AI Aided Annotation

CS 4624
Multimedia, Hypertext, and Information Access

Report
May 11, 2022
Blacksburg, VA, 24061

Instructor: Dr. Edward Fox
Client: Satvik Chekuri
Team: Jonah Bishop, Isaac David, Ishaandeep Lubana
# Table of Contents

- **List of Figures**  
  - 4
- **List of Tables**  
  - 5
- **1.0 Abstract**  
  - 6
- **2.0 Introduction**  
  - 7
  - 2.1 Objectives  
  - 7
  - 2.2 Deliverables  
  - 7
  - 2.3 Client  
  - 7
  - 2.4 Team Members and Roles  
  - 8
- **3.0 Requirements**  
  - 9
  - 3.1 Upgrade Existing Website/Code  
  - 9
- **4.0 Design**  
  - 10
- **5.0 Implementation**  
  - 11
  - 5.1 Artificial Intelligence  
  - 11
    - 5.1.1 Keyword Generation with RAKE  
    - 11
    - 5.1.3 Sentence Selection with Maximal Marginal Relevance  
    - 11
  - 5.2 Server Backend  
  - 13
  - 5.3 Website Frontend  
  - 14
- **6.0 Testing/Evaluation/Assessment**  
  - 15
  - 6.1 Website Walkthrough  
  - 15
  - 6.2 Google Survey Questionnaire  
  - 15
  - 6.3 Survey Results and Implications  
  - 16
- **7.0 User Manual**  
  - 17
  - 7.1 Website Walkthrough  
  - 17
    - 7.1.1 Top of the Homepage  
    - 17
    - 7.1.2 Middle of the Homepage  
    - 18
7.1.3 Diagram of MMR Workflow
7.1.4 Deep Explanation of MMR
7.1.5 Uploading a Document
7.1.6 Document Successfully Uploaded
7.1.7 Selecting the Sentences
7.1.8 Annotating the Data
7.1.9 Exporting the Data
7.1.10 Format of the JSON Data

8.0 Developer's Manual

8.1 Inventory of All the Files

8.1.1 Python Files
  8.1.1.1 mmr.py
  8.1.1.2 sentence.py
  8.1.1.3 server.py
  8.1.1.4 mongo.py
  8.1.1.5 tests.py

8.1.2 HTML and CSS Files
  8.2.1.1 intro.html
  8.2.1.2 index.html
  8.2.1.3 style.css

8.1.3 JavaScript Files
  8.1.3.1 api.js
  8.1.3.2 client.js
  8.1.3.3 demo_article.js
  8.1.3.4 view.js

8.1.4 Data Files
List of Figures

Figure 1: MMR Algorithm Formula

Figure 2: Chart showing the similarity between each document and the query

Figure 3: The top of the Homepage

Figure 4: The middle of the Homepage

Figure 5: Diagram explaining how the MMR algorithm works

Figure 6: Low level overview of MMR algorithm

Figure 7: Upload button pop-up

Figure 8: The three columns to select the diverse sentences

Figure 9: Main three columns with selected sentences filled

Figure 10: The generated summary and questions and answers column

Figure 11: Exporting Data

Figure 12: Format of the JSON data

Figure 13: A function that returns a webpage

Figure 14: A wrapper for mmr.summary_generator

Figure 15: A wrapper for server._testable_phase_two

Figure 16: The json_request lambda function

Figure 17: Method to redirect server JSON request
List of Tables

Table 1: Each of the Python files with where their Section is and a brief description
Table 2: Each of the HTML files with where their Section is and a brief description
Table 3: Each of the JavaScript files with where their Section is and a brief description
1.0 Abstract

Human annotation of long documents is a very important task in training and evaluation in NLP. The process generally starts with the human annotators reading over the document in its entirety. Once the annotator feels they have a sufficient grasp of the document, they can begin to annotate it. Specifically, annotators will look for questions that can be answered, then write down the question and answer. In our client’s case, the chosen long documents are electronic theses and dissertations (ETDs) which are often 100-150 pages minimum thereby making it a time consuming and expensive process to annotate. The ETDs are annotated on a chapter-by-chapter basis as content can vary significantly in each chapter. The annotations generated are then used to help evaluate downstream tasks such as summarization, topic modeling, and question answering.

The proposed model/system will aid the annotators in the creation of a Knowledge Base that is rich with topics/keywords and Questions-Answers for each chapter in ETDs. The core of the system revolves around an algorithm known as Maximal Marginal Relevance (MMR). By utilizing the MMR algorithm – consisting of a changeable lambda value, keywords, and a couple of other elements – we can identify sentences based on their similarity or diversity relative to a collection of sentences. This algorithm would greatly enhance the annotation process in ETDs by automating the identification of the most relevant sentences. Thus, annotators do not have to sift through the ETDs one sentence at a time, instead making a comprehensive summary as fast as the MMR algorithm can work. As a result, annotators can save many hours per ETD, resulting in more human generated annotations in a shorter amount of time.

Previously, a team worked with our client Satvik Chekuri and made an initial website, but the project was more focused on interactivity with the MMR algorithm. They essentially completed the first part of the project of identifying and selecting sentences but left plenty of work for us to make this a useful tool for annotators. In order to expand upon the previous team’s work, we needed to automate the selection of sentences, generate similar sentences and keywords, create a text area for the actual annotations, and finally add the ability to export the selected sentences, keywords, and annotations.

