ArtiCrawl: A Hybrid Recommender System for Researchers Using User-based Filtering and Content-Based Filtering utilizing Topic Modelling

---

A Thesis Project Presented to the Faculty of

the College of Information, Computer and Communications Technology

University of San Jose - Recoletos

Cebu City, Philippines

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

---

by

Kristoffer Francis E. Milan

Jules Russel A. Lucero

Javin Stefan R. Tan

December 2021

ABSTRACT

This study aims to make finding related literature easier for researchers by using a recommender system. The recommender system uses a Hybrid-Recommender that uses both Content-Based Filtering and User-Based Collaborative Filtering. For the Content-Based Filtering system, Latent Dirichlet Allocation is utilized to create a topic model that predicts a given text's topic distribution based on a given trained corpus. The User-Based Collaborative Filtering system recommends articles by learning the user and item features through Matrix Factorization, by factoring the user-item matrix. This Hybrid-Recommender System is fed to a web application where researchers can get suggestions from the system by entering their research abstract.

Keywords:

Recommender System, Hybrid Recommender System, Content-based Filtering, User-based Collaborative Filtering, Topic Modeling, Natural Language Processing, Machine Learning, Computer Science

CHAPTER I

INTRODUCTION

Rationale of the Study

Finding related literature has been a huge problem for many researchers. A simple search on google will give you not that many relevant results, and the problem is most of these results end up being hidden behind paywalls where many researchers can't access them. Because of this, many people get turned off at the idea of research because of the long and tedious process that just one of the sections in the paper requires.

Currently, there are a few solutions to this problem. Google scholar is one popular solution that many scholars use to both find related literature, and share their published articles in their scholar profiles. Google claims that documents are ranked "in a way researchers do" which includes citation statistics, the author, the publication it appears in, and by "weighing the full text" [1]. Bielefeld Academic Search Engine (BASE) is another scholarly article searching platform that offers search results from the "Deep web" as a solution to the small number of results that google and other platforms give [2]. Aside from Google Scholar and BASE, there are many other search engines that also offer a huge repository of articles. Some of them, like Springer Link, allow you to download research and books that are normally behind paywalls, for free.

While many of these solutions offer large repositories, and many others offer free repositories, the problem of making research and finding related literature easier is still present. It still isn't as easy as doing one search and then being recommended search results that actually fit what you're looking for. This is because all of these search engines use keyword-based searches. The problem here is that these keyword searches return articles that have either those keywords in the title, or have those keywords be relevant in the full text. Google scholar, a very popular search engine, has also been found to rank the number of times that text has been cited very highly [3]. Beel and Gipp's study on Google scholar [4] found that Google scholar's citation metrics could easily be manipulated through spam, and therefore can also give results that are non-sense as long as they are highly cited.

This study aims to remedy these problems by making sure that researchers are able to let the search engine know what their study is about and recommend articles that are similar to it. This is possible by allowing researchers to input their research abstracts instead of keywords. This research abstract will then be compared to articles in the repository to find articles that are similar to the abstract. Furthermore, all of the articles that will be recommended to the researchers will be coming from Open Journal Sources which assures them that they won't have to pay anything to be able to access the articles being recommended to them.

THEORETICAL BACKGROUND

This study will focus on the creation of a Hybrid recommender system for its recommendations to users. Recommender systems are systems that provide content recommendations or suggestions to users through the use of certain software and techniques [1]. These are mainly used in applications that offer a large amount of content so users don't waste time scrolling endlessly through the application to find the best content that they want to consume. For example, Netflix uses a Recommender system to help users find shows that they might like, a lot easier. They do this by allowing the Recommender system to take note of what a user's preference is and then suggesting shows that are similar to those that the user prefers. This preference that a user has, in simple terms, is based on what kind of content that they usually consume. If for example, they watch Romantic Comedy shows a lot, the system takes note of this as a preference that a user has. There are many approaches to Recommender systems. This includes approaches like Collaborative filtering, Content-based filtering, and Hybrid recommender systems.

A hybrid recommender system makes use of two different recommendation approaches: Collaborative filtering and Content-based filtering. There are various ways to hybridize a recommender system. One way to do this is by implementing both methods separately and then combining their results to get recommendations. Another method is to use the capabilities of one approach, and add it to another. For example, adding elements of Content-based filtering to a Collaborative filtering approach or vice versa [2].

Content-based filtering, one of the methods used in a Hybrid recommender system, uses item features to recommend items to users [3]. These item features are ways that we can describe and categorize certain items. For example, shows on Netflix can be categorized by type, genre, location, actors, directors, and etc. When a user interacts with an item, either by watching or liking that content, the system takes note of that and adds the item features to the user's preferences. In simpler terms, Content-based filtering recommends users items based on the features that users prefer.

When recommending text content, the topics of these texts could be considered as features. Most of the time, there is no way to know that texts are part of certain topics unless they have been manually categorized. Topic modeling remedies this problem by automatically identifying the topics of the texts so we can use them as features. Topic modeling is a technique that allows us to cluster documents using an algorithm like Latent Dirichlet Allocation. It is an unsupervised machine learning algorithm, which means that you don't have to have documents pre-labeled with topics and then fed to the system to train it to classify topics of the texts. Instead, Topic modeling relies on counting words and finding groups of similar word patterns to find the topics in your data [4]. This method is applied to the collection of documents, or corpus, as a whole.

To start the process of topic modeling, there needs to be a database filled with documents, also known as a corpus. Usually, projects already have a set corpus that they want to analyze and produce topics of. This could be a group of magazines, a group of books, or a group of news articles that you already have, and want to apply topic modeling to. If not, a corpus can be built by getting documents from the internet.

You could do this manually by saving documents or web pages, or by copy-pasting. An automated approach to building your own corpus could be done through Web scraping.

According to Bright Data [5], Web scraping, or web data extraction, is an automated way of extracting specific data from web pages. A scraper, the bot that does the data extraction, identifies specific parts of the web pages that you want to extract data from. For example, a scraper can be given specific HTML elements that it should locate and then extract the data from those elements. Web scraping also involves web crawling in order for it to be able to extract multiple data from different sources. Web crawling is, essentially, a way for a bot called a crawler to move from one web page to another. Web crawling and web scraping work hand in hand in order to slowly build the corpus one document at a time. The data that is scraped by the scraper is then stored in a database for it to be processed for topic modeling.

Before the actual topic modeling happens, data needs to be preprocessed in order for the model to be fed with only the details that are important and then transformed in a way that both the computer and the user can understand. This involves removing null values, checking if values are consistent, and validating the correctness of the data [6]. In text documents, data preprocessing may involve but is not limited to, removing words that hold no meaning or words that are repeated in every document, transforming words back to their root forms, and making sure that the documents you have are all in one language. After these techniques are applied, data then needs to be transformed in a way that the computer can understand it, and this can be achieved through vectorization.

Vectorization involves converting text data into numerical data [7]. Computers are best able to understand numerical values so all of this text data should be converted into something that it can understand. Two of the most popular approaches to text vectorization are Bag of Words and Term Frequency-Inverse Document Frequency (TF-IDF). TF-IDF is a score given to each word based on how important that term is, given its frequency in a certain document and its frequency in the corpus. TF-IDF is a simple multiplication of a term's Term Frequency (TF) and Inverse Document Frequency (IDF) scores. The TF is taken by counting how many times this term appears in a single document divided by the number of total words in that document. The IDF is calculated by taking the inverse, or the natural logarithm of the Document Frequency (DF). DF is taken by counting the number of documents in the corpus divided by the number of documents that contain the term. This results in a document term matrix where the documents headline the rows, the terms headline the columns, and the values added in the matrix are the TF-IDF scores of each term relative to that document.

Once the text data are vectorized and a document term matrix has been produced, topic modeling can finally start. One of the most popular algorithms to conduct topic modeling is called Latent Dirichlet Allocation (LDA). LDA assumes that each document has a probability distribution that it belongs to a topic (theta) and that each topic has a probability distribution that a word is a part of it (phi) [8]. These mixtures are matrices that the LDA algorithm will output at the end. The mix of topics that a document has is dictated by the mix of words that exists in that document. Since each word belongs to certain topics, the existence of that word in that document influences the topic of that document.

LDA is able to get the phi and theta outputs through statistical inference. It starts off assigning random topics to each word in every document. The number of topics it will infer is a user-entered argument. It then samples each word by calculating the conditional probability that each word exists in every topic. After each word is sampled one by one, the algorithm repeats this process a predefined number of times.

Once the LDA algorithm outputs the phi and theta matrices, we now have the topics, or features, of the documents. What's left for us to do in this kind of Recommender system is to find a way to measure the similarity between two documents that have different topic distributions. There are many ways to do this. One approach is the Euclidean distance where we get the difference between two points in a plane. Another is to use cosine similarity which simply gets the cosine of the angle between the two points. One of the most used approaches in finding document similarity in LDA is through the use of the Jensen-Shannon Distance, a smoothed version of the Kullback-Leibler Divergence.

The Kullback-Leibler Divergence (KL Divergence) measures how different one probability distribution is from another [9]. This similarity measure is not symmetric, meaning if you compare probability distribution P from probability distribution Q, the score is different if you compare Q to P. If the score for the divergence comes to 0, that means both probabilities are identical. The Jensen-Shannon Distance (JS Distance) remedies the asymmetry that the KL Divergence has and smooths it out to create a similarity metric that is symmetric [10]. Therefore, the distance between P and Q, and Q and P are identical. The nearer the distance score is to zero, the more similar the two probability distributions are. This is what we use to find documents that have similar

topics, by getting the JS Distance of one document's topic distribution and another document's topic distribution.

Once the topic modeling algorithm has finished, there needs to be a metric to assess the accuracy of the topic model. A way to do this is to evaluate the coherence of each topic. This is called Topic Coherence. It is a measure that scores the individual topics by assessing the semantic similarity between a number of high-scoring words in a topic [11]. This metric tells us exactly how identifiable a topic that LDA generates is. Since LDA does not give topic names, we have to assess if the topics that LDA inferred are identifiable. There are many coherence measures like UCI, Normalized Pointwise Mutual Information (NPMI), and UMass.

The UMass coherence measure bases its score on two words' document co occurrence counts [11]. Other coherence measures like UCI need an external corpus to compare to which makes it an extrinsic coherence measure. UMass only relies on the corpus that the model itself is trained on which makes it an intrinsic coherence measure. It takes the top 5, 10, 15, or 20 words in each topic and evaluates each lower-scoring word in that topic to every other higher-scoring word. Word 10, for example, must be compared to words 1 to 9, while word 9 must be compared to words 1 to 8.

Collaborative filtering, the other method used in a Hybrid recommender system, finds recommendations by using the similarities between users and items [12]. This type of filtering system finds users that are similar to user A and recommends content to user A based on what the similar users are also interested in. For example, user A and user B are both interested in superhero movies and they have both similarly watched

Batman, the Iron Man Trilogy, and Spiderman. If user B has also seen Superman Returns and is interested in it, user A will also be recommended the same movie. An algorithm that helps implement Collaborative filtering is Matrix factorization.

Every time a user interacts with an item, either through viewing or liking an item, the system takes note of that and adds that to your interests. This then creates a user-item matrix that maps every user's rating to every item. Since not every user can rate every item, this matrix is usually sparse [13]. We then use matrix factorization which attempts to decompose the user-item matrix into two matrices with k number of features. These two matrices are a representation of the user features and the item features, where conducting matrix multiplication on the two matrices will result in a user-item matrix which is a combination of the original user-item matrix's original ratings plus new values on the previously unrated or unobserved items. This matrix can then be used as the basis for the prediction.

In order for us to make sure the collaborative filtering model is as accurate as possible; we utilize a loss function. A loss function measures how much error the model has through computing the distance between the current output of the algorithm and the expected output [14]. This value is usually called the "loss" or "error rate", and different algorithms and models use different loss functions. The lower the loss, the better the model. One example of a Loss function is Mean Squared Error.

Mean Squared error is a kind of loss function that measures the average of the squares of the errors. It tells you how close a set of points are to the regression line in a graph [15]. This is done by first subtracting the predicted value from the actual

value and then squaring the result. This will result in the squared error. Finally, the result is divided by the size of the dataset which would make it the mean.

To make sure that we get the lowest possible loss in our model, we will need to use an optimization method whose aim is to minimize the loss. Gradient Descent, one of the most popular optimization strategies in machine learning, does this by looking for the local minima of a differentiable function [16]. To do this, gradient descent gets the derivative of the loss function, which is its slope. In doing so, the model can take repeated steps in the opposite direction to descend through the function to finally reach the local minima, which would mean our loss would be lower.

REVIEW OF RELATED LITERATURE

The use of recommender systems has become more and more popular nowadays when it comes to solving modern-day problems. Each of these researches focuses on various problems to solve using different algorithms and outputs a recommendation based on the goal of the project.

In Joeran Beel, Stefan Langer, Marcel Genzmehr, and Andreas Numberger's work [21], a research paper recommender system called Docear uses content-based filtering methods that enables users to search, organize and create research articles. User data such as papers, references, annotations, etc. are managed in mind maps which are utilized for research paper recommendations. When a user requests for recommendations, the system forwards the request to Docear's Digital Library which creates a user model and returns ten research paper recommendations.

In Khalid Haruna, Maizatul Akmar Ismail, Damiasih Damiasih, Joko Sutopo, and Tutut Herawan's work [22], a collaborative approach is used to recommend research papers. Their initial approach is transforming all the recommending papers in the dataset into a paper-citation relations matrix, the rows being the recommending papers while the columns being the citations. Rating scores are mined between researchers and research papers based on these paper-citation relations which are then used to recommend the most appropriate research papers for the users.

In Bela Gipp, Joeran Beel, and Christian Hentschel's work [23], a hybrid research paper recommender system called Sciencestein is used by combining different concepts like citation analysis, author analysis, sources analysis, implicit ratings, explicit

ratings. Instead of the traditional way of just entering keywords, the users may provide up to six inputs such as text, references, authors, sources, ratings, and documents to receive recommendations for research papers.

In an article by Koren, Bell, and Volinsky [24], they mentioned that there are two main strategies which are used by Recommender Systems—Content-based Filtering and Collaborative Filtering. Content-based Filtering categorizes each user and item (e.g. a movie's genre) which will be used by the recommender system to create suggestions. However, categorizing users and items usually means there will have to be external input to identify their categories. Meanwhile, Collaborative Filtering takes into account past user behavior to create relationships between users and items. This means that since the user's actions are the input, there is no need to identify their categories.

One of the many Collaborative Filtering algorithms is the Matrix Factorization algorithm, a method popularized by Simon Funk [25] during the Netflix Prize Competition, which he shares his findings in his blogpost. He was given a user movie matrix dataset which contained a user's rating to a movie that they have rated. The task was to predict the ratings of movies that the user has rated or in other words—predict if the user would like it or not. They started with the assumption that a user's rating is the sum of preferences regarding the likeness of the movie. After making this assumption, their solution was to decompose into two matrices—where one matrix would represent an aspect of a movie, and the other how much a user likes those aspects. The two matrices will then be used to create predictions on unrated movies.

An LDA-based Book recommender system was created by Andrew McAllister, Ivana Naydenova, and Quang [26]. They used a content-based approach to Book recommendations using Latent Dirichlet Allocation, a topic modeling algorithm that finds topics in a corpus. They first made a topic model with the book dataset that they have and then used that topic model to recommend books by finding similar books to the requested one.

PROJECT OBJECTIVE

The study's aim is to create a recommender system that is able to help researchers have an easier time finding related literature. Specifically, the study aims to:

1. Scrape Articles from Open Journal Systems Sites for data retrieval
2. Perform data preprocessing and vectorization using Term Frequency - Inverse Document Frequency
3. Perform Content-Based Filtering using Latent Dirichlet Allocation
4. Perform User-Based Collaborative Filtering using Matrix Factorization
5. Perform Hybrid Recommendation by combining Content-Based Filtering and User-Based Collaborative Filtering
6. Validate accuracy of recommendations through Topic Coherence

Project Scope and Limitation

The recommender system's main function is to make the finding of related literature easier for researchers. Therefore the target audience of this project are those that are undertaking research studies and finding the related literature they need. Ideally, the system should be able to recommend research from every kind of field that exists. However, because of the constraints of having a very large corpus, the researchers have decided to focus only on Computer-related fields to include in our corpus. Thus the system is only able to properly recommend related literature in fields that are related to computer technology.

Articles that will be used for the study will only be articles that are taken from Open Journal Systems (OJS) since these systems allow open access to all the published research in their database. This is also to make sure that the articles recommended are accessible by all researchers. Thus, research articles that are hidden behind paywalls are excluded in the conduct of this research.

Since the researchers mainly use English as the primary language for research, it is natural that English be the primary language for the system as well. Therefore all the other non-english articles that we may encounter are not included in the corpus. Additionally, non-English articles must be processed in their own special ways considering that grammar is different for these languages. The researchers do not possess the expertise nor the basic knowledge to be able to process foreign language articles.

The system will make use of both Content-based Filtering and Collaborative Filtering. In order to get the content similarity between documents and research abstracts that are entered by the user, Topic modeling is employed, specifically, Latent Dirichlet Allocation. Topic modeling is an unsupervised algorithm therefore when the topic model generates topics, there is no way for the model to label these topics with actual names.

For Collaborative Filtering, the system makes use of Matrix Factorization. It is an algorithm that factorizes the user-item matrix into two rectangular matrices, which will learn the latent features present in the user-item matrix.

RESEARCH METHODOLOGY

This study primarily focuses on the ability for a system to recommend related research based on a given abstract as input. That related research will be processed within the system to find the most related articles.

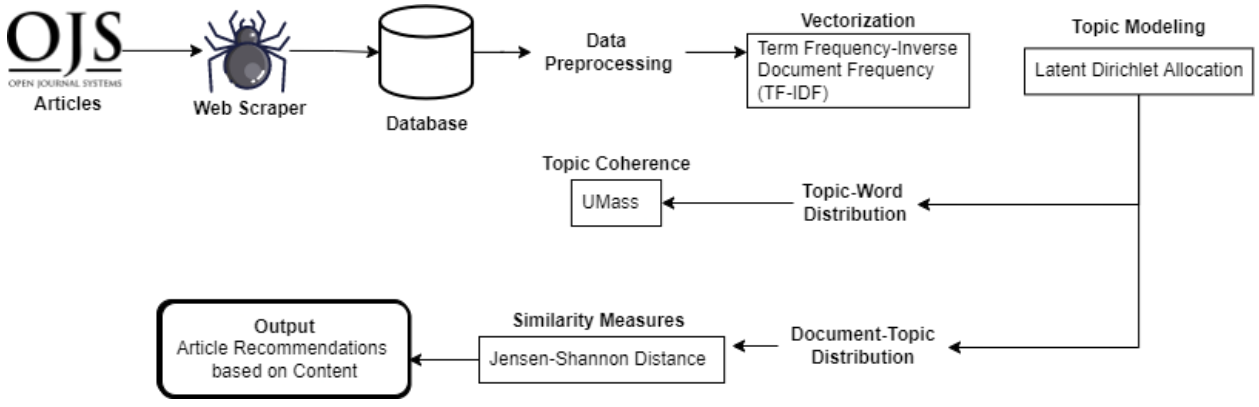


Figure 1: Conceptual Diagram

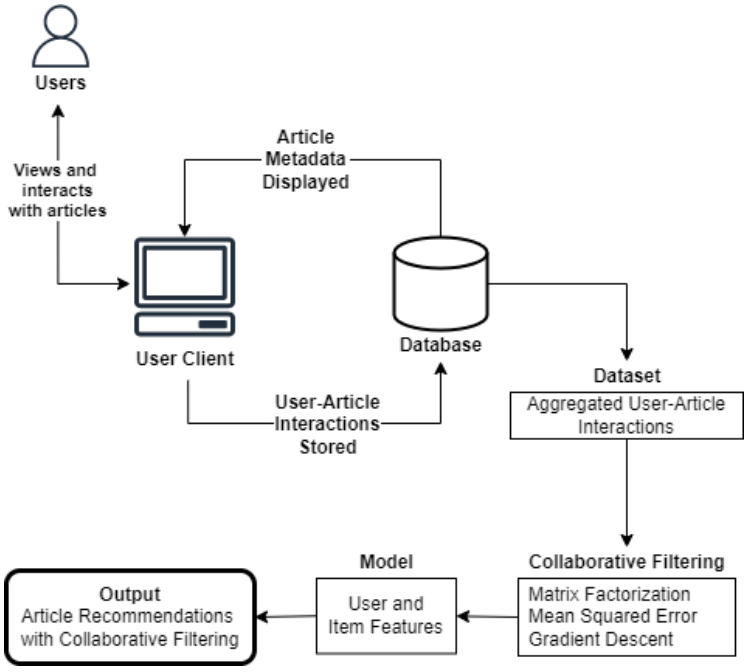


Figure 2. User-based Collaborative Filtering Output

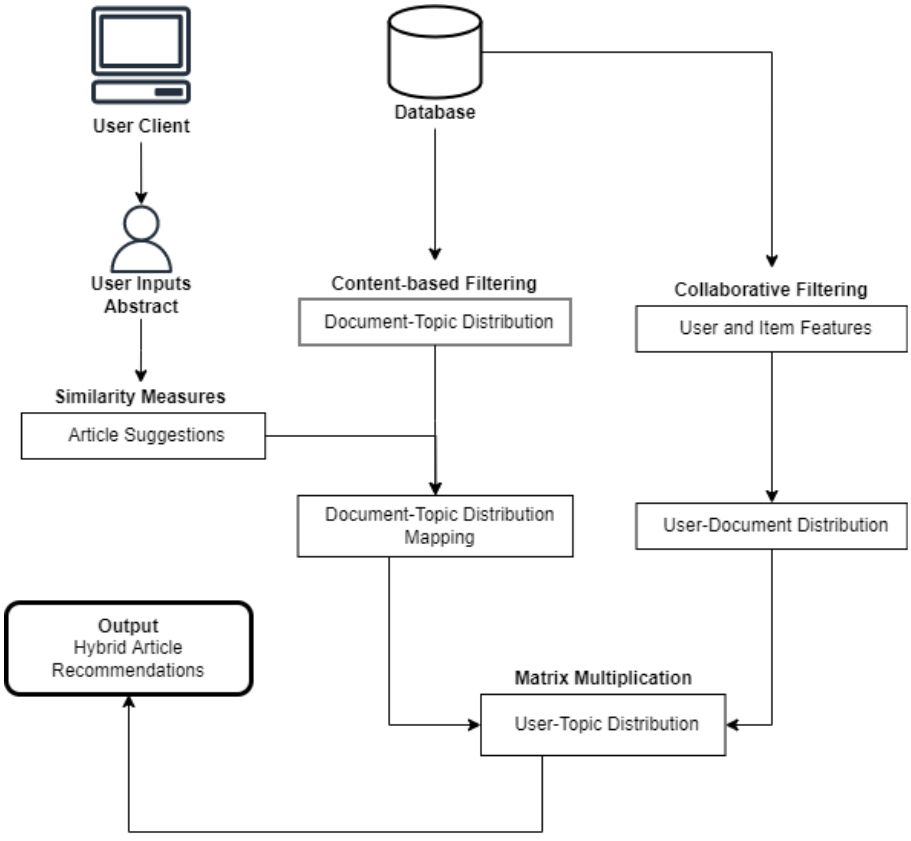


Figure 3 Hybrd

Figure 1 above shows the process to give similar article suggestions when opening an article in the user-client. The articles are first taken from an Open Journal Source using a web scraper. These are then stored into a database where all of the articles go through data preprocessing, and are then vectorized using TF-IDF. The vectorized articles are fed to a Topic Model using Latent Dirichlet Allocation, which produces two results: The topic-word distribution, and the document-topic distribution. The topic-word distribution is used to determine the topic coherence of the model, which ultimately shows the quality of the topic model. The document-topic distribution is used to get the similarity between each document based on the distribution. Once a user opens an article, the similarity measures between that article and all other articles are used to get the most similar articles, and then are suggested to the user.

