1
Models

1.1 Our Different Models

We have multiple models for this project with various purposes. Our control model is that of arbitrary replacement. We have a model for scoring text input on the basis of our emotion categories. We have an experimental model for targeted replacement, and we have another scoring model that uses a latent dirichlet allocation (LDA) model, leaving an LDA-based replacement model for future work.

All of the models are written in Python, which necessitates another web server so that we can interact with them. For this, we are using Flask, which is a pretty minimal web framework for Python. It handles routing rather in a friendly way with a route singleton that registers endpoints with decorators, and the Flask instance itself is a WSGI (Web Server Gateway Interface) server; their documentation suggests that this server is not for production due to it's poor scalability, but given the limited scale of this application, we should be able to get by with a development server here.

> While lightweight and easy to use, **Flask's built-in server is not suitable for production** as it doesn't scale well. Some of the options available for properly running Flask in production are documented here. — Flask documentation

1.1.1 Model Application Structure

We have a rather similar structure for our model API as we do for our CRUD-wrapper API in Golang. Everything is contained to one module and is served from the main script. We have a `config` submodule to initialize our singletons and other data to be used across the application; this includes loading our `.env` file contents, initializing the Flask instance, initializing the route instance, applying middleware (just wildcard CORS), loading our LDA model, and initializing some word processing utilities. Then, we have a submodule for our models and one for our endpoints. The core of the work lies within the `models` module, where we go through the actual text processing. The `endpoints` module simply serves to receive requests, interface with the corresponding logic within `models` and send responses.

1.1.2 Control Model (Arbitrary Replacement)

Our control model is one of arbitrary replacement. This makes use of our thesaurus collection, where we replace a word in the input with one of it's synonyms. There's one key method to control this replacement that belongs to a `Control` class that extends a base class of `Model`.

The base `Model` class contains some important attributes and methods for the `Control` and `Score` models. We pass the base url for the CRUD-wrapper API, a list of stop words and whether or not to ignore them, and provide some utilities for processing the input. There is a method to determine if a word is in a collection—which takes into account the stop words—and will return the response from requesting the word from the collection. This goes hand-in-hand with a `requestWord` method that takes a collection string and a word in order to ease the interfacing with the other API. Outside of request utilities, there are methods to strip and replace punctuation, which allows for a wider array of valid input once split on whitespace.

For arbitrary replacement, we copy the input string before stripping the punctuation. We loop over all the words in the input string, and collect the output string along with a list of words that were encountered but not present in the database, the number of words that were replaced, and the stop words that were skipped. For each word, we assume it is unchanged by default, then strip the punctuation on the token, and with a passed probability, attempt to replace the word. For a replacement attempt, we call the `entryExists` method of the base class to return whether or not the word exists, as well as the response; should the word exist in the thesaurus collection, we select a random synonym from the response for the output and then replace the original punctuation. This punctuation replacement is a little cute: we pass the original word with punctuation and the replacement word, and we replace the original word stripped of punctuation with the new word. Should the word not exist or belong to the list of stop words, then the word remains unchanged and is added to either the list of words not present or the stop words skipped, and the word is added to the output. This ultimately gives output *similar* to the original input on the basis of synonym bag membership.

The purpose of this model is to give a baseline for how natural a sentence can sound after being passed through the model, as well as giving insight on underlying emotional scores. With the control model, we can more appropriately look into the output of something like our targetted replacement model, in that we can determine if scores are innate to the bag of synonyms, or if targetted replacement has a tangible effect.

Control Model Endpoints

Each set of endpoints has a basic `GET` endpoint to return some healthcheck response. This is convenient for some quick debugging to make sure that the routes are registered at a glance. Then, there is some `POST` handler to interface with the actual model. For the control model, we accept parameters for the probability of replacement as well as a switch for whether or not to ignore stop words. We throw an empty input message if there was no text passed, and otherwise return the replacement result.

1.1.3 Base Scoring Model

The base scoring model also extends the base `Model` class, and gives way to our targetted replacement model. We score on a subset of emotions—sadness, joy, fear, and anger—as the affect-intensity lexicon is limited to these emotions, whereas the sense-level lexicon has the following emotion list: anticipation, anger, positive, surprise, disgust, joy, trust, fear, sadness, negative. For our purposes, we went with the intersection of the two emotion-category sets.

