File System Search Engine

EECS 767 Final Project Report

Jace Kline
jace_kline@ku.edu

Ishrak Hayet
ihayet@ku.edu

Manoj Thangavel
manoj.t@ku.edu

I. Introduction

The aim of this project was to implement an information retrieval system to enable local file system-based queries. The nature of the file system domain poses unique challenges and considerations that diverge from the more traditional web search-based information retrieval systems. The unique challenges faced in this domain are as follows:

1. The need to search for files based on the similarity of a query to both content and file name.

2. The program is not assumed to be continuously running unlike in the web scenario. Hence, efficient index storage, retrieval, and updates upon program start are of great importance.

3. File type and format heterogeneity is present across files on file systems. This results in a significant parsing, text-processing, and indexing challenge. This heterogeneity property also makes it difficult to offer a uniform way of opening and viewing selected files.

This project sufficiently addresses both challenges (1) and (2). However, the final challenge proved to fall largely outside our project's scope due to complexity and time constraints.

Some assumptions we make for this project include the following. First, we assume that the target file system exists on a Linux operating system. Next, to address challenge (3) outlined above, we treat all files as text files for the purposes of text processing. We assert that any file able to be rendered by a modern web browser can be viewed from our application. Lastly, we assume that the target system has a web browser able to display our web-based graphical user interface (GUI) for querying and retrieving results.

The remaining sections proceed as follows. In section II, we discuss the high-level architecture and technology stack leveraged in our application. Next, in section III, we outline the general algorithm our application executes upon invocation. We also introduce each module and discuss its implementation details. In section IV, we evaluate the performance and accuracy of our application. In section V, we discuss the current limitations of our approach along with future work suggestions. Lastly, we conclude in section VI.

II. Architecture Overview

The architecture of our application consists of two high-level components. The backend component, compiled to a binary, is written in Rust. We chose Rust as it is a growing language with low level control, good performance, and type safety. The purpose of the backend is twofold. First, it is responsible for performing the setup process, including scraping file system paths, updating the index, and persisting the index to the file system. The backend component is also responsible for deploying a web server. This web server has three roles. First, it is used to serve the web browser-hosted user interface HTML and related resources. Second, it serves REST API endpoints that are accessed from the user interface to make queries and provide relevance feedback. Lastly, it serves files on the filesystem so they can be viewed locally in the web browser.

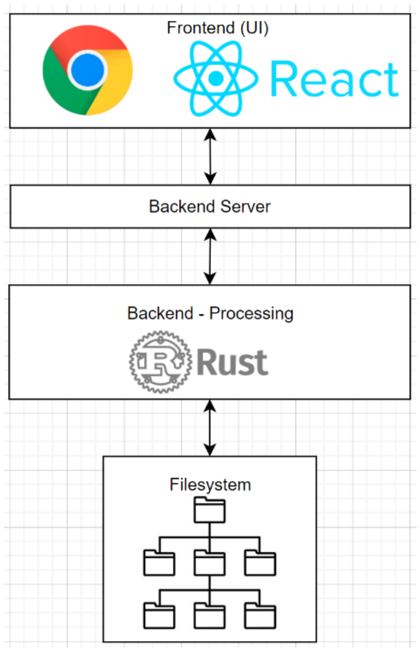


Figure 1 - Application Architecture

This last component is important if a user wants to preview a file's contents when the file appears in a query's ranked results list.

The next major component in our application is the frontend. This is the user interface of our application displayed in the browser and is built using the ReactJS framework. The purpose of the frontend is to allow the user to make queries, provide relevance feedback, and view the history of files they have opened and queries they have made. As mentioned above, the frontend communicates with the backend via the exposed API.

III. Implementation

In this section, we first discuss the high-level algorithm employed by our application. Next, we highlight key details from the implementations of sub-components of our application.

3.1. Algorithm Overview

The inputs to our program consist of (1) the path to the root of the portion of the filesystem to index and, optionally, (2) the file path to the stored index file from the last run, or the path to the location to store the index (if first run). We chose to allow flexibility for the root indexing path to allow for easier testing, easier evaluation, and to allow use cases that don't involve scraping and indexing the entire file system. The program is invoked from a shell on the local machine and should be supplied the arguments corresponding to (1) and (2) above. Figure 2 shows the pseudocode representing the algorithm used by the application.