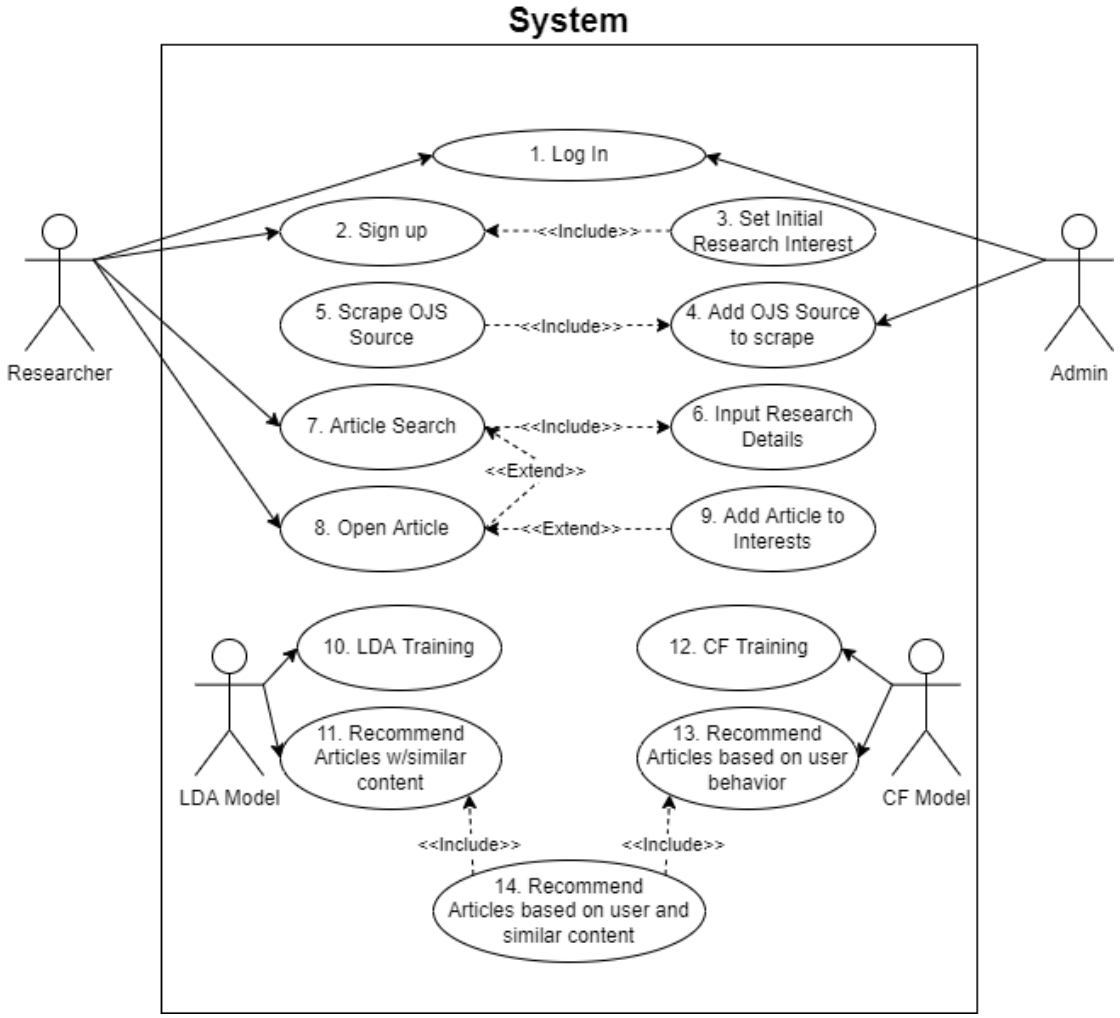
In the home page, the users are suggested articles that they are most likely to view based on their interests. The second figure illustrates how the system is able to generate these suggestions. Everytime a user views or interacts with an article in the user client, it is stored in the database. The data collected here are then used as an aggregated user-article interactions matrix which is fed to the collaborative filtering model. The model uses Matrix factorization with Mean Squared Error as its cost function, and Gradient Descent as its optimization function. The output of the model is a user and item feature which will be used to recommend the articles in the user's home page.

In figure 3, the process for hybrid recommendation is laid out. First the user inputs their abstract in the user client. The abstract is fed to the Content-based filtering model to get it's similarity measures with all the documents in the database. While this is happening, the user and item features are also taken from the database. These two tasks produce a document-topic distribution mapping, and a user-document distribution respectively. These are both used to get the user-topic distribution where Matrix multiplication is utilized. The user-topic distribution is what is then going to be used as the suggestions to the abstract the user has entered.

CHAPTER II

SOFTWARE REQUIREMENTS AND SPECIFICATION

Use Case Diagram



This diagram visualizes how the different users interact with the ArtiCrawl system. A researcher should be able to log in to their user accounts in order to access the application itself, while the Administrator should also be able to login in order to manage the system. An administrator, once logged in, has access to the feature where

Open Journal System (OJS) Sources can be added for the system to start scraping. This will include the use case to scrape articles from these OJS sources.

Once a researcher is logged in, they will be able to search for research articles with the same topic as the research details that they will be entering. The encoding of the research details is a use case included in the Article Search use case. This Article Search use case can extend to a use case where a user opens the source of an article that is suggested to them. Finally, a researcher can manually select research topics of interest to add to their profile so that the recommender system can use this as additional information to make the recommendations better.

The recommender model needs to undergo training whenever data is added to the corpus since the recommender system uses Latent Dirichlet Allocation which makes use of the relationship of words and documents in a whole corpus. Once the model is trained, the model can give recommendations whenever a user asks for suggestions.

Use Case Narrative

The Use Case Narratives of ArtiCrawl show the flow of how the users interact with the application to achieve a certain goal. The narrative provides different scenarios for each use case, it also shows how the system would respond to a certain action that the user does.

1. **Login Use Case Narrative**

This use case details how a user, both Researcher and Admin, are able to log in to their respective accounts. This use case is the first step to use ArtiCrawl since ArtiCrawl requires you to login to use its services.

Use case:	Login
Actors:	Researcher, Admin
Type:	Essential
Precondition:	The user must have an account
Postcondition:	The user will be able to access their account
FLOW OF EVENTS	
Actor Action	**System Response**
1. Enter user email and password.	2. Check the database if the account exists.
	3. Redirects to user's dashboard page

ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: User entered wrong email or password	ALT STEP 1b: The system prompts an error for wrong email or password
	ALT STEP 1c: Redirect to login page
ALT STEP 2a: User has no account and clicked “Sign Up”	ALT STEP 2b: Proceeds to Use Case Narrative 2: Sign Up
ALT STEP 3a: User clicks enter with empty fields	ALT STEP 3b: The system prompts an error for invalid fields.

2. **Sign Up Use Case Narrative**

The purpose of this use case is to allow users to create an account in the system. Doing this will let them log in to the system and be able to perform all the services offered depending on the account type.

Use case:	Sign Up
Actors:	Researcher, Admin
Type:	Essential

Precondition:	User must be on the registration page
Postcondition:	The user will be registered to the system.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Enter user email and password.	2. Checks the database if the account exists
	3. Information will be stored in the database
	4. Redirects user to complete profile information
ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: User clicks enter with empty fields	ALT STEP 1b: The system prompts an error for invalid fields.

3. Set Initial Research Interest Use Case Narrative

This use case sets users' initial research interests to their profile upon registration. This is to help the system recommend the appropriate research articles even if the user has no previous data yet. Research interest in the form of topics will be

used by the system to determine which research articles are closely related to the users'.

Use case:	Set Initial Research Interest
Actors:	Researcher
Type:	Essential
Precondition:	User must have finished Signing Up
Postcondition:	The user's interests will be added to the system
FLOW OF EVENTS	
Actor Action	**System Response**
1. User enters keywords of their topics of interest	2. Find which topics hold these keywords
	3. Add these topics as interests for the user
	4. Use data as the basis to recommend articles on these topics for the user

4. **Add OJS Source to Scrape Use Case Narrative**

The purpose of this use case is to add an OJS Source that will, later on, be scraped by the scraper. The scraped articles will be the documents to be used by the model for training later on.

Use case:	Add OJS Source to Scrape
Actors:	Admin
Type:	Essential
Precondition:	The user must be logged in.
Postcondition:	OJS Sources added to list of supported sites.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Admin inputs the OJS Website URL.	2. Checks the database if the OJS Website already exists.
	3. The OJS Website is added to the list of

	supported websites.
ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: Admin inputted source already exists.	ALT STEP 1b: The system prompts that the source exists and asks the user to input again.
ALT STEP 2a: Inputted field is empty.	ALT STEP2b: The system prompts the user that the field is blank.

5. **Scrape OJS Source Use Case Narrative**

This use case is for when an administrator wants to scrape articles from new sources not yet present in ArtiCrawl to expand the corpus. A prerequisite to this use case is to have a source added first in order to start Scraping, and this source must not be part of the current list of OJS sources in the ArtiCrawl repository.

Use case:	Scrape OJS Source
Actors:	Admin
Type:	Essential
Precondition:	There must be a source added in the Add OJS

	Source Use Case.
Postcondition:	Scraped articles will be added to the corpus.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Press button to start scraping.	2. Checks if there are added OJS sources to scrape.
	3. Starts scraping articles from specified OJS sources .
ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: No OJS source is added.	ALT STEP 1b: The system prompts an error that it can't scrape without sources being specified.
	ALT STEP 1c: Redirects you to add an OJS source.
ALT STEP 2a: OJS source added is already part of the ArtiCrawl	ALT STEP 2b: The system opens a prompt saying that the source is already part of the

repository.	repository.

6. Input Research Details Use Case Narrative

This use case will allow the users to input text (specifically the abstract of a research) in an input field. It will then make the “Get ArtiCrawl” button active and be clickable for the next use case to be performed.

Use case:	Input Research Details
Actors:	Researcher
Type:	Essential
Precondition:	The user must be logged in and have the details ready.
Postcondition:	The “Get ArtiCrawl” button will be clickable.
FLOW OF EVENTS	
Actor Action	**System Response**
1. The user inputs the abstract of the	2. The inputted abstract will be displayed in

research.	the input field.

7. **Article Search Use Case Narrative**

The purpose of this use case is for the user to get articles that have similar topics to what they have previously inputted. The research details that they provide will be utilized by the model to retrieve these similar articles.

Use case:	Article Search
Actors:	Researcher
Type:	Essential
Precondition:	The user must be logged in and finish Input Research Details.
Postcondition:	Articles with similar topics will be given.
FLOW OF EVENTS	
Actor Action	**System Response**

1. Click the “Get ArtiCrawl” button.	2. Request for a hybrid recommendation based on given abstract
	3. Display the returned suggested articles of the hybrid recommendation.
ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: User clicks the button with empty fields.	ALT STEP 1b: The system prompts an error for invalid fields.

8. Open Article Use Case Narrative

This details the use case on opening suggested articles. After the model recommends the articles of a similar topic to the user’s input, the user is able to click on the link provided on the suggested article and read the full text of that article on the given link.

Use case:	Open Article
Actors:	Research
Type:	Essential

Precondition:	The user must have searched for articles.
Postcondition:	The user will be redirected to a page where the full article can be found.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Click on the link provided in the article suggestions.	2. Opens a webpage where the full article can be found.

9. **Add Article to Interests Use Case**

This use case allows users to add a certain article to their list of interests. This helps the system gauge which articles a user believes is very interesting and helps the system recommend articles to the users better.

Use case:	Add Article to Interests
Actors:	Researcher
Type:	Essential

Precondition:	The user must have opened an article
Postcondition:	The article will be added to the user's interests
FLOW OF EVENTS	
Actor Action	**System Response**
1. User clicks the button to add the article to interests	2. The system will add the article to the user's interests
ALTERNATIVE FLOW OF EVENTS	
ALT STEP 1a: User clicks button but article is already part of interests	ALT STEP 1b: The system removes this article from the user's interests

10. LDA Training Use Case Narrative

This figure demonstrates how the training process happens where the model first vectorizes the documents and then uses LDA to get the topics within the corpus. The model is then a distribution of topics over documents and a distribution of words over topics that the model can use to recommend articles to the users.

Use case:	Training
Actors:	LDA Model
Type:	Essential
Precondition:	There must be newly scraped articles in the corpus.
Postcondition:	Topics distribution and word distribution are found.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Actor starts the LDA Training process	2. Starts vectorization of the corpus using Term Frequency - Inverse Document Frequency (TF-IDF) and is stored to the database
	3. The document term matrix is used to get the topic distribution of each document in the corpus and the word distribution of each topic

11. Recommend Articles with similar content Case Narrative

The purpose of this use case is to give the user articles that have similar topics to what they have entered.

Use case:	Recommend Articles w/similar content
Actors:	LDA Model
Type:	Essential
Precondition:	The LDA model must be trained.
Postcondition:	Articles with similar topics will be given.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Actor Requests for recommendations based on content	2. Get the topic distribution of the abstract entered
	3. Compare the topic distribution of the entered abstract to the individual article's topic

	distribution of the whole corpus
	4. Get the list of articles that are most similar to the entered abstract
	5. Send list of most similar articles and display them to the user

12. CF Training Use Case Narrative

This figure demonstrates how the training process happens where the model factorizes the User-Article Matrix using Matrix Factorization. By doing so, the model learns the latent features of both the users and the articles, which the model can use to recommend articles to the users.

Use case:	Training
Actors:	CF Model
Type:	Essential
Precondition:	There must be scraped articles in the corpus.
Postcondition:	User and Item Features are found.

FLOW OF EVENTS	
Actor Action	**System Response**
1. Actor initiates the Collaborative Filtering model training.	2. User-Article Aggregated Matrix is loaded into the model from the database.
	3. The model factorizes the Aggregated Matrix into two matrices, obtaining the User and Item Features.
	4. The User and Item Features are stored into the server for later use.

13. Recommend Articles based on user behavior Case Narrative

The purpose of this use case is to give the user articles based on their past behavior—this includes viewing an article, showing interest in an article, and adding an article as a scholarly work.

Use case:	Recommend Articles based on user behavior
Actors:	CF Model

Type:	Essential
Precondition:	The CF model must be trained.
Postcondition:	Articles that the user may like will be given.
FLOW OF EVENTS	
Actor Action	**System Response**
1. Actor requests for article recommendations for a specific user.	2. The system retrieves the User and Item Features from the database. From there, a Matrix Product is between the specified user features and all the articles.
	3. The articles with the highest scores are returned as recommendations.

14. **Recommend Articles based on user behavior and similar content Case Narrative**

The purpose of this use case is to give the user article recommendations based on the combination of their past behavior and the similar articles from the given input of the user.

Use case:	Recommend Articles based on user behavior and similar content
Actors:	LDA & CF Model
Type:	Essential
Precondition:	Actors must be trained
Postcondition:	Hybrid article recommendations
FLOW OF EVENTS	
Actor Action	**System Response**
1. Actor requests for hybrid recommendations for a specific user.	2. The system retrieves the User-to-Article Matrix from CF Model
	3. The system retrieves the Article-to-Topic Matrix from LDA
	4. User-to-Topic will be generated by doing a matrix multiplication for the two matrices (2

	and 3).
	5. System will determine the top 3 topics and recommend the top articles from those particular topics.

Activity Diagram

Log In

This diagram shows the flow of what happens when a user logs in to the user client.

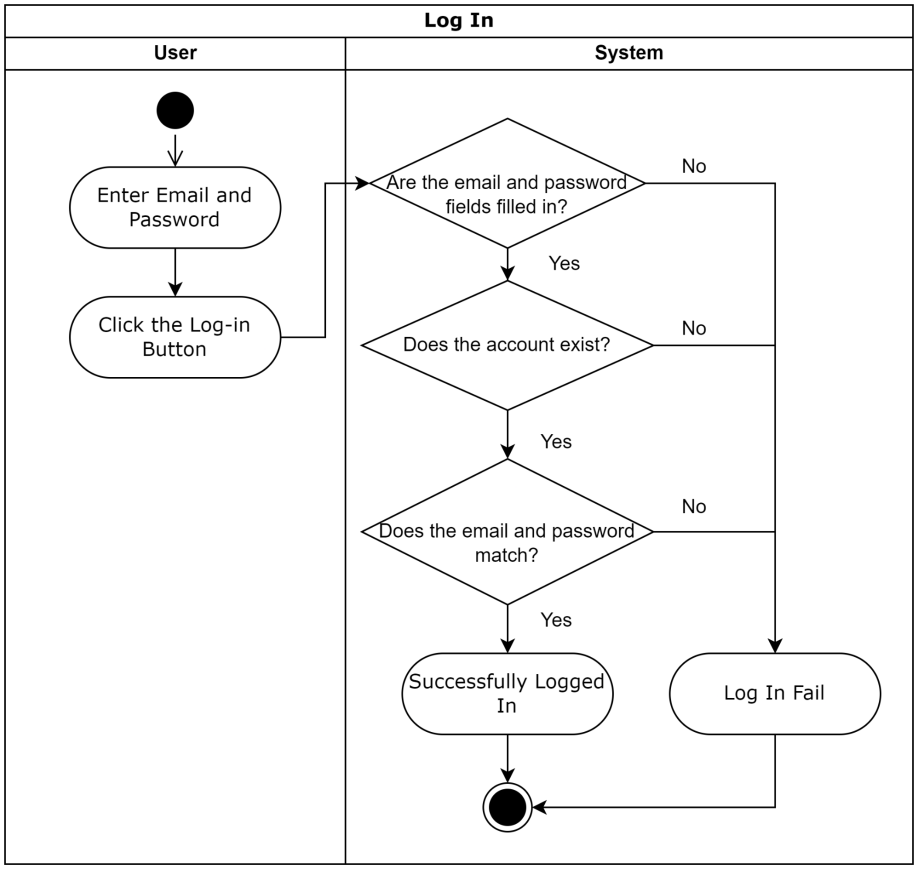


Figure x Log In Activity Diagram

Figure x shows us that once the user has clicked the Log-in button, the system first checks if the username and password fields are filled in. If yes, the system checks if the account exists in the database. If yes, it finally checks if the username and password match according to the database. Finally, after that final check, the user is

able to successfully log in. If either of those three questions came back no, then the login fails.

Sign Up

This diagram illustrates how the user and system interact when a user signs up for ArtiCrawl.

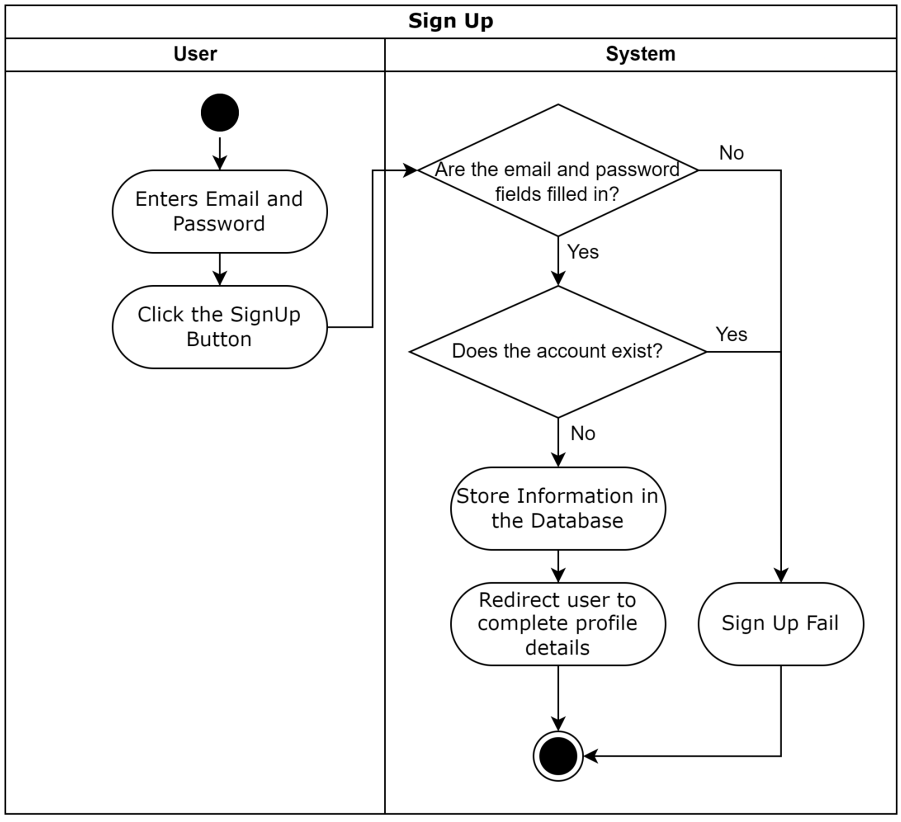


Figure x Sign Up Activity Diagram

The user enters their email and password in the given fields and then clicks the signup button. The system then checks first if the email and password fields are both

filled in. If not, the signup fails. If it is, the system then checks if the account exists. If it exists, the signup fails as the account is already part of the database. If not, the system stores the information in the database. After that, the system redirects the user to complete their profile details.

Set Initial Research Interest

This activity diagram shows how the user and system interact to add the initial research interests for every user after signup.

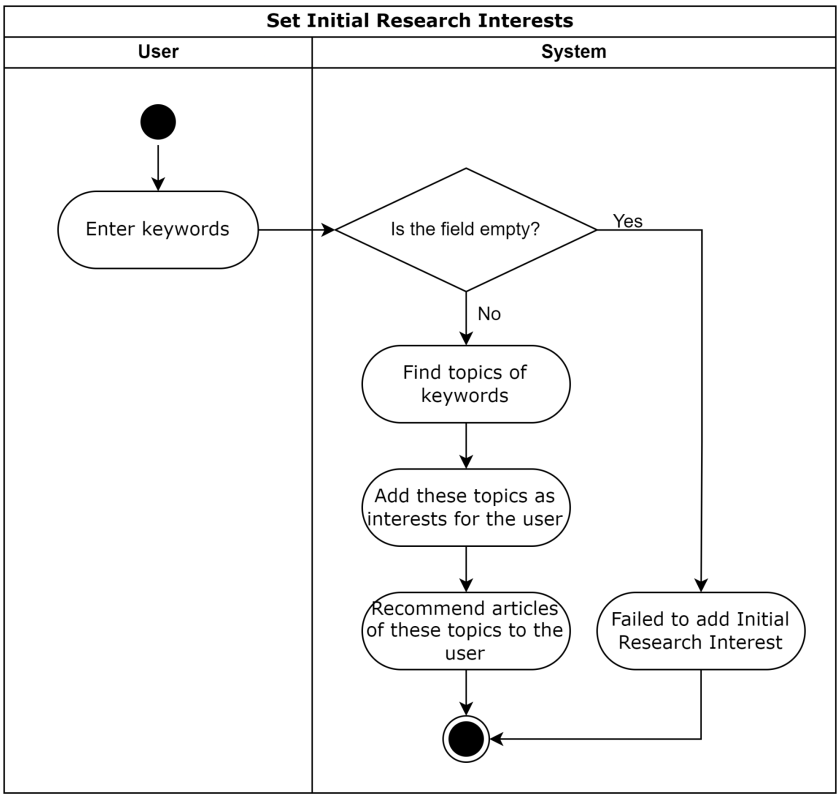


Figure x Set Initial Research Interests Activity Diagram

The user enters keywords of topics that they are interested in. After this, the system checks if the field entered is empty before finding the topics of the given keywords and then adding these topics as interests for the user. The system then recommends articles on these topics to the user in their dashboard. If the fields were empty, the activity fails.

Add OJS Source to Scrape

The diagram below illustrates how OJS websites are added to the system for them to be scraped later on.

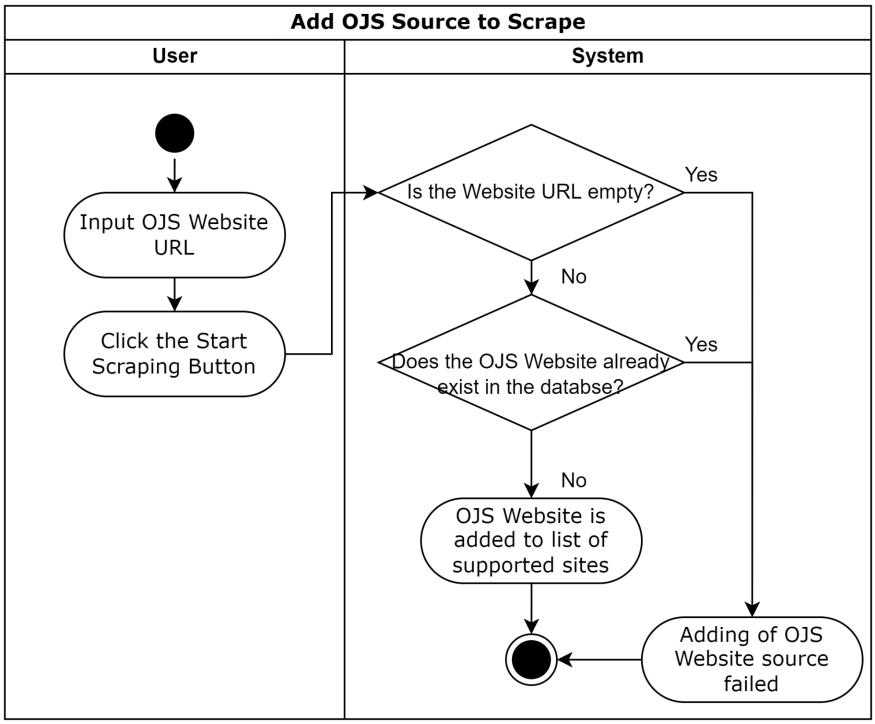


Figure X Add OJS Source to Scrape

The user first inputs the OJS website URL in the field and then clicks the start scraping button. The system first checks if the website is empty and then if the OJS website already exists in the system. If the answer to both of these is no, then the system adds this website to the list of supported sites. If the answer to either one of the questions is yes, then the adding of the OJS website source failed.

Scrape OJS Source

The diagram shows the process of how the system scrapes an OJS Source.

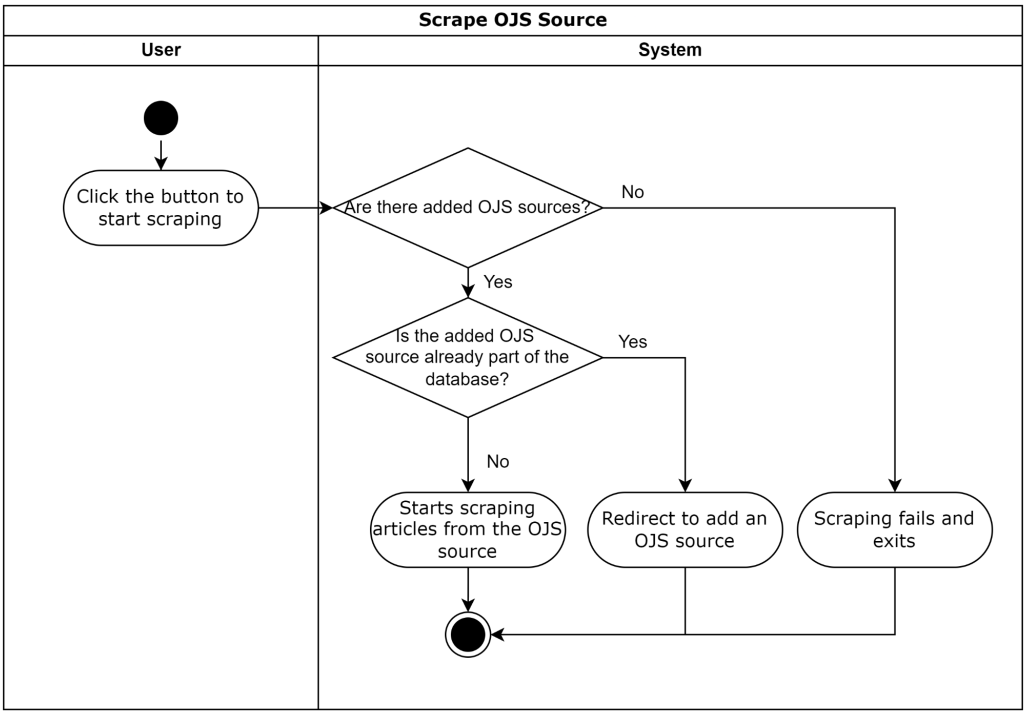


Figure X Scrape OJS Source Activity Diagram

The user starts the scraping process by clicking the button to start. The System checks if there are added OJS sources. If there are none, the scraping fails and exits.

