# OSSFinder

Final Project
COSC 445/545 Fall 2015

Dr. Mockus
Alex Klibisz (undergraduate, senior)
Muhammed Benkhayal (undergraduate, senior)
Yuxing Ma (graduate)

University of Tennessee, Knoxville

*This document is intended to be an informal description of our work. The purpose is to provide anyone else interested in continuing this project or a related project with a sufficient amount of information to understand our goals and implementation decisions.*

## Abstract

As of 2013, the open source hosting website Github.com had 10 million hosted repositories, hosting some of the most widely used languages, libraries, frameworks, and projects in the software industry. For programmers, using and contributing to open source software can boost productivity, teach new skills, and lead to improvements in existing software. However, such a magnitude of available software can make it difficult to find relevant software quickly.

We attempt to solve this problem by build a proof-of-concept recommendation engine for open-source software hosted on Github.com. Our engine uses feature-based searching to find projects that match a required feature and user-interaction data to build representations of similarity among repositories.

With this, we are able to prompt a programmer for a desired feature and repositories that the programmer has used in the past. We then use our engine to recommend repositories that the programmer should try.


## Implementation: Tooling

All of the data and software used in this project was stored and run on Dr. Mockus' da2 server in the EECS building. The data was stored using MongoDB. The software consisted of 1) a collection of python scripts written to retrieve, process, and run calculations on the data and 2) a web app written in python, HTML, and Javascript to create a user interface to the engine.

## Implementation: Data Collection

Our project required the following data from Github.com:

- Repositories
- Users
- Commits
- Pull Requests
- Pull Request Comments
- Issues Raised
- Issue Comments
- Forks
- Watchers

We narrowed down our scope by selecting only repositories with a minimum of 20 forks and 20 watchers.

Dr. Mockus was able to provide us with a snapshot of Github from within the past year. This was a major time-saver as we did not have to retrieve this data ourselves. We ran python scripts to find and transfer all repositories matching our requirements into a separate Mongo database. We then ran scripts to find all of the users, commits, pull requests, etc. that corresponded to any of the matched repositories.

After running these scripts, our data looked as follows:

| Collection | Description | Documents | Size |
|---|---|---|---|
| commits | Information on all of the commits | 493,859 | 233.13mb |
| forks | Information on all of the forks, does not have the user who performed the forks information | 2,347,681 | 589.26mb |
| forks_fixed | Fixed collection of forks, includes the user who forked's information | 2,347,681 | 709.51mb |
| issue_comments | Information on all of the issue comments | 6,384,620 | 12,295.29mb |
| issues | Information on all of the issues | 3,052,225 | 5718.74mb |
| pull_requests | Information on all of the pull requests | 2,075,034 | 3,926.18mb |
| rel_aggregate | Holds the total scores for repos, incomplete due to threading | 3,535 | 0.65mb |
| rel_commits | All the relations between repos based on commits | 19,492 | 2.85mb |
| rel_forks | All the relations between repos based on forks, incomplete | 906,503 | 105.04mb |
| rel_forks2 | All the relations between repos based | 31,670 | 4.83mb |

| | | | |
|---|---|---|---|
| | on forks, threaded, incomplete | | |
| rel_issue_comments 2 | All the relations between repos based on issue comments, threaded, incomplete | 33,354 | 4.72mb |
| rel_issuecomments | All the relations between repos based on issue comments, incomplete | 1,185,858 | 157.63mb |
| rel_issues | All the relations between repos based on issues, incomplete | 1,236,382 | 150.53mb |
| rel_issues2 | All the relations between repos based on issues, threaded, incomplete | 22,016 | 2.75mb |
| rel_pulls | All the relations between repos based on pulls, incomplete | 131 | 16.70kb |
| rel_watchers | All the relations between repos based on watchers, incomplete | 240,373 | 28.49mb |
| rel_watchers2 | All the relations between repos based on watchers, threaded, incomplete | 48,776 | 5.90mb |
| relationships | | 9 | .98kb |
| repos | | 17,000 | 79.95mb |
| repositories | All of the repositories we used, some are duplicated | 22,560 | 105.64mb |
| repositores_unique | All of the repositories we used, no repetition of respositories | 12,368 | 61.86mb |

| | | | |
|---|---|---|---|
| watchers | Information on all watchers | 4,848,895 | 4,393.43mb |

## Implementation: Feature Search

We retrieved all the readme files of the 12,368 repos using the Github API. We try to find the features of these projects by applying natural language processing on these readme files.
We use a python library called simserver in gensim, which is recommended by Dr. Mockus. Basically this library provides the 'Find similar' service. The server works in the following steps:

1. converts documents to semantic representation
2. indexes documents in the vector representation for faster retrieval
3. for a given query, return ids of the most similar documents from the index

Techniques and principles used in numerical calculation and indexing:

1. TF-IDF （term frequency - inverse document frequency）
   Formula for unnormalized weight of term *i* in document *j* in a corpus of *D* documents:
   weight_{*i*,*j*} = frequency_{*i*,*j*} * log_2(D / document_freq_{*i*})
2. LSI (latent semantic indexing)
   An indexing and retrieval method that uses SVD(singular value decomposition) to identify patterns in the relationships between terms in documents.

With this library, we build a constant model(kind of a small database) on webserver, which can give the corresponding reply to user's search request.

