CS4624: Multimedia, Hypertext, and Information Access

Topic Modeling Toolkit

Professor: Edward A. Fox

Client: Aman Ahuja

Jiayue Lin, Mingkai Pang, Kevin Liu

May 9, 2023

Virginia Tech, Blacksburg, VA 24061
The Topic Modeling Toolkit project began with an existing text mining toolkit and aimed to enhance its functionality by incorporating cutting-edge topic modeling techniques. Specifically, BERTopic [9], CTM [2], and LDA [16] were used to extract pertinent topics from a corpus of text documents. The resulting web-based platform provides users with a search engine, a recommendation system, and a usable interface for browsing and exploring these topics.

In addition to these enhancements, our team also implemented a text-filtering framework and redesigned the user interface using Tailwind CSS. The final deliverables of the project include a fully functional website, user documentation, and an open-source toolkit that can be used to train machine learning models and support browsing and searching for various text datasets.

While the current version of the toolkit includes BERTopic, CTM, and LDA, there is potential for future work to incorporate additional topic modeling methods. It is important to note that while the project originally focused on electronic theses and dissertations (ETDs), the resulting platform can be used to explore and comprehend complex subjects within any corpus of text documents.

The topic modeling toolkit is available as an open-source package that users can install and use on their own computers. It is available for use and can be used to support browsing and searching for various text datasets. The intended user group for the platform includes researchers, students, and other users interested in exploring and understanding complex topics within a given corpus of text documents.

The resulting topic modeling toolkit offers features that facilitate the exploration and comprehension of intricate topics within text document collections. This tool has the potential to aid researchers, students, and other users in their respective fields.
Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Problem ................................................ 1

1.2 Motivation ........................................... 2

1.3 General Approach .................................... 3

2 Requirements 4

2.1 Database .............................................. 4

2.2 Topic Modeling ....................................... 5

2.3 User Interface ....................................... 6

3 Design 7

3.1 Overview ............................................. 7

3.2 System Architecture ................................. 8

3.3 Data Flow ............................................ 10

3.4 Frontend ............................................. 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5 Datasets</td>
<td>12</td>
</tr>
<tr>
<td>3.6 Database</td>
<td>12</td>
</tr>
<tr>
<td>3.7 Backend</td>
<td>12</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Backend</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Frontend</td>
<td>18</td>
</tr>
<tr>
<td>5 Testing</td>
<td>19</td>
</tr>
<tr>
<td>5.1 Database</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Backend</td>
<td>20</td>
</tr>
<tr>
<td>5.3 Frontend</td>
<td>20</td>
</tr>
<tr>
<td>6 Users’ Manual</td>
<td>22</td>
</tr>
<tr>
<td>6.1 Prerequisites</td>
<td>22</td>
</tr>
<tr>
<td>6.2 Configuration</td>
<td>23</td>
</tr>
<tr>
<td>6.3 Website</td>
<td>25</td>
</tr>
<tr>
<td>7 Developer’s Manual</td>
<td>33</td>
</tr>
<tr>
<td>7.1 Prerequisites</td>
<td>33</td>
</tr>
<tr>
<td>7.2 Configuration File</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Figures

3.1 An overview of the structure and components of the Topic Modeling Toolkit system ................................................................. 8

3.2 Level 1 & 2 data flow diagram .................................................. 10

4.1 Overall implementation .......................................................... 14

4.2 An example of the JSON data format that is returned by the API .... 15

4.3 Base Database Schema ........................................................... 16

4.4 Database Configuration in config.ini ........................................ 17

6.1 Global configuration file .......................................................... 24

6.2 Home Page ............................................................................. 25

6.3 Pagination Feature ................................................................. 26

6.4 Document Searching Page ....................................................... 27

6.5 An example of sorting .............................................................. 27

6.6 An example of filtering ............................................................ 28

6.7 Topic Browsing Page ............................................................... 29

6.8 Related documents for topics ................................................... 30

6.9 Machine Learning Topics ......................................................... 31

6.10 Document Description Page .................................................. 32
List of Tables

7.1 APIs’ Description ................................................. 39

8.1 Timeline of Topic Modeling Toolkit .......................... 53

A.1 Workflow ............................................................ 70
Chapter 1

Introduction

1.1 Problem

The Topic Modeling Toolkit serves as an essential instrument in enhancing the accessibility and discoverability of text documents. Currently, the labeling system for academic papers is too broad, impeding researchers from efficiently locating the relevant papers. By employing machine learning-based methods to incorporate more fitting tags for each paper, the toolkit enhances the precision of search and recommendation algorithms. Consequently, researchers can readily discover pertinent papers with ease.

Upon inheriting the TextMining project, now renamed to the Topic Modeling Toolkit, previously developed by a team from CS 4624 in the Spring of 2022, our team noticed a few critical issues that required attention. One of the main issues was the lack of extensibility, which required users to manually modify arguments within different backend scripts. As a result, the toolkit’s integration with other systems proved to be a challenge. Another issue that needed to be addressed was improving the user interface of the existing system such that it is easily navigable and seamlessly integrated with the APIs. This would improve the usability of the toolkit and make it easier for researchers to integrate it into their workflows. The shortcomings of the supervised-learning toolkits [18] available from outside sources served as the driving force behind the decision to incorporate topic modeling algorithms in the Topic Modeling Toolkit. These restrictions were specifically related to the toolkits’ ineffectiveness
in classifying and storing datasets. Because they allowed for the identification of latent topics within a corpus of text documents without relying on pre-defined labels or classifications, topic modeling algorithms were therefore thought to be more appropriate for the task at hand. To overcome this, an unsupervised learning algorithm was required. With the use of topic modeling, latent topics could be identified within a corpus of text documents without pre-defined labels or classifications.

1.2 Motivation

A team working on Text Mining in the CS 4624 course developed a framework to extract topics using machine learning-based techniques from a sizable collection of documents during the Spring 2022 semester. We intend to expand the methodology to other types of text-based documents, such as books, which have various smaller units such as chapters, by starting with the TextMining toolkit. To accomplish this, we intend to use more advanced machine learning-based techniques to extract topics from a sizable collection of documents, thereby enhancing the toolkit’s functionality to support a wider variety of document types.

Our team planned to continue the development of the original TextMining toolkit, which has now been renamed the Topic Modeling Toolkit. This open-source toolkit is Python3-based and facilitates efficient searching and browsing of text documents. To this end, we intend to revamp the toolkit’s interface, enabling users to seamlessly select from different trained machine learning models and filter results based on diverse criteria, such as date, university, and document type. The deliverable of our project will have benefits for various stakeholders, including researchers, students, educators, information professionals, librarians, and developers [19]. For example, librarians require a system that allows them to easily manage and organize large collections of academic documents, as well as search and filtering
features that allow them to efficiently retrieve and browse relevant documents. The project aims to deliver a system that expedites the process of identifying and accessing academic papers, thereby assisting researchers in discovering related papers they may have overlooked, and facilitating access to their full text. Additionally, developers will have access to the toolkit’s API and documentation, enabling them to integrate it with their own software and customize it to meet their specific requirements.

1.3 General Approach

Our approach consists of several key steps to improve the functionality and user experience of the Topic Modeling Toolkit [13]. Firstly, we will refine the toolkit by evaluating the quality of the topic modeling results, examining the generated topics for accuracy and meaningfulness, and implement any necessary adjustments. Furthermore, we will create a recommendation system that uses an index of topics and documents to facilitate the searching and browsing features. Browsing and recommendation are two key sections of topic modeling, and our recommendation system will cater to both. The system will be complemented by a search engine with query suggestion capabilities, an auto-complete feature, and an user interface with pagination and filtering capabilities.