Otherwise, the system then checks if the added OJS source is already part of the database. If yes, the system redirects the user to add an OJS Source. If not, the scraping process starts.

Input Research Details

This diagram details how a user is able to input their research details.

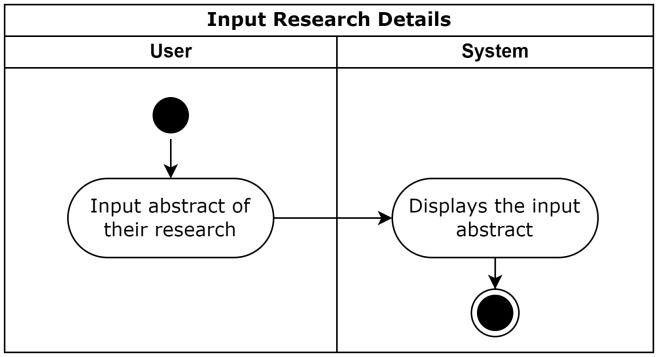


Figure X Input Research Details Activity Diagram

The user inputs their research abstract in the field, and the system displays the input abstract.

Article Search

This activity diagram shows how the user is able to search for articles given their entered research abstract.

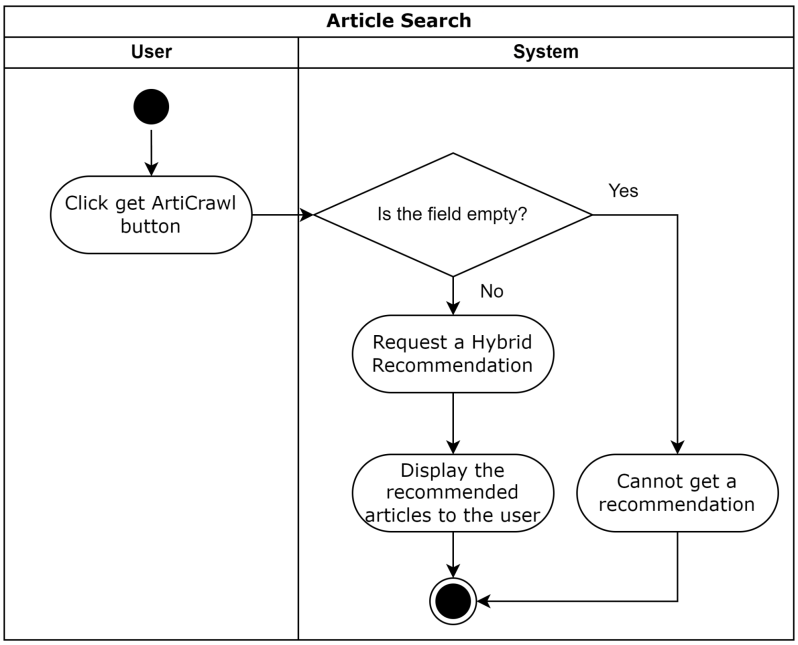


Figure X Article Search Activity Diagram

The user clicks the get ArtiCrawl button to start asking for articles. The system checks if the field is empty and if yes, the system cannot give a recommendation. If not, the system requests for a hybrid recommendation and then presents the recommendations to the user

Open Article

This activity diagram shows how a user opens an article for viewing.

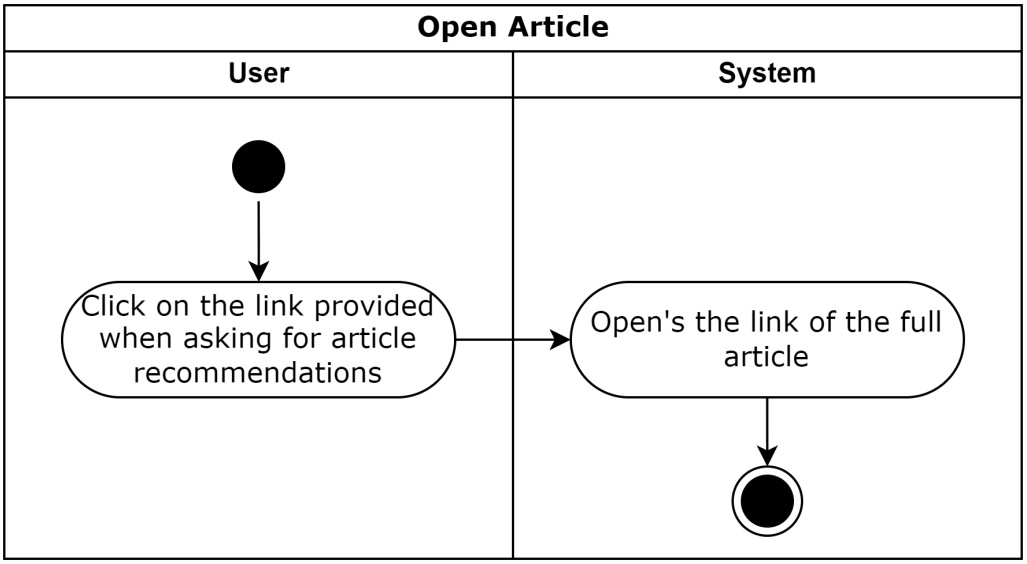
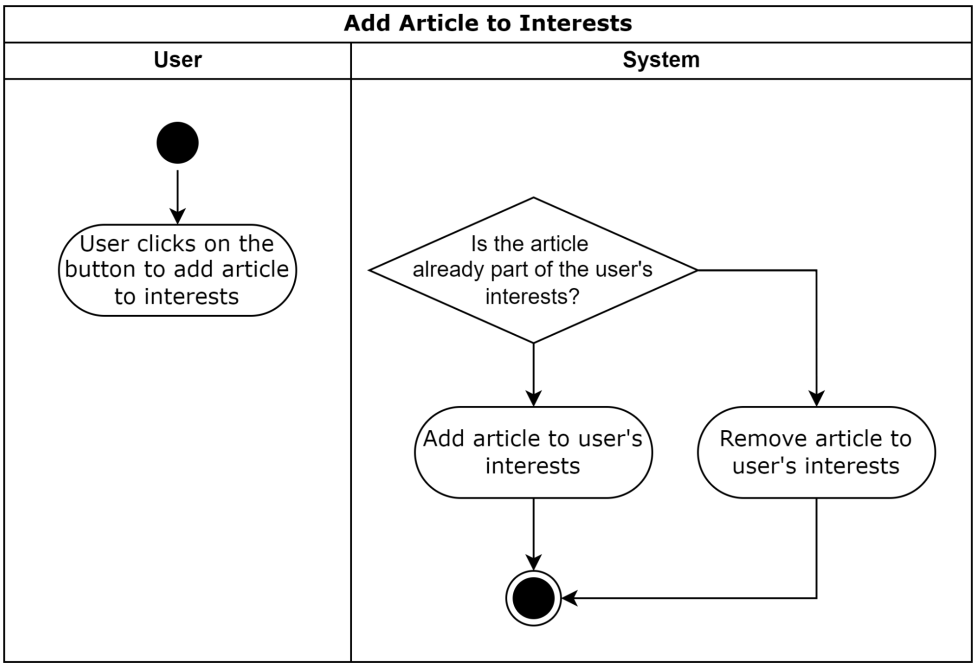


Figure X Open Article

The user will click on a link to an article—obtained from either the feed or via searching. After clicking on the link, the user will be led to the page containing the article's details.

Add Article to Interests

This activity diagram shows how users are able to add an article to their interests.



Once the user is on an article's page, there will be a button that will allow them to show their interest in their article. By clicking that button, the article will be to their interests and will be utilized by the Collaborative Filtering Recommender. In the case that the article is already in their interests, clicking the button again will remove it from their interests.

LDA Training

The diagram shows the process of how the system trains the LDA model.

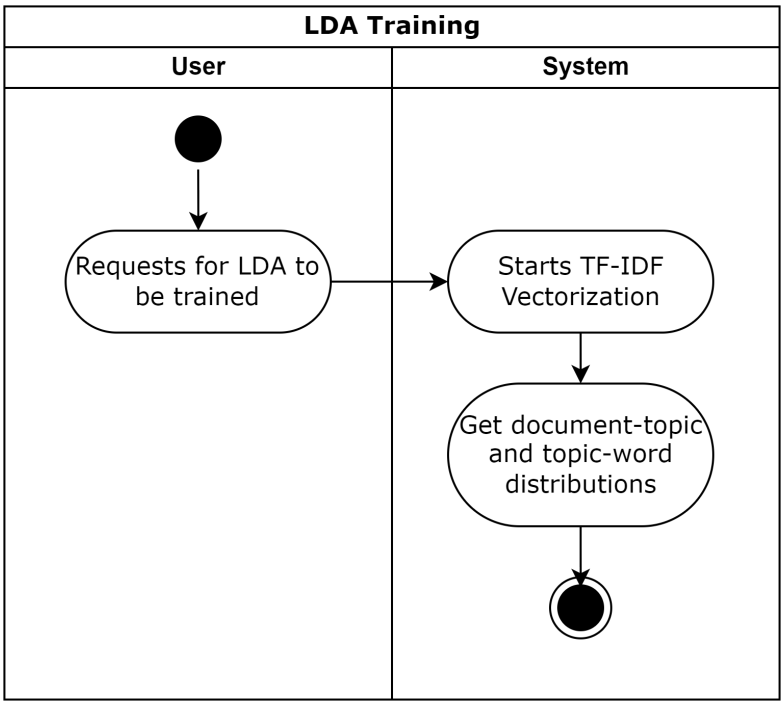


Figure X LDA Training

The LDA model must be trained before it can give recommendations. The user first creates a request to start the LDA training. From there, TF-IDF is conducted to vectorize the corpus. Next, the system conducts topic modeling to get the document-topic and topic-word distributions.

CF Training

The diagram shows the process of how the system trains the Collaborative Filtering Model.

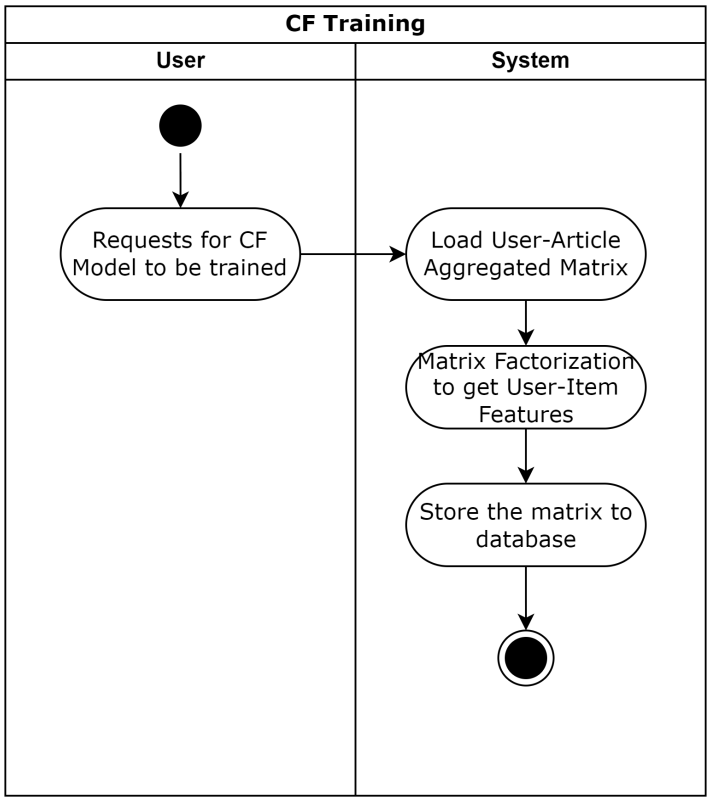


Figure X CF Training

Before any recommendations using Collaborative Filtering can be made, the model must be trained first. The user first issues the command to start the training process, and the system will load the user-article aggregated matrix, which is the model's dataset. The system will then conduct Matrix Factorization to learn the user-item features, and store the learned features later on so that it can be used to create recommendations.

Recommend Articles with Similar Content

The diagram shows the process of how the system …

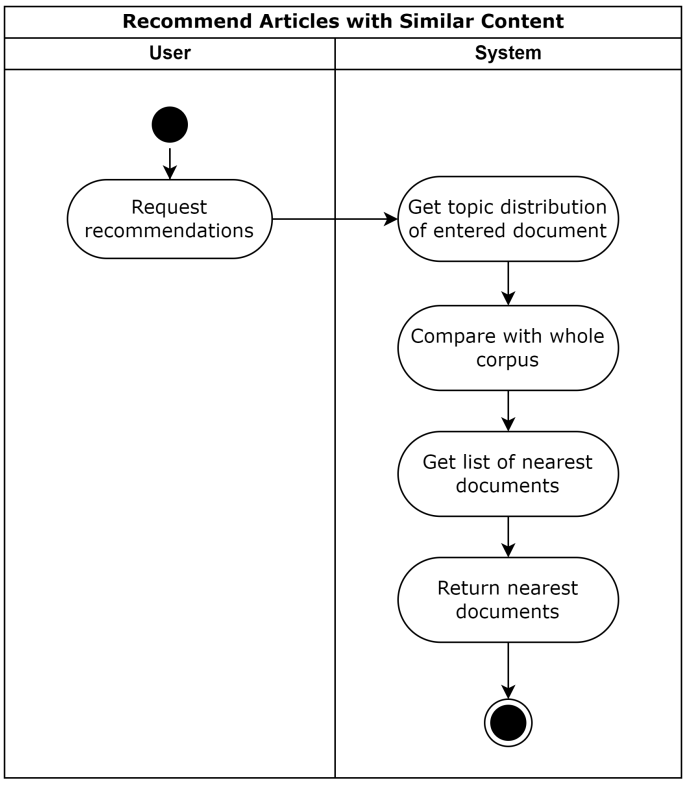


Figure X Recommend Articles with Similar Content

The system requests recommendations for

Recommend Articles Based on User Behavior

The diagram shows the process of how users are recommended articles based on past user behavior.

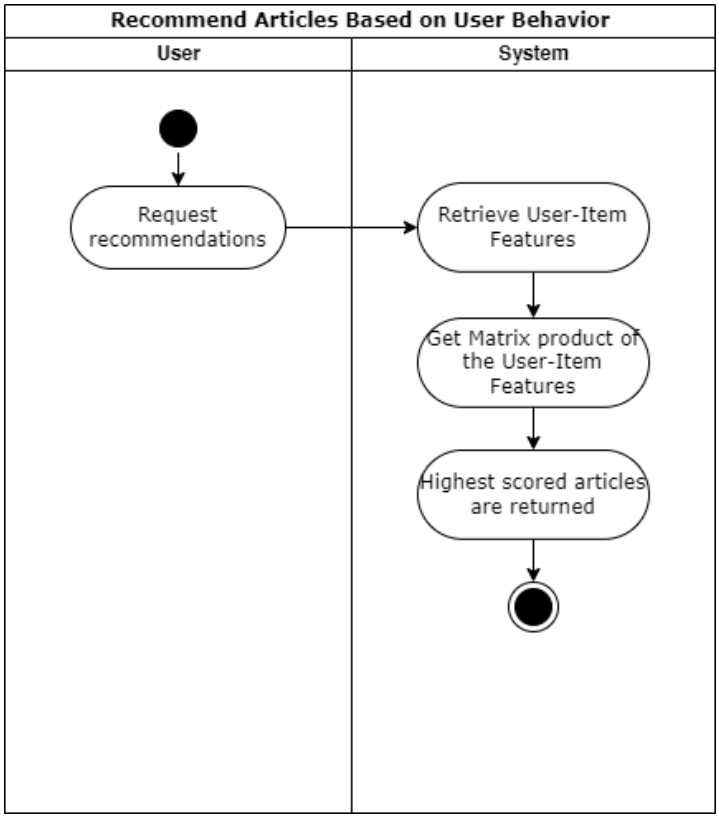


Figure X Recommend Articles Based on User Behavior

Once the user is on a page that displays recommendations, it will send a request to the system for article recommendations. Once the request is sent, the system will create recommendations by first retrieving the model, which contains the user-item features. It will then conduct Matrix Multiplication on the two matrices and return the articles with the highest scores.

Recommend Articles based on User Behavior and Similar Content

The diagram shows the process of how the user can request for article recommendations.

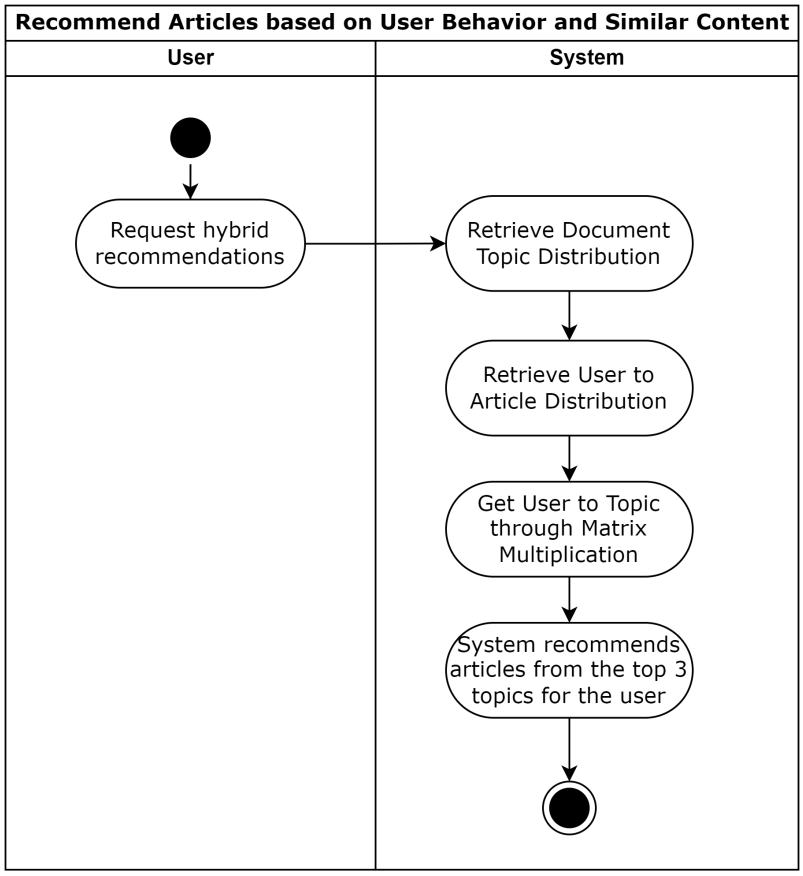


Figure X Recommend Articles based on User behavior and Similar Content

The user provides an input abstract in which they want to get recommendations from. Then, the system retrieves the user-to-article matrix from UBCF and article-to-topic matrix from LDA. The system applies matrix multiplication from the two matrices and outputs a user-to-topic matrix which will recommend articles from the top 3 topics.

Entity Relationship Diagram

The diagram below illustrates the database and it's relationships within each table. The model table holds details of the LDA model, and each model has 1 or more reference to the topic_coherence table. The doctopic and topicterm tables are representations of sparse matrices.

The vocabulary contains a list of all the tokens that are taken from all of the articles in the corpus. A pruned_vocabulary table then holds the tokens from the vocabulary that are part of the final vocabulary after pruning, or removing words that are deemed unnecessary.

Each article has one or more authors, and is being referenced in clean_abstracts, a table that holds filtered articles. Clean_abstract and pruned_vocabulary are both referenced by the doc_term_matrix.

Selectors have a reference to selector_types where each selector type can have many selectors. The vocabulary holds tokens from the corpus and the pruned_vocabulary only takes the tokens that have been filtered after pruning.

The user table holds user information and connects to the tables user_interested articles, user_viewed_articles, and use_scholarly_works. These three tables also reference the article table.

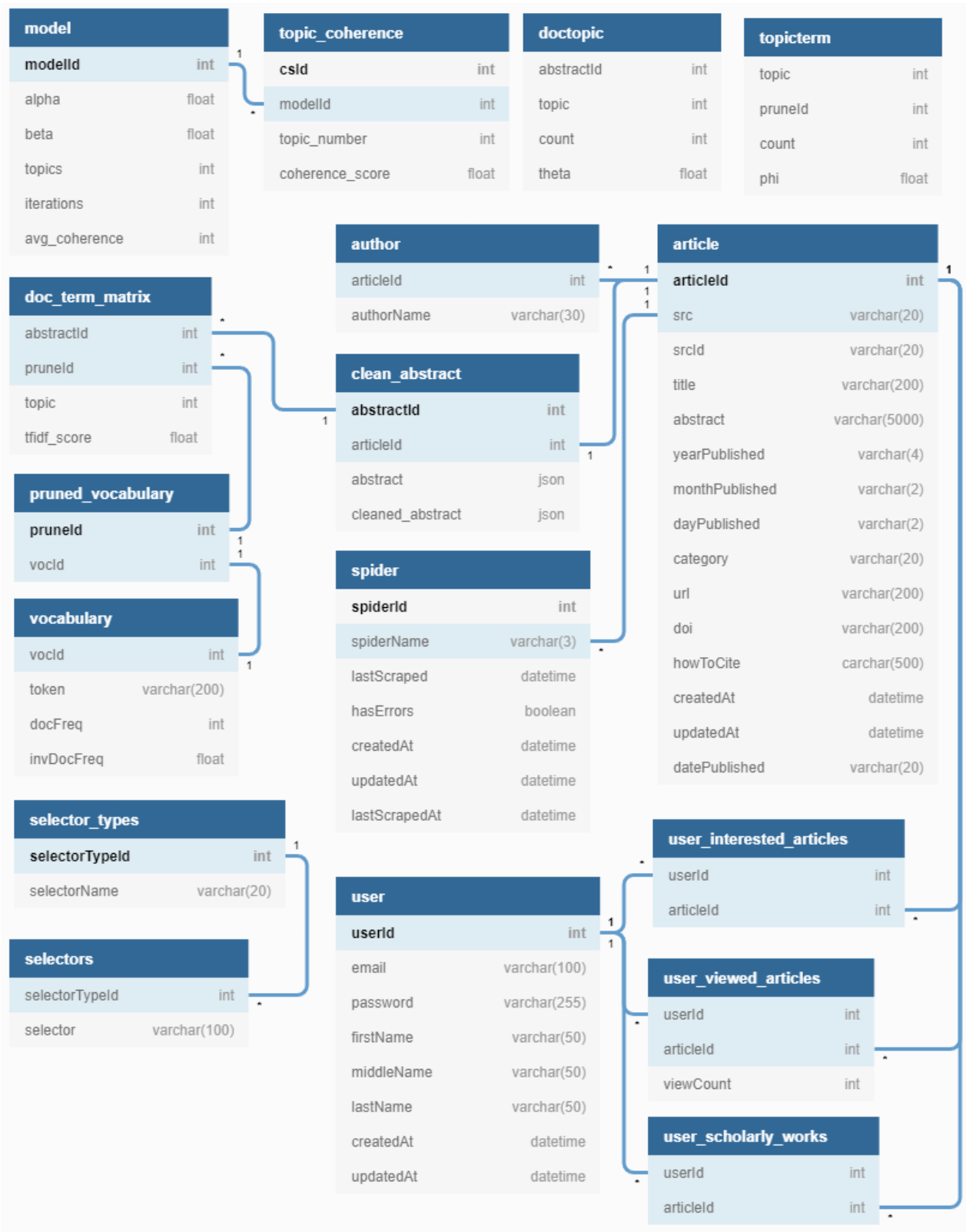
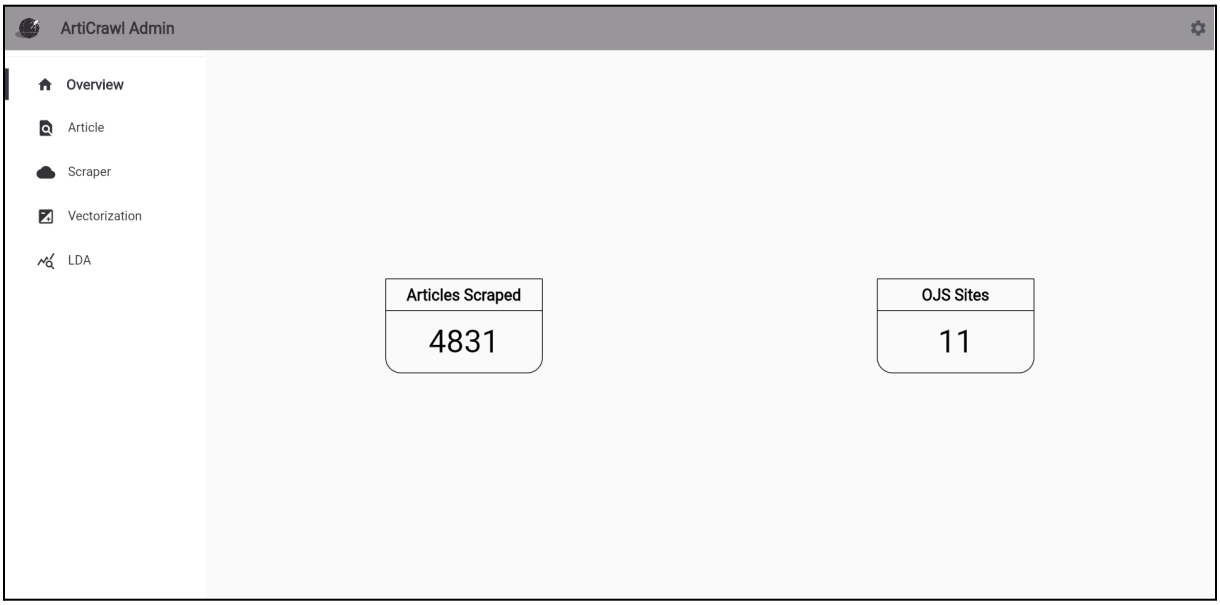


