Benjamin Chen

Reddit Subreddit Recommendation Engine (AKA Recommendit)

**Background**

Reddit is a website where users can submit posts which can be original text content (e.g. a funny story or a joke) or external links (e.g. an article from the New York Time). Other users can vote up (upvotes) and vote down (downvotes) these posts with more popular posts having more up-votes. The website is structured as a collection of various subreddits. Each subreddit has a certain topic ranging from the ordinary such as fitness, cooking, or gardening to the oddly specific such as birdswitharms (a subreddit featuring pictures of birds with photoshopped human arms). However, with over 22,000 subreddits, this should come as no surprise. When users submit posts, they submit to the subreddit that best characterizes the post. Users can also subscribe to their favorite subreddits to receive their content.

With over 22,000 subreddits, it can be difficult to decide and even know of which subreddits one should subscribe to. In this project, I built a Reddit subreddit recommendation engine that uses a user’s content and post history to determine the subreddits that best match his or her interests. Similar applications that already exist face certain problems: only crawling a subset of all subreddits, using humans rather than automation to generate recommendations, and not providing a personalized recommendation.

**Implementation**

***Front End***

HTML/CSS along with JQuery and the Bootstrap toolkit were used to build the website front end for the application. Communication with the backend uses AJAX (Asynchronous Javascript and XML) to interact with a Python based CGI script hosted on a remote server. This CGI script, which imports other custom Python modules, takes in a username and responds with the relevant recommendation.

***Storage***

In terms of data storage, MongoDB (a NoSQL database) is used partly because of its low setup overhead and partly because ACID properties are not important for this application.

***Data Extraction and Cleaning***

Before any recommendations could be generated, a sizeable pool of data had to be scraped from Reddit. PRAW (Python Reddit API Wrapper), a library built on top of the Reddit API, was used to access the data. For each subreddit, the text from the titles of the top 1000 posts (the posts with the most up-votes) was extracted and stored in the database using two keys. One key, called “name”, stored the name of the subreddit and the second key, called “text”, stored the full text from the titles (MongoDB documents are stored in BSON format). The text was further cleaned by removing all stop words, duplicates, numbers, and punctuation, converting all of the words to their stems, and sorting the words by their frequency within the text. The text from all of the subreddits was aggregated and parsed to generate a list of the most common words, and those words were also removed from the text of each subreddit. Because the Reddit API places a limit against many requests in a short period of time, the subreddit data extraction process took a lengthy amount of time of roughly two and a half days.

***Generating a Recommendation***

*Version 1 (Version that is used in the application)*

The recommendation engine takes in a username and generates the appropriate recommendation. Using PRAW, all comment and post title data are extracted for the user. Stop words, duplicates, numbers, and punctuation are removed, and the words are converted to their stems. Recommendations are generated by calculating a similarity value between the 200 most frequently used words used by the user based on post titles and comment text and the 40 most frequently used words of each subreddit as such:

Where is the length of (which is set to 40) and

The above is the theoretical formula used to calculate the similarity values. In the actual implementation code, however, the similarity values are multiplied by a constant weight of as such:

This modification widens the range of possible similarity values, mitigating possible issues with rounding and floating point comparisons. A list of subreddits sorted by similarity is then generated after calculating a similarity value for every subreddit. The top 20 most similar subreddits are then returned to the user as the recommendation. Running the application with this version takes roughly 45 seconds.

*Version 2*

The subreddits were first divided into various clusters. Hierarchical clustering was the ideal method to group all of the subreddits, but the runtime was prohibitively large. The *python-cluster* library runs at for single linkage where is the number of subreddits, which meant that clustering 20,000 subreddits would take roughly 26 days on my computer. The alternative was thus to use a single pass algorithm as such:

**for** subreddit **in** subreddits**:**

cluster **=** getMostSimilarCluster**(**subreddit**,** clusters**)**

**if** sim**(**subreddit**,** cluster**)** **<** threshold**:**

clusters**.**addNewCluster**(**subreddit**)**

**else:**

cluster**.**add**(**subreddit**)**

getMostSimilarCluster() traverses the list of clusters and returns the most similar cluster. The similarity between a subreddit and a cluster is defined as such:

getSim**(**subreddit**,** cluster**):**

total **=** 0

**for** othersubreddit **in** cluster**:**

total **+=** sim**(**subreddit**,** othersubreddit**)**

**return** total**/**len**(**cluster**)**

sim**(**subreddit**,** othersubreddit**):**

weight **=** len**(**subreddit**.**words**)**

i **=** 1

**for** word **in** subreddit**.**words**:**

**if** word **in** othersubreddit**.**words**:**

score **+=** weight**/(**0.5**\*(**i**+**othersubreddit**.**words**.**indexof**(**word**)+**1**))**

i **+=** 1

**return** score

othersubreddit**.**wordsand subreddit**.**words are the top 50 most frequent words for those subreddits. By dividing a constant weight by the average index of the shared term for the two word lists, this similarity function weighs terms that appear earlier in the list more heavily. This algorithm takes time (as opposed to where b is the number of clusters and N is the number of subreddits) because finding the similarity between a cluster and a subreddit takes time proportional to the size of the cluster (rather than constant time). This is in contrast to clustering other objects such as 2D coordinates, which takes time, as the average coordinate for a cluster can be saved and updated on each iteration. Whereas hierarchical clustering takes about 26 days to process, this single pass method takes 16.2 hours.

The clusters are stored in the database as follows. A mapping of subreddit name to cluster ID is stored in one collection and a mapping of cluster ID to the subreddit names contained within that cluster is stored in another collection. Cluster IDs are generated by keeping track of an ID variable and incrementing this variable by one and assigning this ID for every cluster created. Finding the subreddits that share a cluster involves finding the cluster ID mapped to a subreddit and extracting the subreddits mapped to that cluster ID.