We will also develop a filtering system that allows users to filter documents based on specific criteria such as start and end dates, universities, and other parameters. Finally, we aim to redesign the user interface of the Flask-based system [8] from the TextMining Toolkit [13], as implemented by the previous team, to enhance the user experience. This will involve evaluating and selecting appropriate frontend frameworks, integrating the chosen framework with Flask, implementing new UI components, and testing the finished product.
Chapter 2

Requirements

2.1 Database

The Topic Modeling Toolkit is a software application designed to streamline the process of topic modeling for users by providing an intuitive interface for generating topic models from text data. Its objective is to gather a corpus of documents and simplify the topic modeling process. To achieve this objective, the Topic Modeling Toolkit utilizes an SQLite3 database [14] as its primary means of data access. One or more datasets containing documents used by the application are stored in the database as a repository. The database handler (see Section 7.5) converts user-provided CSV datasets into a singular SQLite database, enabling the toolkit to access the data, thanks to the flexibility and self-contained nature of SQLite. This feature ensures that the application can handle datasets with different sizes and formats without requiring users to manually change their dataset’s format.

We used Apache Lucene (v9.0) [10] for effective indexing and searching of the datasets, and we built a Flask-based REST API [8] to communicate with Lucene using JavaGateway [5] since Lucene is implemented in Java. The search service queries the documents in the SQL database using the indexes that Lucene has returned.

Additionally, the SQLite databases are dynamically generated using user-defined configurations as well as user-uploaded datasets. By allowing users to specify their own data structures
and configurations, the toolkit is able to adapt to a wide range of use cases. As a result, the
database only includes the precise columns required to enable the application’s core func-
tions. This reduces the size of the database, allowing for quicker query processing and more
effective data storage.

2.2 Topic Modeling

Topic Modeling Toolkit’s main module is in charge of using topic modeling algorithms to ex-
tract topics from the document corpus. This module receives its input from the preprocessed
data. The topic modeling algorithms supported by our system at this time include:

- **BERTopic**: This topic modeling algorithm is based on the pre-trained deep learning
  model for natural language processing known as the BERT language model. The text
  documents are represented as a collection of topics by BERTopic using a clustering
  technique. By taking into account the documents’ semantic similarity, it groups the
  documents according to coherent topics.

- **CTM**: The CTM (Contextual Topic modeling) algorithm is an extension of the tradi-
tional topic modeling technique that takes into account the context of each word in a
document. CTM uses pre-trained neural language models, such as BERT or GPT, to
encode the contextual information of each word. This enables CTM to generate more
accurate and interpretable topics and group documents by topics by incorporating
contextual information.

- **LDA**: The LDA (Latent Dirichlet Allocation) algorithm is another statistical modeling
algorithm used for topic modeling, which aims to identify topics within a corpus of
documents. LDA assumes that each document in the corpus is composed of a mixture
of different topics and that each topic is represented by a set of words that tend to co-occur in documents. By analyzing the frequency of word co-occurrences across documents, LDA can estimate the probability of each topic being present in each document, as well as the probability of each word belonging to each topic. The LDA algorithm uses the probability generated to accurately group documents into different topics.

The Topic Modeling Toolkit employs the aforementioned topic modeling algorithms to extract the document corpus and subsequently categorize documents into relevant topics. Every model will produce a distinct set of topics, and users of the Topic Modeling Toolkit will have the option to select which model to utilize for training their own set of documents.

### 2.3 User Interface

The combination of using both React [1] and Tailwind CSS [20] enables us to create web pages that are visually appealing and responsive with minimal code.

React allows us to create reusable components that are simple to incorporate into the application, cutting down on development time and improving maintainability. The utility-first approach to CSS styling offered by Tailwind CSS streamlines the styling procedure by offering pre-defined classes that can be quickly applied to HTML elements. With the help of this feature, we can create web pages that are both visually appealing and responsive without writing a lot of complex custom CSS code.

In addition, React Router is what we’ve chosen to use to control page navigation. This library offers a straightforward and understandable method for handling routing in a React application, making it simple to create multi-page applications.
Chapter 3

Design

3.1 Overview

During the Spring 2022 semester in the CS 4624 course, a team focused on topic modeling created a framework to extract topics from a large collection of documents using machine learning-based methods [13]. The objective of our project is to create a powerful Python-based system that can search through sizable text document collections while making suggestions to users based on the document they have already chosen. With support for machine learning model training as well as browsing and searching through various other text datasets, the system is intended to be adaptable and extensible.
3.2 System Architecture

Figure 3.1 provides a visual representation of the various components that comprise the Topic Modeling Toolkit system. The system consists of several components, including a Python-based platform, a search service using Apache Lucene, a SQL database for storing text documents and metadata, and three topic models (BERTopic, CTM, and LDA). In order to handle user requests and communicate with the search service, topic models, and the database, the system also includes a backend server built with Flask. Additionally, a
frontend interface for browsing and searching text documents is implemented using React and Tailwind CSS.
3.3 Data Flow

Figure 3.2: Level 1 & 2 data flow diagram
3.4 Frontend

Figure 3.2 depicts the data flow of the system (DFD) during the initialization phase and when a user queries the documents, respectively.

The level 1 DFD shows the overall flow of data during the initialization phase when a user can modify the training and UI parameters in the configuration file. The user then uploads the dataset to the document system. The document datasets will then be processed by the database handler (see Section 7.5) and loaded into the SQLite DB. The appropriate topic modeling algorithm receives the dataset from the server, and produces a trained model. The training model is then saved on the local system in a pickle file [22].

The level 2 DFD displays the data flow of the system when a user queries documents. The process begins with the user entering a search query, which the frontend processes and sends to the backend server. In response to the search query, the server then retrieves the pertinent documents from the database and sends them back to the frontend for user display.

3.4 Frontend

We implemented the frontend using React (v18.2.0) and Tailwind CSS (v3.3.1). It provides a user interface for browsing and searching collections of text documents. We have adopted the frontend-backend separation approach [11] to achieve our goal. The home page, document searching page, topic browsing page, and document page are the four main pages of the frontend that are navigated using React Router.
3.5 Datasets

To achieve efficient indexing and searching of the datasets, we leveraged Apache Lucene (v9.0) and developed a REST API based on Flask [8]. To facilitate communication with Lucene [10], which is implemented in Java, we employed JavaGateway [5]. The search service queries the documents in the SQL database using the indexes that Lucene has returned.

3.6 Database

Our team has undertaken the complete development of the Topic Modeling Toolkit’s SQLite3 database, which serves as the primary data source for the toolkit. The data contained within the database are write-protected and can only be modified through our proprietary database handler (see Section 7.5), which has been specifically designed to ensure safe and secure interactions with the database. The database has been designed to be fully dynamic, enabling users to customize columns based on their preferences. As a result, the application can seamlessly integrate with any user-configured dataset, providing maximum flexibility and versatility.

3.7 Backend

Our team has implemented the backend server using Flask and a REST API, which handles user requests and interacts with the search service, trained machine learning models, and database. The text documents and related metadata are stored in the centralized SQLite database. All data-related tasks, such as scaling and loading user-uploaded datasets into the database, are managed by the database handler Python script (see Section 7.5).
The database handler script prioritizes modularity to simplify database management tasks like connecting to the database, inserting records into the database, removing records from the database, creating new tables in the database, and returning the database to its initial state.
Chapter 4

Implementation

4.1 Overview

The objective of this project is to develop a Python-based system that can efficiently search through a large collection of text documents, while also providing recommendations to users based on the selected document. The system is designed to be extensible, with the ability to support machine learning model training, as well as browsing and searching on various other text datasets. Our implementation follows the frontend-backend separation approach, with the following components shown in Figure 4.1.