Figure X: ArtiCrawl Entity Relationship Diagram

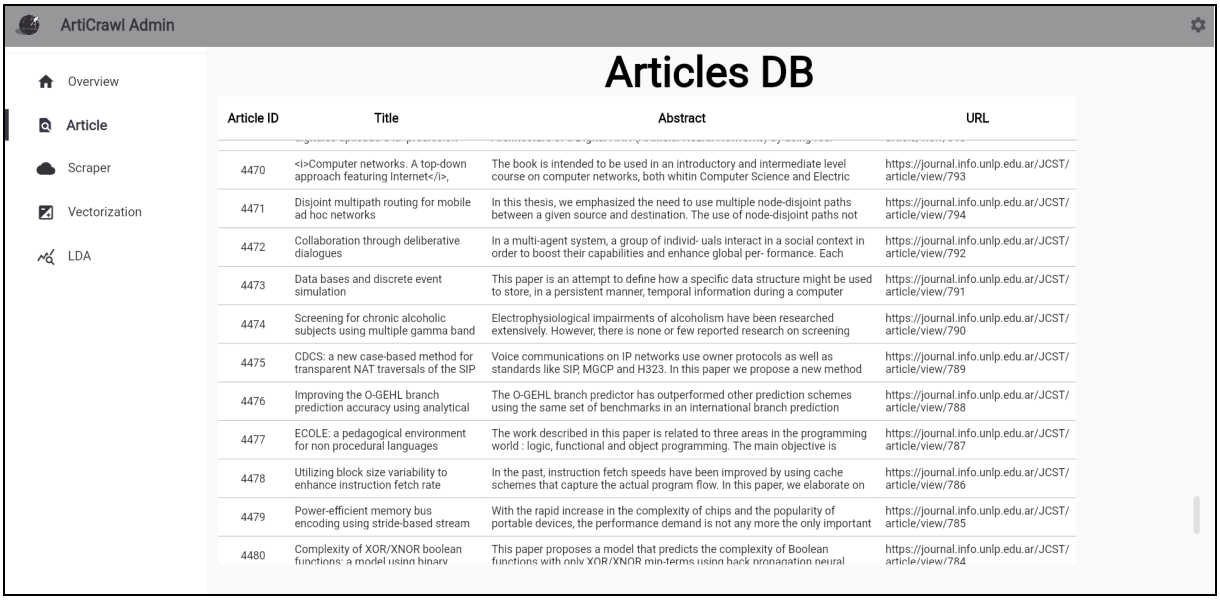
User Interface

Admin Client



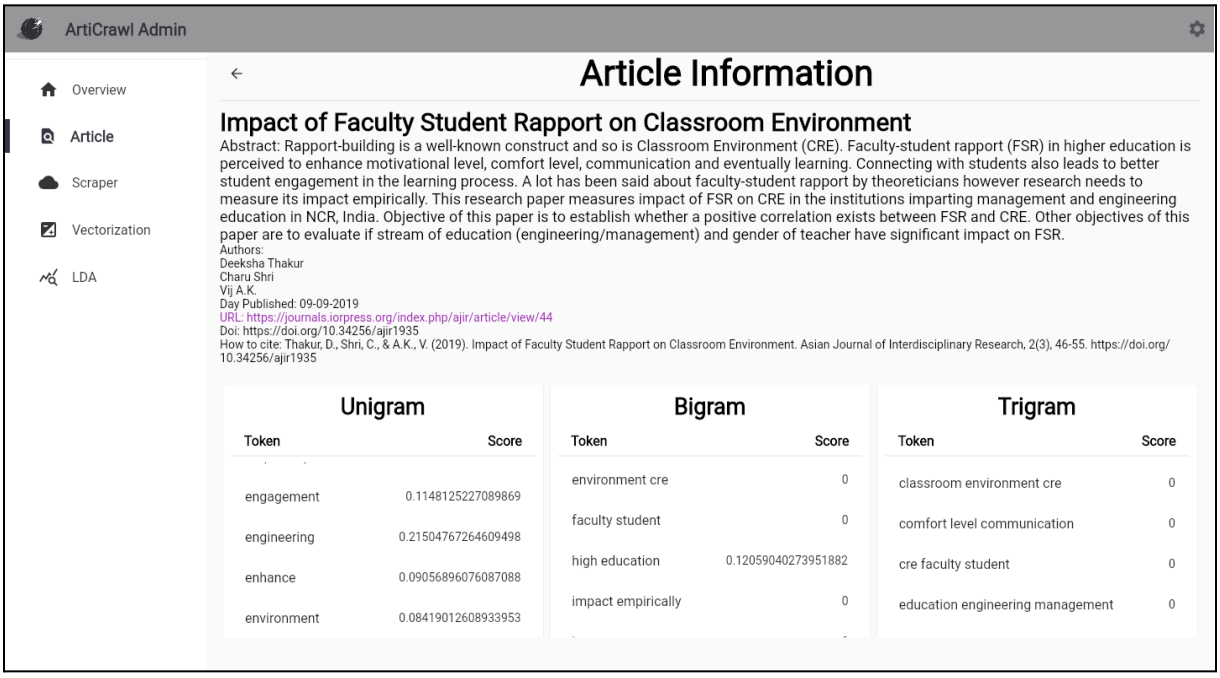
Overview

In this screen, you will see an overview of the overall information of the application. For now, what is displayed is only the number of articles that were scraped and the number of OJS Sites that are supported.



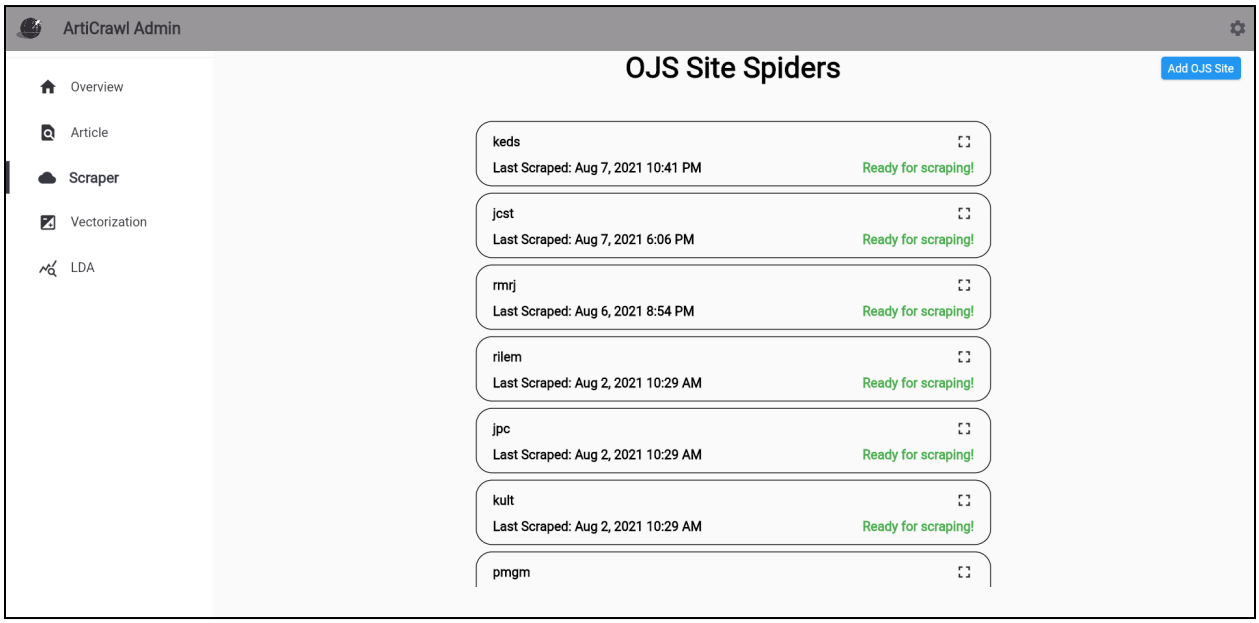
Articles - DataTable

Here, you can find the complete list of articles in the database. By clicking on an article, you can view the complete information of the article.



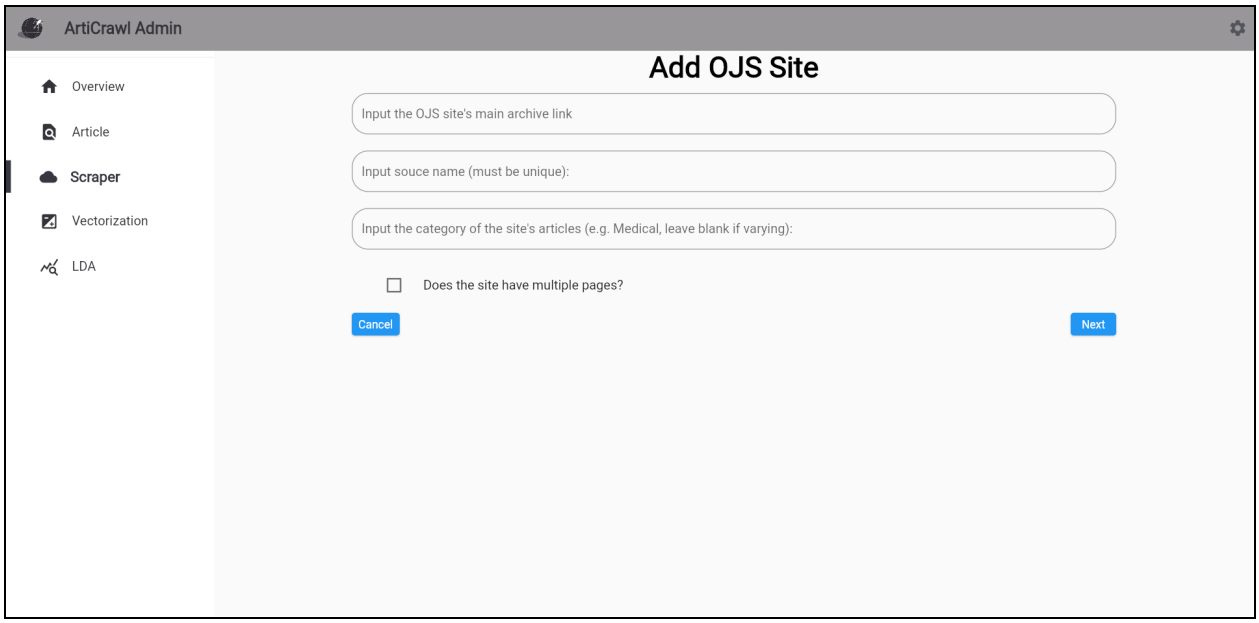
Articles - Article Information

This screen shows all the information regarding the article being viewed - its title, abstract, authors, ngrams, and more. Each token in the n-gram has its corresponding TF-IDF scores and 0 means no score because it is not included in the pruned vocabulary.



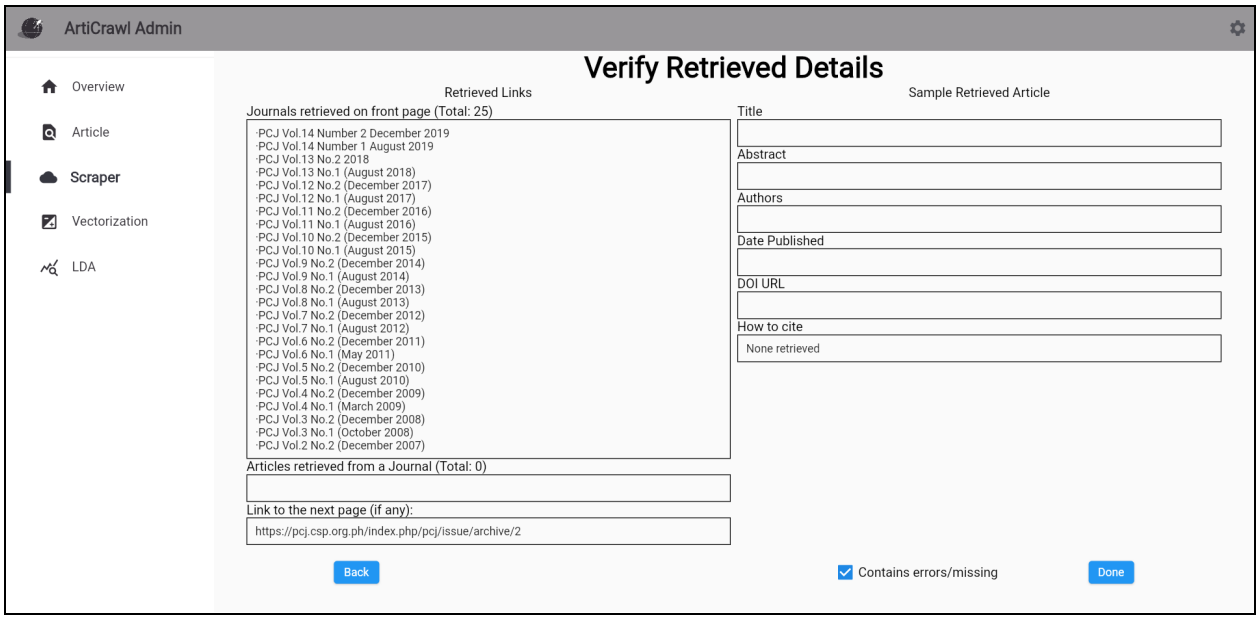
Scraper - Main

The main screen where you can see the different OJS Sources/Spiders, which are clickable and will send you to their screen. However, some that were noted to have errors or missing data will be inaccessible until fixed by the dev. Finally, there is also a button to add more OJS Sites, to be elaborated below.



Scraper - Add Archive

The user will input the OJS site's archive link, which is the URL of the site which contains the list of journals. Additionally, the source name, category, and a check mark that indicate if there are multiple pages should also be provided. Clicking on next will lead you to the verification page.

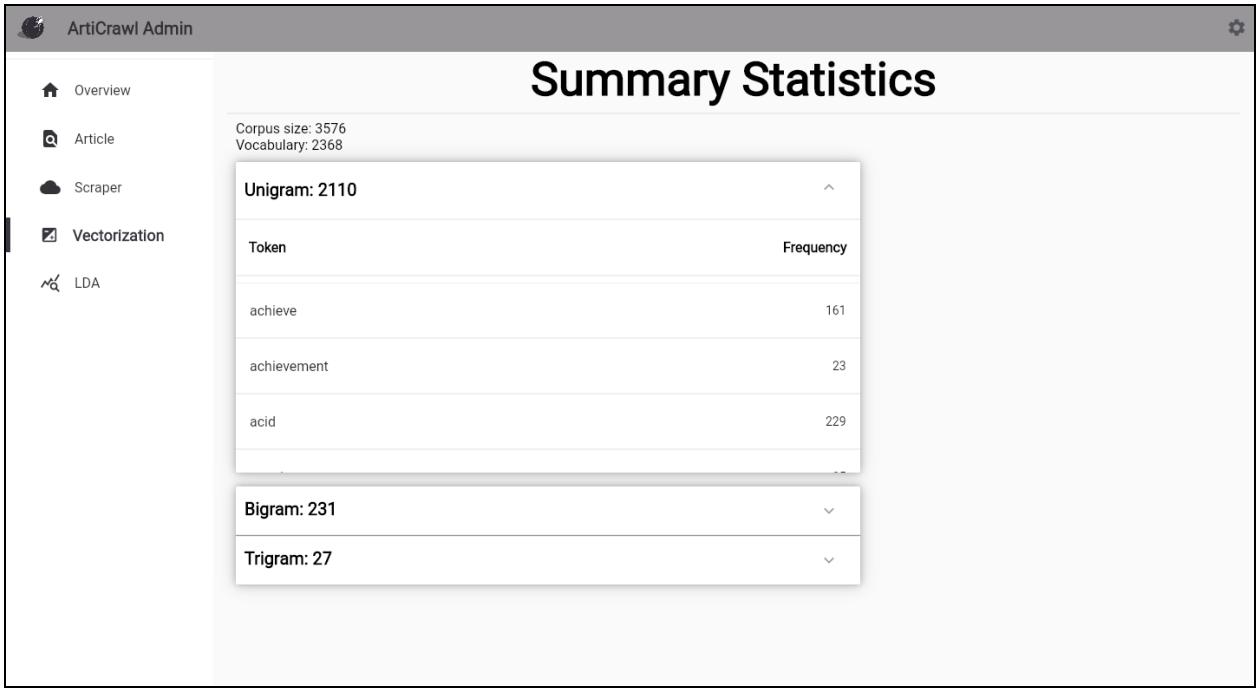


Scraper - Verify Retrieved Details

Here, the user can check if what the dynamic scraper has retrieved is correct or not--if there are any errors or missing values, the user can simply click on the ‘Contains errors/missing’ checkmark on the bottom. This will flag the OJS Site, making it unscrapable until the dev(s) fix it and deem it as ready for scraping.

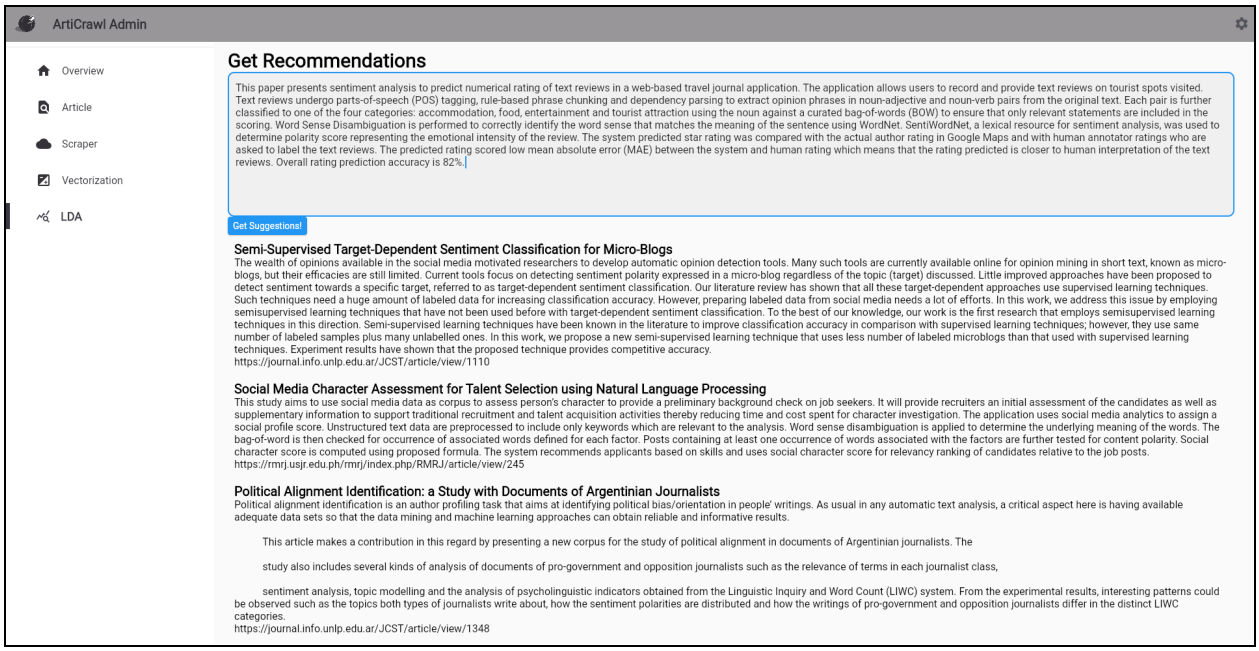
Scraper - Scraper Info

Here we can see the OJS Site Scraper's info, which includes the timestamp of the last scrape, as well as the recently scraped articles. In the upper right corner is a button that lets you start scraping--once clicked, it will turn into a button that lets you cancel the scraping.



Vectorization - Summary Statistics

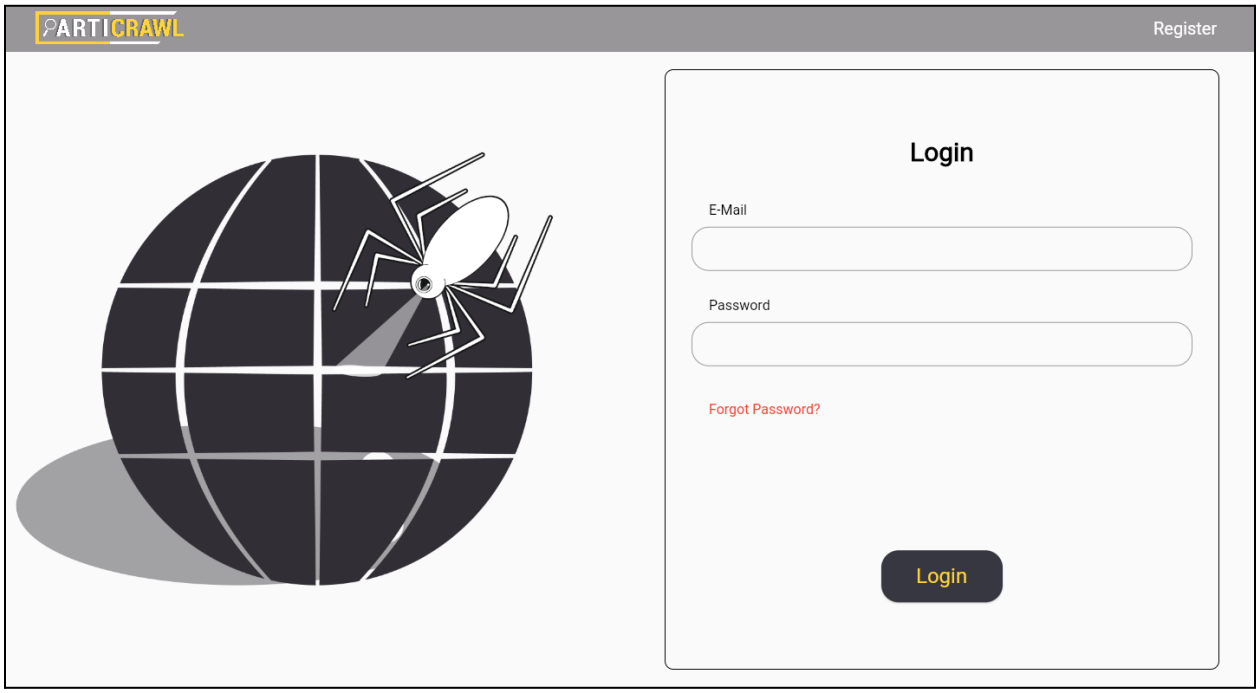
In this screen, all information being utilized during the vectorization process is found here. The **Corpus size** is the count of all non-empty English articles included in the vectorization process. The **Vocabulary** is the count of the pruned tokens having a document frequency greater than 0.5% of the corpus size. The **unigram, bigram,** and **trigram** are ngram subsets of the vocabulary. Clicking the dropdown will display all tokens belonging to the specific n-gram with their document frequency.



LDA - Get Recommendations

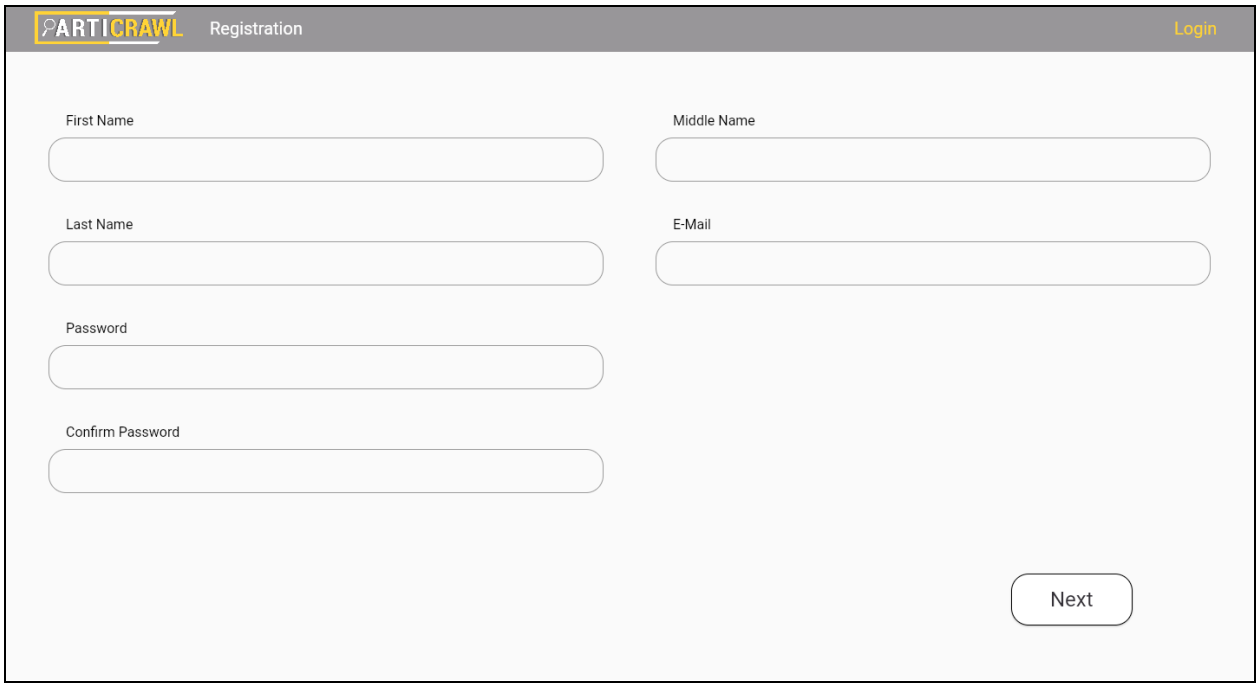
In this screen, we are able to ask for articles that the model can recommend to use given a certain abstract. The screen holds a text area where a user can input their abstract. Below that is a get suggestions button that you press once an abstract is given. Once that button is pressed, the LDA model processes the abstract in the text area and returns article suggestions. The returned article suggestions are displayed in a list view and are arranged in a way where the most similar article is presented first.