## Implementation: Relationship Calculations

The relationship calculations were based on a user's interaction with repos. The idea was to find the repos that a user interacted with, and collect data based on that. For example, if user A has committed to both repo 1 and repo 2, a relationship would be created with a commit score of 1. If user B also committed to both repo 1 and repo 2, the commit score would be incremented to 2. This was done individually for each type of relation: commits, forks, issues, issue comments, pulls, and watchers. Then, the data was aggregated into a single MongoDB collection and given a score based on the relationships that were recorded. The value of a relationship was weighted based on the total amount of each interaction type. Below is a table showing this:

| Type | Approximate Amount | Weight |
|---|---|---|
| Commit | 500 K | 13 |
| Fork | 2.5 M | 2 |

| | | |
|---|---|---|
| Issues | 3 M | 2 |
| Issue Comments | 6.5 M | 1 |
| Pulls | 2 M | 3 |
| Watchers | 6 M | 1 |

The point value for each relationship type was based on how many of that relationship type existed compared to the other types. For example, since there are 500,000 commits (smallest amount) and 6,500,000 issue comments (largest amount), 6.5 M / 500K = 13, each issue comment was worth one point and each commit was worth 13.

The initial scripts written to find the relationships of each relation type were not running fast enough to finish in time, so we did attempt to run threaded versions of the scripts. This is talked about in more depth in the challenges section.

## Implementation: Recommendations

The process by which we generate our recommendations follows the logic:

1. Input:
    a. R1: List of repos that match the user's feature search.
    b. R2: List of repos that the user has used before.
2. Output
    a. R3: List of repos in R1 that have a relationship > 0 with any repo in R2.
    b. Order the repos in R3 based on their total number of relationship points with repos in R2.

This means that R3 will always be a subset of R1. The repos in R1 that make it into R3 are determined by whether or not they have a relationship with any of the repos in R2.

## Implementation: Webapp

We used the collected data, feature search, and relationship calculations to develop a web app that allows users to query the OSSFinder recommendation engine.

The user workflow is as follows:

1. Enter a feature as a text string (ex: *graphing*, *matrix algebra*, *data binding*).
2. Select one or more repositories from a list of roughly 20,000 repositories in our database.
3. Click a button "Get Recommendations" to see the recommended repositories.

On the server-side, the webapp is implemented using the python Flask module to expose several HTTP endpoints. For the client side, the server serves a very minimal single-page application build with Angular.js.

The server exposes the following endpoints:

| Type | Endpoint | Payload | Description |
|---|---|---|---|
| GET | /repos | N/A | returns a list of all repos in our database |
| POST | /search/feature | { query: [string] } | uses the query string to run a feature search against our repos, returns the matching repositories |
| POST | /search/recommendation | { query: [string], repos: [array[object]] } | uses the query string to run a feature search against our repos, returns any repos that were returned from the search that also share a relationship with the repos that the user selected. |

# Results

Here are some examples of the queries and recommendations our system was able to produce.

# Challenges

## Search-Related

While retrieving readme files, we use github API and there is a limitation on the max number of requests that can be sent to github: 60 for normal users and 5000 for authenticated users. We attempted to retrieve readme files directly, but failed because we don't know the specific readme file names.(can be different in file extension)

When we were testing the feature search module, we found that if the query is more than one word, the simserver will return some repos that are highly relevant to only one word in the query. For example, if we do a 'javascript framework' query, many repos returned are highly related to only one word(javascript or framework), which is obviously not what we want. For now, we add an 'AND' filter on the repos returned from search engine to make sure the repos consist all the query words.

### Data-Related

Our most significant challenges arose from having to process such a large amount of data. Some of the scripts for data collection and relationship calculation took several weeks to run. In general, our data processing scripts were structured as:

1. fetch a subset of documents from MongoDB via pymongo,
2. process data in memory using python,
3. send resulting data back to another collection in MongoDB.
4. fetch the next subset of documents, repeat.

Towards the end of the project, we attempted to improve this performance by implementing threading to create a divide-and-conquer approach for processing the data from MongoDB. For example, given we have 10,000 documents, we would:

1. spawn 10 threads
2. tell the first thread to retrieve commits 0 through 999, the second to handle 1,000 through 1,999
3. within each thread, call a common function to process the data, giving the subset of documents as a parameter.

For the aggregate relationship calculations, it worked correctly. This unfortunately did not work properly, and the serial processing yielded much more results for the relationship calculations, so we opted to use those results instead. We see two potential options for the failure of threading with relationship calculations:

1. We spawned too many threads at once, each requesting data from MongoDB, and our requests either failed or timed out because there were so many of them.
2. We attempted to make too many concurrent writes to MongoDB, corrupting the data.

## Continued Work, Improvement

Some suggestions for continuing or improving this work:

1. Create a faster, more repeatable approach for retrieving data and calculating relationships. Right now the collection of scripts that we have used for this is somewhat inconsistent and very slow.
2. Make the feature search more robust. Right now we are using only readme files to populate a search model for our feature search. It could potentially be useful to use the text content from a repo's issues or pull requests to give the search model more to work with. Also it could be better to find new algorithm to do the feature-searching or to

improve the current one by adjusting parameter values. Finding ways to evaluate the searching results always matters.

3. Host the application on a publicly-accessible server. Right now all of the data and the application lives on the da2 server, which is only accessible via SSH and port tunneling to use the webapp. Given the quantity of data and the necessary processing power to do any further processing of data, it could be very expensive to host the data and application on any paid infrastructure.