Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

The Next Generation Personal Research Assistant

Author:
Julian Wong

Supervisor:
Dr Sergio Maffeis

Second marker:
Prof. Alessio Lomuscio

June 18, 2023

Submitted in partial fulfillment of the requirements for the MEng Honours Degree in Computing (Artificial Intelligence and Machine Learning) Part IV of Imperial College London

Abstract

With the exponential increase in the number of academic publications every year [1], discovering relevant academic papers for research is becoming increasingly difficult. While filtering through hundreds of search engine results by hand is immensely inconvenient and taxing, relying entirely on artificial intelligence (AI) and large language models (LLMs) has concerning implications towards result accuracy, recommendation bias, and other factors beyond the user's control. Recommender systems for academic and research papers have been a recent growing area of research, where automated algorithms source and generate relevant papers based on keywords, authors, or entire papers as targets. However, existing solutions suffer from problems such as having to manage large proprietary research databases and thus not being data source independent, and overly relying on proprietary research information [2]. Furthermore, many existing solutions require users to either manually type search queries through websites or rely on web APIs. Therefore, there exists an opportunity to integrate an improved recommender system within some form of research browser. This project report describes the implementation of a lightweight, scalable, and source-independent academic resource recommender system, in the form of an integrated plugin extension for the Zotero research paper management platform. The system utilises state-of-the-art natural language processing techniques including word embedding vectors [3], TF-IDF [4], and TextRank [5], as well as statistical recommendation techniques including co-occurrence relations between papers [6]. Its software architecture is outlined, and in-depth design and engineering decisions are explained and justified. The system is quantitatively evaluated and compared with other existing systems in terms of performance and recommendation quality.

Acknowledgments

I would like to thank my supervisor, Dr Sergio Maffeis, for his enduring patience and academic guidance throughout this project. He has provided me with continuous feedback that have helped shape the direction of the resultant end-product and inspire ideas that have led to significant improvements.

I would also like to thank Angel, my closest friends, and my peers at Imperial, who have been a constant source of motivation for me, and with whom I feel honoured to have shared many of the highlights and happiest moments of my student life.

Lastly, I am immensely grateful to my family for their unwavering love and encouragement throughout this project and over the course of my studies. The completion of this project would not have been possible without them.

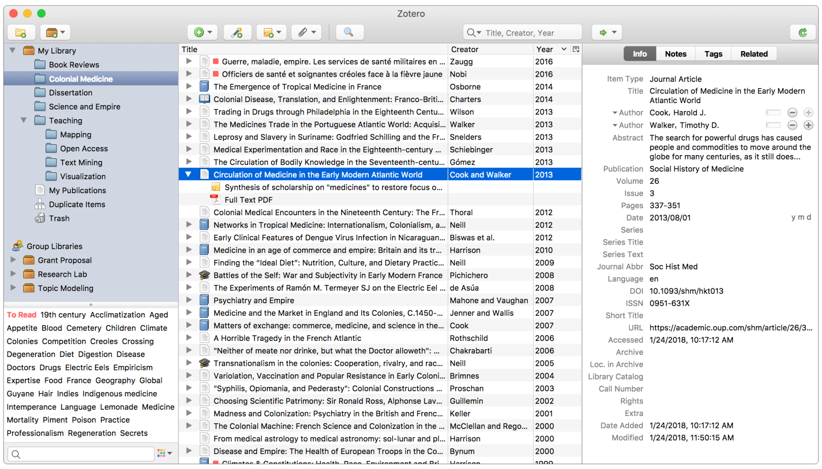
Contents

Chapter 1

Introduction

Zotero is a free-to-use, open-source research paper management platform that aims to help its users collect, organise, annotate, cite and share research [7]. The Zotero platform mainly consists of a client-side application that displays a library of collections of research papers and resources that can be synced and modified online via a web library. Figure 1.1 shows an example of what the main interface of the Zotero client looks like.

Aside from the essential functionality that Zotero offers out of the box, including tagging, annotations, bibliography generation, and web browser extensions for saving online resources, there is also an active community of Zotero users who develop extensions for the open-source Zotero client [8] using Zotero's internal JavaScript API [9], using technologies similar to those found on legacy Firefox extensions [10]. The main areas of development of these extensions can be broadly generalised into categories including metadata aggregation for research papers (e.g. Zotero Citation Counts Manager [11]), attachment file management for PDFs (e.g. ZotFile [12]), advanced graphical annotations (e.g. Better Notes [13]), and integration with other platforms such as Microsoft



Zotero
Title, Creator, Year
My Library
Book Reviews
Colonial Medicine
Dissertation
Science and Empire
Teaching
Mapping
Open Access
Text Mining
Visualization
My Publications
Duplicate Items
Trash
Group Libraries
Grant Proposal
Research Lab
Topic Modeling
To Read 19th century Acclimatization Aged Appetite Blood Cemetery Children Climate Colonies Competition Creoles Crossing Degeneration Diet Digestion Disease Doctors Drugs Electric Eels Empiricism Expertise Food France Geography Global Guyane Hair Indies Indigenous medicine Intemperance Language Lemonade Medicine Mortality Piment Poison Practice Professionalism Regeneration Secrets
Info
Notes
Tags
Related

Title	Creator	Year
Guerre, maladie, empire. Les services de santé militaires en ...	Zaugg	2016
Officiers de santé et soignantes créoles face à la fièvre jaune	Nobi	2016
The Emergence of Tropical Medicine in France	Osborne	2014
Colonial Disease, Translation, and Enlightenment: Franco-Briti...	Charters	2014
Trading in Drugs through Philadelphia in the Eighteenth Centu...	Wilson	2013
The Medicines Trade in the Portuguese Atlantic World: Acquisi...	Walker	2013
Leprosy and Slavery in Suriname: Godfried Schilling and the Fr...	Snelders	2013
Medical Experimentation and Race in the Eighteenth-century ...	Schiebinger	2013
The Circulation of Bodily Knowledge in the Seventeenth-centu...	Gómez	2013
Circulation of Medicine in the Early Modern Atlantic World	Cook and Walker	2013
Synthesis of scholarship on "medicines" to restore focus o...		
Full Text PDF		
Colonial Medical Encounters in the Nineteenth Century: The Fr...	Thoral	2012
Networks in Tropical Medicine: Internationalism, Colonialism, a...	Neill	2012
Early Clinical Features of Dengue Virus Infection in Nicaraguan...	Biswas et al.	2012
Medicine in an age of commerce and empire: Britain and its tr...	Harrison	2010
Finding the "Ideal Diet": Nutrition, Culture, and Dietary Practic...	Neill	2009
Battles of the Self: War and Subjectivity in Early Modern France	Pichichero	2008
The Experiments of Ramón M. Termeyer SJ on the Electric Eel ...	de Asúa	2008
Psychiatry and Empire	Mahone and Vaughan	2007
Medicine and the Market in England and Its Colonies, C.1450-...	Jenner and Wallis	2007
Matters of exchange: commerce, medicine, and science in the...	Cook	2007
A Horrible Tragedy in the French Atlantic	Rothschild	2006
"Neither of meate nor drinke, but what the Doctor alloweth": ...	Chakrabarti	2006
Transnationalism in the colonies: Cooperation, rivalry, and rac...	Neill	2005
Variolation, Vaccination and Popular Resistance in Early Coloni...	Brimnes	2004
"Syphilis, Opiomania, and Pederasty": Colonial Constructions ...	Proschan	2003
Choosing Scientific Patrimony: Sir Ronald Ross, Alphonse Lav...	Guillemin	2002
Madness and Colonization: Psychiatry in the British and Frenc...	Keller	2001
The Colonial Machine: French Science and Colonization in the ...	McClellan and Rego...	2000
From medical astrology to medical astronomy: sol-lunar and pl...	Harrison	2000
Disease and Empire: The Health of European Troops in the Co...	Bynum	2000

Item Type	Journal Article
Title	Circulation of Medicine in the Early Modern Atlantic World
Author	Cook, Harold J.
Author	Walker, Timothy D.
Abstract	The search for powerful drugs has caused people and commodities to move around the globe for many centuries, as it still does...
Publication	Social History of Medicine
Volume	26
Issue	3
Pages	337-351
Date	2013/08/01 y m d
Series	
Series Title	
Series Text	
Journal Abbr	Soc Hist Med
Language	en
DOI	10.1093/shm/hkt013
ISSN	0951-631X
Short Title	
URL	https://academic.oup.com/shm/article/26/3...
Accessed	1/24/2018, 10:17:12 AM
Archive	
Loc. in Archive	
Library Catalog	
Call Number	
Rights	
Extra	
Date Added	1/24/2018, 10:17:12 AM
Modified	1/24/2018, 11:50:15 AM

Figure 1.1: A screenshot of the Zotero 6.0 client user interface.

Word and LaTeX editors.

From this collection of extensions, it is apparent that whilst there is an abundance of support for the modification and analysis of existing resources within a user's library, there is lesser development surrounding the manipulation of external resources that are outside one's library, such as those online. Currently, one major limitation for Zotero users is the inability to conveniently find related resources that are worth exploring further based on their current research interests or the resources they are currently reading in their libraries. Whilst research recommender system solutions exist in various forms outside of the Zotero community, there is still a lack of integration with the Zotero client from a user interface and functionality perspective. Existing solutions are often data source-dependent, meaning the recommendations that these systems generate are limited to the resources that exist in their proprietary databases. Furthermore, these systems rely on information that may be considered proprietary to researchers, including the full text from research papers, citations, and researchers' in-depth user profiles [2].

Therefore, the aim of this project is to design and implement a lightweight, scalable, source-independent, and Zotero-integrated academic resource recommender system as a plugin extension for the Zotero client, and thus create an improved recommender system compared to existing solutions available to the community of Zotero users. This project report discusses existing solutions, designs, and algorithms that have enabled the creation of this recommender system, explains its design and in-depth implementation details, evaluates its performance and recommendation quality, compares and contrasts it with existing solutions, and discusses opportunities for further development and improvements.

Chapter 2

Background

This chapter compares and contrasts different types of generic recommendation methods, and explores the different architectures of existing research paper recommender system proposals, providing a comprehensive list of the advantages and disadvantages of using each approach.

Recommender systems for academic and research papers have been a growing area of research in recent years, and continue to be a future trending research interest among researchers, publishers and students in higher education. In particular, throughout the last decade, various methods of implementing such a system have been suggested and illustrated, most of which utilise the power of generic recommender systems, natural language processing, and machine learning.

Nascimento et al. [2] provided a comprehensive overview of the problem from a more generic, platform-independent perspective, and highlights the advantages and disadvantages of more novel approaches to implementing research paper recommender systems in the past. For example, whilst directly utilising a paper's references for recommenders is an efficient and simple method, citations are ultimately a human-generated piece of 'metadata' from the paper, and thus should not be overly relied upon either as a source for recommendation candidates or as a recommendation criteria. Furthermore, citations alone have no information about their specific relationship to the main concepts of a paper, some important references may not always be cited by the author(s), and citations can be considered proprietary information in some circumstances.

2.1 Recommender system approaches

Before diving into research-specific recommender systems, it is useful to first compare and contrast the major driving algorithms behind modern recommender systems –

collaborative filtering and content-based methods. This leads to the discussion of the feasibility and suitability of using each type for the use case of this project.

2.1.1 Collaborative filtering

It is often the case that recommender systems from other popular domains, especially multimedia content such as movies and music, rely on collaborative filtering. The objective of collaborative filtering was summarised mathematically from a high level by Adomavicius and Tuzhilin [14]. The main problem that a recommender system aims to solve is to generate an ordered list of items, sorted by their usefulness to the target user, or 'utility'. To achieve this, the system must first solve the problem of calculating or predicting the utility of some specified item. Formally, the problem space consists of a set of users C, a set of items S, and a utility function $u(c, s)$ that returns the utility of item s based on target user c, where $u : C \times S \rightarrow R$.

A collaborative approach first compares the target end-user with other end-users in C and selects the k most 'similar' end-users. The similarity of two users c_1 and c_2 is calculated based on the ratings both end-users have given in the past. Their ratings of items that both of them have rated in the past can be represented as two vectors x_1 and x_2, and their similarity score can be found using methods such as cosine similarity. Once a set of similar end-users C' has been collected, the utility $u(c'_i, s)$ of every similar end-user c'_i can then be calculated and used to predict the utility $u(c, s)$ of the target user c and target item s. For example, in memory-based approaches, one could predict the rating user c would give to item s by calculating the average rating other similar users have given to the same item, or:

$$r_{c,s} = \frac{1}{N} \sum_{c' \in C'} r_{c',s}$$

Optionally, the similarity score between the target user c and other user c' can be used as a weighting for how relevant their rating is to the prediction:

$$r_{c,s} = \frac{\sum_{c' \in C'} \text{sim}_{c',c} \times r_{c',s}}{\sum_{c' \in C'} \text{sim}_{c',c}}$$

A problem that arises from this method is the 'cold-start' [15], where the few number of users at the beginning of the lifecycle of a system means that the quality of recommendations is often poor due to a lack of data. This highlights a more fundamental concern regarding a collaborative approach, which is that it heavily depends on all other existing users on the system, and requires the system to have knowledge of each user's ratings and preferences. Relating this to Zotero integration, a collaborative approach would not be ideal as the Zotero platform is not a multi-user platform by design, unlike other online platforms where collaborative filtering would be the obvious approach, such as social media. For example, the Zotero web API requires authentication information such as a client key and client secret in order to access a given client's library and collections [16]. This data would rightly be considered proprietary and private by users, and should therefore not be incorporated into this project's recommender system. Furthermore, this would only be possible if the Zotero user had created an online account on the platform and registered their copy of the client application to their account, which may not always be the case, rendering this an infeasible approach.

2.1.2 Content-based methods

In contrast, content-based methods depend on the target user c and the set of all items S for prediction. The target user c would have a set of items S_c that they are interested in, or have rated highly of previously. Adomavicius and Tuzhilin [14] formalised that a vector ContentBasedProfile(c) could be generated based on the information from S_c, and is meant to represent the preferences of a user based on their interested content. The method could then iteratively generate a vector Content(s) representing the information of each item s in S, and compare the similarity between Content(s) and ContentBasedProfile(c). Therefore, the utility function $u(c, s)$ could be defined as:

$$u(c, s) = \text{score}(\text{ContentBasedProfile}(c), \text{Content}(s))$$

In practice, the vectors Content(s) and ContentBasedProfile(c) can be generated using natural language processing techniques, such as aggregating the term frequency-inverse document frequency (TF-IDF) measure [4] for every keyword in a document, or for every document in a user's profile.

As mentioned above, this approach would be preferable for the use case of this project, since the method solely relies on the target user, their personal research paper library and collections, and other publicly available research publications, potentially online. In simpler cases where the system is recommending related resources for a single paper, the single target paper s_c is the only source of information the system needs from its user.

However, a problem that may arise from an NLP-oriented approach such as TF-IDF is that a paper is solely abstractly represented by its important keywords in the form of a vector. Adomavicius and Tuzhilin [14], and Shardanand and Maes [17] both mention that this could be a problem when two papers, for example, a well-written one and a poorly-written one, are represented by the same keywords, and there is no other metric the system can use to determine which paper would be a better recommendation for the user. Other metrics are therefore necessary to measure a paper's authority and importance within its field of study.