The final deliverables are the actual project, a final slideshow presenting our work throughout the semester, this final report, and a video demonstrating exactly how to use our platform. Since the project is built using GitHub, it is free and available to the public to fork and modify in any way they see fit.
2.0 Introduction

2.1 Objectives

Our primary objective was to create a program that helps annotators provide topic/keywords and Questions-Answers associated with chapters in ETDs more efficiently. This objective is achieved using unsupervised methods such as RAKE for keyword suggestion and the MMR algorithm for sentence selection [3, 9]. The goal was to create a functioning website that allows users to upload documents, select/auto-populate diverse sentences, add annotations, and export their work through the website itself, all with control of the variables that affect the MMR algorithm at work. Using our service, annotators could more effectively and efficiently create rich data from ETDs, making what would be a labor-intensive process much less so. Another objective was to create a walkthrough or demonstration of our website in use to help new users understand how to upload documents, change algorithm variables, select or auto-populate summary sentences, and add individual annotations to each sentence.

2.2 Deliverables

The purpose of this project is to allow for ease of access to annotation of ETDs and generation of rich data. We aim to do this by accomplishing our objectives as stated above and presenting the deliverables listed below:

- A website that supports annotation of ETD chapters
- Searching and ranking methods implemented in Python
- Database for storing the uploaded documents, keywords, and the questions and answers for each user account
- Video demonstrating how to use the website

2.3 Client

The client for our project is Satvik Chekuri, a Ph.D. student and graduate research assistant at Virginia Tech. He has been guiding us throughout this whole project by providing us with resources, directions, and a code base that gave us the best foundation possible to complete our project by the end of the semester. This project will ultimately be delivered to Dr. Edward Fox, our CS 4624 professor.
2.4 Team Members and Roles

Our team consists of Jonah Bishop, Isaac David, and Ishaandeep Lubana. Jonah Bishop and Ishaandeep Lubana are seniors majoring in computer science. Isaac David is a junior majoring in Computer Science.
3.0 Requirements

After discussion with our client Satvik, we came to the conclusion that there is only one core requirement we need to complete this project, the website. Given his team had previously worked on this project, we were able to fork their GitHub repository instead of starting from scratch. This made it so our one requirement was to upgrade the website from an interactive MMR page to a fully functioning site for generating annotations.

3.1 Upgrade Existing Website/Code

Since we are building upon a previous project, we were able to save time and not build from the ground up [4]. This meant a lot of the initial setup and choosing of the tech stack was already done for us. Building upon the previous platform, our client Satvik wanted the website to be used as a tool in aiding annotation of the ETDs. Before we could really start on this, we had to fully understand the code base we were given from the previous team, which is what we spent the first several weeks doing. Once we each felt we had a solid grasp of it, we could finally begin working on the website.

The first coding step was to add two additional columns, one for the generated summary and another, a text area, for putting the actual annotations and keywords. It is important for the second area to contain any number of questions and answers, so we added a button to add a new text area for a question and answer into the column. Next, we had to add a button to auto-select sentences for the users and another button to move those selected sentences to the selected sentences column and find the supplementary sentences. Additionally, we needed a query window for the keywords in the selected sentences as this is also important data for the annotators. Lastly, we needed to add the ability to export the generated sentences, keywords, and annotations. This was arguably the most important feature. We chose to export the data as a JSON file, due to its versatility.
4.0 Design

Since the main purpose of our project was to further our client’s previous work, we tried to incorporate the idea of design preservation while making design decisions for the main site pages. For instance, the only change to the previous team’s project page was the addition of a button that led users to a login/signup page because account functionality is a core aspect of our project. The login page itself is designed to be minimalistic, so the MMR page is designed with plain features. The design changes to the annotation page include the addition of further text boxes to facilitate the annotation functionality that we want to provide with a simple “next” button designed in the same way that the previous team designed the upload button. In this way, we were able to preserve the stylistic characteristics of the previous team’s work while making the required changes.
5.0 Implementation

Our implementation built upon the previous group’s work and can be divided into three main sections. Each corresponds to a source code file or group of files: the mmr.py file which contains the functions that perform the natural language processing, the server.py file which services requests from the frontend and provides the frontend with an interface to use the tools in mmr.py, and a set of JavaScript files that make up the frontend website, in particular, client.js.

5.1 Artificial Intelligence

In order to make this tool useful and a time saver for annotators, we need to incorporate “Artificial Intelligence” into the program. More specifically, we used RAKE from NLTK to perform a keyword analysis, and utilized the MMR algorithm to recommend diverse and similar sentences to rapidly provide a summarization of the ETD chapter [3, 9].

5.1.1 Keyword Generation with RAKE

As previously stated, we used RAKE to perform a keyword analysis on the basis of keyword frequencies. Rake additionally has the added functionality of removing stop words such as “the”, “and”, “there”, etc. RAKE takes in a list of stop words, a set of phrase delimiters, and a set of word delimiters. It then uses stop words and phrase delimiters to partition the document text into candidate keywords. The keywords from RAKE make up the query part of the MMR algorithm, as described in Section 5.1.3.

5.1.2 Cosine Similarity with TF-IDF

The next crucial step in the MMR algorithm is calculating the similarity. We opted for cosine similarity as this is a type of similarity metric used for document-based similarity scoring. There are two different cosine similarities calculated. The first is the similarity between the selected sentence and the query (the keywords), while the second is the similarity between the selected sentence and the best sentences already selected. The weights for terms in the vector representations are based on TF-IDF (term frequency * inverse document frequency).