3.2. Index Construction & Persistence

Our index consists of three sub-indexes, stored as binary tree (BTree) maps. The first index maps file paths to their metadata. This metadata includes the file's last modification time and file length, which is used on the following program invocation to determine whether the file has been modified or removed

Inputs: INDEX_ROOT_PATH, PERSIST_INDEX_FILE
Algorithm:

- Scrape all paths and info recursively from INDEX_ROOT_PATH
- Read and deserialize the stored index from PERSIST_INDEX_FILE
- Calculate the “diff” between persisted and scraped file paths
    - Use path, modification time, and file length to detect changes
    - Tag each scraped and previously indexed path as ‘new’, ‘modified’, ‘removed’, or ‘unchanged’
- For each path, update the index according to its tag (if needed)
    - If tag = ‘new’ | ‘modified’, run text-processing and indexing
    - If tag = ‘removed’, purge from index
    - If tag = ‘unchanged’, do nothing
- Store the serialized updated index back to the PERSIST_INDEX_FILE path
- Precompute document vectors from the updated index frequency information
- Start the local web server
    - Serve the UI page and static resources
    - Serve the API for making queries
    - Serve the local file system files with the root of INDEX_ROOT_PATH
- Listen for query requests at “/api/query”…
- On a request, return rankings based on:
    - Content -> Cosine similarity
    - File name -> Levenshtein similarity
    - If relevance feedback given, compute the centroid of relevant path document vectors and use Rocchio algorithm to compute a shifted query vector, then re-rank

Figure 2 - Application Algorithm

between program runs. The next index, called the “file-term index”, maps file paths to each term present in that file, along with the frequency of that term in the file. This index is used to quickly build the document vectors, as each document vector is represented as a file path mapped to its terms and the associated TF-IDF weight. Hence, the document vectors are computed as a map operation over this file-term index. The last sub-index is a traditional inverted index. This index maps terms to the documents they are present in, along with frequency and position information.

We chose to use binary tree maps as opposed hash maps, as this allowed us to exploit the inherent string-ordering of our keys to perform some operations more efficiently. This ordering property is leveraged when computing the differential between the previous file system state and the scraped file system state when computing the updated index. With both represented as maps with string-ordered file path keys, we “zipper” the maps together. If a key conflict occurs, we check the file info metadata to see whether the file has been modified and tag it accordingly. If there is not a match for a particular file path key, then either the path represents a new file or a removed file, depending on the map it originated from.

To persist our index across runs, we used the BSON binary object format to save our index to a file. This allows us to save considerably on storage space as well as serialization and deserialization time as opposed to using a naïve string-based format such as JSON.

3.3. File System Scraping

The scraping module of our application is responsible for collecting the file paths and metadata of all file paths under a given root path. We implement this using a recursive traversal of the files and directories. The paths outputted by the scraping process are used in computing the file system differential against the previously stored index paths.

3.4. Text Processing

Text processing consists of a series of modifications and normalizations to raw input text. These steps are performed on both the text from files as well as the text from raw queries.

The first step in this process is *tokenization*. This consists of removing punctuation and splitting tokens on white space delimiters.

Next, we perform *case-folding*. This involves mapping over each token found during tokenization and converting the characters to their lowercase equivalents, if applicable.

To represent each token in a standard form that discards inflection, ownership, or multiplicity, we leverage *stemming*. In essence, this converts each term to its “root” form. Particularly, we use the Porter Stemmer, which consists of 5 rounds of iterative suffix reductions each term. The Rust language has a library that implements the Porter Stemmer, and therefore we utilized this in our project.

The last step in our text processing module is used to remove stop words. Stop words are words that are not good differentiators of candidate documents due to their extremely high prevalence and frequency across a document corpus. In addition, stop words inflate the size of the index, increase the size of document vectors, and, in general, increase the processing time for queries. Hence, we simply remove these tokens from the processed token stream.

3.5. Scoring

As discussed in the introduction, a key consideration for the scoring mechanism in a file system search engine is that of scoring based on file content as well as file (or path) name. There are two ways to tackle this issue. The first is to construct a combined score metric that attempts to infer the intent of the user query and weight the file content versus file name similarities accordingly. The alternative approach, employed by our application, involves scoring and ranking based on content and file name, and then returning the results separately.