2.2 Existing recommender system designs

Having established preferable approaches to implementing a recommender system for this project, this section focuses on analysing some of the existing proposals for implementing research paper-specific recommender systems, comparing and contrasting the concrete techniques and technologies used in each approach. This knowledge can then be used to assess the improvements that could be made to existing solutions to this problem, and thus what concrete approaches a Zotero-oriented recommender system could take.

2.2.1 Scalable source-independent architecture

Nascimento et al. [2] proposed a 'source-independent' solution that aims to mitigate the problem of legacy recommender systems overly relying on the use of user profiles and proprietary information, including a user's private library collection and citations. Instead, the objective of the proposed framework is to only rely on information that can be gathered from the single target paper in question and other publicly available sources of data online. All of this eventually accumulates to an end-user experience that requires low effort, which is a property that would greatly suit the interests of implementing such a recommender system on the Zotero client. The flow of a query within the framework can be summarised in the following steps:

1. the target paper is parsed into its sections (e.g. title, abstract etc.) and keywords are extracted from each section,
2. sections of text and keywords are arranged into a pool of queries which are sent to public databases, with the results then aggregated into a single list,
3. the resultant list of recommended papers is removed of duplicates and ranked using content-based methods, before being returned to the end-user as output.

One key advantage that this framework brings is that it is scalable at every intermediate stage of the process. For example, this design enables the end-user to specify which public databases they would like their recommendations to come from, depending on the databases' popularity, authoritativeness, relevance, and so on. It also allows the end-user to specify the method in which queries to the databases are generated depending on their use case. For example, the structure and the keywords used in a query can differ if the user is trying to learn more about a specific research domain, as opposed to if they are trying to find more authoritative sources to cite for a research concept of their interest.

However, such an approach proposed by this framework relies heavily on the search interface, or API, of the public databases in question. How a certain database collates its results and how many fields of query the API accommodates can become key bottlenecks to the overall quality of the recommendations provided by the framework. This distils down to the lack of control the framework has over how it can manipulate the data within a public database.

2.2.2 Utilising k-means clustering and k-NN

Lee et al. [18] proposed an architecturally-similar system that also suggests how the keyword extraction and similarity measurement stages could be implemented differently compared to the previous proposal. The system can be summarised in the following steps:

1. a set of candidate research papers is downloaded using a self-implemented web data gatherer that crawls through public databases such as IEEE Xplore and the ACM Digital Library,
2. the gathered papers are then represented as vectors using a bag-of-words model,
3. a user-item matrix is built based on the end-user's research interests and preferences,
4. both k-means clustering and k-nearest neighbours (k-NN) algorithms are used to select a fixed number of to-be-recommended papers, based on their similarities to the highest-scoring items in the user-item matrix,
5. and finally, the cosine similarity measure is used to rank the selected candidate papers, which are then sent to the end-user as output.

The proposal mentions the use of cosine similarities on the bag-of-words vector representations as a measure of similarity. This is similar to the previous proposal. However, one major difference that is worth analysing between this proposal and the previous is the use of k-means clustering and k-NN algorithms to select candidate papers to recommend to the end-user. Considering the more simple example of having to recommend k related papers based on a single target paper, a k-NN algorithm provides a straightforward method of doing that exact task. However, this may not work directly when having to select a number of candidate papers based on an arbitrary number of target papers, as is the case when using a user-item matrix. Lee et al. [18] proposed the use of k-means clustering before running the k-NN algorithm. Each target paper in the user-item matrix is considered as the centroid of its own cluster. All candidate papers are then compared in terms of similarity and assigned to a cluster whose centroid they are closest to. Finally, k-NN is run on each cluster. Assuming that there are n clusters, this means that the system is able to recommend $n \times k$ related papers as its output. Figure 2.1 from [18] illustrates this process visually.

Although this method of handling multiple target papers is easy to understand and simple to implement, it poses some limitations towards the quality of the recommendations. For example, consider a user-item matrix that contains two target papers A

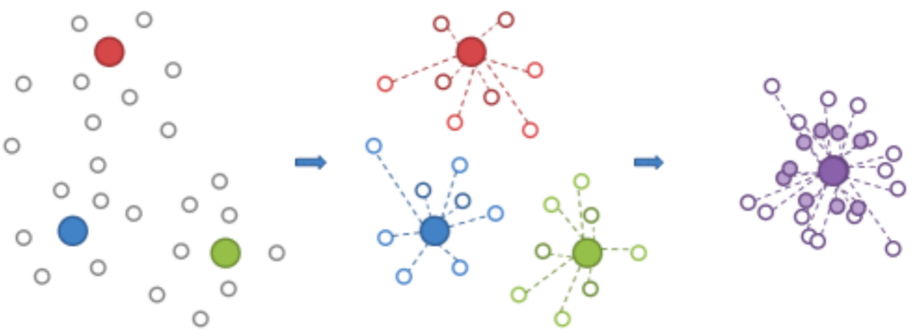


Figure 2.1: A visual representation of how k-means clustering and k-NN can be used to list recommendations, as appeared in [18].

and B that belong to the same research domain. This clustering method is unable to extract commonalities of both A and B, and calculate similarity measures based on those commonalities. Instead, the method is only able to find papers that are related to A and those that are related to B separately. This could be a concern, as an end-user would most likely want to find related papers based on all the research content in their library as a whole, not just separately based on individual papers on their own. This has especially vital implications for a Zotero-oriented recommender system, where researchers commonly organise their papers into collections, where each collection often contains papers of a particular field of study or research area.

Another element to note is that this system prioritises recommending research papers that are similar to papers that have been written by the end-user, i.e. ranking papers that have been written by the end-user themselves above those written by other authors. The system assumes that its end-users are most likely research authors themselves, and is more suited towards such a use case. However, this can easily be modified for a Zotero-oriented system. For example, instead of populating the user-item matrix with papers that are written by the end-user, a Zotero-integrated system can populate it with papers and resources that are in the user's Zotero library or collections.

2.2.3 Utilising deep learning

Hassan [19] proposed an architecturally simple content-based recommender system, where the bulk of the processing and prediction work is done using deep learning. The architecture can be summarised as follows:

1. the end-user's research interests are modelled in the form of a user profile,
2. a set of to-be-recommended papers is downloaded from a public database, such as PubMed, through web crawling,
3. the title and abstract of the set of to-be-recommended papers are represented as vectors through word embedding via the word2vec [20] library and a long short-term memory (LSTM) model,

4. and finally, papers in the user's profile and papers in the to-be-recommended set are compared using cosine similarities, and thus ranked and sent to the end-user as output.

The deep learning element of this architecture comes in the form of the word2vec library, which consists of a continuous skip-gram language model that needs to be trained from a large corpus of text. This trained model is then used to predict and generate word embedding representations. word2vec is a form of deep learning because the features of input keywords are extracted automatically.

While the proposed architecture here is structurally simple, it does come with some areas of ambiguity that will need to be addressed if one were to move towards this approach. Firstly, the proposal does not concretely explain how to-be-recommended papers should be extracted from the PubMed database through the use of the crawler, nor the process of deciding what papers should be considered as part of the to-be-recommended set. There is a possibility that this process also relies on the use of the querying interface of the PubMed database, and would need to be explored further. Secondly, the fundamental issue of using word embedding techniques such as word2vec is that it has to be trained with the corpus of text that you or your use case are interested in. In the case of the proposed system, training the model using a sample collection of scientific publications has been suggested. However, this drastically limits the research areas that the recommender system could be used for and also introduces the additional task of collecting a large sample of publications to be used as training material. This process would require a lot of consideration into whether the training material is biased and whether it covers all of the research domains the system should be able to recommend.

2.2.4 Collaborative filtering for citations

While previous sections have established that collaborative filtering in its traditional form may not be suitable for this project, collaborative filtering methods and algorithms may be adapted for use in other contexts to help enhance content-based approaches. Liu et al. [6] proposed the use of *context-based* collaborative filtering (CCF) for citation recommendation. Whereas traditional user-based collaborative filtering (UCF) methods are based on the notion of users and items, in the situation where some paper i cites some paper j, CCF interprets the citing paper i as a 'user' and the cited paper j as an 'item'.

User-based collaborative filtering

Before explaining the CCF algorithm, it is useful to first gain an insight into how traditional UCF methods could be used for citation recommendation and general resource recommendation.

First, a citation relation matrix C is constructed from the relationships between citing papers and cited papers, where the value of $C_{i,j}$ is 1 if paper i cites paper j. Then, the similarities between citing papers are calculated pair-wise. For two citing papers i_1 and i_2, their two relation matrix vectors C_{i_1} and C_{i_2} and the cosine similarity measure is used to calculate the similarity value sim_{i_1,i_2}. Finally, similar to the collaborative filtering algorithm mentioned before, this similarity value is used as a weighting for how relevant a citing paper's 'rating' is to the prediction of whether some cited paper j would be rated highly by some citing paper i, with $r_{i,j}$ being close to 1 if i would ideally cite j, and close to 0 otherwise.

In the case of this project, there are two suitable types of recommendations CF methods could provide. There are recommendations extracted from the reference lists of citing papers using the CF scoring method, i.e. the cited papers, but there are also the citing papers themselves which could be ranked by their relation matrix vector's cosine similarity to the target papers.

Context-based collaborative filtering

CCF introduces an additional step in the calculation of the final value of $r_{i,j}$, namely the *association matrix*, which aims to represent *significant co-occurrences* between papers, a more subtle and implicit relation compared to the direct overlapping similarity between their reference lists. The CCF method is intended to capture more implicit relationships between citing papers even if they do not explicitly cite the same set of papers, where it is claimed that there is approximately a 10% improvement according to the F1 measure in citation recommendation compared to UCF methods.

An association matrix A represents whether two citing papers are significantly co-occurring or not, where $A_{i_1,i_2} = 1$ if citing papers i_1 and i_2 are significantly co-occurring, $A_{i_1,i_2} = 0$ otherwise. Determining whether two papers are significantly co-occurring requires advanced statistical analysis in the form of hypothesis testing and probability distributions, as described by Liu et al. [6]. First, a contingency table is constructed as demonstrated in Table 2.1, where two papers i_1 and i_2 cite N_{11} papers simultaneously, do not cite N_{22} papers at all, and respectively cite N_{12} and N_{21} papers mutually exclusively:

	paper_{i_2}	$\neg\text{paper}_{i_2}$
paper_{i_1}	N_{11}	N_{12}
$\neg\text{paper}_{i_1}$	N_{21}	N_{22}

Table 2.1: A demonstration of how contingency tables are constructed in the CCF method [6].

Then, the chi-squared metric χ^2 is calculated using the values from the contingency table as follows:

$$\chi^2 = \frac{(|N_{11}N_{22} - N_{12}N_{21}| - \frac{1}{2}(N_{11} + N_{22} + N_{12} + N_{21}))^2}{(N_{11} + N_{12})(N_{21} + N_{22})(N_{11} + N_{21})(N_{12} + N_{22})}$$

In the context of general hypothesis testing, a larger χ^2 value implicitly indicates the lesser likelihood that papers i_1 and i_2 are independently occurring in terms of their citations.

Finally, to calculate the concrete probability value of whether there is a significant co-occurrence between i_1 and i_2, the cumulative density function of the χ^2 distribution $F(x)$ is calculated as follows, with x being the value of χ^2 as calculated before:

$$F(x) = \int_0^x \frac{e^{-\frac{1}{2}t^{\frac{1}{2}}}}{\sqrt{2}\Gamma(\frac{1}{2})}$$

A user-defined threshold ts is set, and a significant co-occurrence is considered to exist if $F(x) >$ ts. These calculations are executed pair-wise for every pair of citing papers in the system to populate the whole association matrix. For efficiency, the association matrix is symmetric along its diagonal, since paper association is a commutative property by definition, and thus only half of the association matrix needs to be constructed in practice.

The subsequent calculations of citing paper similarity and cited paper score are the same as those in UCF, but using the association matrix vectors for each citing paper rather than the relation matrix vectors. In other words, with CCF, two citing papers can be similar even if they do not have directly overlapping references, as long as there is an implicit citation association between them.

Aside from citation recommendation, it is also worth exploring how this algorithm may be applied to author lists as well as reference lists, where a paper i would be interpreted as the 'user' and its author(s) j would be the 'item(s)', to cater for use cases where the system is expected to recommend papers that have the same or similar authors as the target.

2.3 Discussion

Whilst each of the existing proposals mentioned above suggests different methods of implementing certain components of their systems, they do share a somewhat similar underlying structure. Namely, all the recommender systems are composed of the following components:

1. a data gatherer connected to some public database(s),
2. a model that abstractly represents the gathered data,

3. and a mechanism to compare and rank data according to their similarities to the target.

Data essentially goes through each of the above components in that order, in series. This provides some insight as to how this project's resultant recommender system should also be structured from an architectural level; and as Nascimento et al. [2] demonstrated, each of these components can be made to be independently scalable because of their cohesiveness.

Through the analysis of existing proposals, the following design and engineering approaches have been justified and established:

- using content-based methods, in general, to rank resources as opposed to traditional collaborative filtering, since Zotero is predominantly not a multi-user platform, and there are privacy concerns surrounding the storage of researchers' in-depth user profiles,
- using a resource's keywords as the main representation of a resource's content, and using them as comparison metrics for a resource's similarity to recommendation targets,
- training a word embedding model with a corpus of text that best represents the text that is normally found in academic writing while covering a wide range of research topics to avoid bias,
- and using user-based or citation-based collaborative filtering to rank author-based and citation-based criteria specifically as part of the wider content-based approach, and to assist in providing more contextual information about a resource to the ranking algorithm without solely relying on keywords alone.

Chapter 3

System Overview

This chapter introduces PolarRec, a lightweight, scalable, source-independent academic resource recommender system, tailor-made for the Zotero client application. The system consists of a monolithic back-end with a web API, handling the algorithmic logic of the recommender system, and a Zotero plugin extension front-end that provides a simple user interface for interacting with the web API within the Zotero client application.

Zotero users select the academic resources they would like recommendations for, known as *target resources*, from the items list, and a request is sent to the web API where recommendations are sourced, processed, and ranked. Lists of recommended resources are returned by the web API and to the Zotero plugin, where they appear on the PolarRec library tab panel on the right side of the Zotero UI.

3.1 Software architecture

Figure 3.2 illustrates the system architecture of the recommender system using the C4 software architectural visualisation model [21]. The architecture demonstrates that instead of using proprietary research databases, the system can extract research data from an abundance of existing research databases that support search queries and APIs, without affecting the resource ranking algorithm and sacrificing recommendation quality, making the system fully source-independent.