5.1.3 Sentence Selection with Maximal Marginal Relevance

The core of this whole project is centered around using a Maximal Marginal Relevance algorithm to determine diverse and similar sentences. MMR can help the annotators locate those key sentences based on the most relevant words. One idea behind this is that words that appear several times are more significant than others in a sentence. When the user specifies a query including key terms, MMR can rank sentences based on their estimated relevance to the searched
query. After the user selects some sentences, with an adjustable \( \lambda \) factor, they can further choose between accuracy (in terms of estimated relevance to the search query) and diversity (as compared with the already selected sentences); see Figure 1.

\[
\text{MMR} = \arg \max_{D_i \in R \setminus S} \left[ \lambda \text{Sim}_1(D_i, Q) - (1 - \lambda) \max_{D_j \in S} \text{Sim}_2(D_i, D_j) \right]
\]

Figure 1: MMR Algorithm Formula

Assume that we are given a database of 5 documents \( d_i \) and a query \( q \), and we calculated, given a symmetrical similarity measure, the similarity values shown in Figure 2. Further assume that \( \lambda \) is given by the user to be 0.5.

Figure 2: Chart showing the similarity between each document and the query

Initially our result set \( S \) is empty. Therefore, the second half of the equation, which is the max pairwise similarity within \( S \), will be zero. For the first iteration, the MMR equation reduces to:

\[
\text{MMR} = \arg \max \text{Sim} (d_i, q)
\]

\( d_i \) has the maximum similarity with \( q \); therefore, we pick it and add it to \( S \). Now, \( S = \{d_i\} \).

Since \( S = \{d_i\} \), finding the maximum distance to an element in \( S \) to a given \( d_i \) is simply \( \text{sim}(d_i, d_i) \).
For \(d_2\):

\[\text{sim}(d_1, d_2) = 0.11\]

\[\text{sim}(d_2, q) = 0.90\]

Then \(\text{MMR} = 0.90 - (1-\lambda)0.11 = 0.4225\)

Similarly, MMR values for \(d_3, d_4, d_5\) are 0.135, -0.35 and 0.19, respectively. Since \(d_2\) has the maximum MMR, we add it to \(S\). Now \(S = \{d_1, d_2\}\).

This time \(S = \{d_1, d_2\}\). We should find the max of \(\text{sim}(d_i, d_1)\) and \(\text{sim}(d_i, d_2)\) for the second part of the equation.

For \(d_3\):

\[\max\{\text{sim}(d_1, d_3), \text{sim}(d_2, d_3)\} = \max\{0.23, 0.29\} = 0.29\]

\[\text{sim}(d_3, q) = 0.50\]

Then \(\text{MMR} = 0.5*0.5 - 0.5*0.29 = -0.0725\)

Similarly, MMR values for \(d_4\) and \(d_5\) are -0.35 and 0.06, respectively. Since \(d_2\) has the maximum MMR, we add it to \(S\). Now \(S = \{d_1, d_2\}\).

\(d_3\) has the maximum MMR, therefore \(S = \{d_1, d_2, d_3\}\).

If we didn't have diversity at all \((\lambda = 1)\), then our \(S\) would have been \(\{d_1, d_2, d_5\}\).

### 5.2 Server Backend

The server interface is entirely contained in server.py. It is implemented using version 2.1 of Flask for Python [5]. In its current state, for each session, it keeps track of the uploaded document, the sentences in the document and an associated ID, and a list of keywords generated by mmr.py. When the document is first uploaded, this session data is generated and shared with the frontend. These sentence IDs are used instead of sending full sentences in future HTTP requests and responses, to reduce packet size [6].

Whenever the frontend needs to use the tools in mmr.py, server.py provides the interface to handle those requests. This includes a function that re-ranks the suggested sentences based on the current top sentences, and a backend function that takes the provided top sentences and returns the group of similar sentences and suggested keywords. Both of these functions are essentially adapters between client.js and mmr.py. Server.py can also generate a file for export upon a POST request, returning a handle which can later be used in a GET request to download the file.
5.3 Website Frontend

The home page of the website is an explanation of how the MMR algorithm works and a demo video. Users then press the get started button to move to the core functionality webpage. They will be immediately prompted to upload the chapter by copying and pasting the text into the provided window. After uploading, top sentences are automatically generated and can be further selected according to user preference by making changes to the query selector next to the upload button. Users can then drag and drop or auto-populate sentences for their summarization. Then, by clicking the next button at the bottom-right, users are automatically scrolled down to the second half of the web page, where their summary can be annotated directly with the textbox provided on the right.

The website’s frontend interface is divided among 2 HTML files that work in conjunction with JavaScript files to connect to the Python backend. Although our main changes and contributions are in index.html, minor changes were made to intro.html. The index.html file contains the main page that annotators would use to upload documents, select or generate sentences after optimizing variables, add their own annotations, and then export data directly from the page. In other words, all the core functionality of the frontend is contained within index.html.

The frontend is implemented using HTML5 with a stylesheet designed by the last team built in CSS 4.15. In the frontend’s current state, the structure is divided into the 5-box design that is visible on the website. The text model for the upload functionality of the page is also contained entirely in the file. A lot of the page’s aesthetic choices were made by Satvik and the previous team; we decided not to make any major changes to the page other than to remove and optimize features to better suit our project. Since the web-app is hypertext transfer protocol secure, the scripts being run also need to be the same and this is done in the footer of index.html [6].
6.0 Testing/Evaluation/Assessment

In the early stages of development, we continued testing using the short news articles that the previous group’s project used. As a result, all testing in the first half of the semester focused on adding features and ensuring they function as expected with unit and integration tests. We used ETDs provided by our client that have already been divided into chapters within the content.

Our final product is intended to be a tool that makes it easy to generate rich data about a document. For the final stage of testing, we asked students in our class, as well as CS majors ranging from freshmen to seniors, to use our annotation tool and give us feedback on the user experience and the user interface. Specifically, we wanted them to rate the effectiveness of the walkthrough/demonstration video, the main home’s explanation of the algorithm behind our work, and the actual annotation tool’s layout in maximizing productivity and ease of use.