For scoring based on content, we construct a vector representation of each document (file or query). The cosine similarity score computed between two document vectors determines how similar two documents are. The components of the document vectors represent the TF-IDF weight between a document and each term in the corpus. Hence, the dimension of the vector space is equivalent to the number of unique terms present in the document corpus. The TF-IDF weight for term t and document d is computed as follows:

$$w_{t,d} = tf_{t,d} \times idf_t$$

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

In these equations,

- $tf_{t,d}$ represents the term frequency of term t in document d.
- df_t represents the number of documents term t occurs in.
- N is the number of documents in the corpus.

The cosine similarity between two document vectors $\vec{q}$ and $\vec{d}$ is computed by the following:

$$\cos(\vec{q},\ \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} \in [0,1]$$

In our implementation, we represent each document vector as a binary tree (Btree) map from the its terms to the TF-IDF weights associated. We also pre-compute and store the magnitude of each vector to save time at query time. Note that we only store the terms that have non-zero weight for each vector. To compute the dot product between two vectors, we leverage the ordered property of the map keys (terms) to perform a "zipper" of the two vector representations. We multiply the weights of each matching term present in both maps, and then add these together to arrive at the dot product. Our implementation is considerably more efficient than storing and a sparse matrix of weights for each document and term pair.

For a given query, we prune candidate documents by selecting only those that share at least one term with the query. We return the results ranked in descending order based on the cosine similarity of the candidate documents.

For scoring based on file name or file path similarity, we utilize a string similarity function known as Levenshtein distance. In essence, this function computes the minimum number of operations needed to convert one string to another. The fewer number of operations needed, the lower the Levenshtein distance. This function is defined recursively as follows:

$$L(s_0, s_1) = \begin{cases} len(s_0), if\, s_1\ empty \\ len(s_1), if\, s_0\ empty \\ L\big(tail(s_0), tail(s_1)\big), if\, s_0[0] = s_1[0] \\ 1 + \min\begin{Bmatrix} L\big(tail(s_0), tail(s_1)\big) \\ L\big(s_0, tail(s_1)\big) \\ L\big(tail(s_0), s_1\big) \end{Bmatrix}, \text{otherwise} \end{cases}$$

To achieve a score that is independent of the length of the string and results in a higher score for more similar strings, we define a normalized Levenshtein distance as follows:

$$L_n(s_0, s_1) = 1 - \frac{L(s_0, s_1)}{\max\big(len(s_0),\ len(s_1)\big)} \in [0,1]$$

To ensure we account for the user searching for the full path or the file name, we compute the final score for some query string q and document d as:

$$score(q, d) = \max(L_n(q,\ d.path),\ L_n(q,\ d.filename))$$

When given a query string, we compute this normalized score for each of the documents in the index. We do not have a pruning mechanism for this name-based score and therefore consider all documents as candidates.

The last key aspect of scoring involves relevance feedback. This allows a user to select a subset of results that they deem "relevant" for a given query, based on their information need. Our implementation only supports the notion of "positive feedback" since we allow users to only specify which documents are relevant instead of which documents are not relevant. Relevance feedback is modeled in our vector model space by shifting the query vector towards the centroid (mean) vector of the given relevant documents. The cosine similarity scores are then re-computed based on this new query vector. We compute the new query vector by utilizing the Rocchio algorithm, shown below:

$$\vec{q} = \alpha\overrightarrow{q_0} + \beta\frac{1}{|D_r|}\sum_{\vec{d} \in D_r}\vec{d}$$

We use values $\alpha = 1,\ \beta = 0.5$ in our implementation. Note that the set D_r represents the set of document vectors for the documents marked "relevant" by the user.

3.6. Backend Server

The backend server is responsible for serving the user interface HTML and resources, serving the API endpoints for making queries and receiving results, and serving files on the file system to be viewed in the browser. We host the user interface page at the root path "/", the API endpoints at "/api", and the file system files at "/document". To deploy the web server, we leveraged a Rust library called Rocket.