Later sections demonstrate that because of the lack of need to manage a large proprietary database, the system can generate high-quality recommendations within a small memory footprint and with low spacial complexity, making the system sufficiently lightweight for most use cases and deployment environments. Additionally, by making the clear division between the operations of the front-end and back-end through a web API, computational load is shifted away from the Zotero plugin itself, making for a more responsive user experience for the Zotero client user. This architecture

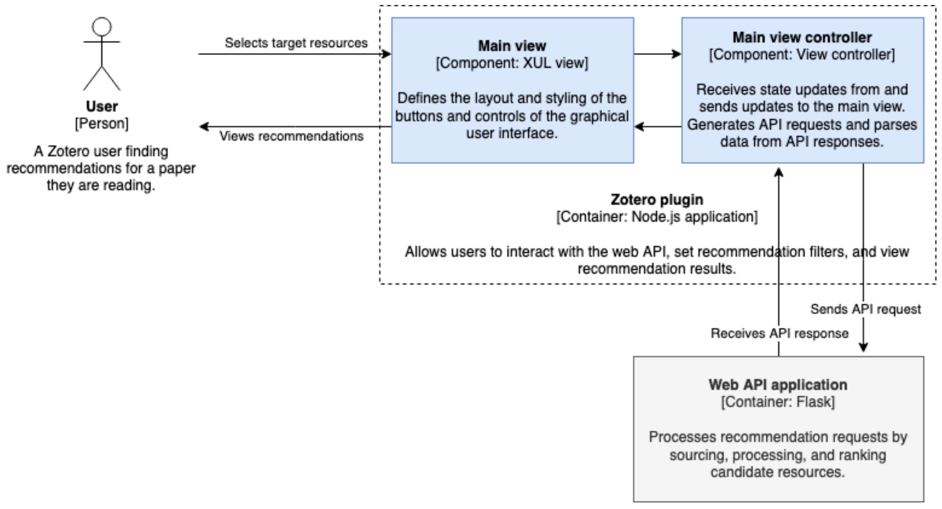

User
[Person]
A Zotero user finding recommendations for a paper they are reading.
Selects target resources
Views recommendations
Main view
[Component: XUL view]
Defines the layout and styling of the buttons and controls of the graphical user interface.
Main view controller
[Component: View controller]
Receives state updates from and sends updates to the main view. Generates API requests and parses data from API responses.
Zotero plugin
[Container: Node.js application]
Allows users to interact with the web API, set recommendation filters, and view recommendation results.
Sends API request
Receives API response
Web API application
[Container: Flask]
Processes recommendation requests by sourcing, processing, and ranking candidate resources.

Figure 3.1: A C4 component diagram illustrating the system architecture of the recommender system's Zotero plugin.

allows for many client platforms to create their own user interfaces for the PolarRec system with minimal development overhead, since the API server itself is completely platform-independent with respect to the client. Future work on this project could include the development of a web app and mobile apps for the recommender system.

Furthermore, the software architecture of the system makes it easily scalable, allowing future developers to add support for additional research databases, citation databases, and resource ranking algorithms easily.

Crucially, with a tailor-made user interface fully integrated into the Zotero client UI, it provides additional ease of use and integration that other existing solutions have yet to provide for the Zotero community.

3.2 Development technologies

The Zotero plugin has been developed using TypeScript programming language, basing its source code on the Zotero Plugin Template [22] for reasons that are explained in subsequent sections. For the web API application, there are many programming languages and frameworks that assist in the development of simple and lightweight web servers, including Express.js (JavaScript) [23] and Laravel (PHP) [24]. However, the Flask microframework [25] and the Python programming language were eventually chosen because of Python's extensive support for natural language processing libraries and linear algebra computation.

To demonstrate the lightweight nature of the system for this project, the API application has been successfully deployed onto a single Amazon Web Services (AWS) T3 Micro instance [26] with 1 GiB of memory and running on 2 virtual CPU units within a

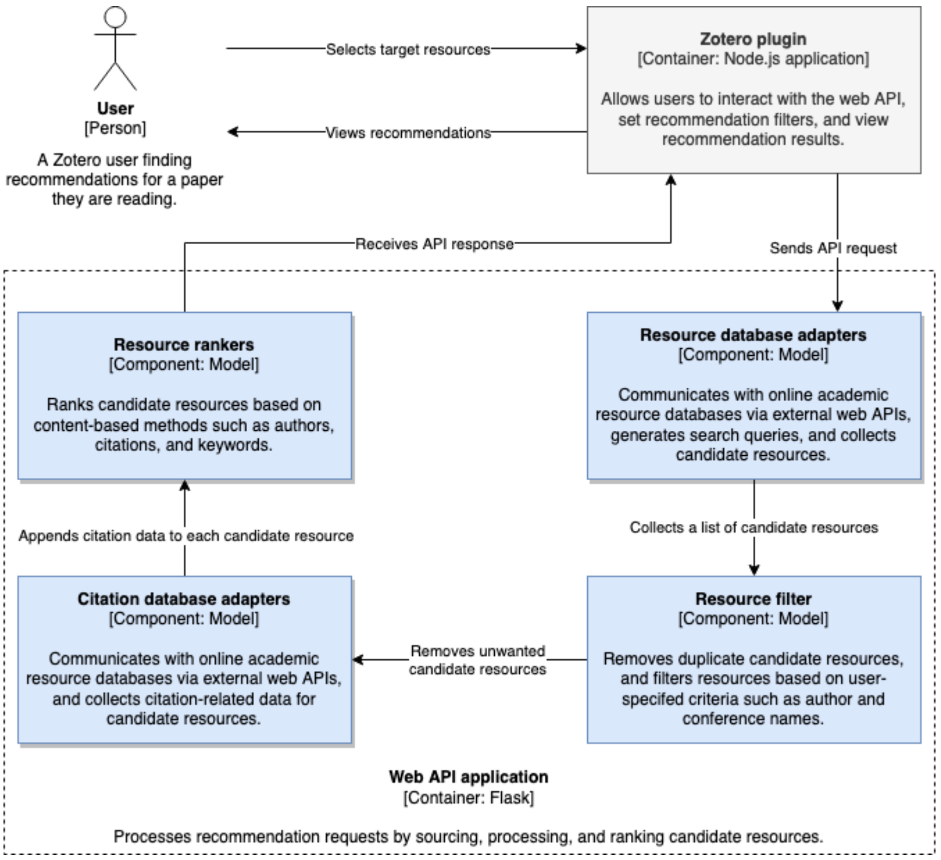

User
[Person]
A Zotero user finding recommendations for a paper they are reading.
Selects target resources
Views recommendations
Zotero plugin
[Container: Node.js application]
Allows users to interact with the web API, set recommendation filters, and view recommendation results.
Receives API response
Sends API request
Resource rankers
[Component: Model]
Ranks candidate resources based on content-based methods such as authors, citations, and keywords.
Resource database adapters
[Component: Model]
Communicates with online academic resource databases via external web APIs, generates search queries, and collects candidate resources.
Appends citation data to each candidate resource
Collects a list of candidate resources
Citation database adapters
[Component: Model]
Communicates with online academic resource databases via external web APIs, and collects citation-related data for candidate resources.
Removes unwanted candidate resources
Resource filter
[Component: Model]
Removes duplicate candidate resources, and filters resources based on user-specifed criteria such as author and conference names.
Web API application
[Container: Flask]
Processes recommendation requests by sourcing, processing, and ranking candidate resources.

Figure 3.2: A C4 component diagram illustrating the system architecture of the recommender system's web API.

larger machine. The instance runs on the Amazon Linux 2/3.5.1 operating system and with a Python 3.8 runtime. The AWS Elastic Beanstalk service [27] that is used offers a built-in continuous deployment pipeline that automatically deploys new versions of the API application based on the `main` branch of the application's Git repository.

3.3 Using the Zotero source code

Zotero is an open-source project managed by the Corporation for Digital Scholarship, with an open-source, publicly viewable code base [28]. The Zotero client is distributed under the GNU Affero General Public License, whereas the 'Zotero' name is a registered trademark, meaning the software deliverable from this project is not and should not be branded, labelled or documented as a Zotero product. Whilst the client source code has not been extracted or modified as part of this project, should this be the case in the future development of the system, the project must 'assert copyright on the software' in the documentation before the software can be copied, distributed and/or modified, and that 'modified source code becomes available to the community'.

The Zotero JavaScript API [9] has been implicitly used in this project through the use of the Zotero Plugin Template, which itself is also distributed under GNU Affero General Public License. Therefore, the terms and conditions stated earlier apply to the modifications made to the template's source code. As a result, the Git repository of the PolarRec Zotero plugin has been made publicly available along with the original terms of this copyleft licence. The detailed role the template plays in the implementation of the plugin is explained in later sections.

3.4 Use of proprietary research data

As mentioned by Nascimento et al. [2], existing research recommender systems often utilise 'privileged' information from users' libraries to generate recommendations, including private document collections, the full inner-text of private papers, sets of citations, sets of user profiles etc. Not only does the process of storing such data have to comply with General Data Protection Regulation (GDPR) terms, but there are also more fundamental privacy concerns that allude from such an implementation, for example, if the academic research in a researcher's library involves personal data from others, or if the research in question is meant to be kept unpublished due to corporate reasons. Therefore, the software design and architecture of the resultant system do not store or utilise such information when generating recommendations.

Chapter 4

User Interface

This chapter explains the detailed implementation of the front-end of the recommender system, the PolarRec Zotero plugin, and justifies design decisions related to the system's user interface.

4.1 Using the Zotero Plugin Template

The Zotero client application was built using the JavaScript programming language, with many of its UI elements implemented using XUL, HTML and CSS [28]. Zotero exposes a local JavaScript API that allows developers to either build plugin extensions or run ad-hoc JavaScript code directly within the client application [9]. The API allows developers to add items to libraries, modify item data, access file contents, filter items with queries etc. Zotero plugin developers leverage the Zotero JavaScript API to add additional functionality to the Zotero application in the form of Chrome- and Firefox-style extensions.

However, the process of developing a plugin for Zotero from scratch can be lengthy and tedious. Aside from the source JavaScript files that implement the core functionality of the plugin, the basic components of a Zotero plugin include `.rdf` files for installation and software updates, a `manifest.json` metadata file, and `.xul` files for defining the design of UI elements. On top of this development overhead, with the documentation for the Zotero API growing continuously out of date and cluttered, and with the lack of static typing from JavaScript, it is increasingly difficult for developers to explore various aspects of the Zotero API and build more advanced plugin functionality without having to comprehend the raw Zotero client source code.

The Zotero Plugin Template was developed by third-party developer windingwind [22] to mitigate these issues. The template offers the ability to develop plugins using the Node.js runtime, exposing Zotero plugin developers to the wealth of Node.js packages many other JavaScript-based applications, including React web apps, are able to access. Using a single `package.json` metadata file defined by the developer, and other pre-defined NPM scripts, the template is able to generate the relevant boilerplate code dynamically during compile-time and compile the final `.xpi` plugin binary automatically – the same binary Zotero users eventually download in order to install and use the plugin. Furthermore, the template uses TypeScript to redefine many of the variables, classes, and methods in the Zotero JavaScript API, thus adding static typing and additional up-to-date documentation to the original API. Therefore, the PolarRec Zotero plugin uses the Zotero Plugin Template to reduce development overhead and allow for potentially more advanced functionality. The main changes made to the original template source code include the `package.json` metadata file and the `src` directory where the core functionality of the plugin resides.

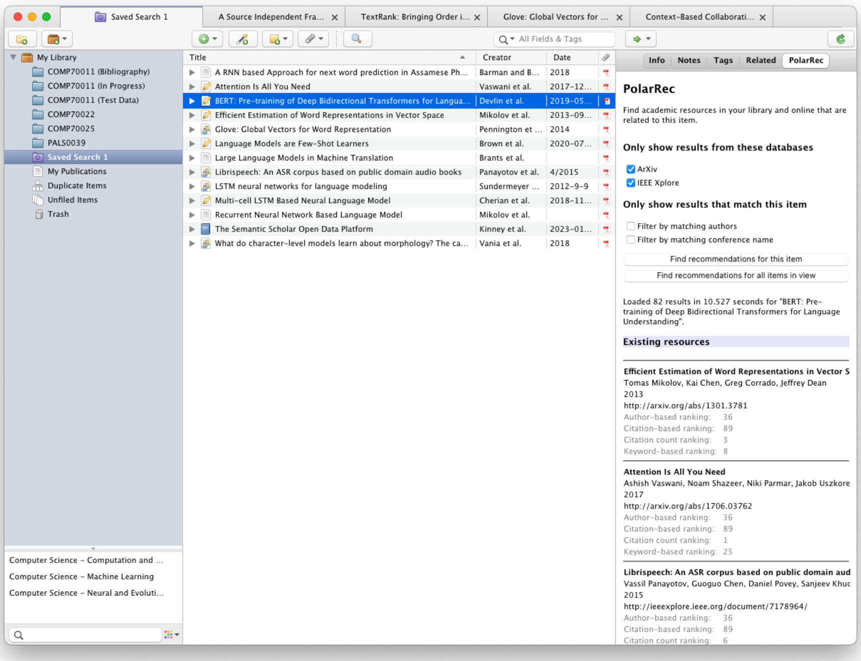
4.2 Software architecture

The implementation of the plugin follows a model-view-controller software design pattern [29] with a single view, the `MainView`, and its controller, the `MainViewController`. The 'model' element is represented by the Flask web API application's role in the whole system, along with several TypeScript-written helper classes and libraries that handle low-priority algorithmic logic, including the `locale` library which enables the translation of UI text between several different Zotero-supported languages, and the `CustomLogger` class which enables the logging of debugging messages on the Zotero developer console.

During typical operation, the user interacts with the main view, which sends its state to the main view controller. The controller is then responsible for the following sequence of events:

1. collecting resource data from the user's Zotero library,
2. constructing the API request along with data fields and query parameters,
3. sending the API request to the back-end as an HTTP POST request,
4. parsing the recommendation data returned by the API response,
5. and updating the main view with all the recommendation results returned by the API.

To aid the development process, the Zotero plugin uses a different HTTP address to communicate with the API application depending on the compilation environment. When compiled in development mode, the plugin assumes the API application has been deployed locally on the development machine, and uses `http://127.0.0.1:5000` as the address for the back-end. When compiled in production mode, the plugin communicates with the AWS-deployed cloud server instead, just as a non-developer user would expect it to during normal operation.



Saved Search 1
A Source Independent Fra...
TextRank: Bringing Order i...
Glove: Global Vectors for ...
Context-Based Collaborati...
All Fields & Tags
My Library
COMP70011 (Bibliography)
COMP70011 (In Progress)
COMP70011 (Test Data)
COMP70022
COMP70025
PALS0039
Saved Search 1
My Publications
Duplicate Items
Unfiled Items
Trash
Computer Science - Computation and ...
Computer Science - Machine Learning
Computer Science - Neural and Evoluti...
Info
Notes
Tags
Related
PolarRec

Title	Creator	Date
A RNN based Approach for next word prediction in Assamese Ph...	Barman and B...	2018
Attention Is All You Need	Vaswani et al.	2017-12...
BERT: Pre-training of Deep Bidirectional Transformers for Langua...	Devlin et al.	2019-05...
Efficient Estimation of Word Representations in Vector Space	Mikolov et al.	2013-09...
Glove: Global Vectors for Word Representation	Pennington et ...	2014
Language Models are Few-Shot Learners	Brown et al.	2020-07...
Large Language Models in Machine Translation	Brants et al.	
Librispeech: An ASR corpus based on public domain audio books	Panayotov et al.	4/2015
LSTM neural networks for language modeling	Sundermeyer ...	2012-9-9
Multi-cell LSTM Based Neural Language Model	Cherian et al.	2018-11...
Recurrent Neural Network Based Language Model	Mikolov et al.	
The Semantic Scholar Open Data Platform	Kinney et al.	2023-01...
What do character-level models learn about morphology? The ca...	Vania et al.	2018

PolarRec

Find academic resources in your library and online that are related to this item.