6.1 Website Walkthrough

To help with user testing, we developed a three-minute video that showed us going through the complete workflow on the website. The walkthrough was recorded on Zoom by Jonah Bishop and Ishaan Lubana. It begins with the user uploading a test document about COVID to the site. The user then changes the lambda value using the slider and next proceeds to add a keyword, as well as remove an auto-generated keyword. After that, the user clicks the auto-populate button that generates the top summary sentences. The user then proceeds to manually add a sentence and remove a different one, thereby highlighting the drag-and-drop functionality that has been introduced. The user then clicks the next button, which takes them to the second half of the page where they proceed to add 2 questions addressing the information in the summary sentences. Finally, the user clicks the export button and the data is downloaded locally.

6.2 Google Survey Questionnaire

For the survey, we wanted to collect both qualitative and quantitative data. Furthermore, we wanted the form to be concise enough to be completed quickly yet collect useful information regarding the three concepts that we want to get feedback on: the walkthrough video, the user interface, and the layout design. With these thoughts in mind, we designed our questions to get user ratings on a scale of one to ten, as well as allow them to give longer responses, should they choose to do so. The questions we asked in our survey were:

- Rate the Effective of the Walkthrough Video (1-10 scale)
- Any questions/comments/concerns on the demo (Free Response)
- Rate the overall experience with the website (1-10 scale)
- Rate the Ease of Use of the service (1-10)
6.3 Survey Results and Implications

Although our form was only live for a week, we gathered 20 responses from our classmates and peers. In terms of quantitative results, for the effectiveness of the walkthrough video, we received a mean score of 7.75; for the overall user experience rating, we received a mean rating of 7.30; for the ease-of-use rating, we received a mean score of 8.50. These ratings indicate that we were able to create a tool that is easy to use, has an intuitive user interface, and is easy to understand. For the qualitative responses, we received a wide range of responses, both positive and constructive. Users seemed to like the auto-populate functionality provided to select summary sentences; more than 25% of the respondents mentioned the auto-populate functionality. For the questions asking about frustrations, the main points of criticism were the walkthrough video. The video, in its form for most of the testing, didn’t have audio. In addition, according to responses, the video moved a little too fast for users to fully understand each feature directly from the video rather than through their own experience with the site.
7.0 User Manual

7.1 Website Walkthrough

In the following subsections, we walk annotators through exactly how they can use our tool to shave time off the process of annotating ETDs. This section will also serve as a walkthrough of the website as a whole.

7.1.1 Top of the Homepage

The first thing annotators see when they visit the website is the Homepage; see Figure 3. This page serves as an introduction to the MMR algorithm and how it works. Annotators can either jump right in and click the get started button or watch the video demonstrating how to annotate documents. If users choose to scroll down first, please continue reading the subsections in order otherwise skip ahead to Subsection 7.1.5.

Figure 3: The top of the Homepage
7.1.2 Middle of the Homepage

Assuming annotators did not want to start annotating immediately, they would have scrolled down and found themselves looking at the middle of the Homepage; see Figure 4. The first paragraph describes the current lengthy process of annotating data. The second paragraph contains a high-level overview of how our process works in speeding up the annotation process.

Figure 4: The middle of the Homepage

7.1.3 Diagram of MMR Workflow

Scrolling down once more, users find themselves looking at a diagram detailing the inner workings of the MMR algorithm; see Figure 5.
Figure 5: Diagram explaining how the MMR algorithm works

7.1.4 Deep Explanation of MMR

Finally, users will have scrolled to the bottom when they can see another get started button; see Figure 6. In this section, annotators can really get a low-level feel for how the MMR algorithm works. The first image in Figure 6 is the actual formula for the algorithm. In figure 6, there is a sample matrix containing the calculated MMR scores given a document. Then there is text explaining the actual steps for computing the scores.
7.1.5 Uploading a Document

After successfully clicking on either of the get started buttons, annotators are brought to the upload page; see Figure 7. The first thing they will see is a small pop-up window asking them to upload text from a document. The annotators then copy and paste the text of one chapter, and then press upload and the pop-up closes.
7.1.6 Document Successfully Uploaded

After the document is uploaded, starting from the top, we can see the upload button; see Figure 8. If annotators made a mistake uploading the document, they can always redo the upload.

To the right of the upload button is the box containing the keywords used by the MMR algorithm. Annotators can press the little plus to add another keyword from either the provided list or enter their own. Each keyword box also has a little x to click to remove the keyword which will cause a reranking of the top sentences column.

To the right of the query box are two sliders. The slider on top labeled “Number of Auto-populate Sentences” is for choosing the number of sentences picked by the auto-populate button. It is described in detail in Subsection 7.1.7. The slider below that, labeled “Number of Similar Selected Sentences,” is for choosing the number of similar sentences, and is described in Subsection 7.1.8.

Next, we have three columns, each with its own distinctive purpose. The first column is simply the text of the document uploaded. Directly to the right, the middle column displays the most diverse sentences as determined by the algorithm. Additionally, take note that the highlighted words in each sentence are the selected keywords from the query box. Finally, the far-right
column is where annotators drag and drop sentences, or auto-populate, as described in Subsection 7.1.7.

Figure 8: The three columns to select the diverse sentences
7.1.7 Selecting the Sentences

The next step is selecting diverse sentences, typically five or more in number. There is a manual option and an automatic option for this process. If annotators want more control over the process, they can manually select multiple sentences by dragging and dropping into the Summary column as seen in Figure 9. Otherwise, they can simply click the auto-populate button on the bottom right and the top diverse sentences will be selected based on the top slider’s value. Furthermore, they can auto-populate then switch out any sentences they do not like, for ones they do. To get to the next step, annotators click the “next” button on the bottom left.