1. A frontend implemented using React and Tailwind CSS that provides an interface for browsing and searching text documents.

2. A search service that uses Apache Lucene, a Java library, to provide document indexing.
and searching capabilities.

3. A backend server implemented using Flask and REST API that handles user requests and interacts with the search service; BERTopic, CTM, LDA topic models; and the database.

4. A SQL database that stores the text documents and its associated metadata.

Search

In this project, we utilize Apache Lucene (v9.0) [10] to index and search the datasets, considering both the title and abstract of the documents. To interact with Lucene, we have created a Flask-based REST API that communicates with Lucene using JavaGateway since Lucene is implemented in Java. The search service utilizes the indexes returned from Lucene to query the documents in the SQL database. Figure 4.2 shows an example of the JSON data format that is returned by the API.
4.2 Backend

SQLite Database

The Topic Modeling Toolkit utilizes an SQLite database as its primary source of data access, whereby the data from user-uploaded datasets are loaded into the database. This is specifically tailored to store one or more datasets containing the documents required by the application. The recommended database schema consists of a fundamental structure represented in Figure 4.3. This schema acts as a framework for creating and arranging datasets, enabling effective information storage and retrieval. As a result, it is essential to follow this schema in order to guarantee the application’s proper operation.

![Figure 4.3: Base Database Schema](image)
Database Configuration

The Topic Modeling Toolkit benefits from dynamic scaling and refactoring of users’ CSV datasets into SQLite databases through its database handlers by utilizing the inherent portability, flexibility, and self-contained nature of SQLite, as shown in Figure 4.4. For additional information on the Database Handler, please refer to the pertinent documentation provided in Section 7.5. The database is designed to be created dynamically based on both user-uploaded datasets and user-defined configurations, taking into account solely those columns necessary for the essential functionalities of the application.

![Image](image)

**Figure 4.4:** Database Configuration in config.ini

Model Training

The Topic Modeling Toolkit is designed with three machine learning models to provide users with a diverse range of machine learning models to choose from, each tailored to cater to unique topic modeling needs and requirements. Each of the trained models and the corresponding data-ID map with the SQLite DB, and will be stored as a pickle file locally. The user has the option to load, save, or validate any trained model.

1. BERTopic [12]: a topic modeling technique that is based on the BERT (Bidirectional
Encoder Representations from Transformers) [9] architecture. It is an unsupervised machine learning model that can be used to identify topics in a large corpus of text data.

2. CTM: a probabilistic generative model used in topic modeling, similar to Latent Dirichlet Allocation (LDA). The CTM is based on the assumption that topics are not independent but are correlated with each other, which allows it to model more complex relationships between topics [6].

3. LDA: a probabilistic generative model used in topic modeling, which is a technique for identifying themes or topics in a large corpus of text data [7].

4.3 Frontend

We decided to use React (v18.2.0) [1], a popular JavaScript library for creating user interfaces, and Tailwind CSS (v3.3.1) [20], a utility-first CSS framework. The combination of them allowed us to create responsive and visually appealing web pages with minimal coding. We used React Router to handle the navigation between pages. This made it easy to switch between the four main pages of the frontend: the home page, the document searching page, the topic browsing page, and the document page.

We started the frontend development by creating wireframes for each of the four pages based on the project requirements and specifications. We then began creating the individual components and styling them using Tailwind CSS. We made sure to keep the code modular and reusable by breaking down complex components into smaller and simpler ones. Throughout the development process, we received feedback from both the client and our team members and made iterative improvements to the design and functionality of the pages.
Chapter 5

Testing

Testing ensures the quality, reliability, and functionality of the system. We want to ensure that our system is flexible and can handle changes in the backend database. We have performed testing and evaluation to ensure that the system can handle changes to the backend database and meta fields without breaking the frontend or backend. As we are not currently implementing Continuous Integration/Continuous Deployment (CI/CD) tests, this will be left as future work for other teams to consider.

5.1 Database

The main objective of database testing is to ensure that the system is capable of accommodating modifications to the backend database and metadata fields without causing any malfunction to either the frontend or the backend. By switching to alternative databases with different columns, we assessed the system’s versatility and scalability, confirming its ability to adapt to distinct database configurations. This process ensured that no errors were present prior to the system’s release to users.

The testing for database is performed in two stages:

1. Functional Testing: The database is tested isolated in functional testing. New Arxiv metadata datasets [4], different from those employed during the development phase,
were used in this process. A new database was first created based on the new Arxiv datasets and the global configuration file was modified accordingly. Subsequently, the database was tested to verify proper data storage and retrieval. This process entailed confirming the presence of all anticipated metadata fields and assessing the correct records from the database.

2. Integration Testing: The database is then tested after integration with the frontend and the backend. This involved verifying that the search and recommendation functions were working correctly with the new database configuration. Any identified issues or errors were addressed by modifying the code or adjusting the database schema.

5.2 Backend

Dedicated test cases were employed to conduct functional testing of the Topic Modeling Toolkit’s backend scripts. Each test case function was customized to match the corresponding function in the base script to guarantee proper execution. The test cases were designed to encompass all possible edge cases, and the results were assessed based on pass or fail criteria and the accuracy of any actions executed during a machine learning model query. This process verified the system’s stability and dependability, confirming that it performed as intended.

5.3 Frontend

We conducted a performance analysis using Chrome DevTools [21] to identify any potential bottlenecks. Through the analysis, we discovered that the CSS styling was overloading the browser with unnecessary code, which was affecting the overall performance of the frontend.
We removed the unnecessary styling and optimized the code based on the analysis, resulting in an improvement in the frontend’s load time.
Chapter 6

Users’ Manual

6.1 Prerequisites

The following is a set of instructions to set up the environment for running the backend and frontend on a Linux server or Unix machine. For optimal performance, it is recommended to create a new Conda virtual environment to prevent package conflicts. It is important to note that a Python version of 3.7 [22] or higher is required for all necessary packages.

1. The environment can be set up using Anaconda as follows:

   (a) Confirm that Anaconda [3] is installed and available. Then, execute the following command to create and activate the environment:

   (b) # Create a Conda virtual environment

       $ conda create --name <env> --file requirements_conda.txt

   (c) # Activate the virtual environment

       $ conda activate <env>

2. Alternatively, the environment can also be set up using pip by executing the following command:

   (a) # Install all required packages.

       $ pip install -r requirements_pip.txt
6.2 Configuration

By following these steps, the user can ensure that the environment is set up correctly and that the backend and frontend can operate seamlessly on the Linux or Unix machine.

After setting up the environment, the user should modify the global config file to customize the application to their specific needs. This config file can be easily adjusted to meet changing requirements, providing flexibility and adaptability. Within the config file, there are several fields that need to be modified, including the pathOfJSON which specifies the location of the dataset file using a relative path, pathOfIndex which specifies the location of the Lucene Indexed files, and pathOfDatabase which provides the location of the database file. Additionally, if the user changes to a different dataset, they must modify the metadata accordingly. Once the config file is modified, the frontend will dynamically render the sorting and filtering options based on the settings provided in the config file. Figure 6.1 shows the global configuration file.
Figure 6.1: Global configuration file
6.3 Website

The user interface of the frontend comprises four primary pages, which include the following: a homepage, a document search page, a topic browsing page, and a document page.