Only show results from these databases

[✓] ArXiv
[✓] IEEE Xplore

Only show results that match this item

[ ] Filter by matching authors
[ ] Filter by matching conference name

Find recommendations for this item

Find recommendations for all items in view

Loaded 82 results in 10.527 seconds for "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding".

Existing resources

Efficient Estimation of Word Representations in Vector S
Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean
2013
http://arxiv.org/abs/1301.3781
Author-based ranking: 36
Citation-based ranking: 89
Citation count ranking: 3
Keyword-based ranking: 8

Attention Is All You Need
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkore
2017
http://arxiv.org/abs/1706.03762
Author-based ranking: 36
Citation-based ranking: 89
Citation count ranking: 1
Keyword-based ranking: 25

Librispeech: An ASR corpus based on public domain aud
Vassil Panayotov, Guoguo Chen, Daniel Povey, Sanjeev Khuc
2015
http://ieeexplore.ieee.org/document/7178964/
Author-based ranking: 36
Citation-based ranking: 89
Citation count ranking: 6

Figure 4.1: A screenshot of the PolarRec Zotero plugin 0.7.0 user interface of the recommender system, running on Zotero 6.0.

4.3 Graphical user interface design

The plugin offers a simple, scrollable, HTML-constructed user interface, with filters and controls placed at the top, and lists of recommendation results placed at the bottom, as illustrated in Figure 4.1, where the main view of the plugin is located on the right-side panel of the Zotero window.

The filters and controls at the top include the following components, as illustrated in Figure 4.2a:

- checkboxes to select which resource databases to use and not to use during the data-gathering stage of the recommendation process,
- checkboxes to select whether to only include results that have matching authors or belong to the same conference as the target resource,
- a button to initiate the recommendation algorithm using the single selected Zotero item in the list view as the target resource,
- and a button to initiate the recommendation algorithm using all Zotero items in the list view as target resources.

Between the controls and the results view is a status view that displays the current loading status of the recommender system, including the processing time of the last recommendation request and the number of results the system has processed. Should the back-end return a large number of recommendation results, only the top 50 results are displayed for a better user experience.

The results view includes two lists of recommendation results, where both lists are sorted in descending order of their relevance to the target(s):

- a list of existing resources in the user's Zotero library, specifically those that are currently in the list view, that are significantly relevant to the target (if a single Zotero item has been selected as the target),
- and a list of resources extracted from the external research databases the user has 'whitelisted' in the filter controls.

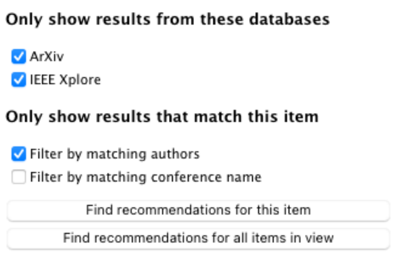
Each result item includes the following information, as demonstrated in Figure 4.2b:

- the title of the resource,
- the authors of the resource,
- the year the resource was published,
- the URL that redirects to the resource's website,
- and the resource's positional ranking according to its author list similarity to the target(s)', its reference list similarity to the target(s)', the number of times it has been cited by others, and how relevant its keywords are to the target(s)' respectively.

With the transparency of the recommender system's ranking algorithm, the user can better understand how and why certain results are considered more relevant than others and thus choose to view results that better fit their personal ranking criteria. This is a benefit that most existing solutions do not provide, where systems do not make the ranking process transparent and where users often do not understand why systems have recommended certain results.

4.4 Communicating with the web API

The main view controller is responsible for generating API requests and parsing API responses between it and the Flask application in the back-end. The following JSON pseudo-code demonstrates the format of a typical API request, which contains the list of the target resources, a list of existing resources in the user's Zotero library that are not included in the targets, and finally, author, conference, resource database filters, all three of which correspond to the filter selections the user has made in the GUI:



Only show results from these databases

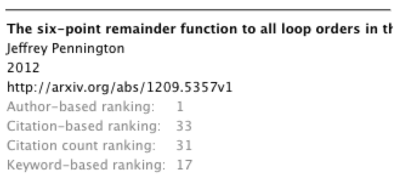
- [✓] ArXiv
- [✓] IEEE Xplore

Only show results that match this item

- [✓] Filter by matching authors
- [ ] Filter by matching conference name

Find recommendations for this item
Find recommendations for all items in view

(a) The filters and buttons.



The six-point remainder function to all loop orders in tl
Jeffrey Pennington
2012
http://arxiv.org/abs/1209.5357v1

Author-based ranking:	1
Citation-based ranking:	33
Citation count ranking:	31
Keyword-based ranking:	17

(b) A recommendation result.

Figure 4.2: A close-up screenshot of the various elements of the PolarRec Zotero plugin 0.7.0 user interface.

```
{
    "target_resources": [{Resource},{Resource},{Resource},...],
    "existing_resources": [{Resource},{Resource},{Resource},...],
    "filter": {
        "authors": ["R. Girshick","J. Donahue","T. Darrell","J. Malik"],
        "conference_name": "2014 IEEE Conference on Computer Vision"
    },
    "resource_databases": ["arxiv","ieeexplore"]
}
```

The following demonstrates the format of a typical API response, which contains the processing time of the recommendation algorithm for this request, the sorted list of recommended resources that already exist in the user's Zotero library, and the sorted list of recommended resources sourced from external databases:

```
{
    "processing_time": 2.56,
    "ranked_existing_resources": [
        {RankableResource},{RankableResource},{RankableResource},...
    ],
    "ranked_database_resources": [
        {RankableResource},{RankableResource},{RankableResource},...
    ]
}
```

The exact format of a `Resource` and `RankableResource` JSON object directly correspond to the data fields that are contained in the `Resource` and `RankableResource` classes in the back-end respectively, which are further explained in subsequent sections. They include common information one would use to identify any general academic resource, including its title, authors, and abstract.

Chapter 5

Recommendation Algorithm

This chapter explains the detailed implementation of the back-end of the recommender system, the PolarRec web API, introduces the various models and libraries that enable its functionality, and justifies design decisions related to the system's recommendation algorithm.

5.1 Data gathering

The first step to generating recommendations is to collect *candidate academic resources* from various sources. The most straightforward source of resources is the user's own Zotero library. It is commonplace for Zotero users to create *collections* – user-defined folders of Zotero items where related resources often reside together, for example, those belonging to the same subject area, or those collectively used for a particular research project. Furthermore, Zotero supports the use of *saved searches* – local search query results that can be saved in the form of a collection in the user's library on the left-side panel. Therefore, when a user selects a single target resource from their Zotero library and makes a recommendation request, existing resources currently in view are also passed onto the web API as a data field so that the system can filter through and rank existing resources alongside those found from other sources. These existing resources can either come from a collection, a saved search, or be *unfiled* (not belonging to any particular collection), depending on which collection the user has selected in the left-side panel.

5.1.1 Resource database adapters

In addition to existing library resources, the recommender system also sources candidates from online resource databases. This is achieved by using *resource database adapters.* This model follows the adapter design pattern from the 'Gang of Four' [30] by exposing the querying functionality of online resource databases in the form of simplified methods. The model enables the system to communicate with databases and collect candidate resources through search queries in a cohesive manner. Search queries are generated by extracting various information from target resources. The system currently generates two search queries for each resource database it communicates with:

1. a keyword-based search query, where a list of top keywords is extracted from the target resources and turned into a query,
2. and an author-based search query, where the target resources' authors are extracted and turned into a query.

For example, the following are examples of two HTTP GET requests are sent to the arXiv API upon a recommendation request:

```
http://export.arxiv.org/api/query?
search_query=all:%22segnet%22+OR+all:%22network%22+OR+
all:%22segmentation%22+OR+all:%22maps%22+OR+all:%22feature%22+OR+
all:%22encoder%22+OR+all:%22decoder%22+OR+all:%22architectures%22+OR+
all:%22also%22+OR+all:%22trainable%22&start=0&max_results=20

http://export.arxiv.org/api/query?
search_query=au:badrinarayanan+vijay+OR+au:kendall+alex+OR+
au:cipolla+roberto&start=0&max_results=20
```

Combined, these queries return candidate resources that are either related to the target resources in terms of textual content or in terms of their authors.

However, by passing the same search queries through multiple resource databases, it is inevitable for duplicate candidates to surface. Therefore, once all the candidate resources have been collected, duplicate resources are removed before executing the next stage of the recommendation algorithm for maximum efficiency.

For demonstration purposes, the system currently exclusively supports the arXiv and IEEE Xplore databases. However, it is a straightforward process to scale the system and include additional resource databases for the future, such as JSTOR, PudMed, ScienceDirect etc. Adding additional databases can help diversify the recommendation results the system generates, broaden the range of research topics covered by the system, and provide more filtering options for the user.

5.1.2 Keyword extraction

To generate keyword-based search queries to resource databases, keywords have to be extracted from the target resources first. Statistical natural language processing has created many different algorithms which achieve keyword extraction and ranking. Two of these algorithms have been implemented in this recommender system, and are interchangeable by updating hyperparameter values in the hyperparameter file within the source code. They are namely term frequency–inverse document frequency (TF-IDF) [4] and TextRank [5]. The implementations of both algorithms can be found as part of the `KeywordRanker` class, whose wider functionality is further explained in the subsequent sections.

Whilst some resource databases provide author-defined keywords as part of their APIs, the implementation of this recommender system has avoided the use of these pre-defined keywords when assigning keywords to resources for multiple reasons. Firstly, not all resource databases provide this information, so to maintain the data source independence and scalability of the system, the ranking algorithm should not overly rely on it. Secondly, not all resource types have pre-defined keywords in the first place, such as websites and generic PDF documents.

TF-IDF

TF-IDF [4] is one of the most popular text mining algorithms, used primarily for ranking keywords and key phrases within documents of text, with over 80% of text-based digital library recommender systems utilising the algorithm as of 2015 [31]. The TF-IDF statistic is composed of two sub-statistics – term frequency (TF) and inverse document frequency (IDF). They are defined mathematically as follows, where t refers to some term, word, or phrase, d refers to some document, $f_{t,d}$ refers to the number of occurrences of term t in document d, and D refers to the complete set of documents:

$$\mathrm{TF}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

$$\mathrm{IDF}(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

Effectively, the TF statistic upweights terms if they are important to the documents they belong to, such as keywords, whereas the IDF statistic downweights overly common terms if they occur across all documents, such as ‘a’, ‘an’, ‘the’ etc. The TF-IDF statistic for a particular term t and document d is thus defined as follows:

$$\text{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D)$$

The keyword ranker, when using TF-IDF, calculates the TF-IDF value for every term or phrase in a resource’s title and abstract, and sorts the terms by their TF-IDF values. The top terms in this list are thus considered ‘keywords’ by the system and are used to represent a resource not just during query generation, but also during keyword-based ranking, which is further explained later on.

TextRank

TextRank [5] is an alternative graph-based ranking model for text processing. Specifically, it bases its model on the PageRank algorithm [32] which itself has been famously used by Google Search to rank website content. Instead, TextRank extracts sentences and keywords of importance from textual data. The algorithm builds a weighted directed graph, with vertices representing text entities such as sentences, keywords, or key phrases, and edges representing the relationship between text entities. A score is assigned to each vertex, which represents the importance of that particular text entity within the text. The score for a particular vertex V_i is defined by the following iterative equation, where d refers to a user-defined damping factor between 0 and 1, $\text{In}(V_i)$ refers to the set of vertices with outgoing edges towards V_i, $\text{Out}(V_i)$ refers to the set of vertices with incoming edges originating from V_i, and weight w_{ij} refers to the 'strength' of the relationship between V_i and V_j:

$$\text{WS}(V_i) = (1 - d) + d \times \sum_{V_j \in \text{In}(V_i)} \frac{w_{jk}}{\sum_{V_k \in \text{Out}(V_j)} w_{jk}} \text{WS}(V_j)$$

This scoring method is run across every vertex in the graph over multiple iterations through a random walk method until the scores for all vertices stabilise and converge to fixed values within a given threshold. A *random walk* treats the graph as a Markov chain, where the weight w_{ij} of a directed edge corresponds to the probability of a transition from the source vertex to the target vertex. In terms of Markov chains, the damping factor d corresponds to the probability of a vertex transitioning to any other random vertex even if they are not connected, thus resolving the issue of reducible and periodic Markov chains not being able to converge to a stable score state, or the *stationary distribution* [33]. In other words, there exists an edge with a small transition probability for every pair of vertices in the graph regardless, which helps convergence for pieces of text that generate more sparsely-connected graphs.

When executing the task of keyword extraction, words within the text are each assigned a vertex, with the edge between them representing their *co-occurrence* in the text. Specifically, if two words appear together within a window of N words in the text, they are considered co-occurring. The words included in the graph can be preprocessed and filtered, with Mihalcea and Tarau [5] stating that only including nouns and adjectives tend to lead to the best results. This intuitively makes sense as keywords and key phrases in academic writing usually only consist of nouns and adjectives. The aforementioned scoring method is then run across the graph, and each word is assigned a score. Thus, the final keyword list returned by the algorithm consists of words that have the highest scores.

Through a user-defined hyperparameter, developers can choose whether to use the TF-IDF or TextRank algorithm for keyword extraction in the recommender system. Their relative performances in terms of how accurately they are able to extract keywords from academic paper abstracts are compared and contrasted in later sections.

5.1.3 Citation database adapters

Whilst resource databases commonly store basic academic resource metadata including titles, authors, and abstracts, more advanced citation-related metadata such as citation counts and reference lists are not necessarily available. Therefore, it is necessary to distinguish the difference between resource database adapters and *citation database adapters*.

As part of the recommendation algorithm, once candidate resources have been sourced from resource databases, citation metadata is appended onto each candidate resource object by querying various citation data-specific, or citation data-compatible, databases. For example, for demonstration purposes, the system currently exclusively uses the Semantic Scholar Academic Graph (S2AG) API [34] as its main citation database. The system passes each candidate resource's title and authors as a search query and parses the response data to extract citation count and reference list data for each candidate. This data must be extracted before the candidate ranking stage of the algorithm as rankers rely on this citation data to rank candidates. However, it should also be extracted after the resource filtering process to prevent duplicate data extraction for the same candidate. The Crossref REST API was also experimented and tested with as a citation database during the development of the system but was later abandoned due to the lack of reference list data and strict request rate limits imposed on client users.

Although the system currently primarily uses S2AG as a citation database, S2AG is also capable of handling queries for academic resources by keywords and authors, so future development could see the S2AG adapter being implemented as both a resource database and a citation database through class inheritance, demonstrating the scalability and flexibility of the recommender system in handling databases of different types and capabilities.

5.2 Data representation

Metadata passed into the API application, collected by resource database adapters, and returned to the user need to be stored in memory in a unified format for clarity and cohesiveness. It is therefore important to define the structure of a `Resource` object, which acts as the fundamental building block of the recommender system, similar to how an *item* is an 'atomic unit' inside the Zotero client. A *resource* is meant to represent any generic academic resource that can be imported into Zotero, not just research papers, but also books, documents, websites etc.