During runtime, user data is extracted with PRAW and for each comment and post submitted by the user, the subreddit for which the comment resides in or post submitted to is recorded. After this process is complete, a sizeable pool of user subreddits will be generated. For each subreddit in this pool, all other subreddits that share a cluster are found by consulting the database and they are placed in another larger pool (for the sake of clarity, this larger pool is called the cluster pool). With this cluster pool, the process continues as in Version 1. For every subreddit in this cluster pool, calculate a similarity score between the subreddit and the user word list using the similarity equation seen in Version 1. A list of subreddits sorted by similarity can then generated. Whereas Version 1 of the recommendation implementation takes 45 seconds to complete, this second version is faster by only 4 seconds with a run time of 41 seconds.

**Example Execution**

The user enters a username into the front end form and this username is sent via AJAX to a Python CGI script hosted on a remote server. Correspondence uses data packed in a JSON format. Using PRAW (Python Reddit API Wrapper), content information for the user is scraped. If the username is invalid (a user does not exist for this username), the “success” key is set to false and the JSON data is immediately sent back to the front end application. If the username is valid, content data from the user’s comments and submission titles are scraped and parsed. For every subreddit stored in the database, a similarity score is calculated that determines how well the content of the subreddit matches the preferences/interests of the user. The top 20 subreddits with the highest similarity scores are then sent back to the front end in a JSON format. The front end receives the data and displays the recommendations to the user in a visually friendly manner. This entire process takes roughly 45 seconds, which is quite slow as far as web applications are concerned.

**Design Decisions**

As seen in Figure 1, the term distribution for a subreddit models Zipf’s law quite well as there is an apparent exponential decline in term frequency. This behavior is the inspiration for the similarity equation. Assuming that the frequency of a term is roughly proportional to how well that term characterizes a subreddit, the similarity calculation incorporates a multiplicative decrease in the weighting for every successive term. On the other hand, suppose the equation is modified to give a linearly decreasing weight.

Where is the length of (which is set to 40) and

After implementing both versions, a qualitative comparison of the recommendations from these two implementations reveals that the linear scheme does not seem to adequately capture the similarity characteristic as well as the multiplicative scheme.

The decision to only use the top 40 words of each subreddit and the top 200 words for each user is wholly based on a qualitative assessment. It seems as if words that fall beyond the top 40 do not properly characterize the subreddit. The top 200 words (as opposed to the top 40) of the user are used because fewer words are needed to characterize a subreddit compared to a user who may have diverse interests. For example, the top words used for the subreddit “cooking” include “recipe”, “cooking”, “chicken”, and “homemade”. A user’s top words may include “cooking”, “chicken”, “python”, “tools”, “parameter”, “C++”, “java”, “bash”, and “Cygwin”. This user is a pretty good match for the “cooking” subreddit, but he also has other words that make him a good match for other subreddits such as “programming”. Because a user generally has many interests (and thus more terms that characterize these interests), a larger pool of user words is needed to be matched with the top 40 words of each subreddit. This model resembles a slightly modified query-document matching system where the top 40 words of each subreddit are the queries matched against the user terms document.

Figure 1: Word Frequency graph. (Not all subreddit words are shown in this graph)

Interestingly, the runtimes for Version 1 (45 seconds) and Version 2 (41 seconds) are quite similar even though Version 2 should theoretically be much quicker than Version 1. This seeming anomaly suggests that the majority of the runtime occurs not in analyzing the subreddit data but in other overhead factors, such as queries with PRAW and the Reddit API and connection setup and teardown with the MongoDB database. Version 1 also takes advantage of spatial locality as all of the subreddit data are stored contiguously or near-contiguously in memory while Version 2 selects specific subreddit data among the database collections.

Therefore, Version 1 was chosen to be used with the application because its runtime is not significantly higher than Version 2 and it provides a more accurate recommendation for the user as it analyzes all subreddits rather than a subset. While one may believe that there is no need to analyze subreddits beyond those in the shared clusters, it may be the case that (1) the clusters are not wholly optimal and (2) certain subreddits may be relevant to the user that do not belong in a shared cluster. For the former case, a single pass rather than hierarchical algorithm was used, which means that the organization of the clusters is not as optimal as an arrangement generated by more sophisticated clustering algorithms. In regards to the latter case, take, for example, the subreddit “lifeprotips” which is a forum dedicated to posts about random tips that can improve one’s quality of life. These types of subreddits do not revolve around a specific topic, so their constituent terms do not necessarily provide an accurate characterization of the subreddit. During the clustering process, a subreddit such as “lifeprotips” may not be clustered correctly. Version 2 of the recommendation implementation does not properly handle such cases, while Version 1, which analyzes *all* subreddits, is much more robust.

**Possible Optimizations**

***Limiting the Subreddit Pool***

Unpopular subreddits are updated less frequently (they have fewer posts posted), which means that the user is probably not interested in these subreddits. A possible optimization is thus to eschew subreddits with fewer than X subscribers. Out of the 24,000 subreddits there are roughly 5000 subreddits with fewer than 30 subscribers, so it can be worthwhile to ignore these subreddits.

***Limiting the Term Pool***

Instead of scoring with the top 40 words of each subreddits and the top 200 words for each user, the size of these pools can be decreased to improve the running time. However, as with many other optimizations, an improved runtime trades off against the accuracy of the recommendation.

**Source Code**

All frontend and back end source code is hosted here: <https://github.com/ben444422/Recommendit>

The front end database, cgi script, and database are all hosted from my computer. Unfortunately, it seems as if Princeton firewall regulations prevent me from using my computer as server, so I will demonstrate the application during the meeting.