1. Home Page

The landing page serves as the main entry point for the website, providing a comprehensive overview of the system’s capabilities and features. From there, users can easily navigate to the document searching page and topic browsing page, which are designed to help users quickly find and explore relevant information as shown in Figure 6.2.

![Figure 6.2: Home Page](image)

2. Pagination The toolkit has been created to include pagination functionality as shown in Figure 6.3, which enables users to conveniently browse through the outcomes of their analysis in a structured manner, further enhancing the user experience. This
feature makes it easier for users to navigate through large datasets and quickly access particular pages of results, which enhances their ability to analyze and interpret the toolkit’s insights.

3. **Document Searching Page** The document searching page is a key feature of the website, enabling users to browse and explore a vast array of documents from the database. Users can search for specific documents based on keywords and refine their search with the following customizable parameters, as shown in Figure 6.4:
6.3. Website

Figure 6.4: Document Searching Page

(a) **Sorting:** By default, search results are sorted by estimated relevance, as determined by the Lucene Index. However, users can choose to sort results by publishing date and order the results in ascending or descending order as shown in Figure 6.5.

Figure 6.5: An example of sorting
(b) **Filtering**: Users can filter their search results by various options generated from the config file, such as university, department, or language. This feature allows users to narrow down their search to more specific categories and find the exact documents they need as shown in Figure 6.6.

![Topic Modeling Toolkit](image)

Figure 6.6: An example of filtering

4. **Topic Browsing Page**

The topic browsing page allows users to browse and explore different topics related to the documents stored in the database as shown in Figure 6.7. Users can select a specific topic list of interest, and the page will display a list of related documents. By default, 100 topic lists are generated using BERTopic when there are 230K documents in the database. Additionally, users can search for specific topics using keywords, and the topic lists will be dynamically updated to show the most relevant topics. The topic browsing page is essential for users who want to discover and explore new topics related to their interests or research areas.
Figure 6.7: Topic Browsing Page

(a) Related documents for topics are shown in Figure 6.8: students, study, music, research, reading, language, teachers, writing, work, design.
Topics related to machine learning are shown in Figure 6.9.
5. **Document Description Page**

The document page presents the complete content of a chosen document, along with relevant metadata. For text document collection that includes the author’s name, publishing date, university affiliation, title, abstract, advisor, URL, and language, as shown in Figure 6.10. It also includes recommendations for related topics and documents, based on the content of the current document, which allows users to easily navigate to similar topics and documents.
Worlds of Desire: Gender and Sexuality in Classical Tamil Poetry

ABSTRACT

This dissertation contributes to the nascent study of the Tamil Cankam corpus, a collection of poetic anthologies produced in the first three centuries CE. The Cankam poems are constructed around the two complementary themes of the 'inner world' relating to emotions, romance and family life, and the 'outer world' relating to kingship, warfare and public life. This dissertation argues that the thematic division within the corpus is gendered, as the 'inner world' is associated with the feminine while the 'outer world' is associated with the masculine. Each chapter explores the way that the poets establish the boundaries of femininity and masculinity through both the form and content of their verses. This dissertation focuses closely on the moments of rupture in the poets' system of gender construction, for these moments suggest that the poets acknowledged that gender is more fluid and complex than it initially appears. To better understand the workings of gender and sexuality in these poems, this study juxtaposes recent theoretical frameworks with these poems from the distant past. Methodologically, this dissertation collapses traditional historical time, bringing the ancient Cankam anthologies into conversation with ideas that are circulating now. In doing so, it seeks to elucidate both the poems and the theory, while also opening up new questions in both fields.

ADVISOR

Hart, George L;

URI

https://escholarship.org/uc/item/0nk260ck

LANGUAGE

eng

Similar Topics

Topic 21: leadership, principals, school, schools, teachers, faculty, principal, study, leaders, superintendents

Similar Documents

An Environmental and Economic Trade-off Analysis of Manufacturing Process Chains to Inform Decision Making for Sustainability
Chapter 7

Developer’s Manual

7.1 Prerequisites

Please refer to the User’s Manual Prerequisites section to configure the development environment.

Getting Started with Frontend and Backend

Before proceeding with the following commands, it is important that developers make sure they have followed the prerequisite requirements step-by-step, installed the necessary dependencies, and activated the development environment.

1. Starting the Frontend

(a) To get started with the frontend, make sure you are in the project directory. Then, navigate to the topic_modeling_toolkit folder by entering the following command:

$ cd front_end/topic_modeling_toolkit

(b) Once you’re in the correct folder, install the required dependencies by entering the following command:

$ npm install
(c) After the installation is complete, start the frontend by entering the following command:

- $ npm start

(d) Access the application on personal machine: Open your web browser and navigate to http://localhost:3000/ to access the application.

2. Starting the Backend

(a) To start the backend, navigate to the project directory and then to the back_end folder by entering the following command:

- $ cd back_end

(b) Once you’re in the back_end folder, start the search service by navigating to the Search.java file located in the following directory:

- $ cd TopicModelingKit/src/utils/SearchEngine/src/Search.java

(c) After navigating to the Search.java file, start the search service by running the Java file.

(d) Next, navigate back to the back_end folder and start the backend server by running the following command:

- $ python server.py

(e) Access the API: Once the backend server is running, you can access the API by using the URL http://localhost:5000/.
7.2 Configuration File

The config.ini file, located at back_end/TopicModelingKit, serves as a global configuration file for the entire project. Both the frontend and backend read and parse the information in this file to dynamically generate the database and frontend. It contains several configuration options, such as database location, metadata information, search engine settings, etc.

You can modify the config.ini file to customize the project’s behavior, such as changing the database location, adjusting the search engine’s indexing settings, and more. However, be careful when modifying the configuration file, as it can affect the overall performance and stability of the project as shown in Figure 7.1. Therefore, it’s essential to ensure that the configuration file is correctly set up before running the backend or frontend to avoid any unexpected issues.
Below is a brief description of each field in the configuration file. The `config.ini` file contains the following sections:

1. **[dataset]**: Specifies the paths to the JSON file, the index files, and the SQLite database.

   (a) **pathOfJSON**: Path to the JSON file containing the dataset.

   (b) **pathOfIndex**: Path to the folder containing the index files generated by the search engine.
7.2. Configuration File

(c) **pathOfDatabase**: Path to the SQLite database file.

2. **[document]**: Defines the metadata fields of the documents in the dataset.
   
   (a) **metadata**: A list of metadata fields with their names, data types, and display options.

3. **[display]**: Specifies the display options for the frontend.
   
   (a) **topRegion**: A list of metadata fields to display in the top section of the document view page.
   
   (b) **bottomRegion**: A list of metadata fields to display in the bottom section of the document view page.

4. **[database]**: Defines the database schema and paths.
   
   (a) **jsonDatasetPath**: Path to the JSON dataset file.
   
   (b) **databaseName**: Name of the SQLite database file.
   
   (c) **databasePath**: Path to the folder containing the database file.
   
   (d) **datasetTableSchema**: A list of dictionaries specifying the schema of the database table for the dataset.

5. **[model-training]**: Specifies the options for topic modeling and search engine.
   
   (a) **threads**: Number of threads used for data cleaning before training the model.
   
   (b) **loadModel**: Whether to load the pre-trained model or not.
   
   (c) **saveModel**: Whether to save the trained model or not.
   