Figure 9: Main three columns with selected sentences filled
7.1.8 Annotating the Data

As we are approaching the last step of exporting the data, we are finally at the stage where the annotators are annotating the data. After clicking the next button, the page will scroll down to these two columns; see Figure 10. Then, based on the selected diverse sentences from the previous step, additional similar sentences for each diverse sentence will be selected based on the value of the second slider in Subsection 7.1.6. The first sentence in each box is highlighted to indicate that it is a diverse sentence. The remaining sentences in the box are the supplemental sentences most similar to the highlighted one.

In the top right corner, there is another query box labeled “keywords.” It functions almost identically to the one mentioned in Section 7.1.6 with the exception of not re-ranking, only changing the highlighting of the keywords in the sentences.

Annotators can begin filling the right column with their annotations, specifically the answers to questions found in the generated summary. By clicking the New Question button seen in Figure 10, two additional text areas will be added to the column, with the top box for a question and the bottom one for the answer. In the next section, we describe the final step in this process.

Figure 10: The generated summary and questions and answers column
7.1.9 Exporting the Data

The final step is exporting the annotations that the annotator created; see Figure 11. All the annotator has to do is press the export button and the data will be exported to them as a JSON file available for download. The format of the JSON is discussed in Section 7.1.10.

![Figure 11: Exporting Data](image)

7.1.10 Format of the JSON Data

In Figure 12, we can see the format of the exported JSON data. First is the summary which contains all of the sentences in the generated summary. Next is the list of keywords selected by the user. Finally, there is the list of the questions and answers where the questions are the keys and the answers are the values.

![Figure 12: Format of the JSON data](image)
8.0 Developer's Manual

8.1 Inventory of All the Files

8.1.1 Python Files

The core functionality of the website is achieved by Python files, as the server was built on Flask, a Python micro-framework. To better navigate this Section, please refer to Table 1.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Section</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmr.py</td>
<td>8.1.1.1</td>
<td>Natural language processing and MMR algorithm functions</td>
</tr>
<tr>
<td>sentence.py</td>
<td>8.1.1.2</td>
<td>A hashable class encapsulating logic for generating word frequencies</td>
</tr>
<tr>
<td>server.py</td>
<td>8.1.1.3</td>
<td>The main class and the interface for the frontend</td>
</tr>
<tr>
<td>mongo.py</td>
<td>8.1.1.4</td>
<td>Interface between server.py and MongoDB instance</td>
</tr>
<tr>
<td>tests.py</td>
<td>8.1.1.5</td>
<td>Unit testing functions for mmr server and tests.py that can run independently of the frontend</td>
</tr>
</tbody>
</table>

Table 1: Each of the Python files with their Section and a brief description

8.1.1.1 mmr.py

Most of this class was inherited from the previous group. All of the natural language processing, keyword generation, sentence comparison, and the MMR algorithm is performed by functions in this file. Our changes to it include simplifying and condensing some functions, improving documentation and variable naming schema, and adding the n_sim_sentences function. This function is new to our project. It takes three lists of sentence.sentence objects, the list of chosen top sentences, the list of candidate similar sentences, and a list containing every sentence in the document. It compares every top sentence with every candidate sentence and then one-by-one it finds the pair of top-candidate sentences which have the strongest similarity score and adds the candidate to the top sentence’s set. This is done so that no candidate sentence is chosen by more than one top sentence. It returns a mapping from the strings of the top sentences to a list of its most similar sentences.
8.1.1.2 sentence.py

This class was inherited from the previous group. Most of its functions and fields aren’t used. It takes a str sentence as a parameter in its constructor and creates a list of word frequencies for the particular sentence. This frequency list is used by some of the MMR functions. Our changes to this file include reformattting the whitespace for formatting consistency and overriding the __eq__ and __hash__ methods so that sentence objects can be compared to strings for searching purposes in server._testable_phase_two().

8.1.1.3 server.py

This class was also inherited from the previous group. It starts the server and handles the HTTP requests. We added a set of functions to handle user accounts, registration, and signing in. Most of its other functions either return a webpage or are wrappers for functions in mmr.py. When the frontend needs to run the MMR algorithm or generate similar sentences, it makes an HTTP request to server.py. You can see Flask decorators in Figure 13. These set up the Flask package to pass on HTTP requests that match the given URL rule to the correct function.

```python
@app.route('/')
def homepage():
    """Displays the homepage."""
    return render_template('intro.html')
```

Figure 13: A function that returns a webpage

In Figure 14, you can see the server.rank function which is a wrapper for the mmr.summary_generator function. In Figure 14, there is an example of some special wrappers that we created. We suffered from an issue where our main backend developer couldn’t run the server from his machine; more on this is in Section 9. In order to facilitate testing and developing the backend in anticipation of the frontend, some functions like server.phase_two, which was originally a wrapper for mmr.n_sim_sentences, were split into two wrappers, one which parses the HTTP request and formats the response and another that runs on the parsed input and returns the unformatted data for the response. These functions exist for testing server.py functions without a running Flask app. Once those were working, the HTTP request-facing functions were also tested without a running server. All of server.py’s accesses to request.json, a Flask variable, were rewritten to be indirects through a lambda function json_request (Figure 15). In normal operation, this doesn’t affect the app, but it allows tests.py to redirect JSON_request to its own function for the use of testing inputs.
```python
@app.post('/rank')
def rank():
    """Rank sentences from an existing session...."""
    res = {}
    summary = []
    sentences = []
    sessionID = int(json_request('session_id'))
    keywords = json_request('keywords')
    summaryIDs = json_request('summary')
    curr_session = SESSIONS[sessionID]
    for i, j in curr_session['sentences']:
        if j in summaryIDs:
            summary.append(i)
        else:
            sentences.append(i)
    print('summary', summary, summaryIDs)
    mmr_scores = mmr.summary_generator(sessionID, sentences, keywords, summary)
    for index, row in mmr_scores.iterrows():
        for se, s_id in curr_session['sentences']:
            if out['sent'] == se:
                mmr_scores.at[index, 'sentID'] = int(s_id)
    res = mmr_scores.to_json(orient='index')
    print('res', res)
if not all(arg in json_request() for arg in ('session_id', 'Keywords', 'summary')):
    print("Malformed request to '/rank' endpoint:", json_request())
    return jsonify({})
return res
```