User Client



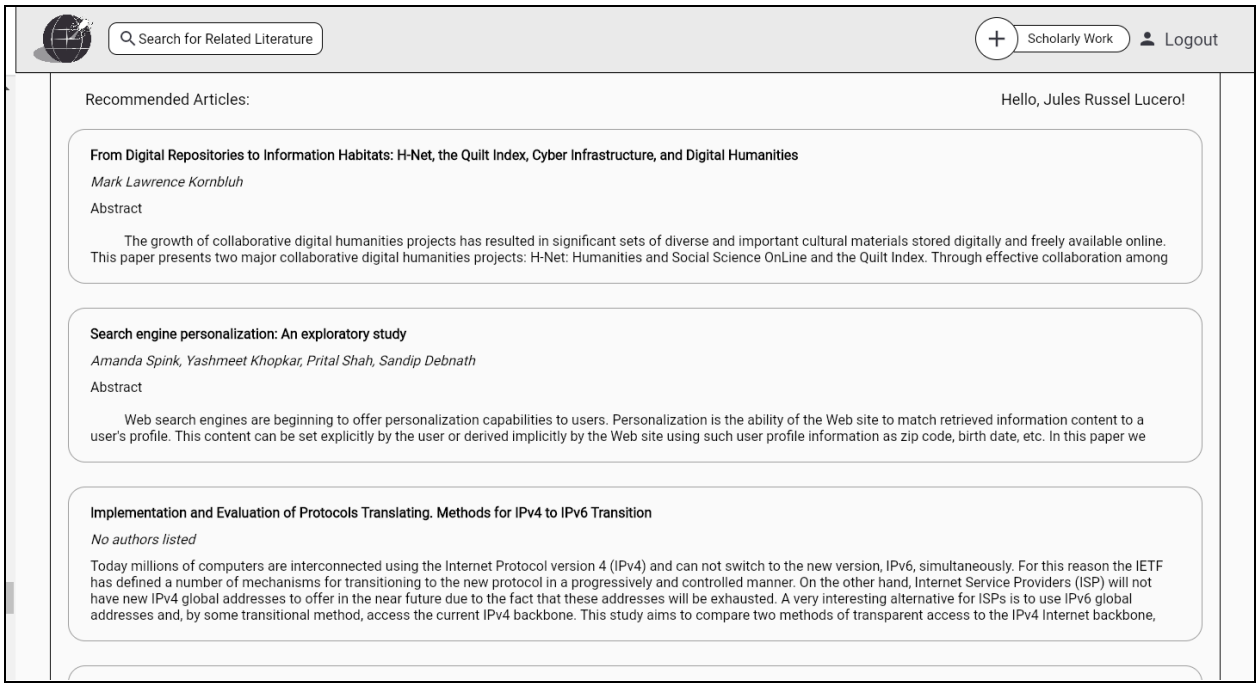
User - Login Screen

In this screen, users can login to the system by providing their registered email address and password. Users can also click on “Forgot Password?” if they forgot their password to make a new one.



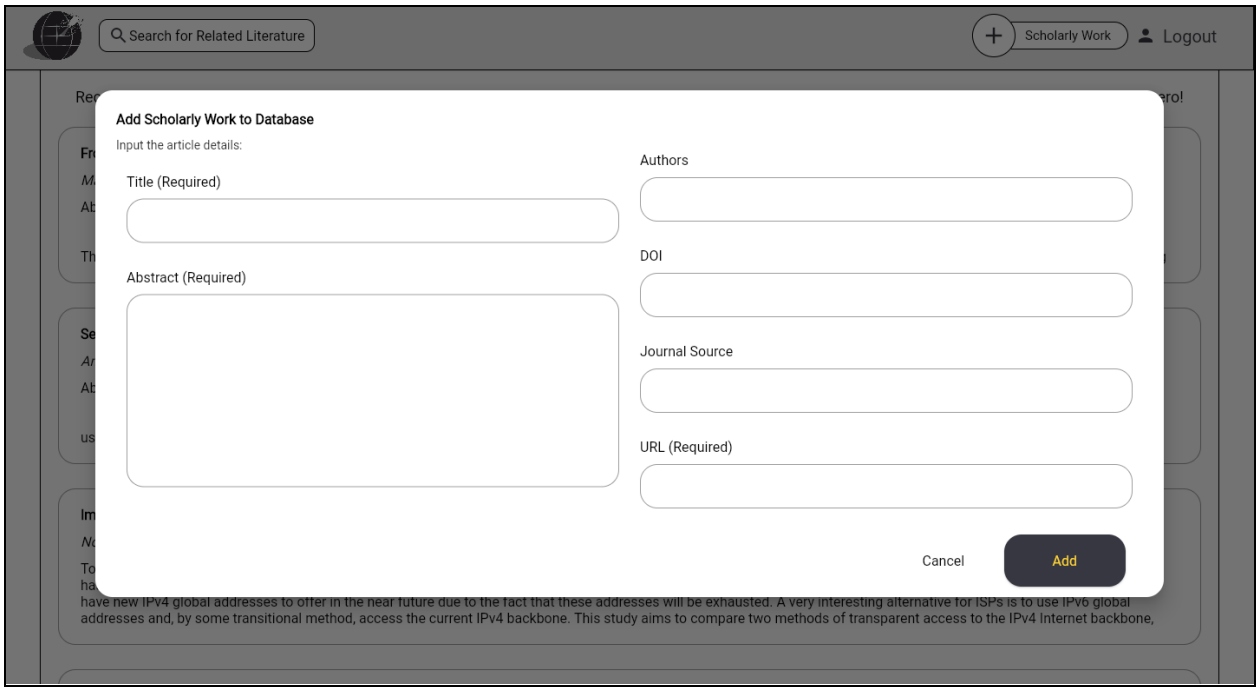
User - Register Screen

In this screen, users can make a new account by providing their first name, middle name, last name, email address, and password.



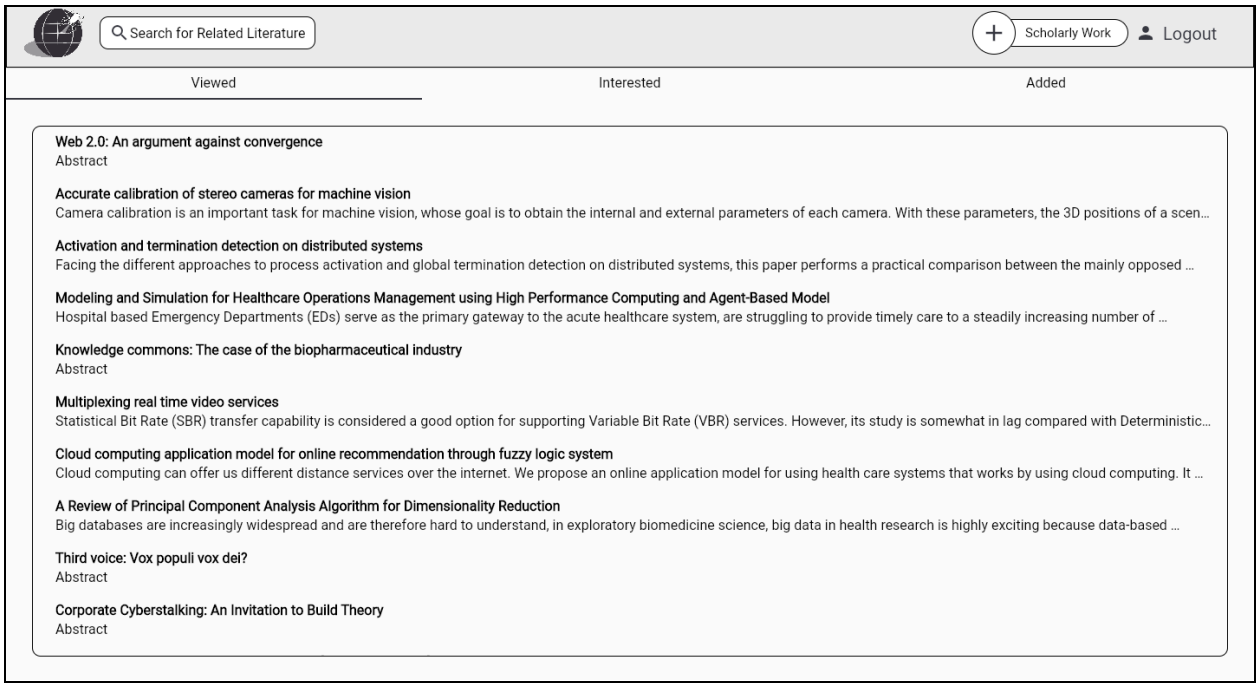
User - Homepage

This screen will be shown after logging in the system. In this screen, users can see the recommended articles provided by the User-based Filtering.



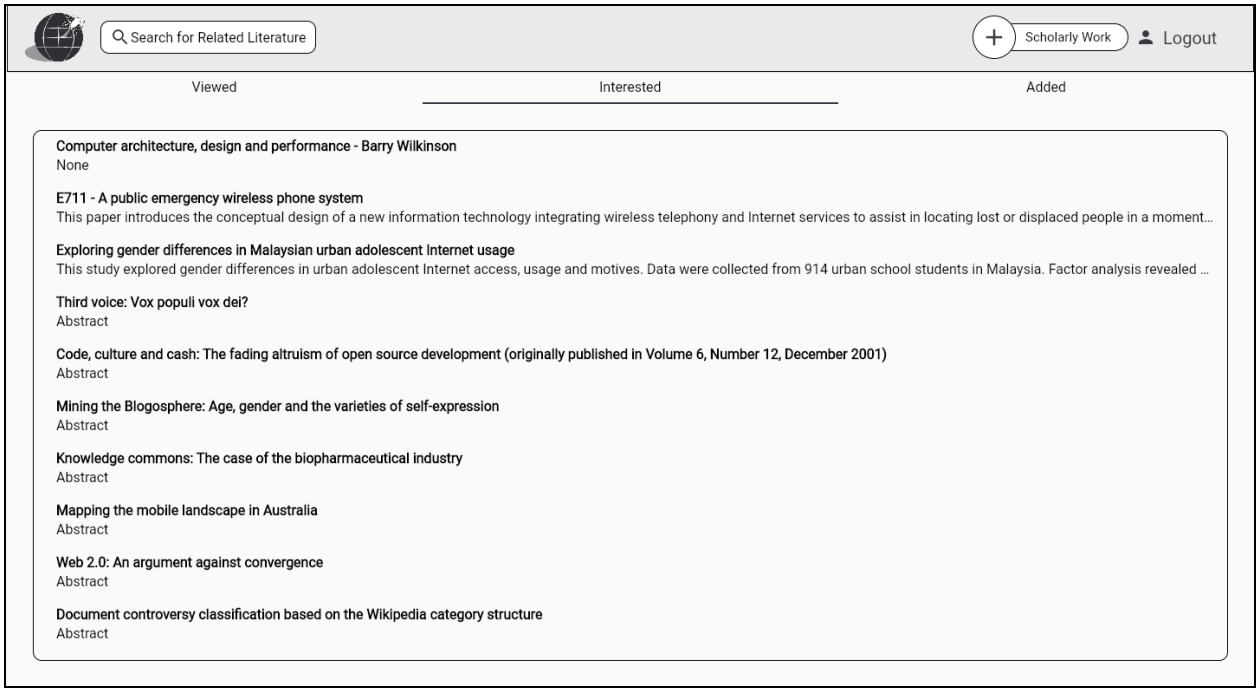
User - Add Scholarly Work

Clicking the “Scholarly Work” on the upper right will display this dialog box for the users to input their own scholarly works by providing the title, abstract, authors, DOI, Journal Source and URL.



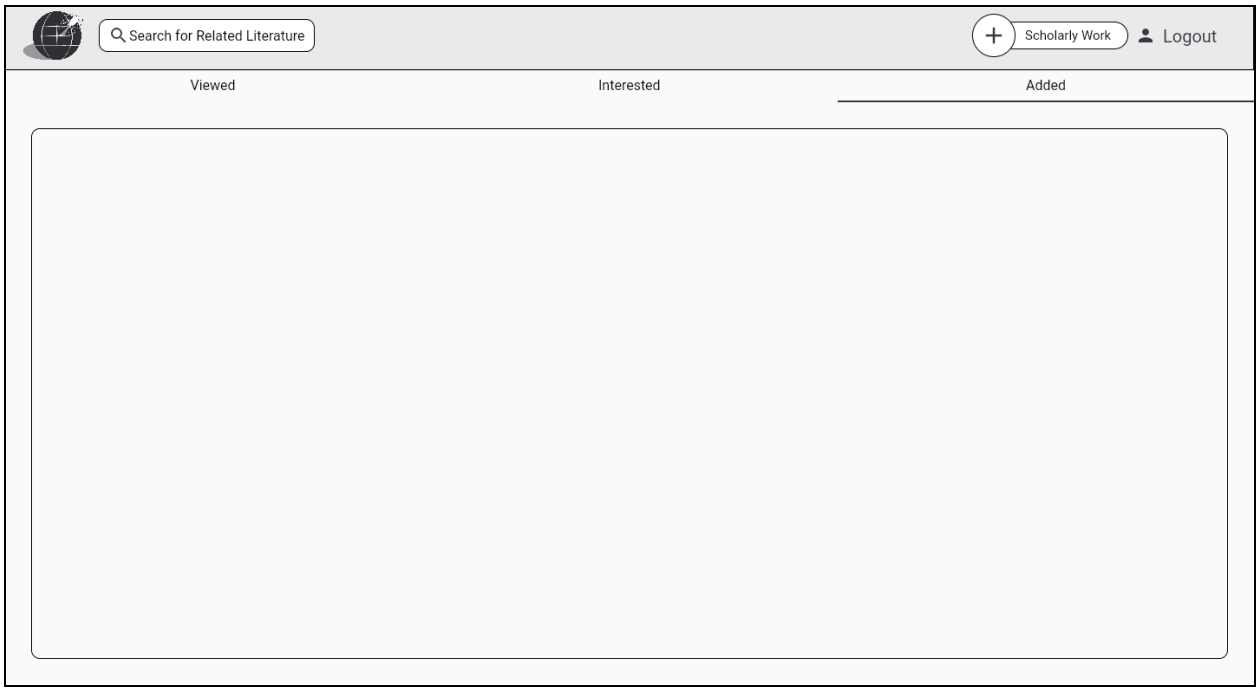
User Profile - Viewed

Clicking the person icon on the upper right will display the user profile. The *viewed* tab shows all the articles being viewed by the user in the past.



User Profile - Interested

Clicking the person icon on the upper right will display the user profile. The *interested* tab is all the articles which show interest to the users.

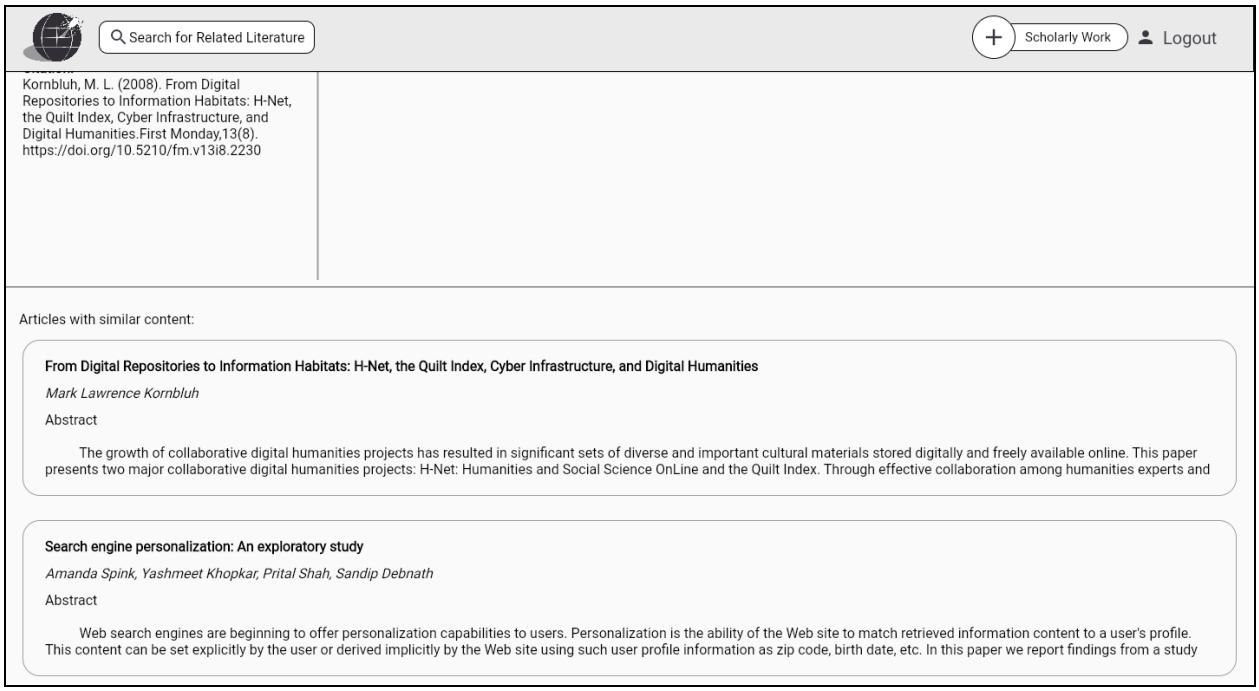


User Profile - Added

Clicking the person icon on the upper right will display the user profile. The *added* tab is all the scholarly works being uploaded by the user to the system.

User - Article Details

Clicking on an article in the homepage will display more information about that article. This screen displays all the details of an article including the title, abstract, authors, source, date published, DOI and citation.



Article Details - Similar Articles

Scrolling down will show all articles that are related to that particular article being opened by the user. Users can then click an article to display more information about that article

CHAPTER III

SOFTWARE DEVELOPMENT AND TESTING

Development Software Platforms, Development Environments, and Tools

The system has 3 parts: the Recommender System, the User Client, and the REST API. The recommender system works to give recommendations to users by working on the corpus. The User client creates a view for the users to interact with the articles, and ask for recommendations. The REST API works to connect both the user client and the recommender system so the two parts can communicate with each other.

The Recommender System is made in Python. Python is a high-level programming language that is widely used because of its language readability and its ability to deal with heavy data science tasks. All of the Recommender System is coded in Python using many of the free libraries. This includes libraries like NumPy, Pandas, and ScraPy.

NumPy is a mathematical library for Python that allows us to deal with large multidimensional arrays and matrices without sacrificing performance. In the project, NumPy is used as a way to easily manipulate and store large matrices that the recommender system uses. Examples of these matrices are the DxK-shaped document-topic matrix (where D is the number of documents and K is the number of

topics), and the KxV-shaped topic-word matrix (where V is the number of words in the vocabulary). Since NumPy allows for easier vectorization and broadcasting, it works well for the many matrix operations used in the Recommender system.

Since the recommender system also deals with large chunks of data from the database (e.g. thousands of articles), the Pandas python library was used. Pandas is a library that allows users to manipulate data a lot easier by storing them in Data Frames. Data Frames are tables of data that have their own column names and column data types. They can be used to store and manipulate data like a database of user information, articles, and etc.

ScraPy is a web scraping library for python. It is free and open-source and can be used to extract data and also as just a general purpose web crawler. This study uses ScraPy as a web scraper to retrieve articles and their metadata from Open Journal Sources.

Flutter, an open-source framework that supports cross platform User Interface creation, was used for the development of the user client. Flutter is a Dart framework that allows you to create beautiful user interfaces by manipulating widgets and how they are laid out. Flutter's cross platform capabilities span operating systems like Android, iOS, Linux, Mac, Windows, and even web platforms. In this project, Flutter is used to create a web application where users are able to interact with articles and request for article recommendations.

The REST API is created through a Python framework called Flask. It is a micro web framework that allows you to create web applications through Python. Aside from

this, Flask is also capable of creating a REST API through the use of the Flask-RESTful library that extends the original Flask library.

The data being used in this system will be stored in a Relational Database Management System. The RDMBS used in this project is MySQL. MySQL is an open source relational database management system and uses tables as a data structure for storing data.

DEVELOPMENT PROCESS

Web Scraper

In order to scrape articles from OJS Sites, a scraper must be created first. The scraper must be able to scrape the different OJS Sites that use different HTML tags to render their metadata. Since scrapers utilize CSS Selectors in order to retrieve the metadata, the scraper must be dynamic. To do so we must store the common CSS Selectors per metadata inside the database. These selectors are added manually since there is no way to just know which selectors hold which metadata. Fortunately, most OJS Sites share common selectors.

selectorTypeId	selectorName	selector
1	title	h1.page_title::text
1	title	div.page-detail-header h1::text
1	title	article.obj_article_details h1::text
1	title	h1.page-header::text
1	title	h1.jatsParser__meta-title::text
1	title	#articleTitle h3::text

Figure X: Common selectors for title metadata

selectorTypeId	selectorName	selector
1	title	h1.page_title::text
2	abstract	.abstract p::text, .abstract p *::text, .abstract *::text
3	authors	ul.entry_authors_list li span.name_wrapper::text
4	datePublished	div.item.published div.value::text
5	doi	span.doi_value a::text
6	howToCite	div.csl-entry *::text
7	journalLinks	div.obj_issue_summary a::attr(href)
8	articleLinks	a.summary_title::attr(href)
9	nextPage	div.cmp_pagination a.next::attr(href)

Figure X: Common selectors for different metadata

We can then use these selectors to check for each metadata to see if any data is scraped using that specific selector.

