# M.A. Computational Linguistics Final Project

Content-Based Recommendation Engine

Derek Andersen

Fall 2020

## 1 Introduction

In this report I will provide an overview of `buzzrec`, a recommendation engine for linguistics academic papers in the LingBuzz database, which is an openly accessible respository of scholarly papers in linguistics. (Starke 2020) In Section 2, I will discuss its motivation, and in Section 3, I will outline the techniques I used for the project's implementation. In Sections 4 and 5, respectively, I will touch on the improvements to be made in the future and my personal learning outcomes from the project.

## 2 Motivation

The motivation for this project was to provide a tool that linguists and regular readers of linguistics papers could utilize to find papers that align with their interests. For example, if a reader is interested in syntax, French, and obstruent devoicing, they could enter these as keyword parameters into the tool so that its algorithm can find papers similar to these topics, and alert the user via email when relevant papers are uploaded to LingBuzz.

Aside from its clear purpose as a tool, I was inspired to tackle this project because of the skills required in creating it. In particular, I was interested in learning about the artificial intelligence-esque functionalities (learning about a user, suggesting new items) of recommendation engines and how to apply them to a variety of problems.

## 3 Techniques and Tools

The engine for `buzzrec` was written in Python, making use of the `nltk`, `pandas`, and `sklearn` libraries. The choices for these particular libraries will be addressed in further detail later in the report. `buzzrec` uses a content-based filtering approach in its recommendation process. This approach is based on the use of item 'features' to recommend other similar items to a user. In this section, I will provide a walk-through of the process of content-based filtering with an emphasis on my specific application of the steps. (Google 2020)

## 3.1 Content generation

The first step in this process is 'candidate generation,' wherein the recommender generates a set of candidates to recommend to the user given some initial query. In this case, that query is realized through the configuration file that sits in the project directory: `config.json`. A user can populate the `keywords` field in this file with a list of keyword strings that interest them. The example from earlier can be seen in 1.

(1) `{keywords: ["syntax", "French", "obstruent devoicing"]}`

With access to these parameters, the recommendation system has a baseline set of keywords that it can use to begin constructing its model of the user.

## 3.2 Embedding space

The next step in the content-based filtering system is to construct a representation of the user in a shared embedding space. This is also sometimes called 'feature space,' as it is a data structure which embeds representations of users or target items to be recommended. In practice, a matrix can be used to accomplish this task. Figure 1 shows an example based on the feature matrix generated with `buzzrec` (using academic papers as the target items).

| | syntax | phonology | French |
|---|---|---|---|
| Paper A | 0 | 1 | 0 |
| Paper B | 1 | 0 | 1 |
| Paper C | 0 | 0 | 0 |
| User | 1 | 0 | 1 |

Figure 1: Feature matrix for academic papers

This matrix encodes the features associated with each paper/user with a binary 1/0 representing having/lacking (a feature). While it is clear which paper would be recommended to our user from this simple matrix, this alone is not enough for the recommender to be able to make suggestions to the user. Since Paper B is identical in terms of feature representation to our user, that would be the top pick. But in practice, the target items cannot be so clearly discerned, especially when the matrix grows in dimensions to be much larger. For this reason, a similarity measure will need to be introduced in order to score and assign a clear rank to items in the feature space.

## 3.3 Similarity measure

A similarity measure is a function that takes a pair of items in a feature space and returns a value that represents their similarity. (Google 2020) Several different similarity functions can be used to calculate this value, but for this project I decided to go with cosine similarity. The equation for this is in 2. (Prabhakaran 2020)

$$cos = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \quad (2)$$

Where $A_i$ and $B_i$ are components of vectors $A$ and $B$, two items in the feature space, and $n$ is the size of the feature space. The resulting similarity values once the feature matrix is converted to a similarity matrix would normally be between -1 and 1 with this equation. However, in practice, they will be between 0 and 1. This is because the method I chose to use for vectorization — TF-IDF, which I will discuss in more detail later — does not allow for negative values.