Figure 14: A wrapper for mmr.summary_generator

```python
@app.route('/phase_two', methods=['POST'])
def phase_two():
    """This is a wrapper for _testable_phase_two. _testable_phase_two has parameters and return values that are easier to test server-side.
    * Expects requests.json['session_id'] to be session_id as usual.
    * Expects requests.json[top_sentences] to be a list of IDs for the top sentences.
    (IDs are the same as in the rank method)
    """
    return jsonify({'similar_sentences': _testable_phase_two(int(json_request('session_id')), json_request('top_sentences'))})
```

Figure 14: A wrapper for server._testable_phase_two
Figure 15: The json_request lambda function

Lastly, there are the functions generate_json and export. Server.generate_json also uses the same design pattern where the function is split into an HTTP adapter and a function that accepts the parsed parameters. Upon a POST request, server.generate_json generates a JSON file containing the important data the user has created and places it in a file with a randomly-generated filename in an exports directory. In its response it provides the frontend with a handle that can be used in a subsequent GET request to download the file with the server.export function.

8.1.1.4 mongo.py

This is an entirely new file we added. In this file we connect to a running MongoDB instance and provide storage functionality for server.py. Although we weren’t able to implement support for uploading and simultaneously annotating multiple chapters in the frontend or server.py, the main hold up was designing the underlying data structures to support this. This file provides an interface for server.py to store session information in a MongoDB collection, and the underlying data structure is designed to easily accommodate multiple chapters. It also has the added benefit of supplementing any future work with user accounts since MongoDB databases, collections, and documents persist on the hard drive during downtime, while also providing efficient scaling and querying.

8.1.1.5 tests.py

This helps with unit testing for the rank function, similar sentence generation, and file export. The functions in this file don’t test for success or failure, but instead print debug output from which major bugs should be apparent. Initially, it was used to test the similar sentence generation function we added to mmr.py. As the backend for this functionality was added to the server and the main server developer had some difficulty connecting to the server, tests.py became responsible for testing the new functions in server.py. At first, he would test the lower-level wrappers that don’t interface with HTTP requests at all. Then, to emulate a running app to test the HTTP interface, tests.py would use a test document string to generate session data and place this data in server.SESSIONS, an action which server.upload would normally perform. It additionally redirects json_request to its own dummy function with test input. From there, the functions in server.py were able to be tested as if there were a running app.
8.1.2 HTML and CSS Files

The HTML and CSS files are responsible for how the webpages of the website appear. There are several HTML files that make up our website but only one CSS stylesheet is used. All of these files can be found in the views folder with the exception of style.css which can be found in the public folder. The stylesheet is self-explanatory as it simply contains the style for most of the HTML elements used. The HTML files are base.html, index.html, intro.html, login.html, and register.html.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Section</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro.html</td>
<td>8.2.1.1</td>
<td>Homepage of the site describing the MMR algorithms</td>
</tr>
<tr>
<td>index.html</td>
<td>8.2.1.2</td>
<td>Main page where users upload and annotate chapters</td>
</tr>
<tr>
<td>style.css</td>
<td>8.2.1.3</td>
<td>Stylesheet for the HTML elements</td>
</tr>
</tbody>
</table>

Table 2: The HTML files, with their section number, and a brief description

8.2.1.1 intro.html

Intro.html represents the homepage of the website. Here, users can read all about the MMR algorithm and its operation. A lot of this information is from the previous group as there was no need to really change anything.

8.2.1.2 index.html

Index.html is the HTML file representing the core functionality of the website. Here, annotators are responsible for uploading a chapter of an ETD. Then the various JavaScript and Python files transform the uploaded document. Similarly to intro.html, this file was also inherited from the previous group, although modified heavily. In addition to cleaning up the code, we added the bottom two columns and the next, auto-populate, and export buttons. The added functionality turned this from an interactive MMR webpage into an effective tool for annotating ETDs by chapter.
8.2.1.3 style.css

Last up for the static display files is the stylesheet, style.css. It contains the styling for many of the HTML elements we used.

8.1.3 JavaScript Files

The JavaScript files are responsible for turning all of the HTML code from static to dynamic by allowing client-side script execution. This is accomplished by running functions on button clicks and facilitating the communication of the frontend with the backend. While the core functionality is all in Python, the JavaScript connecting to the Python functions is essential. Each of the JavaScript files can be found in the folder named public.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Section</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>api.js</td>
<td>8.1.3.1</td>
<td>Connects frontend to backend via GET and POST requests</td>
</tr>
<tr>
<td>client.js</td>
<td>8.1.3.2</td>
<td>Client side functions for dynamic HTML elements (sliders, ex.)</td>
</tr>
<tr>
<td>demo_article.js</td>
<td>8.1.3.3</td>
<td>Text of the three demo articles</td>
</tr>
<tr>
<td>view.js</td>
<td>8.1.3.4</td>
<td>Functions for changes to the view of the web page via HTML</td>
</tr>
</tbody>
</table>

Table 3: The JavaScript files, with their section number, and a brief description

8.1.3.1 api.js

The JavaScript file api.js is relatively simple but crucial to the website. It is the key component connecting the frontend to the backend. It has three functions, each of which connects a specific function on the backend to the frontend by sending a POST request payload. Each of these functions is called in client.js and anything in the response of the HTTPS request is handled by another function in client.js. Two of the three functions were inherited from the previous team and were completely unchanged by our team.