At the beginning, a user must add an OJS Site that they want to use as a source to scrape by providing certain information. The core information that the user will provide are the archive link and if it contains multiple pages. Every OJS Site is structured similarly—there is an archive page that contains links to every journal volume or issue, and in the volume or issue pages that contain links to its articles. First we must see if we can retrieve any issue link from the archive page by testing every journal link selector and checking if it retrieved anything.

```
for selectorDict in journalLinkSelectors:
    selector: str = selectorDict['selector']
    journalLinks = [s.strip()
                    for s in response.css(selector).getall() if s.strip()]
    journalList: list = [s.strip() for s in response.css(
        selector.replace('attr(href)', 'text')).getall() if s.strip()]
    if (journalLinks and journalList and len(journalLinks) == len(journalList)):
        foundMatching = True
        self.selectorsUsed['journal_links'] = selector
        self.journalList = journalList
        break

if (foundMatching):
    yield scrapy.Request(journalLinks[0], callback=self.parseJournal)
```

Figure X: Trying the journal links selectors

If any one of the journal link selectors was successful, it will remember that selector and proceed to parsing each link retrieved. Since we are now inside a journal

issue, it checks for links to articles instead to see if it retrieved anything. Once it has successfully retrieved the links, it will now parse each article one by one.

```
for selectorDict in articleLinkSelectors:
    selector: str = selectorDict['selector']
    articleLinks = [s.strip()
                    for s in response.css(selector).getall() if s.strip()]
    articleList: list = [s.strip() for s in response.css(
        selector.replace('attr(href)', 'text')).getall() if s.strip()]

    if (articleLinks and articleList and len(articleLinks) == len(articleList)):
        for i in reversed(range(len(articleList))):
            if(articleList[i].lower() == 'pdf' or articleList[i].lower() == 'html'):
                articleLinks.pop(i)
                articleList.pop(i)
        foundMatching = True
        self.selectorsUsed['article_links'] = selector
        self.articleList = articleList
        break

if (foundMatching):
  yield scrapy.Request(articleLinks[0], callback=self.parseArticle)
```

Figure X: Testing out the article links selectors

The final step of the dynamic scraper is to parse the article for metadata. We try out all the metadata selectors such as title, abstract, authors, date published, and more. If a match is found, the selector that is used will remember the selector.

```
for selectorDict in selectors:
    selector: str = selectorDict['selector']
    title = response.css(selector).get().strip() if response.css(selector).get() else ""
    if title:
        self.title = title
        self.selectorsUsed['title'] = selector
        break
```

Figure X: Sample snippet of trying the title selectors

The results will be stored in a JSON file so that the user can verify the selectors and retrieved metadata. Once it has been confirmed, the selectors will be applied on a ScraPy Spider template in order to have a spider for that specific OJS Site.


```
{
  "archiveLink": "https://ijritcc.org/index.php/ijritcc/issue/archive",
    "selectors_used": {
    "journal_links": "h2.media-heading a.title::attr(href)",
    "article_links": "div.article-summary a::attr(href)",
    "title": "h1.page-header::text",
    "abstract": "div.article-abstract *::text",
    "authors": "div.authors div.author strong::text",
    "date_published": ".date-published::text",
    "doi": "div.doi a::attr(href)",
    "how_to_cite": "div.csl-entry *::text",
    "next_page": "a.next::attr(href)"
  },
  "sitename": "ijritcc",
  "category": "Computer",
  "multiple_pages": true,
  "next_page_link": "https://ijritcc.org/index.php/ijritcc/issue/archive/2",
  "url": "https://ijritcc.org/index.php/ijritcc/article/view/5471"
}
```


Figure X: Sample post output of dynamic scraper

Each OJS Site Spider pretty much uses the same code as the dynamic scraper, instead of having to try out multiple selectors it uses the selectors for that particular site only.

```
src = self.name
url = response.url

title = (response.css("h1.page_title::text").get().strip()
         if response.css("h1.page_title::text").get() else "")

abstract: str = ''.join(response.css(".abstract p::text, div.abstract p *::text").getall()).strip()
if not abstract:
    abstract = ''.join(response.css(".abstract *::text").getall()).strip()

authors = [s.strip() for s in response.css("ul.authors li span.name::text").getall() if s.strip()]

date_published =
    ''.join([s.strip() for s in response.css("div.item.published div.value::text").getall() if s.strip()])

doi = response.css("div.doi span.value a::attr(href)").get() if response.css("div.doi span.value
    a::attr(href)").get() else ""

how_to_cite = ''.join([s.strip() for s in response.css("div.csl-entry *::text").getall() if s.strip()])
```

Figure X: Sample ScraPy Spider of an OJS Site

To write the scraped metadata to the database, ScraPy has pipelines that will be called once a specific action has been done. By using the pipeline's process_item function, we can let it write the scraped metadata to the database.

```
insert_query = f"""INSERT INTO ARTICLE (src, title, abstract, datePublished, category, url, doi, howToCite,
    createdAt, updatedAt) VALUES (:src, :title, :abstract, :datePublished, :category, :url, :doi, :howToCite,
    CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP());"""
with self.engine.connect() as connection:
    connection.execute(text(insert_query), {'src':article.get('src'), 'title':article.get('title'),
        'abstract':article.get('abstract'),
        'datePublished':article.get('date_published'),
        'category':article.get('category'), 'url':article.get('url'),
        'doi':article.get('doi'), 'howToCite':article.get('how_to_cite')})
    id_query = pd.read_sql(text("""SELECT LAST_INSERT_ID() as id;"""), self.engine).to_dict(orient="records")
    articleId = id_query[0]['id']
    for author in article.get('authors'):
        with self.engine.connect() as connection:
            connection.execute(text('INSERT INTO AUTHOR (articleId, authorName)
                VALUES (:articleId, :authorName);'), {'articleId': articleId, 'authorName': author})
```

Figure X: Writing the scraped metadata to the database

Data Preprocessing

After getting the articles from the web scraper, we need to perform data preprocessing from the abstracts of every article. The first thing to do is to remove all non-ascii characters (or characters with Unicode greater than or equal to 128) from the abstracts.

```
@staticmethod
def remove_non_ascii(word):
    return ''.join(ch for ch in word if ord(ch) < 128)
```

Figure x: Removing all non-ascii characters

The method takes a word as input and iterates through every character. Calling the ord() function will output the character's Unicode. The method returns True if its Unicode is less than 128, otherwise False.

The next step to perform is to include english-only articles in the corpus. This is to help the training of the model later after the next process.

```
def is_english(self, text):
    words = set(nltk.wordpunct_tokenize(text.lower()))
    lang = max(((lang, len(words & stopwords))
                for lang, stopwords
                in self.stopwords_dict.items()),
               key = lambda x: x[1])[0]
    return lang == 'english'
```

Figure x: Getting english-only articles

The method takes an article's abstract as input and performs tokenization stored in the words variable. It will then iterate through the collection of stopwords from different languages and get the length of the union of words and stopwords. The article's language will be determined by whichever language has the most number of stopwords found in the abstract. The method will return True if the language is english, otherwise False.

The next step is to remove all punctuations and numbers in every article's abstract with the help of regular expressions.

```
regex = re.compile('[' + re.escape(string.punctuation) + '\\d\\r\\t\\n]')
punc_num_free = regex.sub(' ', abstract)
```

Figure x: Removing all punctuations and numbers

Then, we will perform Part-of-speech tagging to help us in the next step, which is Lemmatization.

```
pos_tags = nltk.pos_tag(punc_num_free.split())
normalized = [self.lemmatizer.lemmatize(word[0].lower(),
                                        pos=self.lemma_pos(word[1]))
              for word in pos_tags]
```

Figure x: POS Tagging and Lemmatization

Here, we need to pass the correct POS Tag for a token when lemmatizing to ensure that the correct lemma will be produced. We now have a normalized text that will surely help us identify the correct vocabulary.

To get the vocabulary of the corpus, we need to generate the Ngrams of an abstract which are the unigram, bigram, and trigram. For getting the unigram, we just simply remove all the stopwords from the abstract and store it as a separate column in the database. For bigram and trigram, the order of the words in an abstract matters so removing the stopwords will destroy the purpose of getting such. Thus, we are going to use Collocations in order to generate the correct bigrams and trigrams.

```
bigrams = nltk.collocations.BigramAssocMeasures()
trigrams = nltk.collocations.TrigramAssocMeasures()

bigramFinder = nltk.collocations.BigramCollocationFinder.from_documents(corpus)
trigramFinder = nltk.collocations.TrigramCollocationFinder.from_documents(corpus)

bigramFinder.apply_freq_filter(20)
bigram_scores = bigramFinder.score_ngrams(bigrams.pmi)
trigramFinder.apply_freq_filter(20)
trigram_scores = trigramFinder.score_ngrams(trigrams.pmi)
```

Figure x: Initializing collocations

In this figure, we initialized the collocation objects for bigrams and trigrams. By using the collocation finders, the code will generate all the bigrams and trigrams found in the corpus and store them in the bigramFinder and trigramFinder variables, respectively. Then, we applied a frequency filter of 20 which means a token should have at least a frequency of 20 to be included in the list. After that, we will get the

corresponding scores of each token depending on its frequency and with respect to the whole corpus.

	bigram	score
0	(nitric, oxide)	14.477753
1	(peroxisome, proliferator)	14.290821
2	(sars, cov)	14.236744
3	(chi, square)	13.403752
4	(united, states)	13.239747
5	(bl, j)	13.015073
6	(x, ray)	13.014352
7	(waist, circumference)	12.986871
8	(double, blind)	12.621783
9	(laparoscopic, cholecystectomy)	12.552246
10	(proliferator, activate)	12.398025
11	(confidence, interval)	12.285734
12	(logistic, regression)	12.257064
13	(western, blot)	12.195717
14	(gut, microbiota)	12.144162
15	(pi, k)	12.132052
16	(reactive, oxygen)	11.990411
17	(differentially, private)	11.946151
18	(caesarean, section)	11.941289
19	(green, tea)	11.915874
20	(cesarean, section)	11.803785
21	(meonf, ii)	11.495900

Figure x: Bigram scores

Notice that we still have bigrams that do not really make sense but still have high scores like bl j, pi k, meonf ii, etc.

	trigram	score
0	(peroxisome, proliferator, activate)	26.688846
1	(proliferator, activate, receptor)	24.750247
2	(reactive, oxygen, specie)	24.468163
3	(c, bl, j)	23.272777
4	(chi, square, test)	22.771080
5	(dx, doi, org)	21.480876
6	(http, dx, doi)	21.407414
7	(nordic, nutrition, recommendations)	21.270728
8	(doi, org, fnr)	21.195355
9	(acute, coronary, syndrome)	21.086109
10	(activate, protein, kinase)	20.942705
11	(density, lipoprotein, cholesterol)	20.929582
12	(tumor, necrosis, factor)	20.849962
13	(t, l, adipocytes)	20.753626
14	(diastolic, blood, pressure)	20.701512
15	(org, fnr, v)	20.631586

Figure x: Trigram scores

Same with trigrams. c bl j, dx doi org, doi org fnr, etc. are not really valid trigrams. To fix this, we apply a corresponding filter for both bigrams and trigrams.

```
def bigram_filter(self, bigram):
  tag = nltk.pos_tag(bigram)
  if (tag[0][1] not in self.adjective_pos + self.noun_pos)
  and (tag[1][1] not in self.noun_pos):
      return False
  if bigram[0].isspace()
  or bigram[1].isspace():
      return False
  if bigram[0] in self.english_stop
  or bigram[1] in self.english_stop:
      return False
  return True
```

Figure x: Bigram filter

This method takes a bigram as an input and determines whether it is a valid token or not. If the first word is neither an adjective or a noun, and the second word is not a noun, then it is not valid. If either of the words are spaces or stopwords, then it is not valid as well. Otherwise, the bigram is a valid token.

```
def trigram_filter(self, trigram):
  tag = nltk.pos_tag(trigram)
  if (tag[0][1] not in (self.adjective_pos + self.noun_pos))
  and (tag[1][1] not in (self.adjective_pos + self.noun_pos)):
      return False
  if trigram[0].isspace()
  or trigram[1].isspace()
  or trigram[2].isspace():
      return False
  if trigram[0] in self.english_stop
  or trigram[1] in self.english_stop
  or trigram[2] in self.english_stop:
      return False
  return True
```

Figure x: Trigram filter

This method takes a trigram as an input and determines whether it is a valid token or not. If the first and second word is neither an adjective or a noun, then it is not valid. If either of the words are spaces or stopwords, then it is not valid. Otherwise, the trigram is a valid token.

vocId	token
469	artificial intelligence
470	artificial neural
511	association rule
675	base station
763	big data
963	business model
1277	cloud compute
1278	cloud computing
1333	collect data
1385	commonly use
1392	communication technology
1408	comparative analysis
1448	component analysis
1471	compute resource
1473	computer science
1474	computer vision
1879	data center
1880	data mine
1881	data mining

Figure x: Filtered bigrams

vocId	token
97	ad hoc network
471	artificial neural network
674	base image retrieval
1622	content base image
1882	data mining technique
1921	decision support system
4776	mobile ad hoc
5116	open source software
6703	sensor network wsns
7305	support vector machine
8186	wireless sensor network
8222	world wide web

Figure x: Filtered trigrams

Now that we have the vocabulary of our corpus, we can then proceed to the next development process.

Vectorization

Using the vocabulary of the corpus, it is time to get the Document Frequency of each token. To do this, we just need to count how many times a token appears within the articles of the corpus.

vocId	token	docFreq	invDocFreq
7911	use	2213	1.5746016591632275
673	base	1333	2.0812205459562403
7385	system	1283	2.1194222881819034
1878	data	1118	2.256967064119884
3871	information	988	2.3804634408090974
5870	provide	887	2.48818602943964
4788	model	816	2.571518677571807
4679	method	811	2.5776574322701316
7581	time	785	2.610200980002603
5779	process	764	2.6372819386052737
397	application	756	2.6477945189943606
8213	work	750	2.6557521206676746
6795	show	745	2.6624321722280486
4955	network	736	2.6745698802424727
7453	technology	723	2.692366380046093
300	analysis	704	2.718959969619541
5773	problem	689	2.7404661748405044
7451	technique	675	2.7609646963888457
410	approach	670	2.768388635460128
4501	make	645	2.8063582686492077
224	algorithm	634	2.823532773539118

Figure x: Document Frequency and its Inverse

Applying natural logarithm to the frequency of every token will output its Inverse Document Frequency.

```
vocabulary['invDocFreq'] = vocabulary['docFreq'].apply(lambda freq:
                                                       np.log((docs_count + 1) / (freq + 1)) + 1)
```

Figure x: Getting the Inverse Document Frequency

One important step in fine tuning the results of topic modeling is to include only tokens with document frequency greater than or equal to 0.5% of the whole corpus. Tokens that qualify will be part of the pruned vocabulary which will be used in topic modeling. But before proceeding to that, we still need to create a numerical representation of all this data about the pruned vocabulary. We will get the TF-IDF scores of each token in every article or document in the corpus.

Doc-Term Matrix	camera	computer	lens	vision
Document 1	4	1	2	1
Document 2	0	2	3	7
Document 3	5	12	3	9
Document 4	3	7	1	4

Figure x: Term Frequency

Since we already have the Inverse Document Frequency, the next step is to get the raw TF-IDF which is the weighted counts for each token by document.

	camera	computer	lens	vision
tf	4	1	2	1
idf	5.561053	3.462067	5.61512	4.950144
tfidf_raw	22.244212	3.462067	11.23024	4.950144

Figure x: Raw TF-IDF

The next step is to normalize the weighted scores of each token for the model to have a uniform value to manipulate during training. The resulting matrix is now called the Document-Term Matrix.

	camera	computer	lens	vision
tfidf_raw	22.244212	3.462067	11.23024	4.95014470
tfidf	0.033836	0.005266	0.017082	0.007523

Figure x: Document-Term Matrix

In order to store the DTM in the database, we make use of a coo_matrix to represent a sparse matrix as an entity.

abstractId	pruneId	tfidf_score
1	5	0.27447266690055316
1	35	0.20599826958422388
1	42	0.08187037241625451
1	57	0.08503129873438629
1	117	0.10032918671414144
1	149	0.09317216474558176
1	169	0.08427564283515011
1	243	0.10398982693719289
1	249	0.13657592367236132
1	263	0.10945278163043862
1	354	0.06851673547446346

Figure x: COO Matrix Representation of DTM

The coordinate matrix representation follows the format of (row, column, value) which means in a 2-dimensional matrix the abstractId is the row, prunedId is the column and tfidf_score is the value. This is the last step of vectorization and it is now ready to be utilized by the next development process which is topic modeling.

Latent Dirichlet Allocation

Once the Document-term matrix is created through TF-IDF, the LDA algorithm is able to start. The first thing to do is to create an instance of the LDA class that allows you to create an LDA Topic Model.

```
lda_model = LDA(alpha=0.04, beta=0.5, topics=10, iterations=100)
```

Figure x: Instantiating an LDA class

The LDA class takes the hyperparameters as its arguments. Here the hyperparameters are the two dirichlet priors alpha and beta, the number of topics K, and the number of iterations. The alpha and beta are smoothing factors that help us find the conditional probabilities later on. They also serve as density indicators for both the document-topic distribution and the topic-word distribution. A higher alpha score means that there are more topics that can be found in a certain document, and a lower beta score means that the topics are composed of more words, and vice versa. The number of topics K indicates how many topics the topic model will assume is present in the corpus. The number of iterations will determine how many times the LDA algorithm will run.

The first thing that the LDA model does when instantiated is create it's variables and pull up all the important details to make the topic model. The document-term matrix and the vocabulary are read from the database, and the values for D (the number of documents in the corpus) and V (the number of words in the vocabulary) are then determined based on the document-term matrix and vocabulary.

abstractId	pruneId	topic
0	4	NaN
0	33	NaN
0	40	NaN
0	53	NaN
0	112	NaN
0	144	NaN
0	164	NaN
0	238	NaN
0	244	NaN

Figure x: Row-column-value representation of the document-term matrix

The document-term matrix starts off with null values in the topic column. This is because the LDA algorithm starts by randomly assigning the words in every document a random topic from 0 to K-1.

```
def _randomize_topics(self):
    rnd_t = np.random.randint(0, high=self.K, size=len(self.dtm))
    self.dtm['topic'] = rnd_t
```

Figure x: Randomize topics

This is done by using the _randomize_topics() method. The ‘topic’ column in the document-term matrix (known as self.dtm in the code) is assigned an array of random values from 0 to K-1 generated by the random.randint method from numpy.

abstractId	pruneId	topic
0	4	8
0	33	5
0	40	6
0	53	8
0	112	6
0	144	8
0	164	6
0	238	7
0	244	4

Figure x: Document-term matrix with randomized topics

Once the topics have been randomized, we get two matrices that help us keep count of the number of topics that have been assigned to both documents and words. These matrices are the document-topic count and the topic-term count. The document-topic count matrix is a matrix that has documents for rows and topics for its columns. Each value in the matrix represents the number of times a certain topic appeared in a certain document. The topic-term count matrix is similar where it counts the number of times a certain word is assigned a certain topic. The difference is, the topics are now the rows while the words are the columns.

	0	1	2	3	4	5	6	7	8	9
0	3	3	2	3	4	4	7	7	8	4
1	1	3	1	1	3	0	2	1	2	3
2	3	1	1	3	5	1	2	2	4	9
3	0	4	5	2	1	3	1	2	0	0
4	4	3	5	1	1	1	5	3	4	0
5	4	2	2	5	4	4	5	2	1	3
6	2	3	0	3	2	1	3	1	4	0
7	6	1	9	7	4	7	4	5	3	3
8	4	4	4	4	6	6	1	6	4	4
9	2	3	5	3	3	2	3	1	5	3
	...	...	...	...	...	...	...	...	...	...

Figure X: Document-topic count matrix after topic randomization

	0	1	2	3	4	5	6	7	8	9	
0	19	3	2	2	3	39	4	14	13	21	...
1	9	5	1	1	5	48	1	17	10	25	...
2	14	0	1	4	2	37	4	14	17	21	...
3	11	2	3	4	3	38	1	16	10	19	...
4	11	2	3	4	2	52	3	20	12	18	...
5	15	2	4	5	6	40	3	16	7	24	...
6	10	3	0	4	3	45	2	20	10	24	...
7	18	5	2	5	3	58	4	10	17	21	...
8	13	4	1	3	4	40	3	15	8	17	...
9	11	0	3	3	4	42	5	14	6	24	...

Figure X: Topic-term count matrix after topic randomization

Then the LDA algorithm starts calculating for the conditional probability of each word. It starts with the first word of the first document until the last word of the last document.

To calculate the Conditional Probability of a word, we need to sample it. To start, the word must be thought of as a word that has no topic. Since the first word of the first document is word 4, we sample that by treating it as if it doesn't exist. The initial topic of

document 0 word 4 ([0,4]) was 8, so we decrement [0,8] from the document-topic count matrix and and [8,4] from the topic-term count matrix.

abstractId	pruneId	topic
0	4	?
0	33	5
0	40	6
0	53	8
0	112	6
0	144	8
0	164	6
0	238	7
0	244	4

Figure X: Sampling Word 4 from Document 0

0	1	2	3	4	5	6	7	8	9
3	3	2	3	4	4	7	7	7	4
1	3	1	1	3	0	2	1	2	3
3	1	1	3	5	1	2	2	4	9
0	4	5	2	1	3	1	2	0	0
4	3	5	1	1	1	5	3	4	0
4	2	2	5	4	4	5	2	1	3
2	3	0	3	2	1	3	1	4	0
6	1	9	7	4	7	4	5	3	3
4	4	4	4	6	6	1	6	4	4
2	3	5	3	3	2	3	1	5	3
...	...	...	...	...	...	...	...	...	...

Figure X: Decrement [0,8] from document-topic count matrix

	0	1	2	3	4	5	6	7	8	9	
0	19	3	2	2	3	39	4	14	13	21	...
1	9	5	1	1	5	48	1	17	10	25	...
2	14	0	1	4	2	37	4	14	17	21	...
3	11	2	3	4	3	38	1	16	10	19	...
4	11	2	3	4	2	52	3	20	12	18	...
5	15	2	4	5	6	40	3	16	7	24	...
6	10	3	0	4	3	45	2	20	10	24	...
7	18	5	2	5	3	58	4	10	17	21	...
8	13	4	1	3	3	40	3	15	8	17	...
9	11	0	3	3	4	42	5	14	6	24	...

Figure X: Decrement [8,4] from topic-term count matrix

After decrementing the counts from the two matrices, we will use them to calculate the conditional probability. The conditional probability of a word is the probability that it can be found on a topic given the knowledge that we have of other words in the corpus. It is a product of two different probabilities that we calculate using the two matrices.

The first probability is the number of times a word in that document was assigned to topic k + alpha over the total number of words in that document + the number of topics multiplied by alpha. The second probability is the number of times this specific word is assigned to this topic in the whole corpus + beta over the total number of words assigned to topic k + the number of words in the vocabulary multiplied by beta. These two probabilities are then multiplied with each other to get the final conditional probability. The conditional probabilities for all topics then need to be normalized so that the sum of all probabilities sum to 1. This is repeated k-number of times in order to get the conditional probability of this word appearing in each and every topic.

```
td = dt[r] + self.alpha / dt[r].sum() + (self.K*self.alpha)
wt = tt[:,c] + self.beta / tt.sum(axis=1) + (self.V*self.beta)
probs = td*wt
distribution = probs / probs.sum()
```

Figure X: Calculating the Conditional Probabilities for all topics

0	0.062218092567004
1	0.101365808252649
2	0.027588631427748
3	0.060737234091229
4	0.055051753280526
5	0.158480320473455
6	0.141211096571004
7	0.140042507652049
8	0.144459677104143
9	0.108844878580190

Figure X: Conditional probability for Document 0 Word 4

Once the conditional probabilities of the word are found, we need to get the new topic to assign to this word. To do this, we need to pick a word at random based on the conditional probabilities. In the first assignment of topics to a word, the assignment was based on a uniform probability where each word has the same probability that it ends up getting assigned to any topic. In subsequent assignments, the choice is still random, but is now based on the calculated conditional probability instead of a uniform probability.

```
nk = np.random.choice(a=self.K, p=distribution)
```

Figure X: Choosing a new topic

0	0.062218092567004
1	0.101365808252649
2	0.027588631427748
3	0.060737234091229
4	0.055051753280526
5	0.158480320473455
6	0.141211096571004
7	0.140042507652049
8	0.144459677104143
9	0.108844878580190

Figure X: Chosen topic based on conditional probability

After choosing the topics based on the conditional probability, we then increment the two count matrices by 1. We increment the document-topic count matrix by 1 on cell [0,8] where 8 is the new chosen topic, and we increment the topic-term count matrix by 1 on cell [8,4].

Figure X: Increment both count matrices

	0	1	2	3	4	5	6	7	8	9
0	3	3	2	3	4	4	7	7	8	4
1	1	3	1	1	3	0	2	1	2	3
2	3	1	1	3	5	1	2	2	4	9
3	0	4	5	2	1	3	1	2	0	0
4	4	3	5	1	1	1	5	3	4	0
5	4	2	2	5	4	4	5	2	1	3
6	2	3	0	3	2	1	3	1	4	0
7	6	1	9	7	4	7	4	5	3	3
8	4	4	4	4	6	6	1	6	4	4
9	2	3	5	3	3	2	3	1	5	3
	...	...	...	...	...	...	...	...	...	...

Figure X: Document-topic matrix incremented

	0	1	2	3	4	5	6	7	8	9	
0	19	3	2	2	3	39	4	14	13	21	...
1	9	5	1	1	5	48	1	17	10	25	...
2	14	0	1	4	2	37	4	14	17	21	...
3	11	2	3	4	3	38	1	16	10	19	...
4	11	2	3	4	2	52	3	20	12	18	...
5	15	2	4	5	6	40	3	16	7	24	...
6	10	3	0	4	3	45	2	20	10	24	...
7	18	5	2	5	3	58	4	10	17	21	...
8	13	4	1	3	4	40	3	15	8	17	...
9	11	0	3	3	4	42	5	14	6	24	...

Figure X: Topic-term matrix incremented

The document-term matrix will also reflect this change by assigning the topic of that part of the document-term matrix with the new topic.

```
dtm['topic'] = dtm.apply(lambda r: self._conditional_probability(r, dt, tt), axis=1)
```

Figure X: Assigning the returned new topic to the document-term matrix

abstractId	pruneId	topic
0	4	8
0	33	5
0	40	6
0	53	8
0	112	6
0	144	8
0	164	6
0	238	7
0	244	4

Figure X: Document-term matrix after sampling

This process is then repeated for each word in each document in the corpus. This process of calculating the conditional probabilities of all words and choosing new topics is then repeated for a number of times. This set number of times is the given iterations number when instantiating the lda_model.

The final step in the topic modeling process is to get the document-topic distributions (Theta) and the topic-word distributions (Phi). The document-topic distribution is a probability distribution that each document exists in each topic. The topic-word distribution is a probability distribution that each word is part of a topic. Getting the theta and the phi are similar to calculating the two probabilities that make up the conditional probability.

Figure X: Calculating theta

Figure X: Calculating phi

The theta is calculated by taking every value in the doctopic_count matrix and adding the alpha value to it. This value is then divided by the sum of that row + alpha multiplied by the number of topics.

The phi is calculated by taking every value in the topicterm_count matrix and then adding the beta value. This is then divided by the sum of that row + beta multiplied by the number of words in the vocabulary.

	0	1	2	3	4	5	6	7	8	9
0	0.000222	0.000222	0.776275	0.000222	0.000222	0.221951	0.000222	0.000222	0.000222	0.000222
1	0.000585	0.000585	0.000585	0.000585	0.994737	0.000585	0.000585	0.000585	0.000585	0.000585
2	0.000322	0.032476	0.000322	0.000322	0.964952	0.000322	0.000322	0.000322	0.000322	0.000322
3	0.000553	0.000553	0.000553	0.000553	0.000553	0.995028	0.000553	0.000553	0.000553	0.000553
4	0.000369	0.000369	0.000369	0.000369	0.000369	0.000369	0.996679	0.000369	0.000369	0.000369
5	0.000312	0.000312	0.000312	0.000312	0.997196	0.000312	0.000312	0.000312	0.000312	0.000312
6	0.000524	0.000524	0.000524	0.000524	0.995288	0.000524	0.000524	0.000524	0.000524	0.000524
7	0.000204	0.000204	0.000204	0.000204	0.000204	0.000204	0.9778	0.02057	0.000204	0.000204
8	0.000232	0.000232	0.000232	0.000232	0.533875	0.464269	0.000232	0.000232	0.000232	0.000232
9	0.000332	0.000332	0.000332	0.000332	0.000332	0.000332	0.99701	0.000332	0.000332	0.000332
	...	...	...	...	...	...	...	...	...	...