The choice for cosine similarity as a similarity measure was inspired by its widespread use for similar applications (see for example Grimaldi 2020). Specifically, cosine similarity is advantageous when the goal is to determine the similarity of documents regardless of their (possibly differing) size. Euclidean distance, on the other hand, is a metric used to compare documents according to the frequency of words in one document vs. the other, whereas cosine similarity is able to encode their general similarity regardless of the raw frequency of individual words.

## 3.4 Building the user model

A user of `buzzrec` can be represented with a model that encodes his/her tastes, interests, etc. The first problem to be solved with building this model was to decide on the type of data structure in which it would be stored and queried. Earlier I introduced the idea of a feature space in the form of a matrix. The most intuitive way to represent this in a human-readable format that is also easily accessible by code is with a comma-separated values (`.csv`) file. In Figure 2, a rough example of this model is shown, with values omitted for space. It is organized into rows (of paper items) and columns containing their features: title, link to PDF, author list, abstract, and keyword list.

The entire process for building our user model is illustrated in the flowchart in 3, which shows the pipeline of functions and classes I've written for the process. Following the pipeline is an explanation of each function's purpose.

| Title, | Link, | Authors, | Abstract, | Keywords |
|---|---|---|---|---|
| ..., | ..., | ..., | ..., | ... |
| ..., | ..., | ..., | ..., | ... |
| ..., | ..., | ..., | ..., | ... |

Figure 2: `.csv` file for user model

(3) `create_csv()` → `queryLingBuzz()` → `Paper()` → `create_df()` → `merge_df()`

1. `create_csv()`: The user-populated configuration file `config.json` is read to get the list of keywords interesting to our user
2. `queryLingBuzz()`: The list of keywords is used as a search term to query LingBuzz's search feature
3. `Paper()`: The resulting papers' metadata are scraped and fed into this class constructor to keep track of paper objects. These objects are returned to `create_csv` (The `user.csv` is now created as per `create_csv`, and the resulting file sits in the project directory)
4. `create_df()`: A `pandas` dataframe is created according to `user.csv`
5. `merge_df()`: The key words are extracted from the 'Abstract' column via `nltk`'s `Rake` class, which gets only the most relevant words. This overwrites the column. The columns 'Authors', 'Abstract', and 'Keywords' are then merged into one column 'Bag of words', containing space-separated keywords describing each paper

After this, we have the basis for the user model. The dataframe is now separated into three columns: 'Title', 'Link', and 'Bag of words'. To finalize it and make it compatible with the recommendation system, the 'Bag of words' column needs to be vectorized into some value that represents its content. At this point, a cell in the 'Bag of words' column looks something like the example in 4.

(4)
```
technically called sensorimotor existing human music system conditions
sound component formation acted rudimentary units combination emergence
intriguing fact turn critically evaluate darwin evolving might hominid
line word create apes explain two novel factors could language needs
basically designed problematic part must conceptual reach neither story
filled significant precedence prior effect recent proposal evolution
mystery since theory idea process
```

This is where `sklearn`'s `TfidfVectorizer` function comes in — a function which constructs a matrix of TF-IDF weights for all words in all papers in the dataframe. The TF-IDF weight of a term is a measure of its importance relative to the corpus (in our case, the collection of papers). It is calculated by multiplying a term's frequency (TF) by its inverse document frequency (IDF). (tfidf.com 2020) The TF is calculated by multiplying the number of time a term appears in a document by the total length of the document. The IDF is calculated by dividing the total number of documents in the corpus by the amount of documents containing the term, and then taking the natural log of this. As I mentioned earlier, the TF-IDF weight for a term cannot be negative. This will be important for capturing the similarity between two items once the similarity measure is applied; 0 represents not at all similar, and 1 represents the quality of being identical.

Once we have the TF-IDF matrix, the last component necessary is the implementation of the similarity measure (cosine similarity). This is also done with `sklearn`, via their `cosine_similarity` function which applies the cosine similarity equation to the input TF-IDF matrix, and returns a similarity matrix. Now, each of the cells in the matrix for our user model — the relationships between every paper to every other paper — encodes their similarity to each other on a scale from 0 to 1.