The first function, uploadDocument, sends a POST request to the backend containing the document uploaded by the annotator. Upon a successful upload, a function in client.js called uploadSuccessFn is called, but if it fails to upload, then another function in client.js called uploadFailedFn is called. The second function, rank, sends a POST request which contains a payload of the session ID, keywords, and summary sentences. Upon success, a function in client.js calls handleRerank results. Lastly, there is a function called phase_two. This is responsible for sending a payload of the summary sentences and getting back the supplementary sentences. Upon success, a function called handleSimResults is called in client.js.
8.1.3.2 client.js

The JavaScript file client.js is responsible for all the functions handling direct interactions with the website. This includes any time a button is clicked or a file is uploaded. Here, each of the buttons is linked to an event handler which will be triggered every time a button is clicked. Each of the sentence cards in the top middle and top right column is also linked to an event handler to trigger a function when the sentence cards are dragged and dropped into place.

Some of the functions in this JavaScript file were from the previous group and were unchanged. Additionally, this file used to contain a lot of commented out code that was removed. Also removed were functions used to demonstrate the interactivity with the MMR algorithm as it was no longer needed for our project.

Apart from the numerous global variables, there is essentially one function for each button and then some miscellaneous functions. Each of the functions corresponding to a button is relatively self-explanatory and often calls an additional function based on the result of that function. For example, the auto-populate button calls a function called autoPopulateHandler which goes through the top sentences column five times selecting the top sentence and re-ranking after each selection and calling a function in view.js to update the view.

Some of the miscellaneous functions include the function to calculate the MMR score given two similarities and a predefined lambda value. Another set of miscellaneous functions are the two functions to handle the dragging and dropping of sentences. One of those functions is for handling dragging from top sentences to the summary and the other is for the summary to top sentences.

8.1.3.3 demo_article.js

This is the simplest of files and should not really be touched. It simply contains the actual text of the three demo documents: a manually parsed ETD chapter and two noisy ETD chapters.

8.1.3.4 view.js

As with each of the JavaScript files having specific purposes, the view.js file also has a specific purpose of updating the view of the annotators. Each time the view needs to be updated following a function execution in client.js, a separate function in view.js is called. By separating this functionality away from the main client-side file, it makes it easier to differentiate between the various functions. As with the previous JavaScript files, most of this was inherited from the previous team.
Each of these functions in view.js corresponds to a function calling it in client.js. Sometimes, multiple view.js functions are associated with just one client.js function. For example, the creation of the sentence cards in the top middle column has several view.js functions called. First, renderCandidates is called to generate the individual sentence cards. However, each sentence card has some additional visualization aspects such as the highlighting of any keywords. So, an additional function called wrapKeywordsInSentence is called to add that aspect. Using a combination of div elements and some other key ones, view.js works to dynamically adjust the view based on what the annotator clicks on or drags and drops.

8.1.4 Data Files

We were graciously donated lots of ETD data from a colleague of Satvik, Bipasha. The original data she shared with us is the whole ETDs in both .pdf and .txt documents. The newer data is ETDs broken down by chapter which is much more useful to us. None of this data is available with the GitHub repository. However, generating data for use with this tool is simple as all you have to do is find ETDs online to use.

8.2 How to Set Up for Yourself

For anyone wanting to expand upon the project in the future, setting it up is relatively easy. The project is available on GitHub, whether you want to fork the repo or clone for yourself [2].

In terms of the software requirements, we are developing with Python version 3.6.5 and MongoDB version 5.0.6. Both of these are easy to install following guides online; guides have been added for both in the References section [1],[7]. Additionally, we recommend using a virtual environment for development to manage dependencies more easily.

Once all of the necessary dependencies and software routines have been installed, you can clone the GitHub repo onto whatever machine you choose for development. Next, make sure you are in the main directory and run the command: pip install -r requirements. This will install all of the libraries needed. Now you will need to install some specific items from the NLTK library [9]. Run the following commands in this order:

```
python
import nltk
nltk.download('punkt')
nltk.download('stopwords')
```
After exiting the Python interactive interpreter, you are ready to run the server. The final command you have to run is: python server.py. To verify that it is working, navigate to http://127.0.0.1:5000/.

9.0 Lessons Learned

9.1 Timeline/schedule

- **Feb 14**
  - Set up a Github Repository
  - Acquired a website domain
  - Started meeting weekly at regular time
- **Feb 28**
  - Languages, libraries, and frameworks chosen.
- **Mar 21**
  - Important keyword ranking implemented
  - Database of sample texts
  - Removed a lot of previous teams commented out code and useful functions
- **Apr 4**
  - MMR is implemented to select diverse important sentences and similar sentence groups
  - Added auto populate button
- **Apr 11**
  - Added generation of similar sentences on next button click
  - Cleaned up some of the old team’s code
- **Apr 18**
  - Fully working prototype complete
  - Added ability to export data as a JSON file
  - Added diverse sentence and keyword highlighting in bottom row left column
- **Apr 25**
  - Sliders for selecting the number of auto-populate and similar sentences
  - Query box for keywords and dynamic updating of keyword highlighting
- **May 2**
  - Recorded video demo on how everything works
  - Cleaned up documentation and formatting
  - Tutorial for annotators
  - Tested all features
9.2 Problems

We had issues with a slow start. We dedicated much of our first two months to trying to understand the code left to us, parse ETD PDFs, and add user accounts. Parsing ETDs is a hard task, and we were ahead of ourselves to start by trying to parse the ETDs. Our logic was that it was the first step in the dataflow for the project, but what we should have done was start working on the core of the project using single test cases. We also underestimated how hard it would be to parse ETD PDFs and under-budgeted the time for it. User accounts were comparatively not too hard to implement. It took about two weeks to implement them, but nonetheless, it was too early to be working on these things that were add-ons for our project and not the core scope. We should have been meeting more often and been more deliberate in our short-term planning and connecting that to our long-term goals.