Figure X: Theta values

	0	1	2	3	4	5	6	7	8	9	
0	0.0009837	0.0003116	0.0000061	0.0000672	0.0001894	0.0000061	0.0000061	0.0015947	0.0009226	0.0061771	...
1	0.0007894	0.0001493	0.0000071	0.0001493	0.0002205	0.0003627	0.0000782	0.0007182	0.0012872	0.0042739	...
2	0.0008450	0.0000070	0.0000070	0.0000768	0.0005657	0.0055241	0.0004958	0.0012641	0.0007054	0.0005657	...
3	0.0003381	0.0000082	0.0000082	0.0000082	0.0000082	0.0046260	0.0000082	0.0028119	0.0005030	0.0000082	...
4	0.0010068	0.0001236	0.0003592	0.0000648	0.0004769	0.0032442	0.0004180	0.0008302	0.0020078	0.0000648	...
5	0.0009018	0.0004554	0.0000089	0.0000089	0.0000089	0.0016161	0.0000089	0.0004554	0.0000982	0.0014375	...
6	0.0006392	0.0002361	0.0001209	0.0005240	0.0005240	0.0030579	0.0000058	0.0006968	0.0006392	0.0000058	...
7	0.0009972	0.0000066	0.0007331	0.0005349	0.0002708	0.0024501	0.0000066	0.0013274	0.0003368	0.0009972	...
8	0.0009352	0.0000103	0.0000103	0.0000103	0.0000103	0.0029904	0.0002158	0.0017573	0.0000103	0.0009352	...
9	0.0020203	0.0006270	0.0000851	0.0010140	0.0000077	0.0082901	0.0010140	0.0000077	0.0007818	0.0003174	...

Figure X: Phi values

Getting Topic Coherence

In this study, the UMass coherence measure was used. The UMass measure is taken by getting the top N words in each topic and comparing them with each other. Here the top 10 words are being used.

Each lower ranking word is then compared with every other higher ranking word and the mean of all the scores in that topic is taken as the coherence score for that topic. Word 10, for example, must be compared to words 1 to 9. Word 9 must be compared to words 1 to 8, and so on.

To get the UMass score for each comparison, we take the number of times the two words have appeared together in a document + a smoothing factor 1. This is then divided by the number of times the higher ranking word appears in a document. The logarithm of the resulting value is then returned as the UMass score for this comparison. The closer the score is to 0, the better the coherence.

```
qry = f"SELECT COUNT(abstractId) FROM doc_term_matrix WHERE pruneId = {j};"
qry2 = f"""SELECT COUNT(abstractId)
FROM doc_term_matrix WHERE pruneId = {i}
AND abstractId IN (SELECT abstractId
FROM doc_term_matrix WHERE pruneId = {j});"""

dj = pd.read_sql_query(qry, eng).values[0,0]
dij = pd.read_sql_query(qry2, eng).values[0,0]
return np.log((dij+e)/dj)
```

Figure X: UMass Coherence Measure formula

topic_number	coherence_score
0	-1.37929
1	-1.41507
2	-1.6041
3	-1.51001
4	-1.68072
5	-1.6111
6	-1.80505
7	-1.45072
8	-1.65414
9	-1.60333

Figure X: Coherence scores for every topic

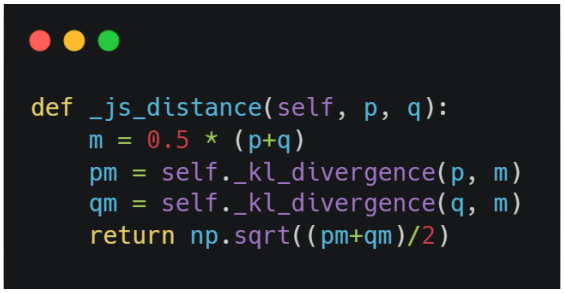
Jensen-Shannon Distance

The Jensen-Shannon distance uses the Kullback-Leibler Divergence (KL Divergence) since it is a symmetric version of it. We start by creating the KL Divergence. For every value in probability distribution P, the logarithm of that value P divided by its corresponding value Q should be multiplied with it. Therefore P_1 should be multiplied by $Log(P_1 / Q_1)$. This is repeated for each value in the probability distribution P and Q. The sum of every calculation is taken as the Distance between Probability Distributions P and Q. It is worth noting that P and Q would never be 0 given that the calculations of theta, the values we want to compare using this measure, were smoothed out by the hyperparameter alpha. P and Q must also be the same size.

```
def _kl_divergence(self, p, q):
    return np.sum(np.where(p!=0, p*np.log(p/q),0))
```

Figure X: Kullback-Leibler Divergence

Once we have the KL Divergence, we can use it for the Jensen-Shannon Distance (JS Distance). The JS Distance first takes the mean M of P and Q so that it can calculate the KL Divergence of P and M, and Q and M. The sum of the KL Divergence of P and M, and Q and M are then divided by two. The value resulting from this division is passed through a square root function to get the JS Distance.

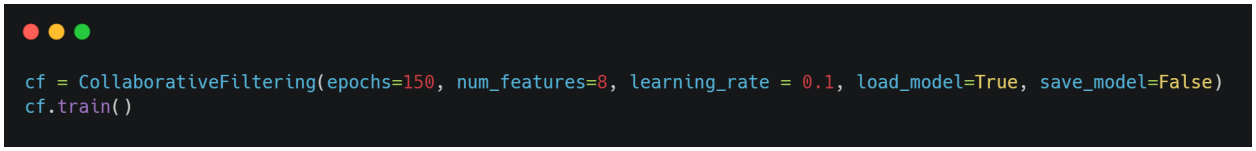


```
def _js_distance(self, p, q):
    m = 0.5 * (p+q)
    pm = self._kl_divergence(p, m)
    qm = self._kl_divergence(q, m)
    return np.sqrt((pm+qm)/2)
```

Figure X: Jensen-Shannon Distance

Matrix Factorization

Once the user has interacted with the articles, we can now create a user-item matrix based on the user's behavior and conduct Matrix Factorization. But first, we need to instantiate the model and pass it the hyperparameters. This includes the epoch—which dictates how many iterations the model will be trained for, the number of features, and its learning rate. Afterwards, we can begin the training, which is conducting Matrix Factorization.



```
cf = CollaborativeFiltering(epochs=150, num_features=8, learning_rate = 0.1, load_model=True, save_model=False)
cf.train()
```

Figure X: Initializing the model with hyperparameters and starting the training

But first, we must load the user-item dataset, which is done by retrieving the user item interactions and assigning weights to them. Cold start interactions are weighted with a 0.5 score, viewed articles are weighted with 1, interested articles weigh 3, and scholarly works are the highest with a weight of 5.

```
dense_matrix[[coo_matrix['userId']-1],[coo_matrix['articleId']-1]] = weight
```

Figure X: The general code for creating the dense matrix after querying

After retrieving each user interaction matrix, we combine all into an aggregated matrix. This matrix will become the dataset and is what Matrix Factorization will be applied on.

```
user_article_aggregated_matrix = (
    self.get_user_cold_start_matrix(user_article_shape) +
    self.get_user_viewed_matrix(user_article_shape) +
    self.get_user_interest_matrix(user_article_shape) +
    self.get_user_scholarly_works_matrix(user_article_shape)
)
```

Figure X: Creating the aggregated matrix

To prepare for Matrix Factorization, we need to create two matrices that will store the user and item features. The user features matrix will have its shape based on the number of users, the item features matrix will by the number of items (in this case articles), and both are affected by the number of features to learn. We initialize these matrices with by generating random numbers with uniformed distribution, and from there we can start conducting Matrix Factorization.

```
self.user_features = np.random.uniform(low=0.1, high=0.9, size=(self.max_users, self.num_features))
self.item_features = np.random.uniform(low=0.1, high=0.9, size=(self.num_features, self.max_items))
```

Figure X: Initializing User and Item Features

The Matrix Factorization itself is simple—per iteration, we adjust all the features of every user and item by using gradient descent. By conducting gradient descent, we will know by how much we need to modify the user and item features to make the error closer to zero. This is because using gradient descent will give you the cost function's gradient, and by going in the opposite direction to go down we can reach the point of local minima given enough iterations.

```
def matrix_factorization(self):
    for epoch in range(self.epochs):
        for k in range(self.num_features):
            for user in self.valid_user_ids:
                self.user_features[user][k] += self.lr * self.get_user_batch_gradient_descent(user, k)
            for item in self.valid_article_ids:
                self.item_features[k][item] += self.lr * self.get_item_batch_gradient_descent(item, k)
```

Figure X: Code for Matrix Factorization

Batch Gradient Descent

Gradient Descent is what helps determine how large the step must be taken when adjusting the features. Too close, and it will take too long to reach the local minima—too far and you might go beyond it. One of the methods of Gradient Descent is the Batch Gradient Descent, which uses the entire dataset and measures its gradient. This is done by taking the derivative of the cost function (Mean Squared Error in this case), however since we have two inputs—user and item—then we will need to derive both with respect to the themselves. By doing so we can create the User and Item Batch Gradient Descent functions, which will help adjust the user and item features in each iteration, until we finish training the model.

```python
def get_user_batch_gradient_descent(self, user, k):
    prediction_matrix = np.dot(self.user_features, self.item_features)
    items = np.array(np.where(self.data_matrix[user][:] > 0)[0])

    sum = np.sum(2 * (self.data_matrix[user][items] - prediction_matrix[user][items]) * self.item_features[k][items])

    return sum / self.num_items
```

Figure X: User Batch Gradient Descent

```python
def get_item_batch_gradient_descent(self, item, k):
    prediction_matrix = np.dot(self.user_features, self.item_features)
    users = np.array(np.where(self.data_matrix.T[item][:] > 0))[0]

    sum = np.sum(2 * (self.data_matrix.T[item][users] - prediction_matrix.T[item][users])
                 * self.user_features.T[k][users])

    return sum / self.num_users
```

Figure X: Item Batch Gradient Descent

Once we're finished training the model, we store the learned user and item features which will be used later on to create recommendations.

Mini-Batch Gradient Descent

Mini-batch Gradient Descent is the same as Batch Gradient Descent, but instead of using the entire dataset, it derives a subset by drawing an *n* amount of data from it (typically without replacement). This number is usually a power of two, e.g. 16, 32, 64, 128, and so on. This helps when the dataset is too large, where a single epoch will take too long. For this project's model, a mini-batch of 256 was used.

Early Stopping

One of the problems a model can encounter is becoming too fit to a dataset or simply "overfitting"—meaning it instead "memorizes" the existing dataset, and the model's performance suffers and will have trouble adjusting to new, incoming data. To mitigate this problem, early stopping can be done—by stopping the training before it can overfit to the dataset.

One way early stopping can be implemented is by splitting the entire dataset into training sets and validation sets—the model will learn from the training set and at the same time use the validation set as a secondary source of accuracy. During training, if the accuracy for the training set goes up but the accuracy for the validation set goes down, this means that the model is starting to overfit to the training set. At this point, the training can be stopped to prevent overfitting.

```python
def to_early_stop(self, epoch):
    if epoch < 5:
        return False

    acc_increase_on_training = 0
    acc_decrease_on_validation = 0
    for i in range(1, epoch):
        if self.training_accuracy_per_epoch[i - 1] < self.training_accuracy_per_epoch[i]:
            acc_increase_on_training += 1

        if self.validation_accuracy_per_epoch[i - 1] > self.validation_accuracy_per_epoch[i]:
            acc_decrease_on_validation += 1

        if acc_decrease_on_validation >=3 and acc_increase_on_training >= 3:
            print(i, acc_increase_on_training, acc_decrease_on_validation)
            return True

    return False
```

Figure X: Function that checks if early stopping is appropriate in the current epoch

Content-based Recommendation

Opened Article's Similar Articles

Content-based recommendations are used when an article is opened. The opened article will also suggest the articles in the corpus that are similar in topic to the current article. This process is done by measuring the Jensen-Shannon Distance between the current article and every other article in the corpus.

```python
article_theta = self.theta[aid-1, :]
distances = self._get_distances(article_theta)
```

Figure X: Calling the get_distances method using the article's theta

First thing to do is to get the article's theta. This is possible by selecting the row of that article using its article id. That article's theta is then sent to the _get_distances method where it's theta is compared to each and every document in the corpus.

```
def _get_distances(self, doc):
    distances = np.zeros(self.D)
    for d in range(self.D):
        distances[d] = self._js_distance(doc, self.theta[d])
```

Figure X: Getting the distances for all documents

The next step is to get all of the articles. These are stored in the nearest variable which holds a dataframe of all articles. A distances column is then created to store all of the calculated distances between the document and every other document in the corpus. The dataframe is then sorted in ascending order based on distance. Since the smaller the distance score is the more similar the article, the articles that are sorted at the top are the nearest articles.

```
nearest['distance'] = distances
nearest.drop(labels=aid-1, axis=0, inplace=True)
nearest.sort_values(by=['distance'], inplace=True)
```

Figure X: Sorting the articles by distance

User Input Abstract's Similar Articles

_____Content-based recommendations are also done when a user asks for a Hybrid recommendation. This will detail what happens in the content-based recommender side before sending it to the hybrid recommendations.

The first thing that happens after the user asks for a hybrid recommendation is to preprocess the abstract the same way all other articles are preprocessed. Words that are part of the vocabulary are retained and are then sent back as an array of pruneIds.

```
doc = np.array(vectorizer.vectorize_input(abstract))
```

Figure X: Vectorize the input

Once the input has been vectorized, a document term matrix for this abstract is created with random topics for every word.

```
rand_topics = np.random.randint(0,high=self.K,size=len(doc))
doc_dtm = pd.DataFrame({'abstractId': [self.D] * len(doc), 'pruneId': doc, 'topic': rand_topics})
```

Figure X: Create temporary dataframe for this document

Once the document term matrix is created, we are now able to get the document-topic and topic-term count matrices and then send the document-term matrix to be trained the same way we trained the whole corpus. This is done to get the theta or document-topic distributions of this inputted abstract.

```
doc_doctopic, doc_topicterm = self._get_suggestion_counts(doc_dtm)
doc_dtm = self._workers(doc_dtm, doc_doctopic, doc_topicterm, 100)
```

Figure X: Get document-topic count and topic-term count matrices

After the final topics of every word in the inputted abstract are found, the next thing to do is to find the theta scores of the document. The theta scores will then be used to get the distances between the input abstract and all the other articles in the corpus.

```
num = np.array([doc_doctopic[self.D, k] + self.alpha for k in range(self.K)])
den = doc_doctopic[self.D].sum() + (self.alpha*self.K)
doc_theta = num/den

distances = self._get_distances(doc_theta)
```

Figure X: Getting theta and distances

Once the distances are taken, they are appended as a column to the dataframe of all articles. The articles are then sorted by distance in ascending order. This is then sent to the Hybrid recommender.

```
nearest = pd.read_sql(query, sa.create_engine(self.engine))
nearest['distance'] = distances
nearest.sort_values(by=['distance'], inplace=True)
```

Figure X: Sorting articles by distance

User-based Collaborative Recommendation

Using the user and item features learned from Matrix Factorization, we can create suggestions for a specific user. This can be done by conducting matrix multiplication on the specific user's features and the article's features.

```
def get_user_recommendations(self, user_id: int, num_suggestions: int):
    if user_id <= 0:
        return {'articleIds': np.array([]), 'scores': np.array([])}
    else:
        num_features = 8
        user_features = self.get_specific_user_features(user_id, num_features)
        if (np.all(user_features == 0)):
            return self.get_most_viewed_articles(num_suggestions)

        item_features = self.item_features

        actual_scores = np.dot(user_features, item_features)
        transformed_scores = self.affine_transformation(actual_scores, 0, 6, 0, 1)
        df = pd.DataFrame(
          {'articleId': np.array(range(len(item_features[0]))) + 1,
           'actual_score': actual_scores,
           'transformed_score': transformed_scores}
        )
        df = df.iloc[self.valid_article_ids]
                .sort_values('actual_score', ascending=False).head(num_suggestions)

        select_query = "SELECT a.* FROM article a, clean_abstract b
            WHERE a.articleId = b.articleId AND a.articleId IN :articleIds;"
        queried_articles = pd.read_sql(text(select_query), self.engine,
            params={'articleIds': df['articleId'].to_list()})
        merged = pd.merge(left=df, right=queried_articles, on='articleId')

        return merged.to_dict(orient="records")
```

Figure X: Function that returns suggestions given User ID

However, due to utilizing the values 0.5, 1, 3, and 5 when creating the aggregated dataset, the numbers aren't uniform as we will be utilizing these results for the hybrid recommender. In order to have a uniform score, we utilize affine

transformation to transform the values into a range of 0 to 1, making it possible to be utilized in the hybrid recommender while retaining the distance between the ratings.

```
def affine_transformation(self, input, orig_range_a, orig_range_b, target_range_c, target_range_d):
    return (input - orig_range_a) * ((target_range_d - target_range_c) /
                                     (orig_range_b - orig_range_a)) + target_range_c
```

Figure X: Transforming the different scores to a range between 0 to 1

The result is a matrix that contains the predicted scores the user will give on every article in the database. In other words, the output is a user-article matrix that contains the scores of every article—with higher scores meaning that the user is likely to like that article.

```
[
  {'articleId': 8985, 'actual_score': 3.223018995447891, 'transformed_score': 0.5371698325746485},
  {'articleId': 6249, 'actual_score': 3.2138310144865594, 'transformed_score': 0.5356385024144266},
  {'articleId': 13988, 'actual_score': 3.178170782511125, 'transformed_score': 0.5296951304185208},
  {'articleId': 12449, 'actual_score': 3.165769388254465, 'transformed_score': 0.5276282313757441},
  {'articleId': 7022, 'actual_score': 3.1036606160108793, 'transformed_score': 0.5172767693351465},
  {'articleId': 7608, 'actual_score': 3.093515928798911, 'transformed_score': 0.5155859881331518},
  {'articleId': 12671, 'actual_score': 3.0910023202366785, 'transformed_score': 0.5151670533727797},
  {'articleId': 11277, 'actual_score': 3.090671702034479, 'transformed_score': 0.5151119503390797},
  {'articleId': 5617, 'actual_score': 3.058872031159101, 'transformed_score': 0.5098120051931835},
  {'articleId': 9456, 'actual_score': 3.055376248504034, 'transformed_score': 0.5092293747506723}
]
```

Figure X: Sample Recommendations for a given user, along with scores

Hybrid Recommendation

Now that we have both recommender systems working individually, the final step is to hybridize their results by doing a matrix multiplication of the User-Article matrix from the user-based collaborative recommendation and Article-Topic matrix (Theta) from the content-based recommendation.

First, users will have to input the abstract of their choice and this will be used to get suggestions from the content-based recommender. Articles in the corpus will be scored based on the similarity of their abstracts to the inputted abstract.

	articleId	distance
2713	7184	0.185774
2181	6645	0.197121
3783	8305	0.210291
600	4722	0.217680
10	4048	0.220605
1273	5634	0.234323
1529	5983	0.237064
2078	6542	0.238123
4183	8713	0.238360
4520	9053	0.239819
1420	5819	0.242332
280	4384	0.243532
1321	5700	0.247565
3432	7947	0.248002
1455	5856	0.252534
4237	8767	0.255037
4472	9005	0.255406
4190	8720	0.255512
2321	6789	0.260522
2985	7459	0.263150
3657	8179	0.264327
4378	8911	0.264824
2645	7116	0.265764
4087	8614	0.268715
3362	7843	0.270250

Figure X: Article scores based on input abstract

Scores less than or equal to 0.3 (greater than or equal to 70%) will be shortlisted. The new list will be used to filter the aforementioned matrices above.

User-To-Article

0	10	280	459	600	951	1273	1321	...	4237	4378	4442	4453	4472	4520	4670
1	0.368671	0.399752	0.346458	0.290399	0.182089	0.298513	0.244822	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0

[1 rows x 46 columns]

Figure X: Hybrid User-Article

Article-To-Topic

	0	1	2	3	4	5	6	7	8	9
10	0.001408	0.001408	0.071831	0.001408	0.212676	0.036620	0.001408	0.212676	0.212676	0.247887
280	0.001639	0.001639	0.042623	0.001639	0.411475	0.042623	0.083607	0.124590	0.083607	0.206557
459	0.001515	0.001515	0.115152	0.039394	0.153030	0.153030	0.039394	0.115152	0.266667	0.115152
600	0.048113	0.000943	0.071698	0.024528	0.213208	0.024528	0.048113	0.118868	0.378302	0.071698
951	0.003226	0.003226	0.003226	0.083871	0.245161	0.083871	0.003226	0.083871	0.406452	0.083871
1273	0.002985	0.002985	0.002985	0.002985	0.450746	0.077612	0.002985	0.226866	0.152239	0.077612
1321	0.062733	0.031677	0.062733	0.016149	0.124845	0.031677	0.031677	0.233540	0.357764	0.047205
1368	0.001099	0.001099	0.028571	0.001099	0.083516	0.165934	0.056044	0.275824	0.248352	0.138462
1420	0.001869	0.142056	0.001869	0.001869	0.235514	0.048598	0.001869	0.142056	0.235514	0.188785
1455	0.001786	0.046429	0.091071	0.001786	0.403571	0.001786	0.001786	0.046429	0.314286	0.091071
1529	0.002985	0.002985	0.002985	0.002985	0.152239	0.002985	0.002985	0.152239	0.376119	0.301493
1546	0.056522	0.056522	0.002174	0.002174	0.328261	0.110870	0.110870	0.110870	0.110870	0.110870
1593	0.001460	0.037956	0.074453	0.037956	0.074453	0.074453	0.110949	0.147445	0.366423	0.074453
1696	0.001408	0.001408	0.071831	0.036620	0.388732	0.071831	0.036620	0.036620	0.318310	0.036620
1896	0.001408	0.001408	0.001408	0.071831	0.212676	0.036620	0.177465	0.177465	0.107042	0.212676
2078	0.103401	0.001361	0.001361	0.001361	0.171429	0.035374	0.035374	0.137415	0.341497	0.171429
2175	0.087069	0.022414	0.043966	0.022414	0.194828	0.130172	0.087069	0.108621	0.194828	0.108621
2180	0.000922	0.047005	0.070046	0.000922	0.208295	0.070046	0.000922	0.392627	0.185253	0.023963
2181	0.025121	0.025121	0.073430	0.000966	0.290821	0.049275	0.000966	0.194203	0.242512	0.097585
2321	0.002299	0.002299	0.059770	0.002299	0.174713	0.002299	0.002299	0.059770	0.519540	0.174713
2436	0.001786	0.001786	0.046429	0.046429	0.180357	0.001786	0.001786	0.314286	0.135714	0.269643
2645	0.001786	0.001786	0.046429	0.001786	0.269643	0.001786	0.001786	0.225000	0.448214	0.001786
2713	0.001460	0.037956	0.037956	0.001460	0.147445	0.001460	0.074453	0.110949	0.366423	0.220438

Figure X: Hybrid Article-Topic

The matrix product of the two matrices is called the User-Topic matrix.

User-To-Topic

	0	1	2	3	4	5	6	7	8	9
1	0.173335	0.186151	0.389332	0.109685	2.056785	0.465013	0.507094	1.56492	2.547311	1.166066

Figure X: Hybrid User-Topic

After getting the User-Topic matrix. We need to consider the top 3 topics based on their scores to proceed to the next step.

Top 3 topics	
8	2.547311
4	2.056785
7	1.564920

Figure X: Top 3 topics

	8
4378	0.541176
2321	0.519540
2645	0.448214
3679	0.440659
951	0.406452
600	0.378302
1529	0.376119
3459	0.370428
3525	0.366551
1593	0.366423
2713	0.366423
3783	0.360479
4237	0.358929
1321	0.357764
2078	0.341497
3432	0.329508
1696	0.318310
1455	0.314286
2947	0.307317

Figure X: Theta scores for top topics

In every topic, each article has a corresponding score and in order to decide which articles will be recommended to the user, we consider only articles with scores greater than 40%. After this, we will now get the details of the recommended articles to be displayed in the user interface.

TESTING PROCESS

Here the processes are going to be tested for their accuracy and their performance. The tests that are going to be done are Topic Coherence for the Topic model, Mean Squared Error for the User-based Collaborative Filtering model, and a performance test to see how fast the system is able to generate a recommendation.

Topic Coherence is a way to evaluate how interpretable topics are. Since the topics in the topic model are not created using a supervised model, there is no way to know what the topics mean therefore there is no easy way to understand if the topics make sense. There are many different coherence measures like perplexity, CV, UMass and UCI that all measure topic coherence for topic models. In this model, the UMass coherence function will be used. UMass is an internal topic coherence evaluation that measures semantic similarity between the top words in a topic. In this test, the top N words of every topic are evaluated to get its coherence score. An average of all topic coherence scores is taken to get the overall coherence score of the model.

Mean Squared Error is used to find the “error” between the original user-item matrix and the predicted user-item matrix. This is done because the goal of Matrix Factorization is to find the factors that can recreate the original matrix with as few errors as possible and at the same time give “predictions” to the unobserved cells of the matrix. The lower the error between the original matrix and the predicted matrix, the more we can say that the factors can be an intrinsic representation of the latent features of the user-item matrix.

A performance test is done to check the speed at which the model is able to send recommendations to the users. This makes sure that the users have a smooth experience when asking for recommendations from the model.

Accuracy Testing

One use of topic coherence is to find the best number of topics to use for the model. One way to approach this is to plot the average coherence scores of the model in a graph and use the so-called elbow technique to find an optimal number of topics. The usual behavior of topic coherence scores is that it tends to increase as the number of topics increase. This increase tend to become smaller as the number of topics go higher and the elbow technique tells us that we pick the number of topics where there the increase of the topic coherence scores decrease.