   (d) **loadLuceneIndex**: Whether to load the pre-built index for the search engine or not.
7.3 Backend APIs

The APIs inside the server.py are built using the Flask framework, which allows developers to build RESTful [15] web services easily and quickly. They provide a standard interface for clients to interact with the document database and machine learning models over HTTP.
### 7.3. Backend APIs

<table>
<thead>
<tr>
<th>API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/docs</td>
<td>Returns all the documents in the database, with each document represented as a JSON object containing its ID and metadata fields.</td>
</tr>
<tr>
<td>/api/search</td>
<td>Returns a list of documents that match the given query. The query parameters include q (the query string), sort (the field to sort on), order (the sort order), filter_field (the field to filter on), and filter_input (the value to filter on).</td>
</tr>
<tr>
<td>/api/get_allow_sort</td>
<td>Returns a list of fields that can be used to sort the documents. The list is based on the configuration file.</td>
</tr>
<tr>
<td>/api/get_allow_filter</td>
<td>Returns a list of fields that can be used to filter the documents. The list is based on the configuration file.</td>
</tr>
<tr>
<td>/api/get_top_region</td>
<td>Returns a list of fields that are displayed at the top of the document details page. The list is based on the configuration file.</td>
</tr>
<tr>
<td>/api/get_bottom_region</td>
<td>Returns a list of fields that are displayed at the bottom of the document details page. The list is based on the configuration file.</td>
</tr>
<tr>
<td>/api/document/<a href="">string:document_id</a></td>
<td>Returns a single document based on its ID, with the document represented as a JSON object containing its ID and metadata fields.</td>
</tr>
<tr>
<td>/api/all_labels</td>
<td>Returns a list of all topic labels.</td>
</tr>
<tr>
<td>/api/labels</td>
<td>Returns a list of document labels that match the given topic query. The query parameter is topic_query (the topic query string).</td>
</tr>
<tr>
<td>/api/get_similar_topics</td>
<td>Returns a list of similar topics for a given document ID. The query parameter is doc_id (the document ID). The list is based on the similarity of the document’s topic to other topics in the model.</td>
</tr>
<tr>
<td>/api/get_similar_documents</td>
<td>Returns a list of similar documents for a given document ID. The query parameter is doc_id (the document ID). The list is based on the similarity of the document’s topic to the topics of other documents in the model. The number of documents returned is determined by the SIMILAR_DOC_COUNT constant.</td>
</tr>
</tbody>
</table>
7.4 Topic Models

The Topic Modeling Toolkit supports three different types of models: BERTopic, CTM, and LDA. All trained models can be saved locally as pickle files and loaded in future runs based on the provided parameters. If the model is saved, then a document index to ID mapping in the form of a dictionary will also be generated and saved as a pickle file along with the model. The mapping of the document index to ID is utilized to accurately match documents trained from models with the corresponding documents within the database, as the output order of the training process may not always be consistent.

- **BERTopic**

![Figure 7.2: BERTopic Training Function](image)

The BERTopic training function as shown in Figure 7.2 is responsible for training and outputting the trained model.

- **Parameter `num_topics`:** This parameter refers to the number of terms each topic may have. Setting it to (1, 1) ensures that all topics only contain a single
word. Setting the max value higher would result in a longer training time.

- **Parameter load_trained_model**: Set to True to load a local model and set to False otherwise. If this parameter is set to True while no model is found locally in the target directory, a `BaseException` will be raised.

- **Parameter offload_trained_model**: Set to True to save the trained model and set to False otherwise. If this parameter is set to True while `load_trained_model` is also set to True, the model loaded will not be saved again.

- **CTM**

```python
def train_model(self, load_trained_model=True, offload_trained_model=True):
    ...
    Train/Load CTM Model based on the provided (clean) documents.
    Models are trained with CombinedTM in CTM (contextualized topic model).
    ...
    def offload_model():
    ...
    def load_model():
        # Load model
        load_success = False
        if load_trained_model: --

        # If model load failed || model not loaded
        if not load_trained_model or not load_success:
            print("Training model...")

        NUMBER_OF_TOPICS = 100
        qt = TopicModelDataPreparation("all吩咐-base-es")
        training_dataset = qt.fit(text_for_contextual=self.documents_clean, text_for_how=self.documents_clean)
        ctm = CombinedTMbow_size=len(qt.vocab), contextual_size=768, n_components=NUMBER_OF_TOPICS, num_epochs=10000)
        ctm.fit(training_dataset) # run the model

        testing_dataset = qt.transform(text_for_contextual, text_for_how)
        res = ctm.get_doc_topic_distribution(testing_dataset, n_samples=20)
        for r in res:
            self.DOCUMENT_IDX_TOPIC_MAP.append(r.index(max(r)))
        self.trained_model = ctm
```

Figure 7.3: CTM Training Function

The CTM training function as shown in Figure 7.3 is responsible for training and outputting a trained CTM model.

- **Parameter load_trained_model**: Set to True to load a local model and set to
False otherwise. If this parameter is set to True while no model is found locally in the target directory, a `BaseException` will be raised.

- **Parameter offload_trained_model**: Set to True to save the trained model and set to False otherwise. If this parameter is set to True while `load_trained_model` is also set to True, the model loaded will not be saved again.

The CombinedTM model is trained with the help of a training dataset containing a collection of preprocessed documents. The toolkit is first initialized and fitted to the dataset using a method called “Quantized Transformer” (abbreviated as “qt”). This method takes the preprocessed documents as input and creates a representation of the documents that can be used by the combined model. Once the training dataset has been processed by the qt, the CombinedTM model is initialized and trained using the processed documents. The CombinedTM is initialized using a bag-of-words representation (bow_size), which counts the number of times each word appears in the documents, and contextual embeddings (contextual_size), which capture the meaning of words in their context. The CTM will generate a specified number of topics (n_components) and train for a specified number of iterations (num_epochs) to find the most relevant topics for the given dataset.
7.4. Topic Models

- **LDA**

```python
def train_model(self, load_model, save_model):
    """
    Train/load LDA Model based on the provided (clean) documents.
    Models are trained with LDA (with inbuilt TF-IDF vectorizer).

    :param load_model: True to load a trained model and False otherwise. Use False to force a model training.
    :param save_model: True to save the model and False otherwise.
    """
    def offload_model(): ...

def load_model(): ...

    # Load model
    load_success = False
    if load_trained_model: ...

    # If model load failed || model not loaded
    if not load_trained_model or not load_success: ...

    vect = TfidfVectorizer(stop_words=STOPWORDS, max_features=1000)
    vect_text = vect.fit_transform(self.documents)

    NUM_OF_TOPICS = 300
    from sklearn.decomposition import LatentDirichletAllocation
    lda_model = LatentDirichletAllocation(n_components=NUM_OF_TOPICS,
                                           learning_method='online',
                                           random_state=42,
                                           max_iter=1)
    lda_top = lda_model.fit_transform(vect_text)
    self.trained_model = lda_model

    # Save trained model (if model is loaded, don’t offload it again)
    if not load_trained_model and offload_trained_model: ...

Figure 7.4: LDA Training Function
```

The LDA training function as shown in Figure 7.4 is responsible for training and outputting a trained LDA model.

- **Parameter** `load_trained_model`: Set to True to load a local model and set to False otherwise. If this parameter is set to True while no model is found locally in the target directory, a `BaseException` will be raised.

- **Parameter** `offload_trained_model`: Set to True to save the trained model and set to False otherwise. If this parameter is set to True while `load_trained_model` is also set to True, the model loaded will not be saved again.

