1
Thesaurus Write Up

1.1 Why do I need a thesaurus?

A moldable thesaurus is seemingly very important for my project. I'm looking at building a model to shift the tone of a body of text while attempting to preserve semantics. Simply going off of intuition, we can figure that swapping a word for a synonym of that word may alter the tone of its enclosing sentence. Consequently, we would like to aggregate some relative relationship between some set of tones and groups of synonymous words. Upon creating such a dataset, we may interchange a word with a synonym according to a difference in tone.

1.2 Why not use an existing API?

There are a few reasons to construct or reconstruct our own thesaurus. First and foremost, we are interested in keeping additional data regarding tone that is not present within existing thesaurus APIs. It is far easier to manipulate data if it is all on hand, and it should save some server-side complexity in dealing with the decoupling of the thesaurus and the word-tone relationships. There is an issue with cost as well: APIs are rarely free past some established number of calls in a given time interval, and I would like for this software to function with minimal cost–ideally the only cost is in hosting. Finally, this is a project in the realm of software engineering. While it

is often better to rely on existing services–standing on the shoulders of those who came before you–it is also important to know how to create your own services.

1.3 How did we do it?

We created the thesaurus by web scraping, which is a large aspect of data-collection and thus is often a necessity in the sphere of machine learning. Of course, it is possible to compile a thesaurus using other means–surely one could buy a physical thesaurus and type up all the entries or even automate such menial tasks with computer vision–but we are interested in constructing a thesaurus as painlessly as possible. It is only one aspect of our project after all.

1.3.1 Choosing a site

Web scraping often comes with a give and take. There are several existing online thesaurus services, so we did looked into a few of them and landed on John Watson's Big Huge Thesaurus. We began with trying thesaurus.com, but they have protections against traditional scraping. There are many of such protections including lazy-loading content, providing fake data, services like Captcha, even automatically altering the page's HTML. It seemed as if thesaurus.com had been randomizing their CSS classes and either lazy-loading content or providing fake data. While we could likely get around this with an automated browser like Selenium, as the CSS classes are only a deterrent if scraping over a large time interval and the content would almost certainly exist within an automated browser session, we should respect that this site has practices in place to prevent scraping.

There may be protections against web scraping in place that we ought to honor, or there are often poorly laid out websites that would be a pain to use despite being open source, or the site may simply not provide all the information we would like. The latter is best exemplified by moby which we may come back to if we decide that we care not about parts of speech. John Watson's Big Huge Thesaurus, on the other hand, seems to have all that we want: synonyms by part of speech, a clean interface, and permission for use given credit is provided.

Protections against web scraping from JonasCz

1.3.2 Scraping the site

Web scraping for purposes such as gathering content from a page is a basic process: get the raw HTML of the page and retrieve what you want. Due to the ubiquity of HTTP requests and string processing, we can use just about anything we want for building our scraper. We will be using Python for its simplicity in our project, although we will create something similar in a bash script as a proof of concept for demonstration (see `thesaurus/webscrape/scrape.sh`).

We are using the `requests` module to get the HTML for each page and `bs4` for our HTML parsing. I am running Ubuntu on my computer, and thus have access to the `words` file present across Unix operating systems: this is a raw text file with a collection of words separated by line. This will be the basis of our thesaurus. We will first reduce this file by removing all entries with apostrophes with `thesaurus/webscrape/fixWords.py` whose essence is:

```
if word.find("'") == -1:
    outfile.write(word)
```

We do this as to eliminate repetition in our database to expedite searching as all nouns present in `words` have a possessive form; note that this does come with the loss of conjuntions. Now that we have the words we will use to construct our thesaurus, we may do exactly that. Each page takes the form of the same base url followed by the word: this makes for easy access. We go through word-by-word in our reduced file, make a GET request for that word, and then aggregate all the word's synonyms and antonyms by part of speech. There's just one little trick to this process: the antonyms are not in a concrete section, but rather one of several possible subsections under each part of speech. To circumvent this issue, we just have to do some checks to make sure that any present antonym section belongs to the part of speech we are considering and not a later-occuring part of speech. We allow the addition of antonyms to the synonym list and remove them prior to returning; this allows for us to have less rigorous checks in adding synonyms.

There was an earlier version of this scraper that did not deal with antonyms. This catch is attributed to Ariadne, who—when I showed her my progress—brought up the word "beautiful" which had "ugly" as the first entry under the synonym section. This prompted quite a refactoring,

and we should now be free of these bugs. The basis of this refactoring was largely tested against "beautiful" and "well" which both have antonyms in the thesaurus, and "well" had all the parts of speech.

Upon tweaking our scraper to suit our needs, we must output our results. As I only have so much RAM, and Python can be rather resource hungry, we segment our data by first letter. We will restructure our data into one object, but we will do this after collecting all of our data. We may naively write each dictionary to its corresponding JSON file and correct the result. This allows us to keep less in memory, which may otherwise present itself as a problem. This also allows us to segment our program as a failsafe; if we are to lose connection, crash, hit a request limit, or otherwise fail to run the script to completion, we may easily start again from where we have left off rather than the very beginning.

Then, upon building our thesaurus, we may move to the next part of our project. We would like to store this thesaurus in a database and create an API to interface with it.