3.7. User Interface

The user interface (UI), built using ReactJS, is a web page that communicates with the backend server described in the previous section. The main features of the UI are the search bar, results lists, search history, and visit history. As discussed in the scoring section, we have two results lists, one for content similarity and the other for file name (path) similarity. When a result in either result list is clicked, the associated file from the file system is viewed in a new tab in the browser. Each result also has an associated checkbox signifying whether it is marked as "relevant" for the purposes of relevance feedback. When a checkbox is clicked, the file path is stored in a list. If a query is submitted with a non-empty relevance list, the backend performs relevance feedback and responds with new results. Other features of the UI include the time to fetch results from a query, and a dropdown box to specify the number of results to return. Below we have included images that show layout of the UI. In addition, a video demo (MP4) of the UI can be downloaded at the following link in our GitHub repository: https://github.com/jace-kline/eecs-767-project/blob/main/767-demo.mp4.

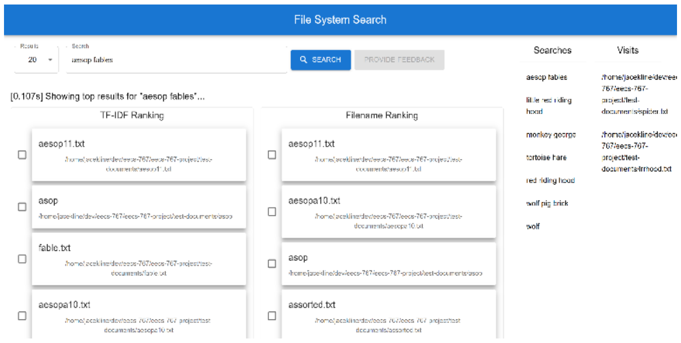


Figure 3 - UI view 1

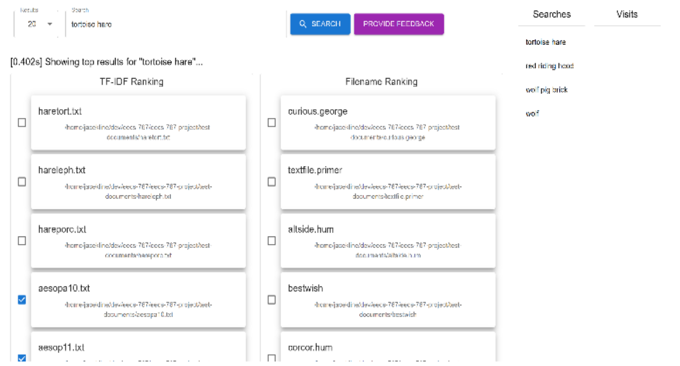
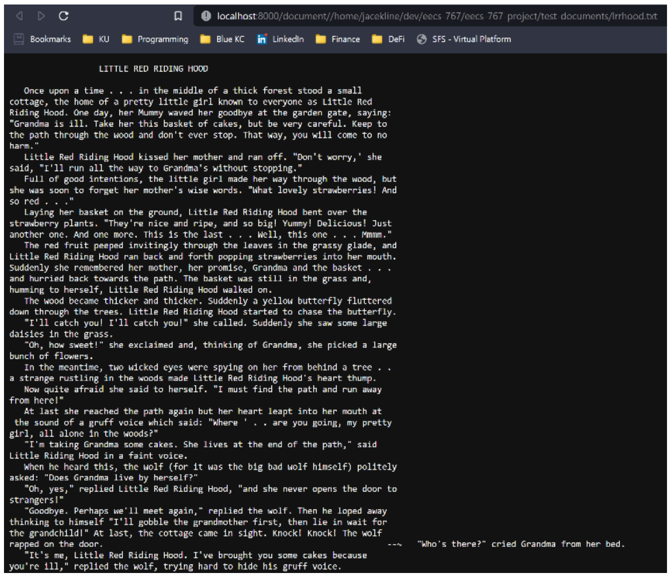