The algorithm is initialized with a specified number of topics, which is determined
by the value of “NUM_OF_TOPICS”. The “learning_method” parameter is set to “online”, which means the algorithm updates the model parameters incrementally as new data is processed, instead of all at once. The “random_state” parameter is set to 42, which is a simple random seed used to ensure the reproducibility of the results. The “max_iter” parameter is set to 1, meaning that the algorithm will perform one iteration of training.

After the initialization, the algorithm is fit to a matrix of document-term frequencies called “vect_text”. This matrix represents the occurrence of each word in each document of the dataset. The “fit_transform” method is used to perform the training of the algorithm on the dataset and to extract the topic distribution for each document in the input dataset.
7.4. TOPIC MODELS

- Sample Command-Line Training Output

A sample of the model training results is displayed in Figure 7.5, which includes various important fields utilized by both the backend and front end for filtering, browsing, and search functionality.

- **Document ID**: The ID of the document generated from Lucene Index. This unique value is used to identify a specific document.

- **Topic ID**: The topic's ID to which the document was matched to. This unique value is used to identify a specific topic. Multiple documents may correspond to the same topic.

- **Topic Accuracy**: The accuracy for the topic matched with a document.

- **Topic Terms**: The terms associated with a specific topic.
7.5 Database Handler (dataset_dbtool.py)

The database handler is a script coded in Python 3 for handling all data-related tasks in the Topic Modeling Toolkit. This includes all SQLite operations as well as scaling and loading user-uploaded datasets into the database.

1. Base SQLite Functions

The database handler script has been designed to prioritize modularity, which brings several advantages to the application. The breakdown of the database handling functionality into smaller, self-contained modules simplifies the maintenance and update process of the codebase.

Specifically, the script is equipped with several core functions that enable developers to perform essential database management tasks. These functions include connecting to the database, inserting records into the database, removing records from the database, creating new tables in the database, and resetting the database to its original state. Figure 7.6 shows the implemented SQLite functions.
```python
def get_sqlite_conn():
    ...
    Establish connection to the target SQLite DB.
    :return: SQLite Connection
    ...
    print(os.path.join(DB_PATH, DB_NAME))
    return sqlite3.connect(os.path.join(DB_PATH, DB_NAME))

def drop_all_sqlite_tables(db_conn):
    ...
    Drops SQLite data tables. Used to setup a clean run.
    :param db_conn: SQLite DB connection
    ...
    cur = db_conn.cursor()
    cur.execute("DROP table IF EXISTS Dataset;")
    db_conn.commit()
    cur.close()

def create_sqlite_db(db_conn):
    ...
    Create new tables for the SROM SQLite database with the given schema (if they don’t already exist).
    :param db_conn: SQLite DB connection
    ...
    schema_list = []
    for col in DATASET_TB_SCHEMA:
        schema_list.append(f"(list(col.keys())[0])\t\{list(col.values())[0]}")
    schema = ",\n\".join(schema_list)
    create_table_sql = f""
    CREATE TABLE IF NOT EXISTS Dataset ( 
        {schema}
    );
    ""
    cur = db_conn.cursor()
    cur.execute(create_table_sql)
    cur.execute(""
    CREATE TABLE IF NOT EXISTS Metadata ( 
        version TEXT NOT NULL DEFAULT '0.0.4',
        db_created_on DATETIME NOT NULL DEFAULT current_timestamp
    );
    ""
    cur.execute("INSERT INTO metadata DEFAULT VALUES;")
    db_conn.commit()
    cur.close()

def load_database_table(db_conn, column="*", table_name="Dataset"):
    ...
    Load the the given table from the database.
    ...
    cur = db_conn.cursor()
    sql = f"SELECT {column} FROM {table_name} ORDER BY id ASC;"
    cur.execute(sql)
    return [i[0] for i in cur.fetchall()]
```

Figure 7.6: SQLite Functions in dataset_dbtool.py
2. Database Generation

The database generator is responsible for cleaning, scaling, and loading the data into the SQLite database based on a JSON file generated from the user-uploaded CSV file and the user-defined configuration, as shown in Figure 7.7.

```python
def load_dataset(auto_correct=True):
    ...

def load_to_db(conn, dataset_dict, validate=False):
    cur = conn.cursor()

    validate_ct = len(dataset_dict)
    validate_id = dataset_dict.keys()

    ...
    INSERT INTO table_name (column1, column2, column3, ...)
    VALUES (value1, value2, values, ...);
    ...
    print("Generating new database...")
    for doc_id, metadata in tqdm(dataset_dict.items(), ncols=100, desc="Loading records to DB: ", ascii='='):

        # == DATA PARSE ==
        # ==
        for k, v in metadata.items():
            if not v:
                metadata[k] = "None"
            if isinstance(metadata[k], str):
                metadata[k] = metadata[k].replace("\", "\")
            metadata["year"] = int(metadata["year"])

        # == DATA INSERT ==
        # ==
        dataval = [doc_id] + list(metadata.values()) + [datetime.datetime.now().strftime("%Y-%m-%d")]
        values = [f"\"{v}\"\" for v in dataval]
        sql = f"INSERT INTO Dataset VALUES({', '.join(values)})"
        cur.execute(sql)
        conn.commit()
    print(f"Database generated at {os.path.join(DB_PATH, DB_NAME)}")
```

Figure 7.7: DB Generation in dataset_dbtool.py

3. Database Validation

Database validation is a crucial aspect of data management, and as such, we have designed two simple methods to ensure data integrity for the datasets generated. Given the large sizes of some of these datasets, the validation process needs to be both efficient and effective:
(a) **Count Validation**: Checks that the number of records in the given dataset is equivalent to the number of records in the database. Count validation’s execution is extremely fast and is always run before content validation. In the event that the count validation fails, content validation will not be executed, to save time.

(b) **Content Validation**: Ensures that all document IDs in the given dataset exist in the generated database. Executing content validation ensures that the data contained in the database is accurate and complete, as shown in Figure 7.8.

```python
def validate:
    print("Validating database's integrity...")
    cur.execute("SELECT count(*) FROM Dataset;")
    row_ct = cur.fetchall()[0][0]
    if row_ct == validate_ct:
        print_res(success=True, type="Count")
    else:
        print_res(success=False, type="Count")

    cur.execute("SELECT * FROM Dataset;")
    rows_content = cur.fetchall()
    doc_ids = [doc_id[0] for doc_id in rows_content]
    for i in validate_id:
        if int(i) not in doc_ids:
            print_res(success=False, type="ID"); return
    print_res(success=True, type="ID")
```

Figure 7.8: Simple DB Validation in dataset_dbtool.py
7.6 Standalone Execution (command line)

The design of the script allows it to be executed as a standalone application in the command line interface as shown in Figure 7.9. This feature provides developers with a dynamic tool to generate, validate, and create portable and testable databases. The dynamic nature of the script enables developers to generate and validate databases based on user-defined configurations, making it a highly customizable and adaptable tool. This functionality ensures that developers can create databases that are specifically tailored to meet the needs of their applications.