The first test is to find out the number of topics that will be used in the topic model. The model is run multiple number of times. Each new training is done with a different topic number. Once the topic coherence scores start to stabilize, we can find out the number of topics that would be best used for the topic model.

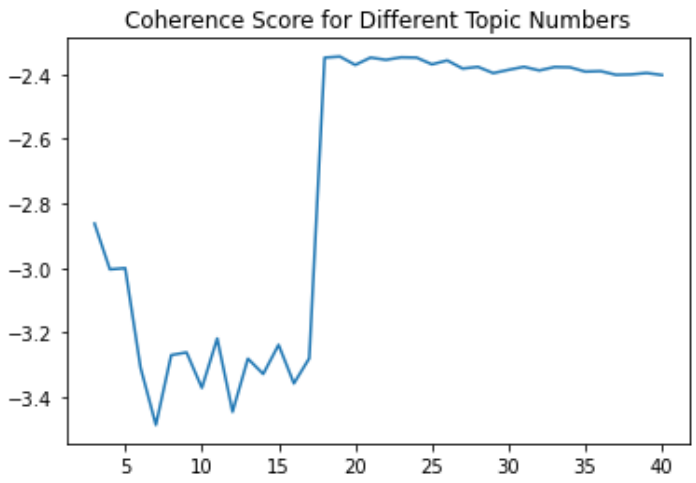


Figure X: Coherence Scores for different Topic Numbers

It is evident in the graph that as the number of topics approach 20, there is a significant increase in the topic coherence score. After it reaches it's highest topic coherence score, the subsequent models start to stabilize and it's scores start to also slowly go down. Using the aforementioned elbow technique, we can find the point at which the model does stops gaining marginal increases, and use that as the number of topics for out model. In this case, the number of topics to be used is 19.

The next test would be to figure out the optimal alpha and beta values for the topic model.

	modelId	alpha	beta	topics	iterations	avg_coherence
55	498	0.03	0.6	19	100	-2.32279
53	496	0.01	0.6	19	100	-2.32439
75	518	0.03	0.8	19	100	-2.33224
20	463	0.08	0.2	19	100	-2.33338
28	471	0.06	0.3	19	100	-2.33484

Figure X: Alpha and Beta testing sorted descending order

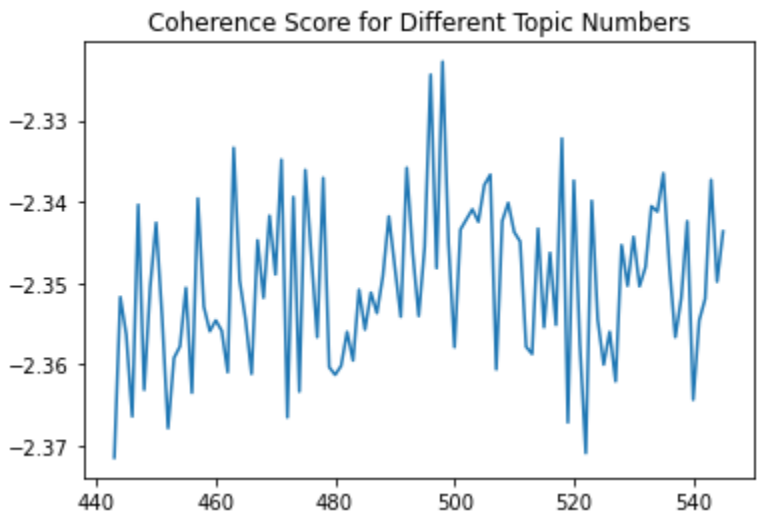


Figure X: Alpha and Beta test graphs

Looking at both the sorted table (based on descending coherence scores) and the graph, we can see that a beta value of 0.6 and an alpha value of 0.3 score the highest in this test.

Remember: an alpha value determines how much topics are distributed within a document, the higher the number the higher the number of topics found in a document are. Similarly, a beta value determines how much words can be found across a topic. A 0.6 beta score would theoretically be ideal in this scenario where we expect a lot of overlap in words in the computing field. A 0.3 alpha score would also be ideal since we expect there to be some sort of overlap between 2-3 topics in each document. We could possibly find both AI and Networking topics in a single document, for example, and the 0.3 alpha value would be able to represent that.

Mean Squared Error (User-based Collaborative Filtering)

In Matrix Factorization, the mean squared error is used to find how close the predicted matrix is to the original matrix. This is done by subtracting the cells in the predicted matrix to the original matrix, then squaring the result and finally getting the mean. Due to its nature of squaring the difference, it will always return a 0 or positive number. By conducting Mini-Batch Gradient Descent, we slowly head towards the cost function's minima, slowly lowering the Mean Squared Error over the course of the training. For this model's parameters, we used a mini-batch of 256 data with 8 features. After 50 epochs, we end with a model with an MSE of 1.0513. Even though the MSE isn't less than 0, it is OK since the model might become overfitted to the dataset—making it unable to deal with future data.

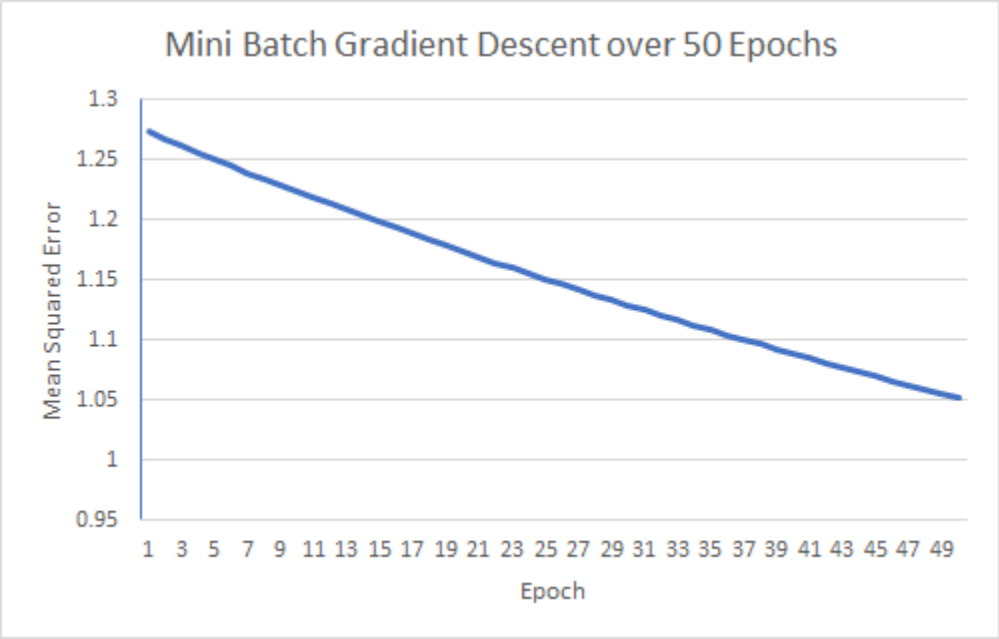
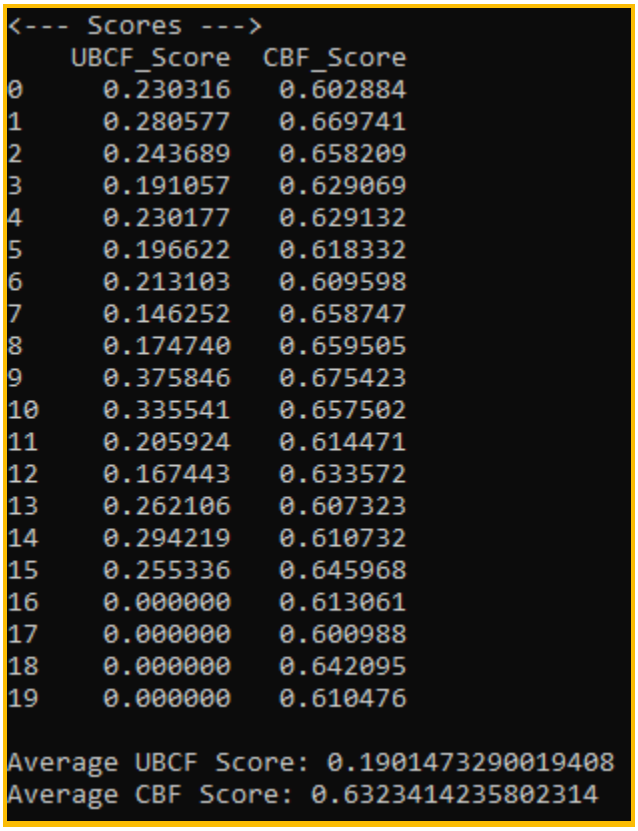


Figure X: Mini Batch Gradient Descent over 50 Epochs

Weighted Hybrid Recommendations Experiment

The prediction score is computed using the weighted technique, which considers all recommendation approaches as variables in a linear combination to calculate the prediction score. Each score for a corresponding article in the corpus will be given a specific amount of weight to determine the articles to be recommended.



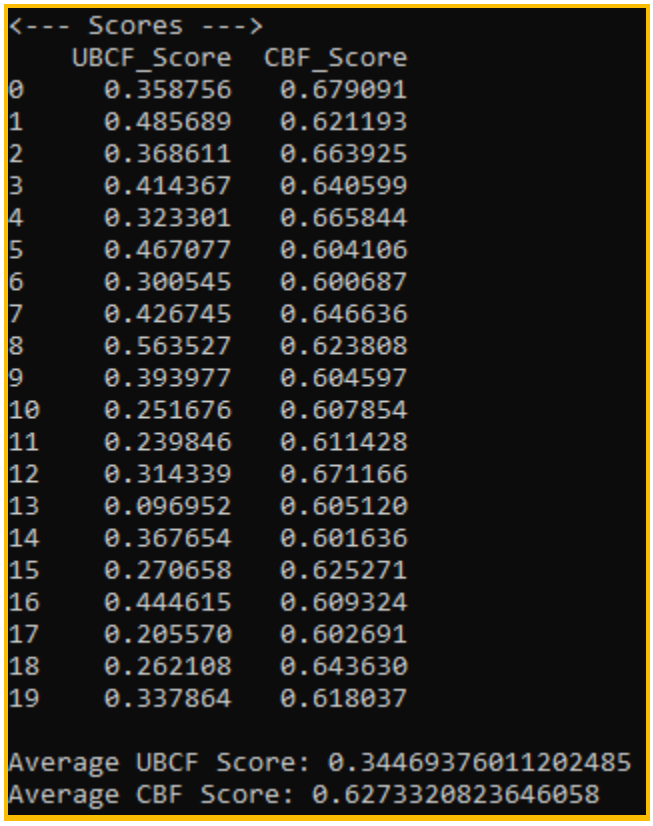
```
<--- Scores --->
    UBCF_Score  CBF_Score
0     0.230316   0.602884
1     0.280577   0.669741
2     0.243689   0.658209
3     0.191057   0.629069
4     0.230177   0.629132
5     0.196622   0.618332
6     0.213103   0.609598
7     0.146252   0.658747
8     0.174740   0.659505
9     0.375846   0.675423
10    0.335541   0.657502
11    0.205924   0.614471
12    0.167443   0.633572
13    0.262106   0.607323
14    0.294219   0.610732
15    0.255336   0.645968
16    0.000000   0.613061
17    0.000000   0.600988
18    0.000000   0.642095
19    0.000000   0.610476

Average UBCF Score: 0.1901473290019408
Average CBF Score: 0.6323414235802314
```

Figure X: No weights

For the base experiment, no weights are added in both user-article and article-topic matrices. After performing the matrix multiplication of the two matrices and getting the top 3 topics, the hybrid recommendations are produced with their

corresponding scores in user-based collaborative filtering and content-based filtering shown in the figure above. We can see that the average score for the user-based collaborative filtering is 19% and 63% for the content-based filtering.



```
<--- Scores --->
    UBCF_Score  CBF_Score
0     0.358756   0.679091
1     0.485689   0.621193
2     0.368611   0.663925
3     0.414367   0.640599
4     0.323301   0.665844
5     0.467077   0.604106
6     0.300545   0.600687
7     0.426745   0.646636
8     0.563527   0.623808
9     0.393977   0.604597
10    0.251676   0.607854
11    0.239846   0.611428
12    0.314339   0.671166
13    0.096952   0.605120
14    0.367654   0.601636
15    0.270658   0.625271
16    0.444615   0.609324
17    0.205570   0.602691
18    0.262108   0.643630
19    0.337864   0.618037

Average UBCF Score: 0.34469376011202485
Average CBF Score: 0.6273320823646058
```

Figure X: CF - 1.3, CBF - 1.7

For the second experiment, weights are added in both user-article and article-topic matrices. 1.3 for the user-based collaborative filtering and 1.7 for the content-based filtering. After performing the matrix multiplication of the two matrices and getting the top 3 topics, the hybrid recommendations are produced with their corresponding scores in user-based collaborative filtering and content-based filtering shown in the figure above. We can see that the average score for the user-based collaborative filtering is 34% and 63% for the content-based filtering.

```
<--- Scores --->
    UBCF_Score  CBF_Score
0     0.244918   0.617472
1     0.360187   0.635530
2     0.330002   0.646814
3     0.254432   0.602509
4     0.296429   0.619217
5     0.268342   0.626230
6     0.428449   0.619710
7     0.406543   0.619699
8     0.505706   0.626401
9     0.371680   0.615970
10    0.299230   0.605100
11    0.513447   0.620492
12    0.300301   0.632168
13    0.293502   0.611407
14    0.455827   0.606677
15    0.186501   0.621507
16    0.444890   0.629383
17    0.400208   0.630965
18    0.372417   0.602643
19    0.382485   0.611788

Average UBCF Score: 0.35577480483829016
Average CBF Score: 0.6200840449503545
```

Figure X: CF - 1.3, CBF - 3

For the third experiment, weights are added in both user-article and article-topic matrices. 1.3 for the user-based collaborative filtering and 3 for the content-based filtering. After performing the matrix multiplication of the two matrices and getting the top 3 topics, the hybrid recommendations are produced with their corresponding scores in user-based collaborative filtering and content-based filtering shown in the figure above. We can see that the average score for the user-based collaborative filtering is 36% and 62% for the content-based filtering.

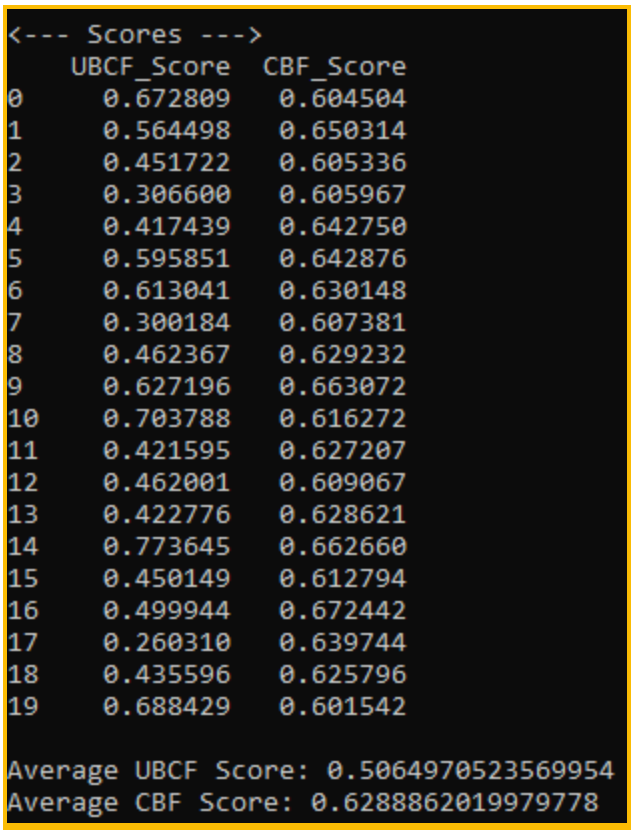


Figure X: CF - 2, CBF - 2.5

For the fourth experiment, weights are added in both user-article and article-topic matrices. 2 for the user-based collaborative filtering and 2.5 for the content-based filtering. After performing the matrix multiplication of the two matrices and getting the top 3 topics, the hybrid recommendations are produced with their corresponding scores in user-based collaborative filtering and content-based filtering shown in the figure above. We can see that the average score for the user-based collaborative filtering is 51% and 63% for the content-based filtering.

From the four experiments tested, only the scores of user-collaborative filtering changed while the scores of content-based filtering remains consistent.

Performance Testing

Here the performance speeds of the different models are tested to check if they are reasonable. The first test is to check the training time of the Topic model. The training time can be called reasonable when it doesn't take a very long time to train the model. Furthermore, when there are changes in the hyperparameters like topics and iterations, the increase in training shouldn't be more than 100% increase when increasing the hyperparameters by 100% as well.

Test Case I - Training the Topic Model

	Linear Computing	Parallel Computing	Parallel Computing	Parallel Computing
Topics	10	10	10	20
Iterations	100	100	200	100
Time	78.64 Minutes	15.03 Minutes	29.85 Minutes	21.98 Minutes

Figure X: LDA Training time

The table above illustrates that when trained sequentially in a linear computing pattern, a topic model with 10 topics that is trained for 100 iterations will finish in approximately 79 minutes. When the model's training is moved to parallel computing, we are able to diminish the training time by more than 5-times the original training time.

When the topics, or even the iterations, are increased two-fold, the increase in time is almost two-fold as well, with the increase in number of topics just lower with a

1.4-times increase. Given the results of this test, it is clear that parallel computing, the method being used in this topic model, is very reasonable as it scales pretty well when the hyperparameters are changed. Apart from that, the base training time of 10 topics and 100 iterations only take 15 minutes to complete compared to the approximately 79 minutes it would take to finish a sequential, linear approach.

Test Case II - Hybrid Recommendations

The second test case involves measuring the speed that the hybrid model gives recommendations. A recommendation time can be considered reasonable if it doesn't take more than 10-20 seconds to give the recommendations. This is because the model has to first analyze the given abstract and match it with the most similar articles in the corpus. Therefore there is a time when the model analyzes, and then a time when the model gives suggestions.

For this test case, 10 published articles from Google Scholar are gathered. Each article is timed for how long the hybrid recommendation takes and the number of articles recommended.

Article No.	No. of articles recommended	Time Elapsed
1	22	4.175 seconds
2	21	4.325 seconds
3	71	4.448 seconds
4	18	4.454 seconds
5	12	3.795 seconds
6	73	4.432 seconds

7	17	4.202 seconds
8	15	4.285 seconds
9	13	4.219 seconds
10	14	4.578 seconds
Average	27.6	4.219 seconds

Figure X: Hybrid Recommendation performance

CHAPTER IV

SUMMARY, CONCLUSION, AND RECOMMENDATION

Summary of Findings

Though the topic coherence scores for alpha and beta testing were variably different, we could still pinpoint that using 0.6 as the beta value and 0.03 as the alpha value would yield the highest topic coherence score.

Conversely, the testing for the different number of topics was more or less expected, with the graph going up as the topics went up past 15 topics and then stabilizing at it’s peak.

Pruning the vocabulary has shown to be very impactful in the Latent Dirichlet Allocation process. Only removing words that appeared in 0.5% documents in the corpus showed that topics would have the same top words. These words were all words

that appeared in more than 10% of the documents. These words become useless as the model finds these words too common in the corpus, and ends up filling all the topics with these words since they appear to be the most statistically important words. Removing the highly occurring words helped the topic model find much more sensible topics with different top words appearing in different topics.

For Gradient Descent, there are three different kinds—Batch (BGD), Stochastic (SGD), and Mini-batch Gradient Descent (Mini-BGD) and they all have different tradeoffs. One epoch of BGD takes the longest but converges to the local minima faster. One epoch of SGD is quick, but the loss constantly fluctuates and might go in and out of the local minima. Mini-BGD is right between the two in terms of computational time and convergence.

This begs the question, which one to use? Testing out Batch Gradient Descent takes 8 seconds per epoch, with a current dataset of 30 users, 4000 articles, and 300 entries for user-article interactions. 150 Iterations would take an approx. 23 minutes which lowers the cost function to a favorable amount. This is already fast for the initial user and item features, and there is currently no need to train the initial model faster. Thus, opting for Batch Gradient Descent in this scenario is most likely the best scenario. However, as the dataset scales and as more users, articles, and interactions are added, it might be good to consider switching to Mini-batch Gradient Descent.

As for Matrix Factorization, it proved well once the dataset was established and the model trained. However, once updates were introduced into the dataset via the users' interactions with the application, things became iffy. Since the rating of the item

would be changed upon an interaction, this meant that the model would have to be updated once a change was introduced—hence, the model would have to be trained again, albeit not fully. This, however, is not ideal—imagine having multiple users utilizing the application, all conducting different interactions. This means that the dataset would have to be updated every time an interaction occured—which means multiple model training would be happening.

Conclusion

Using Latent Dirichlet Allocation, an unsupervised topic modeling algorithm, to create a topic model and find topics of documents creates an easy way to automatically categorize documents quickly and easily. LDA's results are based heavily on many factors including hyperparameters, corpus size, and vocabulary quality. LDA works well and gives better results when given a larger corpus and a better vocabulary through pruning. Hyperparameters are chosen based on the purpose of the algorithm. Since LDA needs to recommend similar articles in this application, a larger amount of topics is necessary in order to give proper suggestions.

As for User-based Collaborative Filtering, Matrix Factorization was used in order to learn the latent factors of the user-item matrix. This seemed ideal for recommending at first—being able to intrinsically learn the preferences of users and the likeness of items is an amazing feat. However, an unnoticed flaw showed up later on—having users that continuously interact with the user-item matrix meant that training would have to be conducted upon every interaction. If this wasn't done, the recommender system wouldn't be dynamic. This didn't mean that having a dynamic recommender with Matrix Factorization was unfeasible but hard nonetheless. Ergo, it turned out that Matrix Factorization isn't the optimal algorithm when it comes to volatile datasets.

Ultimately, combining the above-mentioned filtering algorithms will fulfill the hybrid approach of solving the problems of each individual algorithm, especially the cold-start problem. When one algorithm is not doing well with recommending articles to the user, the other algorithm will be able to solve the issue. There is also a possibility

that both algorithms will not do well and cannot recommend useful articles, then that will also be the time the hybrid recommender will choose to exclude it from the list.

Recommendation

Articles are plenty and they cover a plethora of fields and studies. Aside from Computer Science, researchers could create a model that utilizes a wide array of topics. The number of articles from different areas of specialization must be significant enough for the model to be able to pick up on these topics.

Another thing to consider would be to consider the nature of the dataset—are new users added frequently, meaning, is it dynamic? Are there constant updates to the dataset? If so, how frequent are these updates? Pondering such things before zoning on an algorithm would be the best course of action, as choosing a suboptimal algorithm would drastically affect the recommender's performance and ease of implementation.

To make the recommender system flexible, future researchers may include an algorithm to translate non-English articles to English to increase the variety of research to be recommended. Users can input articles that are not written in English and will receive article recommendations that can be English or the language specified or even both.

BIBLIOGRAPHY

APPENDICES

CURRICULUM VITAE

PERSONAL INFORMATION

Name	**:**	Kristoffer Francis E. Milan
Nickname	**:**	Milan
Date of Birth	**:**	March 17, 1999
Address	**:**	Bulacao, Cebu City, Cebu
Status	**:**	Single
Citizenship	**:**	Filipino
Religion	**:**	Roman Catholic
Email Address	**:**	kfemilan@gmail.com
Contact Number	**:**	09164524878

ACADEMIC INFORMATION

Tertiary	**:**	University of San Jose - Recoletos (2018-2022)
Secondary	**:**	University of San Jose - Recoletos (2012-2018)
Primary	**:**	University of San Jose - Recoletos (2007-2012)

PROFESSIONAL SKILLS

Language	**:**	C, Java, Python, JavaScript, Dart/Flutter

Software : MS Office, Visual Studio Code, MySQL, GitHub

Operating Systems: Windows

PERSONAL INFORMATION

Name	:	Javin Stefan R. Tan
Nickname	:	Habin
Date of Birth	:	March 29, 1999
Address	:	Punta Princesa, Cebu City, Cebu, Philippines
Status	:	Single
Citizenship	:	Filipino
Religion	:	Roman Catholic
Email Address	:	javin.tan37@gmail.com
Contact Number	:	0917 163 1338

ACADEMIC INFORMATION

Tertiary	:	University of San Jose Recoletos	(2018 - 2022)
Secondary	:	Don Bosco Technology Center	(2012 - 2018)
Primary	:	Don Bosco Technology Center	(2006 - 2012)

PROFESSIONAL SKILLS

Language	:	C, Java, HTML, CSS, Angular, Python, Flutter (Dart)
Software	:	MS Office, Visual Studio, VSCode, Netbeans, Postman, Git
OS	:	Windows

PERSONAL INFORMATION

Name	:	Jules Russel A. Lucero
Nickname	:	Jules
Date of Birth	:	March 4, 2000
Address	:	Soliman Street, Poblacion, Argao, Cebu
Status	:	Single
Citizenship	:	Filipino
Religion	:	Roman Catholic
Email Address	:	russelucero@gmail.com
Contact Number	:	0906 413 2581

ACADEMIC INFORMATION

Tertiary	:	University of San Jose Recoletos (2018-2022)
Secondary	:	Argao National High School (2012-2018)
Primary	:	Argao Central Elementary School (2006-2012)

PROFESSIONAL SKILLS

Language	:	C, Java, HTML, CSS, JavaScript, Python, Flutter (Dart)
Software	:	Microsoft Office, Netbeans, Visual Studio, Visual Studio Code, Postman, Git
OS	:	Windows