## 3.5 Querying the model

The final step of the recommendation engine is its ability to actually recommend new papers to a user, taking their tastes into consideration. To do this, it will rely on the model it built for the user as a frame of reference, and compare new papers to it, considering their similarity scores. The method I decided to use for this task involves creating a new model each time a paper needed to be compared to the user's model. The program takes the original `.csv` file for the user, makes a copy (`user_plus.csv`), and creates the model from *this* file as opposed to from the original file. The motivation for this may not seem clear, because it raises the question of why the original file is necessary in the first place.

For clarity, the flow for the main program is as follows: A model for the user is created if it doesn't already exist $\rightarrow$ The ten latest papers are taken from LingBuzz $\rightarrow$ A new model is created for each paper (the original model plus the new paper) $\rightarrow$ The new paper is compared to the rest of the model (excluding itself) and the average of its similarity scores against each other paper is saved as a variable $\rightarrow$ The paper with the highest average similarity score against the rest of the model is selected as the recommendation.

With this method, the original model encoding the user's taste can be saved as a standalone, reusable model. When new papers need to be compared against the user's model, it can simply be copied and added to.

A full example output is shown in 5.

(5)
```
Building user model ...
Fetching papers ... this may take a few minutes.
Done
Checking new uploads against user model ...
Recommendation: This is the title of a paper
Link to PDF: https://link-to-paper.pdf
```

## 4 Next Steps

Originally, the goal was for the tool to be able to 'learn' more about a user's taste based on the user's binary responses to the tool's recommendations, in the form of Yes (like) or No (dislike). Taking the user's feedback into consideration, the tool would ideally be able to adjust its model of the user's taste accordingly. As it stands right now, the engine doesn't do this. Instead, it relies on the initial model it creates according to the user's configuration file. While this may be a good baseline for representing a user, I believe that the ability to improve the model over time via explicit user feedback would give the engine a more accurate representation of the user, and possibly the ability to recommend papers that are more in line with the user's interests. Neural network libraries could also prove useful on this front — this might allow the engine to take a more predictive approach, and 'learn' about the user on its own, in addition to with explicit feedback.

The code also still lacks the email function. The original idea was for `buzzrec` to run in the background 24/7, only requiring an initial setup. Then, on a regular basis (possibly weekly), the user would be emailed with `buzzrec`'s weekly digest for the user in the form of a list of recommendations for the week. Because of the lack of this feature, the user has to run the engine manually whenever they want a recommendation.

## 5 Learning Outcomes

This project was a valuable experience in terms of practical skill accumulation. I was confronted with several problems throughout the course of the project, most of which I was able to overcome with a byproduct being some new understanding of how recommendation systems, or more specifically natural language processing, works. The first of these was the problem of how to encode similarity, which introduced me to a prerequisite problem that had to be solved: how to represent a paper object in terms of its characteristics. Once I had a better idea of how this would work on paper, I was confronted with the problem of applying these skills in Python. I was introduced to new tools in new libraries, including `nltk`'s `Rake` and `sklearn`'s `TfidfVectorizer` and `cosine_similarity`.

The project was also satisfying in terms of personal interest. I was interested in understanding how to encode similarity between documents, and my research for this project exposed me to several methods to do so, and gave me practical experience with one of them. I feel that this project has left me with new skills that are very useful on their own, while also acting as valuable building blocks to new skills and methods in the field of natural language processing.

## References

Google (2020). *Recommendation Systems.* URL: `https://developers.google.com/machine-learning/recommendation/content-based/basics` (visited on 12/15/2020).

Grimaldi, Emma (2020). *How to build a content-based movie recommender system with Natural Language Processing.* URL: `https://towardsdatascience.com/how-to-build-from-scratch-a-content-based-movie-recommender-with-natural-language-processing-25ad400eb243` (visited on 12/18/2020).

Prabhakaran, Selva (2020). *Cosine Similarity – Understanding the math and how it works.* URL: `https://www.machinelearningplus.com/nlp/cosine-similarity/` (visited on 12/18/2020).

Starke, Michal (2020). *LingBuzz.* URL: `https://ling.auf.net/lingbuzz` (visited on 12/15/2020).

tfidf.com (2020). *Tf-idf :: A Single-Page Tutorial - Information Retrieval and Text Mining.* URL: `http://tfidf.com/` (visited on 12/18/2020).