5.2.1 The `Resource` class

Within the system, a resource is represented in the form of a class instance and has data fields including the resource's authors, conference name, conference location, title, publication year and month, abstract, DOI number, URL, list of references, and citation count. The `Resource` class has been implemented such that it efficiently handles non-existent, undefined, or poorly-defined values, and can be instantiated directly from a JSON-like object, such as a `dict`, as well as be exported into a JSON object. This smoothens the process of receiving API requests and generating API responses at the endpoint. For example, the instantiation function checks the types of values in argument dictionaries, ensuring the formats of publication dates and DOI numbers are valid. Furthermore, it checks if keys do not exist in argument dictionaries, and assigns class member variables as `None` if so.

The identity of a `Resource` instance, or its *hash value*, is primarily defined by its title. This is based on the assumption that the likelihood of two different academic resources having the same title is sufficiently small, especially given a limited number of candidates, and that metadata such as authors and references are not always available depending on the source from which the resource was extracted from. When generating its hash value, the `Resource` object pre-processes its title by transforming the string into lowercase letters, removing extraneous whitespace and punctuation, and replacing Unicode-only characters with their ASCII equivalents, creating what is internally referred to as a *comparable string* (a string suitable for direct comparison with other strings). This process is necessary since different sources can format resource titles slightly differently, thus making it difficult for the recommender system to identify duplicate resources. For example, without string pre-processing, two instances of the same title, one using Unicode characters and the other having been translated to ASCII characters, could potentially and undesirably fail string equality tests. To illustrate, consider the following list of differently-formatted resource titles that actually refer to the same paper:

```
GloVe: Global Vectors for Word Representation
Glove: global vectors for word representation
glove global vectors for word representation
```

All these strings resolve to the same comparable string and thus the same hash value, as follows:

```
gloveglobalvectorsforwordrepresentation
```

5.2.2 The `RankableResource` class

A *rankable resource* is derived from the `Resource` class, but with additional data fields to keep track of its positional rankings assigned by resource rankers. This data is used as part of the sorting and ranking process of the recommendation algorithm, which is further explained in detail in the following sections.

5.3 Resource ranking

Having collected and pre-processed all candidate resources, the final stage of the recommendation algorithm is to sort and rank the candidates according to how related they are to the target resources. Currently, the recommender system consists of four resource rankers, each of which assigns a positional ranking to each candidate resource according to some specific criteria:

- an author-based ranker, which ranks candidates based on how similar their authors are to the targets',
- a citation-based ranker, which ranks candidates based on how similar their references are to the targets',
- a citation count ranker, which ranks candidates based on the number of times they have been cited by others,
- and a keyword-based ranker, which dynamically extracts keywords from both targets and candidates using natural language processing methods, and ranks candidates based on how similar their keywords are to the targets'.

The system is designed such that it is easily scalable if developers wish to add additional rankers in the future to further enhance the ranking process. Figure 5.1 illustrates the software architecture of resource rankers within the system in the form of a C4 component diagram.

Once each ranker has assigned their rankings for each candidate, the candidates are sorted by their weighted mean ranking across all the rankers. Currently, the system is tuned such that keyword-based rankings contribute the most to the final weighted mean, due to the importance of a candidate's textual content as an indication of its relatedness to a target. In contrast, citation count rankings contribute the least, as while they aim to further differentiate resources according to their authority within their respective fields of study, they should not be the major determining factor in how related a candidate is to a target. For example, a candidate that has been cited by many researchers but belongs to a completely different research area compared to the target should not be ranked highly. Figure 5.2 demonstrates how the weighted mean rankings of candidates are calculated by aggregating the rankings based on individual content-based criteria.

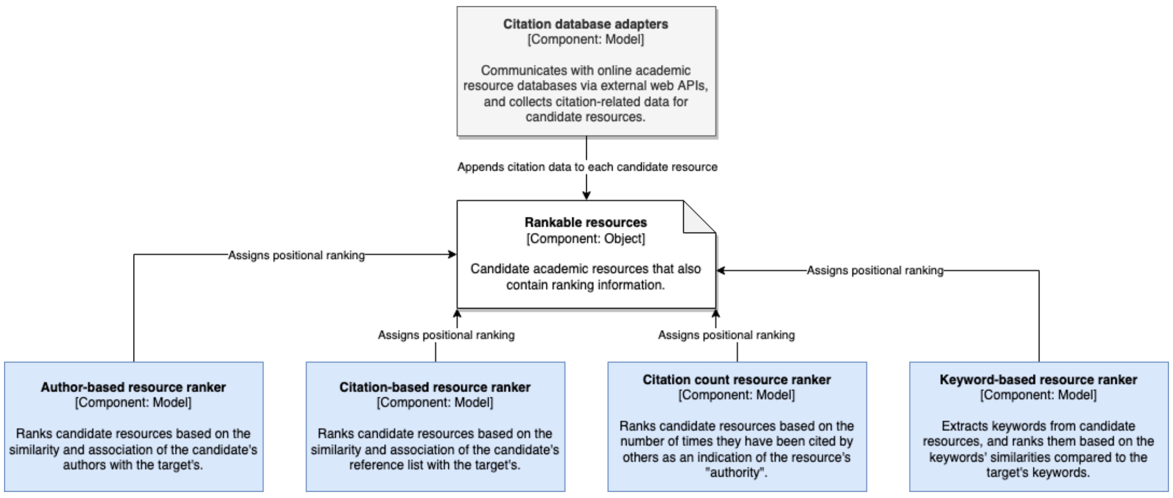
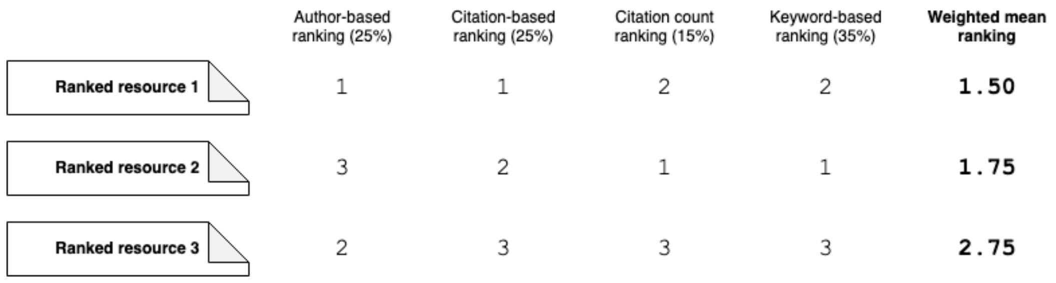

Citation database adapters
[Component: Model]
Communicates with online academic resource databases via external web APIs, and collects citation-related data for candidate resources.
Appends citation data to each candidate resource
Rankable resources
[Component: Object]
Candidate academic resources that also contain ranking information.
Assigns positional ranking
Assigns positional ranking
Assigns positional ranking
Assigns positional ranking
Author-based resource ranker
[Component: Model]
Ranks candidate resources based on the similarity and association of the candidate's authors with the target's.
Citation-based resource ranker
[Component: Model]
Ranks candidate resources based on the similarity and association of the candidate's reference list with the target's.
Citation count resource ranker
[Component: Model]
Ranks candidate resources based on the number of times they have been cited by others as an indication of the resource's "authority".
Keyword-based resource ranker
[Component: Model]
Extracts keywords from candidate resources, and ranks them based on the keywords' similarities compared to the target's keywords.

Figure 5.1: The C4 component diagram illustrating the system architecture of the ranking process of the recommender system.



	Author-based ranking (25%)	Citation-based ranking (25%)	Citation count ranking (15%)	Keyword-based ranking (35%)	**Weighted mean ranking**
Ranked resource 1	1	1	2	2	**1.50**
Ranked resource 2	3	2	1	1	**1.75**
Ranked resource 3	2	3	3	3	**2.75**

Figure 5.2: An example of how rankings assigned by resource rankers are aggregated to generate a weighted mean ranking for each candidate resource.

5.3.1 Author-based and citation-based ranking

Author- and citation-based ranking are implemented in the `AuthorRanker` and `CitationRanker` classes respectively. They share very similar structures as both rankers use collaborative filtering techniques, as mentioned before, to compare the pair-wise similarities between two resources' author and reference lists. Developers can choose whether to use UCF or CCF methods to calculate these similarities by updating the hyperparameter file. Their relative performances are compared and contrasted in later sections.

For both ranking metrics, once similarity scores have been calculated for each candidate resource, they are sorted and their positional rankings are assigned to them through their `RankableResource` instances.

5.3.2 Keyword-based ranking

Keyword-based ranking is implemented in the `KeywordRanker` class. The keyword ranker uses a relatively different class structure compared to the previous two rankers since keyword-based ranking requires natural language processing techniques instead, even though each candidate resource is eventually also assigned a positional ranking through their `RankableResource` instances. Generally, a candidate should rank highly if its keyword list is semantically similar to the targets', and vice versa.

Keyword extraction

The first step of keyword-based ranking is to extract keywords from both the target and candidate resources. This is achieved via the same methods mentioned before during the data-gathering stage of the recommendation process, either using TF-IDF or TextRank. However, what differs is the number of top keywords that are extracted from each resource and used to represent each resource. Instead of database query generation, the purposes of these keyword lists are for similarity comparison. This number is, again, a developer-defined hyperparameter. Later sections evaluate how changing these hyperparameters affect recommendation quality, and highlight what the ideal hyperparameter values are.

Word embeddings

Having extracted keywords, the next step is to compare how similar each candidate's keyword lists are to the targets'. To achieve this, the system must be able to 'comprehend' the semantic meaning of keywords as humans do. Word embeddings are a widely used technique in natural language processing, where each word in a corpus of text is represented by a fixed-dimensional vector. Compared to one-hot encoding and count-based vector methods, word embeddings enable the representation of an arbitrary number of words within a fixed-sized low-dimensional space, significantly reducing the spacial complexity for any computation surrounding the processing of text.

To gain some insight into how word embeddings are constructed, word2vec [20] is a state-of-the-art example of a continuous Skip-gram machine learning model that learns the values of word embedding matrices when trained. In terms of input and output, a Skip-gram model takes in a target word, in the form of a one-hot encoded vector, and predicts the one-hot encoding of a word that could possibly appear near the target word in a piece of text (referred to as *context words*). The size of this context window (the window within which words are considered 'nearby') is defined during the training process. Figure 5.3 illustrates the architecture of a Skip-gram model and provides an example of how target word representations are learnt through its context words.

Given a target word w_t, context words w_{t+j} for a range of j, the size of the context window c, and the number of training samples T, the theoretical training objective is as follows:

$$\max_{\theta} \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t; \theta)$$

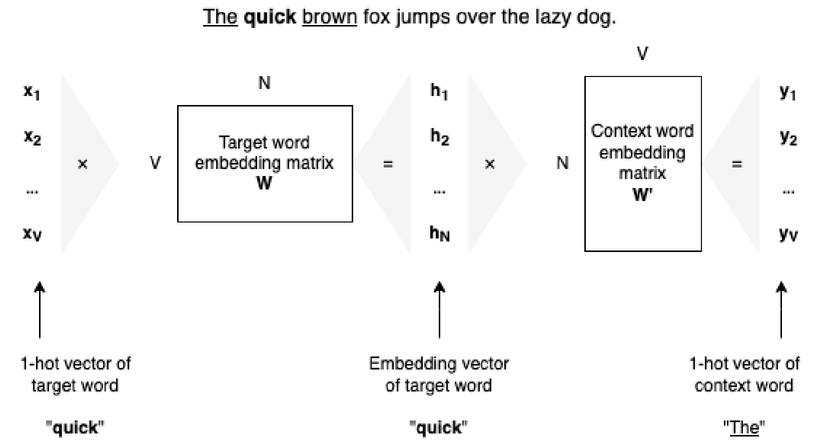

The quick brown fox jumps over the lazy dog.
x_1
x_2
...
x_V
×
V
N
Target word embedding matrix W
=
h_1
h_2
...
h_N
×
N
V
Context word embedding matrix W'
=
y_1
y_2
...
y_V
1-hot vector of target word
"quick"
Embedding vector of target word
"quick"
1-hot vector of context word
"The"

Figure 5.3: An example of how the architecture of a continuous Skip-gram model enables the learning of word embedding vectors.

Effectively, the training objective aims to maximise the average log-likelihood of a predicted word being a context word of the target. In practice, the training objective is slightly different since negative sampling is used as an approximation method to reduce the computational complexity of the training process, but the idea remains the same.

In PolarRec, GloVe word embedding vectors [3], which are trained on a slightly differing word-word co-occurrence statistical model, were used instead, because of their widespread use in recent years and their availability through the Gensim library. Specifically, for demonstration purposes, a pre-trained GloVe word embedding matrix, trained on textual data from the Wikipedia 2014 dump and the Gigaword 5 corpus [35], with 6 billion tokens in total, was used. This particular corpus was chosen among the list of available pre-trained models because of its similarity to the text found in academic writing. The Wikipedia website has over 6.6 million published English articles as of June 2023 [36], many of which cover a wide range of academic and research topics, written in professional styles of writing similar to that found in academia. The Gigaword 5 corpus also covers many topics by aggregating the archives of press publications from various news agencies.

Although a custom-trained word embedding model could potentially be better catered for the specialised and technical vocabulary that is commonly found in research papers, tests run on the titles and abstracts of 18 sample academic resources have shown that only 1.8285% of words were considered out-of-vocabulary by the pre-trained GloVe model, which includes extremely rare vocabulary such as proper nouns for science- and technology-related entities. This value would have been higher had a less appropriate corpus been chosen for the task, such as the social media-based pre-trained models that Gensim also offer.

Keyword list similarity

To calculate the similarity between two words, their word embedding vectors are extracted and compared against each other using the cosine similarity measure. However, additional consideration is required when comparing the similarity between two *lists* of keywords, as is the case when comparing a target and a candidate resource where both their own lists of top keywords extracted from the aforementioned keyword extraction process. Not only does the similarity measure have to capture the similarities of all the keywords within the lists, but it should also consider the *importance* of each keyword to its respective resource. For example, consider a research paper discussing the topic of linguistics, where the term 'machine learning' happened to be extracted as one of the keywords from the abstract of the paper. Consider another paper that introduces a new deep-learning model for medical imaging. The system should not consider these two papers as highly relevant.

The most straightforward method would be to calculate the pair-wise similarities of each pair of keywords between the two lists and aggregate the results through averaging. However, this would mean that for each candidate keyword, there would be many unrelated target keywords that would decrease the value of the mean similarity metric. Furthermore, this metric does not consider the importance of these keywords to their resources. For example, consider the following two arbitrary lists of words:

```
["python", "jumble", "easy", "difficult", "answer", "xylophone"]
["java", "scramble", "simple", "hard", "response", "piano"]
```

Theoretically, these two lists should be considered semantically very similar. However, such a similarity metric would return a very small similarity value. For example, the word 'python' is very much dissimilar to 'piano', 'response', 'hard' etc., but only similar to 'java'. This logic applies to every word in the lists.