Another thing that slowed us down was working with the code provided to us. In the code we inherited, there were many commented out code fragments and in some files much of the documentation referred to previous versions of their code instead of the current version. There were inconsistent formatting practices; documentation was spotty. Occasionally it was thorough, but often it was sparse. In one case, a parameter was called input_file_path which was later passed to another function as file_name, however, the parameter was actually expected to be the full text of a document. The frontend, meanwhile, was nearly devoid of any documentation. This made building off their code difficult, as we couldn’t trust the documentation or adherence to conventions. On our part, we took too long to make a thorough pass to eliminate these issues and we continued working with them for too long.

As mentioned in Sections 8.1.1.3 and 8.1.1.5, our main developer for server.py had longstanding difficulties connecting to the remote server’s running web app. While we immediately noticed that it was unusual that the team members who could successfully connect were able to use localhost:5000 to connect to a remote server, we unfortunately didn’t fully investigate that aspect. Instead, the team member who couldn’t connect spent a lot of time troubleshooting their device.

Even though it also failed on campus, where the connection shouldn’t have been blocked, he tried to set up the Pulse Secure VPN. This ran into unusual issues that even Virginia Tech IT couldn’t resolve. So, he tried borrowing a laptop from the library lending desk to see once and for all if it was really a problem with his personal laptop. Connecting from the borrowed laptop also failed, but the VPN worked, so he spent some time attempting to set up an SSH tunnel. Finally, we noticed when another team member started the server that a notification displayed in the corner of their device; see Figure 16. This is a notification from Visual Studio Code that it had automatically set up an ssh tunnel from localhost:5000 to the remote server. This was the source of the confusion. One team member was starting the server through PuTTY, while the
others were starting it through VSCode and none of us knew that VSCode was handling that for us. The lesson here is that our first instinct was the right one and if we stuck with it, we would’ve saved a lot of development time.

![Visual Studio Code automatically creating an SSH tunnel](image)

Figure 16: Visual Studio Code automatically creating an SSH tunnel

### 9.3 Solutions

After spring break, we knew that we had to redouble our efforts to meet our deadlines. We met with Satvik and committed to a working website with significant progress in 10 days. We worked through the week to assemble the necessary parts for that. We met that deadline and we were able to start testing. Our solution to the problem with the SSH tunnel was simply to uniformly use Visual Studio Code to run the server.

### 9.4 Future work

In terms of progress beyond our team and this semester, we believe we have helped our client grow out the project he had been working on far before our contribution. We believe the future of this service depends on the direction that the team wants to build towards. For instance, an area of interest can be the keyword generation algorithm that is currently being used (as well as any optimizations with the MMR algorithm). Another potential area of improvement is the UI and the variable customization offered to users. Ideally, users should have complete control over the sentences and the logic behind keyword selection that is used in the summarization process. We were interested in helping annotators not only better annotate ETDs but also compile a list of questions that the information gathered through the summaries could be used to answer, thereby helping annotators extract a more thorough understanding of the document itself.

### 9.4.1 User Accounts

Currently, there is no way to save progress on a chapter being annotated by a user. So, if the website were to crash or a user wanted to come back later, they would lose all their progress. By adding user accounts, we could enable the saving of documents to each individual user account.
This would enable the user to chip away at a document little by little instead of having to do it all at once.

To make this work optimally, we could add a dashboard displaying all the documents the user is working on or has worked on. Additionally, if a user lost some of their annotations, they would be able to go to this dashboard to download the lost annotations.

9.4.2 Annotating a Whole Document

Another potential area of improvement would be giving annotators the ability to work on one whole document at a time as opposed to just one chapter. To start with this part, we would have to revamp how we were uploading documents to the website. Currently, there is an upload button which opens a small window to paste the text in. Now, we would have a separate page for the upload to be complete before users go on to the actual annotation step. Additionally, we want annotators to seamlessly be able to switch between different chapters they are annotating.

First, on this upload page, annotators still use the upload button but they also can keep uploading text with the goal of uploading each chapter individually. When all of the chapters have been uploaded, the annotators then press a button to take them to a slightly different webpage. The key differences between the current page and the new one is that the upload button is removed and there are numbers in a row on a navbar at the top and bottom. The navbar is used to seamlessly navigate between the different chapters. Finally, there will be an export button at the bottom which will be used to export all of the individual chapters at once, thus generating all the data for the ETD.
10.0 Acknowledgements

We would like to thank Satvik Chekuri for proposing this project and working with us to develop it and make it the best it can be. We would also like to thank Bipasha Banerjee for providing us with the formatted ETD test data. Additionally, we would like to thank our GTA Ryan Wood for helping us when needed. Lastly, we’d like to thank Dr. Fox for his advice, feedback, and guidance throughout this project. Their contact information can be found below:

- **Client: Satvik Chekuri**
  
  satvikchekuri@vt.edu

- **Instructor: Dr. Edward A. Fox**
  
  fox@vt.edu

- **GTA: Ryan Wood**
  
  ryanw23@vt.edu

- **Bipasha Banerjee**
  
  bipashabanerjee@vt.edu
11.0 References


Accessed 19 April 2022.