To generate the database in the command line, configure the config.ini file and run the shell script:

- $ source back_end/TopicModelingKit/src/database/generate_db.sh

![Figure 7.9: Command Line Execution of dataset_dbtool.py](image)
7.7 Config.ini Dependent

The database handler is intricately linked to the application’s configuration through the use of a config.ini file as shown in Figure 7.10. This configuration file is used to specify various settings and parameters that the database handler relies on to function properly. By utilizing the config.ini file, the database handler is able to seamlessly integrate with the rest of the application without requiring users to directly edit the file themselves.

```python
JSON_DATASET_ABS_PATH = config.get('database', 'jsonDatasetPath').replace(\'', '')
DB_NAME = config.get('database', 'databaseName').replace(\'', '')
DB_PATH = config.get('database', 'databasePath').replace(\'', '')
DATASET_TB_SCHEMA = eval(config.get('database', 'datasetTableSchema'))
DEFAULT_DB_COL_ORDER = [list(v.keys())[0] for v in DATASET_TB_SCHEMA]
NON_OPTIONAL_METADATA_COL = [list(v.keys())[0] for v in DATASET_TB_SCHEMA if v['optional'] == False and v['default'] == False]
OPTIONAL_METADATA_COL = [list(v.keys())[0] for v in DATASET_TB_SCHEMA if v['optional'] == True]
```

Figure 7.10: Config Loading in dataset_dbtool.py

7.8 Frontend

To organize our code, we created four main folders within the src directory of our React project as shown in Figure 7.11. The API folder contains the JavaScript files that connect the frontend to the backend via GET requests. The assets folder consists of the static images used for the home page. The components folder has the Footer.js and Header.js which are used throughout the entire website. The pages folder contains all pages used for displaying the frontend. The App.js uses BrowserRouter to navigate between different pages.
Figure 7.11: An overview of the frontend development
Chapter 8

Lessons Learned

8.1 Timeline

Table 8.1: Timeline of Topic Modeling Toolkit

<table>
<thead>
<tr>
<th>Month</th>
<th>Task</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>January</td>
<td>Familiarize with existing code and system; Understand existing system’s functionalities; Learn backend components; Get familiar with BERTopic modeling</td>
<td>Familiarization, BERTopic Modeling</td>
</tr>
<tr>
<td>February</td>
<td>Finalize system architecture; Implement SQLite Database; Implement BERTopic; Implementation of UI; Prepare for presentation 1</td>
<td>System Architecture Finalization</td>
</tr>
<tr>
<td>March</td>
<td>Implement SciBERT Topic Model; Implement search engine; Prepare for presentation 2</td>
<td>Backend Model Training</td>
</tr>
<tr>
<td>April</td>
<td>Improve recommendation system using text-filtering framework; Integrate data into Python-based toolkit; UI Improvement; Submit to VTechWorks; Prepare for final presentation</td>
<td>Interactive UI, Improved Recommendation System, Integration with Python-Based Toolkit, UI Improvement</td>
</tr>
<tr>
<td>May</td>
<td>Finalize and review project</td>
<td>Project Finalization and Review</td>
</tr>
</tbody>
</table>

8.2 Problems

There are several potential problems that must be considered to ensure the success of our system. The quality of the text documents, which can differ greatly in terms of completeness and accuracy, is one important factor. The performance of our system may be negatively impacted by incomplete or inaccurate data, which could produce search results that are
incorrect or irrelevant. In order to use the text document collection in our system, it is essential to make sure that it has been thoroughly validated and cleaned.

The machine learning models that we are using, BERTopic, CTM, and LDA are also subject to potential issues. While BERTopic is a powerful and popular model, it may not always perform optimally for our specific use case. We need to ensure that we have thoroughly tested the model to ensure that it is providing accurate results for our text documents. Both the CTM and LDA machine learning models may have disadvantages. Despite the fact that CTM is renowned for its capacity to record information about topic and word co-occurrence, it can be sensitive to model initialization and may call for extensive parameter tuning. On the other hand, LDA is a popular model but can experience topic fragmentation, where multiple related topics rather than a coherent set of topics are found.

Hence, it is essential to consider these potential problems and implement strategies to address them to ensure the success of our system. Thorough testing and validation are crucial to ensure that our system meets the desired requirements and specifications, and provides an accurate and efficient user experience.

8.3 Solutions

To ensure the quality of the text documents, several solutions were implemented:

1. Validate and clean the data: To eliminate any inaccurate or incomplete data, a thorough validation and cleaning process was used. The user-uploaded dataset was cleaned by removing all stopwords using NLTK (Natural-Language Toolkit) and applying other criteria based on the user-specified configuration file, such as the number of available threads. These steps ensured that the user-uploaded dataset was thoroughly cleansed
and prepared for further analysis and processing.

2. Worked together with subject-matter experts: Collaboration with subject-matter experts, such as professors or librarians, proved instrumental in identifying and resolving issues with the text documents.

To improve the accuracy and efficiency of the search engine, the following solutions were implemented:

1. Advanced search algorithm was used to improve the accuracy of the search results. Lucene, a widely-used information retrieval library, was incorporated into our toolkit to enhance the search experience for our users and provide more accurate and relevant search results. The integration of Lucene into our toolkit provided efficient indexing and search capabilities, ultimately improving the overall search accuracy and user experience.

2. Configured various parameters, such as the indexing algorithm, query parsing, and ranking algorithms, to optimize the search engine’s configuration.

To address potential issues with the topic models, the following solutions were implemented:

1. Use model fine-tuning: To optimize the accuracy of the model for each dataset, we implemented fine-tuning parameters for the model-training functions that allow users to better fit the model to their unique use case.

2. To ensure that the topic models consistently delivered accurate results for text document collections, the model underwent regular retraining. Additionally, during the retraining process, modifications to the model’s hyperparameters were made to optimize its performance.
8.4 Future work

There are several areas of future work that could be explored to enhance the capabilities and functionality of our text document collection search system. Here are some potential areas for future work:

1. Implementation of additional machine learning models: Although BERTopic, CTM, and LDA are strong machine learning models, our system’s performance might be improved by incorporating additional models, such as SciBERT. To increase the precision of our document search results, for instance, we could investigate the use of deep learning models or ensemble models.

2. Evaluation of different machine learning models: A future consideration would be to conduct a comprehensive evaluation of various machine learning models on the text corpus. This evaluation will enable the identification of the best models for specific use cases and improve the precision and robustness of the topic modeling method. Explore the possibility of using model ensembles or hybrid models to further enhance the performance of the system. Potential models that could be considered include SciBERT and other deep learning models that are suitable for the use case.

3. Integration with additional data sources: For document search and retrieval at the moment, our system only support flat text documents. To improve our system’s search capabilities, though, additional data sources might be integrated. To broaden the scope of the search results, for instance, we could use external databases or repositories.

4. Enhancements to the user interface: Although our current user interface is functional, there may be opportunities to enhance the user experience and make the program more simple to use. To enhance the browsing experience for users, we might, for instance,
implement more sophisticated search filters or offer more thorough document previews.

5. Optimization for scalability and performance [17]: As the amount of data in our system grows, it will be important to ensure that our system is optimized for scalability and performance. This could involve optimizing the search algorithms, improving the database architecture, or implementing caching strategies to improve the speed and efficiency of the system. By estimating the amount of memory and processing power needed to handle the data, a developer can determine the necessary computer configuration for a specific document collection. This can be accomplished by tracking the system’s resource usage and running tests to determine how well it operates under various loads and configurations. A developer may also think about utilizing distributed computing frameworks, like Hadoop or Spark, to split the workload among several computers and increase the system’s scalability.

6. Integration of additional search engines: To offer more complete search results, we might integrate search engines in addition to our current ones. It would involve creating a mechanism for our system to communicate with the other search engine’s APIs. This would enable our system to query these search engines alongside our current search engine, allowing us to provide more comprehensive search results to our users. When a user enters a query, for instance, our system could simultaneously search Google Scholar and our current search engine as well as other academic search engines, then combine and present the user with the results from all sources.