Figure 4 - UI view 2 (relevance feedback checked)


localhost:8000/document//home/jacekline/dev/eecs 767/eecs 767 project/test documents/lrrhood.txt
Bookmarks
KU
Programming
Blue KC
LinkedIn
Finance
DeFi
SFS - Virtual Platform

```
                 LITTLE RED RIDING HOOD

   Once upon a time . . . in the middle of a thick forest stood a small
cottage, the home of a pretty little girl known to everyone as Little Red
Riding Hood. One day, her Mummy waved her goodbye at the garden gate, saying:
"Grandma is ill. Take her this basket of cakes, but be very careful. Keep to
the path through the wood and don't ever stop. That way, you will come to no
harm."
   Little Red Riding Hood kissed her mother and ran off. "Don't worry,' she
said, "I'll run all the way to Grandma's without stopping."
   Full of good intentions, the little girl made her way through the wood, but
she was soon to forget her mother's wise words. "What lovely strawberries! And
so red . . ."
   Laying her basket on the ground, Little Red Riding Hood bent over the
strawberry plants. "They're nice and ripe, and so big! Yummy! Delicious! Just
another one. And one more. This is the last . . . Well, this one . . . Mmmm."
   The red fruit peeped invitingly through the leaves in the grassy glade, and
Little Red Riding Hood ran back and forth popping strawberries into her mouth.
Suddenly she remembered her mother, her promise, Grandma and the basket . . .
and hurried back towards the path. The basket was still in the grass and,
humming to herself, Little Red Riding Hood walked on.
   The wood became thicker and thicker. Suddenly a yellow butterfly fluttered
down through the trees. Little Red Riding Hood started to chase the butterfly.
   "I'll catch you! I'll catch you!" she called. Suddenly she saw some large
daisies in the grass.
   "Oh, how sweet!" she exclaimed and, thinking of Grandma, she picked a large
bunch of flowers.
   In the meantime, two wicked eyes were spying on her from behind a tree . .
a strange rustling in the woods made Little Red Riding Hood's heart thump.
   Now quite afraid she said to herself. "I must find the path and run away
from here!"
   At last she reached the path again but her heart leapt into her mouth at
 the sound of a gruff voice which said: "Where ' . . are you going, my pretty
girl, all alone in the woods?"
   "I'm taking Grandma some cakes. She lives at the end of the path," said
Little Riding Hood in a faint voice.
   When he heard this, the wolf (for it was the big bad wolf himself) politely
asked: "Does Grandma live by herself?"
   "Oh, yes," replied Little Red Riding Hood, "and she never opens the door to
strangers!"
   "Goodbye. Perhaps we'll meet again," replied the wolf. Then he loped away
thinking to himself "I'll gobble the grandmother first, then lie in wait for
the grandchild!" At last, the cottage came in sight. Knock! Knock! The wolf
rapped on the door.                                                  --~   "Who's there?" cried Grandma from her bed.
   "It's me, Little Red Riding Hood. I've brought you some cakes because
you're ill," replied the wolf, trying hard to hide his gruff voice.
```

Figure 5 - Viewing a result file in the web browser

IV. Evaluation

4.1. Efficiency Evaluation

To test the indexing and query processing efficiency of our application, we design a test over a directory of 423 sample text documents. These documents contain a total of 31409 terms.

We first perform indexing with no pre-stored index file and find the average scraping, text-processing, and indexing time to be 37.58 seconds. Upon repeating the experiment with the stored index, we find that this process takes only 3.81 seconds. We conjecture that most of this time is spent reading, deserializing, reserializing, and writing the index back to the persistent index file. Overall, we see that persisting the index results in a startup time reduction of nearly 90% in this scenario. We also find that the stored index file size is 30.2MB.

Next, we perform an analysis of the query processing and response times by the server. For these tests, we implement the benchmarking code on the server side to reduce possible network delay variability. We test 5 different queries and find the average response times for these queries under 3 scenarios. The first scenario involves using only the cosine similarity score while excluding the file name (path) similarity score. In this case, we find an average processing time of 0.044 seconds per query. Next, we include the file name similarity score and find the average processing time to be 0.105 seconds. This shows that including this additional scoring mechanism results in a 239% increase in processing time. Lastly, we consider the case where the user includes relevance feedback. In this scenario, we assume that the user supplies 3 randomly chosen relevant documents out of a set of originally returned results for each query. We also disable the file name similarity ranking. We find that including relevance feedback also results in a slowdown, with an average response time of 0.106 seconds.

4.2. Accuracy Evaluation

For the accuracy evaluation, we use the same set of test documents from the previous section. This document corpus consists of text documents that contain common English language fables. These include stories such as Little Red Riding Hood, Three Little Pigs, Aesop’s Fables, Curious George, and more. To test our scoring mechanism, we perform 4 different queries while restricting the number of returned results to 5. We discuss the expectations and results of each query.

