincides with the features we were unable to implement in this amount of time. These stretch goals include:

- Recursive term lookup and intermediary words and phrases
- A robust scoring system for each medical word or phrase
- Granular sorting tags, filters, or score values stored in the database for ease of implementation on the front end

Lessons Learned

- Web scraping and multithreading are hard to use together. The lack of reliability of internet functions often causes problems with multiple threads working together. It was much faster to use several threads to process the information, but often the program would freeze several thousand terms in, seemingly for no reason. Because of this, it was much more reliable to just set a single process running this software and leave it overnight.
- It is important to save the outputs as you go. When processing large amounts of data, we shouldn't expect the program to be able to connect to every URL perfectly and then save all the outputs. There were several times when the web scraping process crashed or the internet disconnected, and we lost the data that it had in memory, waiting to be saved. The implementation of saving outputs as the program runs also helps save memory. If it writes the output to the file every so often, it doesn't have to keep all the found terms in memory and we can clear up unnecessary memory use.
- We should have clarified our final project plan with the client earlier. We didn't understand exactly what the project output was or what code we had to write to implement the client's specifications until late in the semester, which caused a lot of confusion and bottlenecked our software design. We had to change the course of our project midway through (from Machine Learning to Web Scraping) which caused a lot of our progress to be reset.

Appendix

Table 1.
A sample of word mappings generated by the program

cytosine arabinoside : cytarabine
cytisine : sophorine
baptitoxine : cytisine
cytidine monophosphate : simply cytidylate
cytidine monophosphate : 5'-cytidylic acid
cytopempsis : transcytosis
mimosa pudica : sensitive plant
touch keyboarding : touch typing
touch type : touch typing
malolactic fermentation : malolactic conversion
deadliness : lethality
perniciousness : lethality
nandrolone : 19-nortestosterone
endoscopic laser cordectomy : kashima operation
beheading : decapitation
decarboxylases : carboxy-lyases
radioactive decay : nuclear decay
lymphoedema : lymphedema
lymphatic edema : lymphedema
passivity : deference
deference : submission
alopecia unguium : onychoptosis defluvium
calumny : defamation
oxydimorphine : pseudomorphine
dehydromorphine : pseudomorphine
dejerine–sottas neuropathy : dejerine–sottas disease
juvenile offending : juvenile delinquency
delirium : acute confusional state
meperidine : pethidine
demodicosis : red mange
demodicosis : demodectic mange
methylated spirits : denatured alcohol
toothache : dental pain
divinization : apotheosis
apotheosis : deification from latin
follicular cyst : dentigerous cyst
dentistry : dental medicine

dentistry : oral medicine
hydroxocobalamin : vitamin b12a
hydroxocobalamin : hydroxycobalamin
epilation : hair removal
depilation : hair removal
thessalonica : thessaloniki
selegiline : emsam among others
l-deprenyl : selegiline
dermanyssus gallinae : the red mite
eczema : dermatitis
dermatophilosis : rain scald
dermatophytosis : ringworm
deuterium : heavy hydrogen
devonshire : devon
pantothenol : panthenol
carbamide : urea
proflavine : proflavin
diaminoacridine : proflavine
methandrostenolone : metandienone
methandienone : metandienone
perspiration : sweating
osteopetrosis : albers-schönberg disease
osteopetrosis : marble bone disease
asymmetric synthesis : enantioselective synthesis
electronic heating : dielectric heating
mechlorethamine : chlormethine
dicyclomine : dicycloverine
dideoxycytidine : zalcitabine
etidronate : etidronic acid
dienoestrol : dienestrol
amfepramone : diethylpropion
dihydromorphinone : hydromorphone
root canal treatment : endodontic treatment
root canal treatment : endodontic therapy
quezon city : kyusi
latrepirdine : dimebolin
latrepirdine : sold as dimebon
dimercaprol : british anti-lewisite
dimethazine : mebolazine
metocurine chloride : dimethyltubocurarinium chloride
dimethyltubocurarine chloride : dimethyltubocurarinium chloride

diminazen : diminazene
hymenolepis diminuta : rat tapeworm
meticillin : methicillin
diphthongia : diplophonia
propanoic acid : propionic acid
diprosopus : craniofacial duplication
dirofilaria immitis : dog heartworm
dirofilaria immitis : heartworm
phenoxymethylpenicillin : penicillin vk
phenoxymethylpenicillin : penicillin v