Normally, a resource's list of keywords consists of the various different research topics the resource discusses. Instead of needing a candidate keyword to be 'similar' to every target keyword there is, the metric should rather reflect whether there *exists* a matching target keyword to each candidate keyword, and if so, how important that keyword is to both resources.

Therefore, the recommender system uses a modified keyword list similarity metric that augments the cosine similarity measure, illustrated in Figure 5.4. For each candidate keyword, the algorithm searches for the target keyword that achieves the highest cosine similarity with that candidate keyword. For example, in Example 2 in Figure 5.4, for candidate keyword k_4, k_3 achieves the highest similarity. This similarity measure is then mathematically weighted using the negative exponential distribution and according to the positional ranking of the two keywords in their respective lists. The full definition of the keyword list similarity metric is found in Equation 5.1, where l_C and l_T refer to the candidate and target keyword lists respectively, $R(k_C)$ refers to the

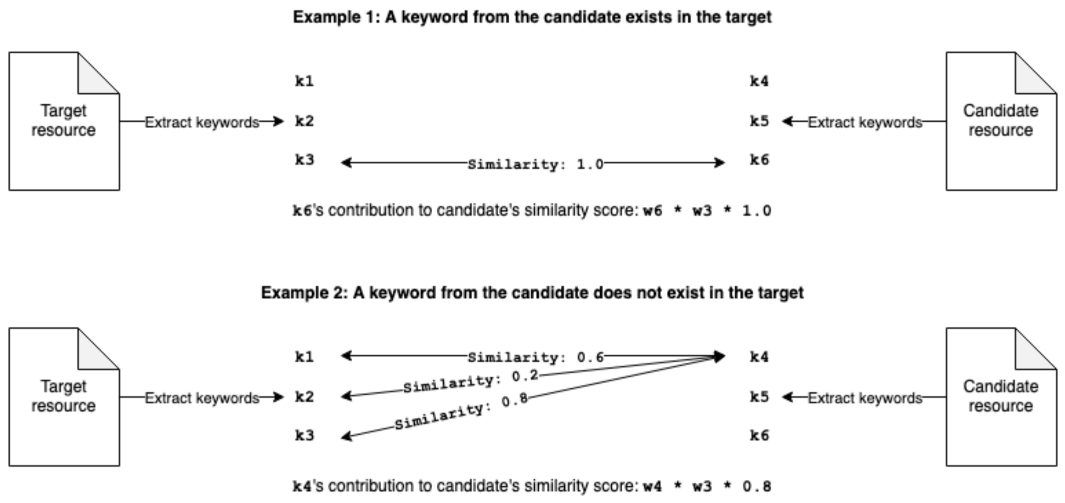

Example 1: A keyword from the candidate exists in the target
Target resource
Extract keywords
k1
k2
k3
Similarity: 1.0
k4
k5
k6
Extract keywords
Candidate resource
k6's contribution to candidate's similarity score: w6 * w3 * 1.0
Example 2: A keyword from the candidate does not exist in the target
Target resource
Extract keywords
k1
k2
k3
Similarity: 0.6
Similarity: 0.2
Similarity: 0.8
k4
k5
k6
Extract keywords
Candidate resource
k4's contribution to candidate's similarity score: w4 * w3 * 0.8

Figure 5.4: Examples of how the keyword ranker compares the similarities of lists of keywords between a target and candidate resource.

positional ranking of k_C within l_C, and d refers to an adjustable damping factor.

$$\begin{aligned} \max_{k_C} &= \operatorname{argmax}_{k_T \in l_T} \mathrm{sim}_{k_C,k_T} \\ w_{k_1,k_2} &= d^2 e^{d(R(k_1)+R(k_2))} \\ \mathrm{sim}_{l_C,l_T} &= \frac{\sum_{k_C \in l_C} w_{k_C,\max_{k_C}} \mathrm{sim}_{k_C,\max_{k_C}}}{\sum_{k_C \in l_C} w_{k_C,\max_{k_C}}} \end{aligned} \tag{5.1}$$

w_{k_1,k_2} transforms the definition into a weighted average of cosine similarities. Intuitively, keyword pairs that rank highly in their lists and that are similar contribute the most to the overall similarity between the two lists. In contrast, irrelevant and dissimilar keyword pairs contribute the least. For reference, increasing the value of d (the λ parameter in a negative exponential distribution) increases the weighting of more important keywords and decreases the weighting of less important ones, and vice versa.

5.4 Engineering challenges

5.4.1 The IEEE Xplore resource database

The IEEE Xplore database is one of the most well-known research databases for electrical engineering and computer science, with nearly 6 million records of academic resources as of June 2023 [37]. During development, the database has been extensively tested and experimented with because of its well-documented API and the wealth of research data it provides, including DOI numbers, citation counts, and author-defined keywords. However, its API, along with many others, does come with some limitations that highlight the fundamental problem of overly relying on an API's search interface in a source-independent system.

Lack of support for multi-author queries

Firstly, the IEEE Xplore API does not support search queries for resources with multiple matching authors. Rather, it only supports queries for one author at a time. To cater for this, the `IEEEXploreQueryBuilder` class had to be adapted, where the query builder would send one individual API request per author for queries with more than one author, setting the API parameter for the number of results to return as an excessively large number. For example, the following three API requests are sent to the IEEE Xplore API as part of a single 20-candidate recommendation request for a paper authored by Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla:

```
https://.../search/articles?max_records=100&author=vijay%2Bbadrinarayanan
https://.../search/articles?max_records=100&author=alex%2Bkendall
https://.../search/articles?max_records=100&author=roberto%2Bcipolla
```

Once all the results had been collected, they would be filtered iteratively, and only resources with all the required authors from the query would be kept and returned as the final output by the query builder. Therefore, future development of the recommender system should further consider how to cater for APIs with limited search interfaces in a more generalised manner.

API request rate limits

Secondly, like many other databases that provide their services as web APIs, the IEEE Xplore API restricts the number of API requests a user can make to 200 per day. This has proven to be a hindrance to software development during phases when the system had to be repeatedly run and tested for compilation and runtime errors. As a development-oriented solution, a rudimentary persistent cache system has been implemented in multiple classes, including the IEEE Xplore query builder and the S2AG adapter. After every API request made to IEEE Xplore, the API response data is stored in a `dict` object with the request URL as the cache key, which is then written to a persistent JSON file and stored locally on the machine. Before every API request, the system reads the file and checks if a record exists for the required request URL. If so, the system uses the cached result instead of sending a new request to IEEE Xplore. During development when the system is run on an extremely limited number of target resources, the API request URLs are largely the same for every run, and thus the persistent cache significantly reduces the number of API calls made to IEEE Xplore.

However, additional engineering would be required before such a caching system could be implemented in production. A problem that could easily arise in production is managing the size of the cache when the number of target resources the system is exposed to is significantly increased. More advanced cache replacement policies, such as the Least Recently Used (LRU) policy or Least Frequently Used (LFU) policy, would need to be considered. Another problem is managing stale cache results. Time-sensitive information such as citation counts would need to be constantly updated in order to maintain the 'correctness' of the system's recommendation results. Furthermore, it is not uncommon for authors to make updates to their publications on research databases after the initial publication date.

5.4.2 Citation database adapters and processing time

In order to implement citation database adapters, additional engineering was required to reduce the per-request processing time of the system. Whilst the number of API requests made to resource databases grows linearly to the number of resource databases supported by the system, the number of requests made to citation databases grows dramatically more quickly, as a request would need to be made for every candidate resource considered by the system. The S2AG API itself provides the option to query a batch of academic papers within a single request by providing their DOI numbers as part of the POST request's body. However, this information is not necessarily available for every academic paper returned by resource databases, let alone any generic academic resource.

Therefore, the solution was to utilise parallelism and send the large collection of API requests in batches concurrently rather than sequentially. The AIOHTTP third-party library was used to achieve this effect within the Python runtime, which itself has only recently introduced support for asynchronous and concurrent programming in newer versions of the language. The number of requests in each concurrent batch has been capped to cater for the API request rate limit of the S2AG API, which is 5,000 requests per 5 minutes for non-partners and 100 requests per second for partners with registered API keys. This solution has led to the critical reduction in processing time per request from a typical magnitude of minutes to seconds.

Chapter 6

Evaluation

This chapter explains the experiments that were designed to analyse the resultant recommender system, reports the evaluation results obtained, and compares and contrasts these results with other existing solutions. As part of evaluating a consumer-oriented software product, the metrics introduced here are aimed at reflecting the typical service-level agreement (SLA) terms that would be included in a recommender system of this nature, primarily in terms of performance and recommendation quality.

6.1 Experimental setup

For each of the following experiments, the recommender system was run across 9 sample academic papers as target resources individually, not collectively, either sourced from the arXiv or IEEE Xplore database. The sample papers were equally divided into three subject areas – human-computer interaction, machine learning, and earth and planetary astrophysics. Each run would target a specific metric, and the final value of the metric would be the aggregated mean of the 9 runs. As previously mentioned, because of the request rate limits that many resource databases, including IEEE Xplore and Semantic Scholar, impose on their general API users (i.e. non-partners), the experiments had to be run on a limited number of sample resources and a limited number of candidates per request.

For each of the following experiments, apart from hyperparameters that were independent variables, all other hyperparameters (acting as controlled variables) were set to the fixed default values as shown in Table 6.1. This list corresponds to a subset of the hyperparameter values found in `models/hyperparams.py` in the source code, the hyperparameter file of the recommender system.

Hyperparameter	**Value**
Disable the use of cached data	False
Number of query keywords extracted (i.e. keywords extracted from each target resource during query generation)	10
Number of candidate resources extracted (from all resource databases combined)	80
Collaborative filtering method used	User-based
Keyword extraction method used	TF-IDF
Number of ranking keywords extracted (i.e. keywords extracted from each candidate resource during keyword-based ranking)	20

Table 6.1: The fixed default hyperparameter values of the recommender system.

6.2 Performance

It is crucial for any web API to produce desired results whilst demonstrating high computational performance and efficiency. The following experiments show how varying certain key hyperparameters affect the recommender system's performance and highlight where areas of performance-quality trade-off may be.

For fairness, the use of cached data has been disabled in performance-related experiments, since the metrics could easily be influenced by extraneous variables such as what academic resources the recommender system had been 'exposed to' prior to the experiments.

6.2.1 Metrics

Processing time is defined as the time taken for the recommender system to handle a single recommendation request from start to finish. This does not include the time taken to send and receive the request to and from the client, which would otherwise be the *response time*.

Peak memory usage is defined as the maximum size of memory allocated during the processing of a single recommendation request. For these experiments, the memory allocated by the language model used for keyword extraction ($\approx$ 65 MB for the 50-dimensional GloVe matrix) has been removed from the metric, as this size is constant throughout the operation of the recommender system, and is pre-loaded into memory as the system starts up, rather than being loaded on a per-request basis. Therefore, this metric assesses the additional load a hosting machine would have to endure based on other variable factors on a per-request basis, such as data gathering and candidate ranking.

6.2.2 Results

Tables 6.2 and 6.3 show the performance-related evaluation results obtained from the experiments. For reference, the experiments were run on a local machine with a 2.3 GHz 8-Core Intel Core i9-9880H processor on a Unix-based operating system.

# of candidates	Mean processing time	Mean peak memory usage
40	9.9467	2.8847
80	14.0138	4.6546
120	17.7040	6.6769
160	21.9788	8.1756
200	25.9383	9.9547

Table 6.2: A demonstration of the number of candidate resources extracted influencing mean processing time (in seconds) and mean peak memory usage (in MB).

# of ranking keywords	Mean processing time	Mean peak memory usage
10	12.5068	4.9089
20	14.4647	4.8769
30	18.1410	4.8295
40	21.0353	4.2121

Table 6.3: A demonstration of the number of ranking keywords extracted influencing mean processing time (in seconds) and mean peak memory usage (in MB).

6.3 Recommendation quality

It is difficult to assess recommendation quality on an absolute scale for any recommender system, since what makes a ‘good’ recommendation greatly depends on the user’s personal preferences and the specific use case. Therefore, the evaluation process of this recommender system has avoided the use of human subjects and the collection of subjective user feedback but instead relied on more objective quantitative metrics on a per-component basis to highlight how the recommender system is capable of returning results that are related to the target.

6.3.1 Metrics

Keyword extraction accuracy (KEA) assesses the keyword extraction method used by the recommender system, and is defined as the proportion of extracted keywords that match up with a pre-defined keyword assigned to the target resource, or mathematically:

$$\text{KEA} = \frac{\text{number of matching extracted keywords}}{\text{number of extracted keywords}}$$

For key *phrases*, two key phrases are considered a 'match' if at least half of the words in the key phrases are the same, or mathematically:

$$\text{match} := ||\text{extracted phrase} \cap \text{pre-defined phrase}|| >= 0.5 \times ||\text{pre-defined phrase}||$$

This is to account for slight morphological differences in words that may affect the comparison. For example, 'pixel-wise classification' and 'pixel-wise classifications' would be considered a 'match'.

Classification accuracy (CA) assesses how closely related in subject area the recommendations are compared to the target. Experimentally, it is defined as the proportion of recommendations within the top 10 that belong to the same assigned subject area category as the target. However, some subject areas can be relatively broad, whereas others can be niche, and some can be subsets of others. For example, consider 'cs.HC', an arXiv subject area classification. 'cs' indicates that a paper broadly falls into the computer science subject area, whereas 'HC' indicates that the paper is specifically related to human-computer interaction, a subtopic of computer science. Therefore, it is also necessary to define *macro-CA*, the proportion of recommendations with matching umbrella fields of study, and *micro-CA*, the proportion with matching specific research areas. For these experiments, the subject area classifications as defined by arXiv are used as the 'gold standard'.

6.3.2 Results

Tables 6.4, 6.5, 6.6, 6.7, and 6.8 show the recommendation quality-related evaluation results obtained from the experiments.

Extraction method	KEA
TF-IDF	0.2556
TextRank	0.1537

Table 6.4: A demonstration of the keyword extraction method used influencing keyword extraction accuracy (KEA).

6.4 Discussion

Overall, PolarRec has achieved respectable performance and recommendation quality results given its current lack of data sources and its small memory footprint. The

Dimension	Macro-CA	Micro-CA
50	0.8728	0.4738
100	0.8592	0.5179
200	0.8922	0.5373

Table 6.5: A demonstration of the dimensional size of the pre-trained GloVe word embedding matrix used influencing classification accuracy (CA).

# of candidates	Macro-CA	Micro-CA
40	0.8247	0.4586
80	0.8728	0.4738
120	0.8605	0.4630
160	0.9037	0.5593
200	0.9160	0.5778

Table 6.6: A demonstration of the number of candidate resources extracted influencing classification accuracy (CA).

system has achieved a maximum macro-classification accuracy of 92% and micro-classification accuracy of 58% when given 100 candidates and using 20 ranking keywords per resource. This result can be compared with the k-means clustering- and k-NN-based recommender system introduced by Lee et al. [18], where it achieved a significantly better micro-classification accuracy of 89% based on similar subject areas through slightly different experimental setups. The k-NN recommender system was evaluated within an environment where 10,386 candidate papers were constantly available to the ranking algorithm through the use of a database, whereas the number of candidate papers the PolarRec system was exposed to per request was relatively limited.