The first query we perform is “wolf pig” with the intention of finding the Three Little Pigs story document. The first result is indeed ‘3lpigs.txt’ with a cosine similarity of 0.79. Each of the other results has a cosine similarity below 0.44 which indicates that the top document was overwhelmingly the best match.

The next query we perform is “wolf forest grandma” with the intention of finding the Little Red Riding Hood story. As expected, the first result is ‘lrrhood.txt’ with a score of 0.61. The second result is a parody of Little Red Riding Hood called “Big Red Riding Cape”, which resulted in a cosine similarity of 0.59. The remaining documents have cosine similarity scores less than 0.3. This is the result we expect based on our query and information need.

Next, we query with the term "Aesop Fable" to attempt to return the top documents that contain Aesop's Fables. Out of the top 5 results, we see that 3 of them are relevant to the query as they contain a variation of "Aesop" in the document name. Interestingly, these 3 documents are the top 3 documents based on file name search.

In the last query, we attempt an exploratory search to find stories that contain witches. We therefore query with the term "witch". We find that the top documents are all notable stories that feature witch characters, namely Magic Tinder Box, Hansel and Gretel, and Little Miss Mermaid. These results meet our information need.

We have shown that our scoring mechanism functions as intended. We can query with the goal of searching by either content or file name to retrieve expected results.

V. Limitations & Future Work

The implementation of our file system search engine covers all the facets of a working and functional information retrieval system. With this, there are potential improvements that could be addressed in future iterations of the application.

The first improvement involves extending the text processing module to recognize and uniquely parse documents based on their file type. In the current implementation, we assume that all files are raw text files for the purposes of text processing. However, a more future implementation could include special parsing of HTML, PDF, JSON, DOCX, etc.

Another consideration for the future is the security aspect of our implementation. Currently, the backend serves the entire file system via a webserver. Although the web server is only accessible via the local machine, this solution could still pose security risks. In addition, our current solution leverages the web browser instead of each file type's preferred application to view file system files. Overall, this is not an ideal setup for a production grade application.

To improve efficiency at scraping and indexing time, another improvement could be to calculate and store document fingerprints based on a hash of their content. This could address the issue of determining whether files are modified and could also be used to detect duplicate and moved files.

The next consideration is that of periodic runtime re-scraping and re-indexing of modified files on the file system. This could be implemented on separate threads to ensure continued availability of the web server. One downside of this is that each modification of the index requires an entire re-computation of all document vectors. However, if performed on background threads, this would allow the user to make changes to the file system and still maintain a nearly up-to-date index.

Empirically, another useful feature to add to the application shall include the filtration of which documents and directories are indexed based on supplied lists of "include" or "exclude" patterns. In essence, this would have the same type of behavior as the ".gitignore" file has for Git repositories. This could allow a user to define targeted document collections to index and query while ignoring those they do not care about.

Lastly, a rethinking of the file name similarity measure would be beneficial in a future iteration of our application. Currently, the Levenshtein distance-based metric is time consuming and lacks a pruning mechanism. In addition, users may not want to have receive two separate results lists based on the file name (path) and the file content.

VI. Conclusion

The goal of this project was to implement an information retrieval system targeted at local Linux file systems. In this IR system, the document corpus consists of all text-based files that the given user has permission to read under a given root path on the file system. The architecture consists of a backend written in Rust, and a frontend user interface developed in ReactJS. The frontend sends queries and receives ranked results from the backend via an API exposed by the backend. Unique challenges encountered in the file system domain include (1) the need to answer queries based on both file name (path) and content similarity measures, (2) the fact that the application is not continuously running, and therefore must take measures to persist its data structures across executions, and (3) the file types and formats found on file systems are diverse. The first challenge is tackled by implementing two separate scoring and ranking mechanisms. We use a vector model based on TF-IDF weighting, along with a cosine similarity measure, to score the similarity of queries against documents. We leverage a variant of the Levenshtein string distance measure to address file name-based similarity. We address the second challenge through serialization and persistence the index used across executions. The last challenge, due to its complexity, was not fully confronted and handled by our implementation. As a result, we assert that all files and queries are text-processed as if they were plain text files. Future work includes further addressing this challenge by invoking different parsing and text processing modules based on recognized file types.

Our source code is available at https://github.com/jace-kline/eecs-767-project.