7. Implementation of Continuous Integration/Continuous Deployment (CI/CD) tests: When changes are made to the codebase, define a set of tests that cover various aspects of the system and configure the tool to run these tests automatically. Additionally, the tool would need to be set up so that it only deploys the code to the production environment after each test has been successfully completed.
Chapter 9

Acknowledgements

Aman Ahuja: Client
Email: aahuja@vt.edu

Dr. Edward Fox: Instructor
Email: fox@vt.edu

Chenyu Mao: Student Volunteer
Email: mchenyu@vt.edu

We want to express our gratitude to our client, Aman Ahuja, and instructor, Dr. Edward Fox, for giving us the opportunity to work on this project. We also thank student volunteer Chenyu Mao for assisting us with technical issues. We want to acknowledge the previous work done by students in CS 4624.
Bibliography


[17] David Mimno, Hanna Wallach, Edmund Talley, Miriam Leenders, and Andrew Mc-
BIBLIOGRAPHY


Appendices
Appendix A

Methodology

A.1 Goals

The goals of each of the types of users that our system can support:

1. Researchers/students/educators require a system that allows them to quickly locate and access relevant academic papers. They should be able to quickly locate papers related to their research topic, discover related papers they may not have been aware of, and access the full text of the papers.

2. Information professionals/librarians require a system that allows them to easily manage and organize large collections of academic documents, as well as search and filtering features that allow them to efficiently retrieve and browse relevant documents.

3. Developers require API access and documentation in order to integrate the toolkit with their own software, as well as an extensible architecture that can be easily customized to meet their specific requirements.
A.2 Tasks and Subtasks

Figures A.1 through A.5 present a detailed breakdown of each goal, consisting of various units of tasks and subtasks.

Figure A.1: Building a recommendation system
Building a recommendation system

The recommendation system begins with the creation of an index of topics and documents, which is used to provide search and recommendation functionality to users. The search and recommendation algorithms can be further improved through user feedback and other techniques.

Figure A.2: Refine the topic modeling toolkit based on the results
Refine the topic modeling toolkit based on the results

The program should evaluate the quality of the topic modeling results. This involves examining the topics generated by the model to determine if they are meaningful and accurate. Afterward, the corresponding implementation will be added to the system and the refined toolkit will be used on a new set of documents to determine if it is effective at document categorization.
The goal is to build a search engine with query suggestion capabilities in order to improve the user experience of searching. This includes creating a query analysis system that will provide suggestions for related queries, as well as implementing an autocomplete feature.
to allow for faster and more accurate searches. Then we’ll provide a user interface with pagination and filtering capabilities. Finally, we will iteratively test the search results to improve performance.

![Diagram](image.png)

**Figure A.4: Filter documents based on user-specified parameters**

---

**Filter documents based on user-specified parameters**

The filter system involves taking a list of documents and filtering them based on certain criteria, such as start and end dates, universities, and other parameters. The result is a list
of filtered documents that meet the specified criteria.

Figure A.5: Redesign new UI
The goal is to enhance the user experience of a Flask-based system by creating a new and improved user interface. We first need to evaluate and select suitable frontend frameworks, integrate the chosen framework with Flask, and implement new UI components. Then we have to test the finished user interface and make improvements based on user feedback.

### A.3 Workflow Table

<table>
<thead>
<tr>
<th>Service ID</th>
<th>Service Name</th>
<th>Input file name(s)</th>
<th>Input file IDs</th>
<th>Output file name</th>
<th>Output file ID</th>
<th>Libraries; Functions; Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>Topic Modeling Toolkit</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>BERTopic, CTM, LDA</td>
</tr>
<tr>
<td>1B</td>
<td>Search Functionality</td>
<td>Document List</td>
<td>1A</td>
<td>Indexed Document List</td>
<td>1C</td>
<td>Apache Lucene, Java Gateway</td>
</tr>
<tr>
<td>1C</td>
<td>Pagination Functionality</td>
<td>Indexed Document List</td>
<td>1B</td>
<td>Paginated Document List</td>
<td>1D</td>
<td>Python Standard Libs</td>
</tr>
<tr>
<td>2A</td>
<td>Query Suggestion Algorithm</td>
<td>Search Query List</td>
<td>1B</td>
<td>Query Suggestions List</td>
<td>2B</td>
<td>Apache Lucene</td>
</tr>
<tr>
<td>2B</td>
<td>Autocomplete Algorithm</td>
<td>Partial Query String</td>
<td>1B</td>
<td>Autocomplete Suggestions List</td>
<td>2C</td>
<td>Python Requests, JSON</td>
</tr>
<tr>
<td>2C</td>
<td>Recommendation Algorithm</td>
<td>Search Query String</td>
<td>1B</td>
<td>Recommended Topics List</td>
<td>2D</td>
<td>Elastic</td>
</tr>
<tr>
<td>3A</td>
<td>Query Suggestion Integration</td>
<td>Query Suggestions List, Search Query String</td>
<td>2A, 1B</td>
<td>Updated Search Query String</td>
<td>3B</td>
<td>Python Requests, JSON</td>
</tr>
<tr>
<td>3B</td>
<td>Autocomplete Integration</td>
<td>Partial Query String, Autocomplete Suggestions List</td>
<td>1B, 2B</td>
<td>Updated Search Query String</td>
<td>3C</td>
<td>BERTopic</td>
</tr>
<tr>
<td>3C</td>
<td>Recommendation Integration</td>
<td>Search Query String, Recommended Topics List</td>
<td>1B, 2C</td>
<td>Recommended Topics List</td>
<td>3D</td>
<td>Python Requests, JSON</td>
</tr>
<tr>
<td>4A</td>
<td>Search Functionality Integration</td>
<td>Search Query String, Indexed Document List</td>
<td>1B</td>
<td>List of Matching Document IDs</td>
<td>4B</td>
<td>Python Standard Libs</td>
</tr>
<tr>
<td>4B</td>
<td>Document Retrieval Functionality</td>
<td>List of Matching Document IDs, Document List</td>
<td>4A, 1A</td>
<td>List of Matching Documents</td>
<td>4C</td>
<td>Python Standard Libs</td>
</tr>
<tr>
<td>4C</td>
<td>Pagination Integration</td>
<td>Paginated Document List, List of Matching Documents</td>
<td>1C, 4B</td>
<td>List of Paginated Documents</td>
<td>4D</td>
<td>Milvus</td>
</tr>
<tr>
<td>5A</td>
<td>UI Design and Implementation</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>React, Tailwind CSS</td>
</tr>
</tbody>
</table>
A.4 Workflows

A list of workflows covering each goal follows, with services detailed in Table A.1.

Goal 1: Build a topic modeling and search engine for articles

Workflow 1: Service 1A + Service 1B + Service 1C

Goal 2: Build query suggestion, autocomplete, and recommendation features for the search engine

Workflow 2: Service 2A + Service 1B + Service 2B
Workflow 3: Service 2C + Service 1B + Service 2A

Goal 3: Integrate query suggestion, autocomplete, and recommendation features into the search engine

Workflow 4: Service 3A + Service 2A + Service 1B
Workflow 5: Service 3B + Service 2B + Service 1B
Workflow 6: Service 3C + Service 2C + Service 1B

Goal 4: Add pagination and document retrieval functionality to the search engine

Workflow 7: Service 4A + Service 1B
Workflow 8: Service 4B + Service 1A + Service 4A
Workflow 9: Service 4C + Service 1C + Service 4B

Goal 5: Design and implement a user interface for the search engine

Workflow 10: Service 5A + Service 4C