As any score is ultimately arbitrary, the important aspect in scoring is consistency. For our scoring, we construct a list of elligible words by filtering in a similar way to the processing in the control model: we split on white space, remove stop words, and remove punctuation. Then we loop over the elligible words, and average the score across each affect dimension. For an individual score, we must consider a word within both the affect intensity collection and the sense level collection. We construct a dictionary with keys for the response by collection if it exists, and then determine the score based off of this dictionary. The score will be a dictionary mapping affect to score (with score between 0 and 1). For each affect within the affect associations from the sense level collection, we increase the corresponding score by one half. Similarly, we add one half of the score from each affect dimension from the affect intensity collection to the corresponding score. This gives a minimum score of zero if the word is not present in either collection (or if it is only present in the affect intensity collection with a score of zero) and a maximum score of one if the word is present in both collections with a score of one in the affect intensity collection.

Basic Scoring Model Endpoints

This also has a basic `GET` handler as a healthcheck. The `POST` handler interacts with the scoring model almost identically to the way the control model endpoint does, but here we only accept a parameter for a switch on the consideration of stop words.

1.1.4 Targeted Replacement Model

This model utilizes both the scoring model and the thesaurus collection to search for a nearby maximization of the score in one of the affect categories. In order to maximize an affect score, we iterate over the input text in the usual way, and we score all synonyms of each word and replace the original with the synonym that achieves maximum score in the given affect.

One can imagine that this is a dreadfully slow process given the number of requests that we have to make. The most synonyms of any word in our thesaurus collection is for “cardinal” which has 406 synonyms (mostly numbers as per the cardinal numbers), the least is, naturally, zero synonyms. Thus the maximum number of requests we would ever have to make would be assuming every word is "cardinal," in which we would make $406 \cdot 2 \cdot n + n$ requests: one request for the affect intensity one for sense level associations for each of the 406 synonyms for each occurence, and one request to get this list of synonyms. This translates to a long time twiddling one's thumbs while waiting for their output. Luckily, most input will not be an onslaught of "cardinal," and the average number of synonyms for a word is roughly 7.66 with the $\frac{\text{total synonyms}}{\text{number of entries}} = \frac{553539}{72285}$. The median number of synonyms is 14, and the mode is 2. This translates to a more manageable request time, although it still isn't too glamorous. Luckily we can eliminate any cycles if we were to choose to branch to the synonyms of a word's synonyms (and so on) by keeping track of the requests we have made for each word; although, in the interests of remaining close to the original word and saving computation time, we limit requests to immediate synonyms.

Targeted Replacement Model Endpoints

This also has a basic `GET` handler as a healthcheck. The `POST` handler here takes a switch for the consideration of stop words as well as a route parameter for the affect to be targetted.

1.1.5 Latent Dirichlet Allocation Scoring Model

Building the model

For the LDA scoring model, we are to construct an LDA model and then determine some score from the output. We are using `gensim` for our implementation, and thus are almost exclusively concerned with determining the corpus to feed it. For this, we have grouped all entries in both the affect-intensity and senselevel lexicons by affect and gave all words by affect as our four seed documents.

$$\text{corpus} = \{\text{anger}, \text{fear}, \text{joy}, \text{sadness}\}\ \text{anger} = \{\text{word} \mid \text{word has affect score for angry}\}$$

This grouping was relatively painless, for affect intensity, we looked for non-zero scores in a given affect as criteria for membership, and we looked for affect existence within the senselevel lexicon; from here, we just combine them.

It is worth noting that this is an unusual means of gathering a corpus. Typically, a corpus is some large collection of documents, which would lend itself to repeated words with various frequencies throughout the documents. Our usage, on the other hand, holds that a word has a maximum document frequency of one. This then has implications on the model that uses things like tf-idf as a means of determining topic likelihood: with such strong limitations on intra-document frequency, uniqueness becomes more important. As a nicety for our model, we are stemming all tokens within our corpus. This maps words onto their root (i.e. $\{\text{stem}, \text{stems}, \text{stemmed}\} \to \text{stem}$), and it allows for removing an inconsistency in the scoring of similar words. With stemming, all words sharing a root are given the same score, which is the average score of entries mapping to the stem. We also remove stop words from the tokens if present.

From here, building our model is as simple as passing our corpus and vocabulary along with some K for number of topics and some number of epochs to gensim's `LdaModel` and save the model to disk. Note, we must save the model once and load it for all future use for the sake of consistency.