It is worth noting that given the same number of candidates and ranking keywords per candidate, CCF methods for author- and citation-based ranking resulted in a higher CA compared to UCF, as demonstrated in Table 6.8, highlighting the benefits of using co-occurrence relations compared to explicit cosine similarity measures. Given the limited number of candidates the system is exposed to per request, the probabilities of two resources sharing the same authors or the same referenced resources are relatively low, resulting in a very sparse CF relation matrix for both authors and citations. This is similar to the ‘cold-start’ problem that most CF algorithms face when supplied with insufficient user-item relational data. Using UCF leads to many author and citation comparisons that result in zero similarity, providing the author and citation rankers with little useful information. In contrast, CCF is capable of finding implicit relations between author and reference lists that have no explicit overlap using association matrices, meaning fewer comparison operations result in zero similarity, thus providing more useful information for ranking.

# of ranking keywords	Macro-CA	Micro-CA
10	0.9062	0.4772
20	0.8728	0.4738
30	0.8713	0.4802
40	0.8552	0.4145

Table 6.7: A demonstration of the number of ranking keywords extracted influencing classification accuracy (CA).

CF method	Macro-CA	Micro-CA
User-based CF	0.8728	0.4738
Context-based CF	0.8750	0.5392

Table 6.8: A demonstration of the collaborative filtering (CF) method used influencing classification accuracy (CA).

6.4.1 Performance-quality trade-off

There is a very clear performance-quality trade-off suggested through the experimental results. Specifically, whilst the classification accuracy of the system generally increases as the number of candidates increases (Table 6.6), the mean processing time and mean peak memory usage increase as well (Table 6.2). However, whilst the mean processing time increases linearly to the number of candidates (which implies an approximately $O(n)$ time complexity where n is the number of candidates), the rate at which CA increases slows down as the number of candidates reaches approximately 160. This suggests that it is less justified to set the number of candidates to be above 160 in a production system in order to maintain a reasonable processing time per request.

Remarkably, CA generally increases as the number of ranking keywords used *decreases* (Table 6.7). As expected, increasing the number of ranking keywords also increases the computational load of the system, and thus increases processing time linearly (Table 6.3), suggesting that the ideal number of ranking keywords used is actually a smaller number. One reason could be because, according to experimentation, the top 10 keywords extracted from a paper's abstract typically best represent its semantic content in full. Keywords that are lower in ranking than the top 10 may be more irrelevant such that they skew the semantic representation of a paper, negatively affecting the keyword similarity comparison operations in the keyword ranker, and thus reducing the number of relevant recommendations that appear at the top of the recommendations list.

It is worth noting that whilst increasing the dimensional size of the word embedding matrix used does not have significant effects on macro-CA, it does increase micro-CA, helping make recommendations more relevant to the targets' precise research area. However, whether it is worth moving to a word embedding matrix of a larger

KEA	Macro-CA	Micro-CA
0.2556	0.8812	0.5483

Table 6.9: The final values for keyword extraction accuracy (KEA) and classification accuracy (CA) of the recommender system using a combination of all the ideal hyper-parameter values according to experimentation.

dimension depends greatly on the machine the system is run on, as not only does this increase the computational load upon every keyword similarity comparison operation, but it also significantly increases the memory required during system start-up when the model is pre-loaded. For reference, for the same corpus, the 50-dimensional GloVe matrix on Gensim is 65.9776 MB in size, whereas the 200-dimensional GloVe matrix is 252.0913 MB in size, suggesting a significant increase in memory footprint for a slight increase in accuracy.

In terms of KEA, the TextRank algorithm has performed significantly worse than the more traditional TF-IDF method for keyword extraction. Both TF-IDF and TextRank actively filter common articles such as ‘a’, ‘an’, and ‘the’. However, the difference in performance suggests that keywords in a piece of text are better identified using their occurrence frequencies rather than word-to-word co-occurrence. For future development, it is worth investigating whether this is to do with the limited amount of text in abstracts and whether TextRank performs any better for longer pieces of text such as the full text from research papers.

6.4.2 Metric limitations

While CA is a good indication of whether the system is capable of recommending resources that belong to the same subject area, it is still limited in indicating whether the recommendations are truly relevant. As Lee et al. [18] states in the evaluation of their k-NN-based system, researchers often conduct research across multiple subject areas and disciplines, and furthermore, two resources that belong in different subject areas do not necessarily mean they are irrelevant. For example, a research paper on robotics could be highly relevant to both machine learning and human-computer interaction. However, one paper classified under robotics and one classified under machine learning would still be regarded as a mismatch by the CA metric regardless.

While monitoring the KEA of the system can help the development and improvement of the keyword extraction algorithm, it is, again, not necessarily an accurate indication of the recommendation quality. Extracting representative keywords from text helps both in making relevant keyword-based queries to resource databases and in comparing whether two resources contain similar semantic content. However, apart from the quality of the query, the results that resource databases return based on search queries highly depend on the implementation of their search interface, coinciding with the fundamental limitations of any completely source-independent system.

Furthermore, the current definition of the KEA metric highly depends on the keywords the authors have chosen to represent the resource. As is with any form of human-driven sample labelling, there is always some form of labeller variability, inaccuracy and bias. For example, some keywords generated by the keyword extraction algorithm that do not appear at all in the authors' lists of keywords do not necessarily mean they are not relevant to the resource. An improved KEA metric could use keyword list similarity comparisons, the same as those found in the keyword ranker, to compare how similar the list of generated keywords is to the author-defined list. However, these particular experiments have avoided this approach because using another sub-component of the system (the keyword ranker) to evaluate itself could introduce unwanted bias.

6.5 Comparison with Semantic Scholar

To qualitatively evaluate PolarRec, it is worth comparing its web API with one of its latest key competitors. The Semantic Scholar (S2) open data platform [34], set up by the Allen Institute for Artificial Intelligence in 2015, is a large-scale scientific literature data platform with over 200 million papers, 550 million paper-authorship edges, and 2.4 billion citation edges as of June 2023. The platform excels at paper search queries, paper summarisation, and paper recommendation by storing a large knowledge graph of all its papers in its database. The advanced data processing pipeline consists of the following components:

- a *PDF Content Extraction* system which dynamically structures both textual and graphical information within the PDFs of research papers, including paragraphs, graphs, tables, and diagrams,
- a *Knowledge Graph Construction* system that deduplicates papers, disambiguates authors, and creates edge relationships between papers and authors,
- and crucially, a semantic analysis process that is capable of TLDR summarisation and the generation of *paper embeddings* – fixed-dimensional vector representations of the semantic meanings of papers, analogous to how word embeddings represent the semantic meaning of words, using the Bidirectional Encoder Representations from Transformers (BERT) language model architecture.

In particular, the S2 platform consists of a recommendation model, offered to users in the form of the Semantic Scholar Recommendations (S2R) web API. A typical recommendation process consists of candidate selection and candidate ranking stages, similar to the structure of PolarRec's algorithm. However, instead of querying external resource databases for candidate selection, S2R uses the existing paper embedding vectors from the aforementioned data processing pipeline and uses a k-NN method to cluster candidate papers that are closest in vector similarity to the targets. The candidate ranking model is implemented using two trained Support Vector Machine

(SVM) models, where one model is trained on the TF-IDF keyword representations of papers, whilst the other is trained on paper embedding vectors. The candidates are ranked using the average of the two scores generated by the two models.

Whilst PolarRec and S2R offer very similar services to their users on the surface, the methods through which they achieve their objectives are notably different, with implications towards the differences in their respective target audiences. Therefore, it is worth comparing these two services in-depth, especially to justify the reasons behind developing the custom PolarRec web API as the back-end of the Zotero plugin, as opposed to directly developing a Zotero plugin for the S2R API. The following sections each discuss and compare an individual aspect where PolarRec and S2R differ.

6.5.1 Data processing

PolarRec puts an emphasis on scalability and greater flexibility towards the data sources it supports because of the wide variety of academic content and content types that are supported by the Zotero platform. This objective has led to a software architecture where academic resources are not permanently stored in large-scale proprietary databases, but instead, scraped from external sources upon request, unlike S2R. Scraped data is always up-to-date relative to what the public is able to view. This is evident in its user interface, where users are able to choose the sources from which their recommendations are extracted, as well as the ability to generate recommendations for all content types, not just research papers. In contrast, the contents of the S2 database need to be regularly maintained and updated through version numbering.

Furthermore, the S2 platform processes scholarly content from both public and proprietary data sources, meaning proprietary research data that would normally not be available to the general public is incorporated into the candidate ranking process of the recommendation model, specifically the full text of non-open access research papers. While this provides the ranking model with an abundance of semantic information for each paper, this goes against the ideas of a ‘source-independent framework’, where only public, open-access data should be relied upon. In contrast, PolarRec only processes information that is either in the user’s own Zotero library or publicly available data offered through web APIs such as titles, author lists, and abstracts.

6.5.2 User interface

The S2 recommendation model is currently offered exclusively as a web API. In order to make a recommendation request for a single paper, the user must know the unique identifier for the paper in advance, which can either be a DOI number or some form of ID used in other research databases including arXiv, the Association for Computational Linguistics (ACL), and S2. If this information is not initially available, the user would need to manually make a search query for the paper based on its title or keywords, find the correct paper from the list of results, and obtain a unique identifier from there.

In contrast, through PolarRec's web API, the user is able to flexibly input metadata for the target resource directly via data fields, including the resource's title, authors, and abstract, without the need for a DOI number or ID. Again, this enables further flexibility towards the types of resources the system can support and de-restricts the system from only recommending resources that are included in its own database, which is a crucial element of this project considering the wide variety of academic resources that Zotero libraries support. Furthermore, with the addition of a Zotero plugin directly integrated into the Zotero UI, the user can avoid having to manually input metadata and send HTTP requests themselves, but instead, request and view recommendations for items in their library through the single click of a button.

However, one element the PolarRec interface lacks in comparison is the ability to define *positive* and *negative* target resources, which is offered in S2R. This idea is based on the k-NN algorithm of the candidate selection process, where some resources can be treated as centroids for positive samples and some as centroids for negative samples. By doing so, the user can better specify the types of recommendations they would like to see, instead of relying on the example of a single paper.

6.5.3 Recommendation quality

In order to compare the differences in recommendation quality between PolarRec and S2R, an attempt at defining a third quality-related evaluation metric was made. Specifically, the calculation of *recommendation accuracy (RA)* takes the top 30 recommendation results from both PolarRec and S2R for the same target resource, and counts the number of matching resources m from the two lists of recommendations. The value of RA is thus mathematically defined as $\frac{m}{30}$. Effectively, the metric takes the recommendations of S2R as a 'gold standard' and compares how similar PolarRec's recommendations are. However, running the aforementioned experiments with every available variation of the set of hyperparameters all resulted in RA values of 0 or nearly 0.

There are multiple reasons why this may be, and why this is not necessarily an indication of poor performance from the PolarRec system. Firstly, the S2 platform processes scholarly content from a vastly wider range of external databases and sources than the demonstration version of PolarRec. The number of candidates available to the S2R ranking model is therefore significantly greater, and many candidates of better quality may be left undiscovered by the PolarRec system. In relation to this point, and as mentioned before, the S2 platform processes scholarly content from both public and proprietary data sources, and its data processing pipeline requires proprietary information from all papers it processes. The information that the S2R ranking model is exposed to for each paper is thus significantly greater compared to PolarRec's, which can have a direct effect on its performance. Furthermore, there is a significant difference in the ranking models themselves. Whilst PolarRec considers a variety of factors when ranking candidates, including authors, references, and extracted keywords from abstracts, the S2R ranking model puts a much stronger emphasis on the similarities between the semantic textual content of papers, as evident in its use of TF-IDF and paper embeddings. The criteria on which the two ranking models are based thus have a direct effect on what they respectively consider as ‘good recommendations’, causing the notable disjointedness in their top recommendations lists.

Chapter 7

Conclusions and Further Work

This project report has introduced PolarRec, a lightweight, scalable, and source-independent academic resource recommender system, which allows its users to input target resources they would like recommendations for, either through the Zotero plugin or web API, and generates lists of recommendations based on a variety of factors including keyword list similarity, author list similarity, reference list similarity, and citation count.

Through explaining the detailed implementation of PolarRec, the report has explained, analysed, and compared the different approaches to building the components for each stage of the recommendation algorithm:

1. **Data gathering.** Resource database adapters and citation database adapters, classes that follow the adapter design pattern, are used to generate candidate resources and aggregate citation data for them respectively by communicating with external research databases. Generating high-quality search queries is crucially enabled by keyword extraction algorithms including TF-IDF and TextRank, where the former approach excelled in accuracy for this particular implementation. Support for parallelised HTTP requests had to be engineered to reduce processing time per request to a reasonable level.

2. **Data representation.** The `Resource` class unifies the data representation of an academic resource throughout the entire system from input to output. A resource is primarily represented by its title, abstract, author list, and reference list, as far as ranking is concerned. *Comparable strings* had to be introduced to enable more accurate candidate deduplication, since different data sources can format resource title strings differently, and reduce the spacial complexity of the algorithm.

3. **Resource ranking.** Four resource rankers make up the ranking algorithm, namely the keyword ranker, author ranker, citation ranker, and citation count ranker (in order of their importance to the algorithm). The keyword ranker

uses GloVe word embedding vectors to represent the semantic meaning of each resource's keyword lists, and uses *keyword list similarity* to compare the relevance between candidates and targets, taking into account each keyword's pair-wise cosine similarity and its importance to its respective candidate. The author and citation rankers use either UCF or CCF to compare the association between a candidate's author and reference lists and the targets', with CCF showing better performance in this instance because of its ability to infer implicit associations between lists when there are no direct overlaps in authors or references. Finally, the citation count ranker uses citation databases to retrieve the number of times each candidate has been cited by other resources, acting as a metric for how authoritative a particular candidate is in their respective fields of study.

The main contribution of this project is towards streamlining the process researchers and students go through to discover academic resources of interest. Not only do Zotero users now have a fully-integrated user interface with which they can use to find recommendations, but the system itself, because of its source-independent architecture, has the potential to recommend a much wider variety of resource types than many existing solutions which exclusively handle academic and research papers. Data source independence enables the system to be easily scaled up to support a large number of research databases and ranking algorithms, providing many more options for users to filter through recommendation results and find results that match their personal ranking criteria.

The following sections discuss areas in which the recommender system is still lacking and can be improved upon, and where opportunities for future development lie.

7.1 Processing time

The evaluation process shows that whilst the system can be run within a small memory footprint, its processing time per request is still far from ideal, where most web APIs in general would be expected to return results in the magnitude of milliseconds or single-digit seconds. Using databases to cache API response data or recommendation results directly could significantly reduce processing time. The rudimentary caching method introduced for the IEEE Xplore query builder and S2AG adapter demonstrates the potential of this approach. For 80-candidate requests run on the sample resources used for evaluation, the mean processing time was reduced by 30% from 14.0138 seconds to 9.0935 seconds with caching enabled. For 200-candidate requests, the time was reduced by 50% from 25.9383 seconds to 13.0879 seconds. However, it is worth investigating whether this approach would be significantly less memory efficient compared to managing a proprietary resource database like other existing solutions. For reference, during the development process, 4,284 candidates' reference lists were stored locally in the S2AG adapter's persistent cache, which resulted in a cache size of 28.7100 MB, meaning an average of 6.8625 KB per reference list record. It is also worth comparing whether storing results in a local database or setting up a separate microservice to handle caching would be the preferable approach.