Using the model

In order to use the model, we ought to give our input the same treatment that our corpus was given. We will case-transform, filter by stop words, and stem the input strings. From here we loop over the inferred topic distribution in order of decreasing probability, and gather a few scores. For each topic with probability greater than a passed threshold, we gather: the topic probability, topic index, the n topic keywords, and a few different means of scoring. We give four scores for each topic (and the net scores weighted by respective topic probabilities): the `input score` is the sum of the product of input token scores and their probability within the topic; the `topic score` is the sum of the product of vocabulary token scores and their probability within the topic; and there are two forms of `keyword` scores, one of which takes the topic keywords product with the normalized probability of the keywords, and the other takes the topic keywords product with the whole topic's probabilities.

LDA Model Endpoints

This also has a basic `GET` handler as a healthcheck. The `POST` handler here takes a float for the topic probability-threshold and an integer for the number of keywords to gather. There is an additional `GET` with a route parameter for topic number, should one want to observe the word probabilities for a given topic; this is not accessed by the frontend, but is accessible from sproj.model.colehollant.com/lda/topic/¡topic-number¿. One of the interesting pieces that arose from the translation from CLI to web server here was that `numpy`'s `float32` wasn't serializable; but as this is used within `gensim`'s model that we use, we have to provide a means of serialization which converts to `float64`. This eases the process of tracking down instances of `float32`, and lets the `json` module handle this for us instead.

1.1.6 LDA Model On Seeds

It's worth considering the inferencing of the seed documents to see how they are scored. For this, we will look at the different net scores determined by feeding in each document with a 1% probability threshold, gathering the top 100 keywords (scores will be rounded to 5 places).

Anger

Category	Anger	Fear	Joy	Sadness
Input Score	**0.94886**	0.76935	0.03475	0.67535
Topic Score	0.59074	**0.59339**	0.03516	0.53103
Keywords by Keywords	**0.66098**	0.65198	0.01993	0.58062
Keywords by Topic	**0.25617**	0.24547	0.00825	0.20950

Fear

Category	Anger	Fear	Joy	Sadness
Input Score	0.65673	**0.87143**	0.01797	0.76317
Topic Score	0.51107	0.59343	0.04255	**0.60067**
Keywords by Keywords	0.58840	0.64486	0.03415	**0.64810**
Keywords by Topic	0.20910	**0.22217**	0.01214	0.21510

Joy

Category	Anger	Fear	Joy	Sadness
Input Score	0.04641	0.06371	**0.97432**	0.07960
Topic Score	0.05247	0.07051	**0.51985**	0.06841
Keywords by Keywords	0.05356	0.07832	**0.61078**	0.08025
Keywords by Topic	0.01923	0.02722	**0.20237**	0.02747

Sadness

Category	Anger	Fear	Joy	Sadness
Input Score	0.62026	0.80163	0.01106	**0.99642**
Topic Score	0.47046	0.60619	0.03291	**0.66378**
Keywords by Keywords	0.55249	0.64956	0.02865	**0.70967**
Keywords by Topic	0.18193	0.20938	0.00980	**0.22397**

This may make some sort of intuitive sense. The input scores for each affect category are championed by their affect. The other scores are affected by the topic definition and keywords without direct scoring of the input, leading to greater fluctuation, although they tend to follow the trends of the input scores. We may notice the tendency of “anger,” “fear,” and “sadness” sharing similar scores that seem to oppose that of “joy.” This can be infered to be a similarity between those emotions, but additional data may give further insight. We’ll consider the mean Valence Arousal Dominance (VAD) scores for each document.

Category	Valence	Arousal	Dominance
Anger	0.25627	0.66386	0.46530
Fear	0.28537	0.65718	0.47638
Joy	0.7726	0.51821	0.59808
Sadness	0.23668	0.58389	0.38429

Here we can see a tangible difference in valence scores between the group of “anger,” “fear,” and “sadness” having rather low mean valence scores, whereas “joy” has a rather high mean valence score. All the arousal and dominance scores across the affect categories have far less extreme values. We can use this as a means of understanding the uneven groupings of our affect categories as a latent bias regarding valence. This correlation also held for the most prevelant color associations across each document with “anger” and “fear” having “black, red, gray;” “sadness” having “black, grey, red;” and “joy” having “white, yellow, pink” as the top three most prevalent color associations.