Aside from the data-gathering stage, improvements can also be made to the resource ranking stage of the algorithm. Further performance profiling experiments showed that the keyword ranker occupies approximately 90% of the execution time of the entire ranking algorithm, as illustrated in Figure B.3. Attempts had been made to add support for concurrency to the implementation of the keyword ranker, where the keyword list similarity calculations for each candidate would be done in parallel within a multi-process worker pool. However, these attempts did not significantly reduce the execution time of the keyword ranker due to the relatively large size of the word embedding matrix and the memory transfer overhead between processes that was associated with it.

7.2 Keyword extraction

For academic papers, the keywords extracted by both TF-IDF and TextRank are very much limited to the words that appear in the title and abstract, which is one of the causes for low KEA. Mihalcea and Tarau [5] demonstrated that with ideal hyperparameters and with the ability to perform keyword *generation*, they achieved a maximum precision of 31.2% using a metric that is similar in definition to KEA. The exact keyword generation methods that can be applied to this system should be further explored. One possibility would be to train an LLM, either LSTM- or transformer-based, in keyword extraction and generation, given that a sufficiently large and diverse dataset of academic papers with author-defined keywords can be collated.

Additionally, further development should also include adding support for extracting textual summaries for resource types other than academic papers, including websites, PDFs, and books. Zotero itself offers some support for this, as a copy of a website's or PDF's full text is often stored in the user's Zotero library when a user saves an online resource or imports it from elsewhere on the machine.

7.3 Resource ranking

The citation count ranker may give some insight into the authoritativeness and popularity of a candidate resource, but in its current implementation, unwanted bias may be introduced, since the mean citation counts of papers often differ slightly depending on the subject area. Some research topics may be much more popular than others. This metric should therefore be improved to be relative to the candidate's subject area instead.

To further improve the keyword ranker, a custom word embedding matrix could be constructed by training a custom word2vec model specifically on academic writing instead of using a pre-trained model. This allows the model to learn word embedding representations that are more accurate to the semantic meaning of words in the context of academia and research. For example, the word 'paper' itself has a very different meaning in academia compared to its other meaning when referring to the physical object used for writing on. Learning more accurate representations within context helps improve the accuracy of keyword similarity comparison operations, and thus enable the keyword ranker to assign rankings more accurately.

7.4 Subject area diversity

One vital improvement that can be made to the development process of this recommender system, and almost all the existing proposals mentioned before, is their exclusive use of STEM-related research databases in their data gatherers, affecting how their modellers are trained and tuned, and therefore restricting the research domains in which these recommender systems can be used. For example, Lee et al. [18] use IEEE Xplore, an engineering- and technology-oriented research library, as their database source, and solely use computer science-related papers as part of their evaluation process. Nascimento et al. [2] also use IEEE Xplore, as well as ScienceDirect, as their database source as part of their evaluation. Meanwhile, Hassan [19] uses the PubMed database, a library oriented towards biological and medical sciences. In the case of PolarRec, only the arXiv and IEEE Xplore databases were used in the development and evaluation process, where both databases are very much oriented towards research topics within science and engineering.

Even though the PolarRec system was designed with data source diversity and scalability in mind, improvements can still be made towards future recommender systems being more inclusive, and covering more research domains such as those in arts, humanities and social sciences. This could potentially be done simply by changing the data sources that are used to shape the models and prediction algorithms in these proposals.

Bibliography

[1] Michael Fire and Carlos Guestrin. Over-optimization of academic publishing metrics: Observing Goodhart's Law in action. *GigaScience*, 8, June 2019. doi: 10.1093/gigascience/giz053. pages i

[2] Cristiano Nascimento, Alberto H.F. Laender, Altigran S. da Silva, and Marcos André Gonçalves. A Source Independent Framework for Research Paper Recommendation. In *Proceedings of the 11th Annual International ACM/IEEE Joint Conference on Digital Libraries*, pages 297–306, Ottawa Ontario Canada, June 2011. ACM. ISBN 978-1-4503-0744-4. doi: 10.1145/1998076.1998132. URL `https://dl.acm.org/doi/10.1145/1998076.1998132`. Cited 2023-01-19. pages i, 2, 3, 6, 12, 16, 50

[3] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL `http://aclweb.org/anthology/D14-1162`. Cited 2023-04-16. pages i, 31

[4] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing Co., Inc., USA, 1989. ISBN 978-0-201-12227-5. pages i, 5, 24

[5] Rada Mihalcea and Paul Tarau. TextRank: Bringing Order into Text. In *Conference on Empirical Methods in Natural Language Processing*, July 2004. URL `https://www.semanticscholar.org/paper/TextRank%3A-Bringing-Order-into-Text-Mihalcea-Tarau/7b95d389bc6affe6a127d53b04bcfd68138f1a1a`. Cited 2023-06-18. pages i, 24, 25, 49

[6] Haifeng Liu, Xiangjie Kong, Xiaomei Bai, Wei Wang, Teshome Megersa Bekele, and Feng Xia. Context-Based Collaborative Filtering for Citation Recommendation. *IEEE Access*, 3:1695–1703, 2015. ISSN 2169-3536. doi: 10.1109/ACCESS.2015.2481320. pages i, 9, 10

[7] Corporation for Digital Scholarship. Zotero — Your personal research assistant, 2023. URL `https://www.zotero.org/`. Cited 2023-01-20. pages 1

[8] Corporation for Digital Scholarship. Plugins [Zotero Documentation], 2023. URL `https://www.zotero.org/support/plugins`. Cited 2023-01-20. pages 1

[9] Corporation for Digital Scholarship. Dev:client coding:javascript api [Zotero Documentation], 2023. URL `https://www.zotero.org/support/dev/client_coding/javascript_api`. Cited 2023-06-08. pages 1, 15, 17

[10] Corporation for Digital Scholarship. Dev:client coding:plugin development [Zotero Documentation], 2023. URL `https://www.zotero.org/support/dev/client_coding/plugin_development`. Cited 2023-01-20. pages 1

[11] Erik Schnetter. Zotero Citation Counts Manager, June 2023. URL `https://github.com/eschnett/zotero-citationcounts`. Cited 2023-06-01. pages 1

[12] Joscha Legewie. ZotFile: Advanced PDF management for Zotero, June 2023. URL `https://github.com/jlegewie/zotfile`. Cited 2023-06-01. pages 1

[13] windingwind. Zotero Better Notes, June 2023. URL `https://github.com/windingwind/zotero-better-notes`. Cited 2023-06-01. pages 1

[14] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, June 2005. ISSN 1041-4347. doi: 10.1109/TKDE.2005.99. URL `http://ieeexplore.ieee.org/document/1423975/`. Cited 2023-01-19. pages 4, 5

[15] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 175–186, New York, NY, USA, October 1994. Association for Computing Machinery. ISBN 978-0-89791-689-9. doi: 10.1145/192844.192905. URL `https://dl.acm.org/doi/10.1145/192844.192905`. Cited 2023-06-01. pages 4

[16] Corporation for Digital Scholarship. Dev:web api:v3:oauth [Zotero Documentation], 2023. URL `https://www.zotero.org/support/dev/web_api/v3/oauth`. Cited 2023-01-20. pages 5

[17] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 210–217, USA, May 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 978-0-201-84705-5. doi:

10.1145/223904.223931. URL https://dl.acm.org/doi/10.1145/223904.223931. Cited 2023-06-01. pages 5

[18] Joonseok Lee, Kisung Lee, and Jennifer G. Kim. Personalized Academic Research Paper Recommendation System, April 2013. URL http://arxiv.org/abs/1304.5457. Cited 2023-01-18. pages 7, 8, 40, 42, 50

[19] Hebatallah A. Mohamed Hassan. Personalized Research Paper Recommendation using Deep Learning. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, pages 327–330, Bratislava Slovakia, July 2017. ACM. ISBN 978-1-4503-4635-1. doi: 10.1145/3079628.3079708. URL https://dl.acm.org/doi/10.1145/3079628.3079708. Cited 2023-01-20. pages 8, 50

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In C. J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf. pages 8, 30

[21] Simon Brown. The C4 model for visualising software architecture, 2023. URL https://c4model.com/. Cited 2023-06-16. pages 13

[22] windingwind. Zotero Plugin Template, June 2023. URL https://github.com/windingwind/zotero-plugin-template. Cited 2023-06-08. pages 14, 17, 57

[23] StrongLoop and IBM. Express - Node.js web application framework, 2023. URL https://expressjs.com/. Cited 2023-06-14. pages 14

[24] Taylor Otwell. Laravel - The PHP Framework For Web Artisans, 2023. URL https://laravel.com/. Cited 2023-06-14. pages 14

[25] Pallets. Welcome to Flask — Flask Documentation (2.3.x), 2023. URL https://flask.palletsprojects.com/en/2.3.x/. Cited 2023-06-14. pages 14

[26] Amazon Web Services. Amazon EC2 T3 Instances – Amazon Web Services (AWS), 2023. URL https://aws.amazon.com/ec2/instance-types/t3/. Cited 2023-06-14. pages 14

[27] Amazon Web Services. Website & Web App Deployment - AWS Elastic Beanstalk - AWS, 2023. URL https://aws.amazon.com/elasticbeanstalk/. Cited 2023-06-15. pages 15

[28] Corporation for Digital Scholarship. Zotero. Zotero, June 2023. URL https://github.com/zotero/zotero. Cited 2023-06-01. pages 15, 17

[29] T. Reenskaug. The Model-View-Controller (MVC) Its Past and Present. 2003. URL `https://www.semanticscholar.org/paper/The-Model-View-Controller-(MVC)-Its-Past-and-Reenskaug/ff2ada602c96499c0f8a634e26c2c58ef8ec490f`. Cited 2023-06-15. pages 18

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 978-0-201-63361-0. pages 22

[31] Joeran Beel, Bela Gipp, Stefan Langer, and Corinna Breitinger. Research-paper recommender systems: A literature survey. *International Journal on Digital Libraries*, pages 1–34, July 2015. doi: 10.1007/s00799-015-0156-0. pages 24

[32] Lawrence Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking : Bringing Order to the Web. In *The Web Conference*, November 1999. URL `https://www.semanticscholar.org/paper/The-PageRank-Citation-Ranking-%3A-Bringing-Order-to-Page-Brin/eb82d3035849cd23578096462ba419b53198a556`. Cited 2023-06-10. pages 25

[33] Henry Maltby, Samir Khan, and Jimin Kim. Stationary Distributions of Markov Chains — Brilliant Math & Science Wiki, 2023. URL `https://brilliant.org/wiki/stationary-distributions/`. Cited 2023-06-18. pages 25

[34] Rodney Kinney, Chloe Anastasiades, Russell Authur, Iz Beltagy, Jonathan Bragg, Alexandra Buraczynski, Isabel Cachola, Stefan Candra, Yoganand Chandrasekhar, Arman Cohan, Miles Crawford, Doug Downey, Jason Dunkelberger, Oren Etzioni, Rob Evans, Sergey Feldman, Joseph Gorney, David Graham, Fangzhou Hu, and Daniel Weld. *The Semantic Scholar Open Data Platform*. January 2023. pages 26, 43

[35] Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English Gigaword Fifth Edition, June 2011. URL `https://catalog.ldc.upenn.edu/LDC2011T07`. Cited 2023-06-16. pages 31

[36] Wikipedia. Wikipedia:Statistics. *Wikipedia*, May 2023. URL `https://en.wikipedia.org/w/index.php?title=Wikipedia:Statistics&oldid=1155280342`. Cited 2023-06-16. pages 31

[37] IEEE. IEEE Xplore, 2023. URL `https://ieeexplore.ieee.org/Xplore/home.jsp`. Cited 2023-06-16. pages 33

Appendix A

Third-Party Software Libraries

Table A.1 contains a list of all the third-party software libraries that have been leveraged to build the PolarRec system as a whole.

Library	**Description**	**Licence**
AIOHTTP 3.8.4	Enables the system to send asynchronous HTTP requests concurrently within the Python runtime when communicating with resource and citation databases.	Apache License 2.0
Flask Version 2.2.3	The Python microframework on which the recommender system's web API application is based upon.	BSD-3-Clause License
Gensim 4.3.1	An open-source library that provides natural language processing functionality and pre-trained word embeddings for keyword similarity comparison during ranking.	GNU Lesser General Public License v2.1
Gensim GloVe 50-Dimensional Word Vectors Trained on Wikipedia 2014 and Gigaword 5 Data	The pre-trained GloVe word embedding matrix used for keyword similarity comparison during ranking.	Open Data Commons Public Domain Dedication and License v1.0
Gunicorn 20.1.0	A WSGI HTTP server framework used for hosting the web API application publicly on cloud servers.	`https://github.com/benoitc/gunicorn/blob/master/LICENSE`
NLTK 3.8.1	A natural language processing library whose stop words corpus is used in the keyword extraction process.	Apache License 2.0

Library	**Description**	**Licence**
NumPy 1.24.2	A mathematical library that enables multi-dimensional array and matrix calculation, primarily used for cosine similarity and collaborative filtering calculations.	BSD-3-Clause License
pandas 2.0.1	Enables the temporary storage and debugging of tabular numerical data related to keyword similarity calculations.	BSD-3-Clause License
Pylint 2.17.1	A static code analyser used for automated Python code linting and formatting during development.	GNU General Public License v2
PyTextRank 3.2.4	An implementation of the TextRank algorithm in the form of a spaCy pipeline extension in Python, used as an alternative to TF-IDF during keyword extraction.	MIT License
scikit-learn 1.2.2	Provides a TF-IDF vectoriser that extracts keywords from text.	BSD-3-Clause License
spaCy v3.5.3	An open-source library that provides natural language processing functionality, primarily enabling the use of the PyTextRank algorithm for keyword extraction.	MIT License
spaCy `en_core_web_sm` Version 3.5.0	A lightweight, CPU-optimised version of the spaCy English language pipeline model, primarily enabling the use of the PyTextRank algorithm for keyword extraction,	CC BY-SA 4.0
urllib3 1.26.15	Enables the system to send sequential HTTP requests within the Python runtime when communicating with the arXiv resource database.	MIT License
xmltodict 0.13.0	A lightweight library that translates XML data into JSON-like `dict` objects in Python, primarily used for translating API response data from the arXiv resource database.	MIT License

Library	**Description**	**Licence**
Zotero Plugin Template 0.1.6 [22]	Used for developing the Zotero plugin using the Node.js runtime, and dynamically generating the relevant boilerplate code.	GNU Affero General Public License v3.0

Table A.1: The comprehensive list of third-party libraries used in the implementation of the recommender system, as of 10 June 2023.

Appendix B

Additional Evaluation Results

Figures B.1, B.2, and B.3 are performance profiling visualisations generated by SnakeViz (2.2.0), which illustrate the cumulative execution time of each child function (rectangles near the bottom) as a proportion of the total execution time of their parent functions (rectangles near the top). Specifically, the visualisations profile the system executing a single recommendation request for one of the sample resources used for evaluation, with the hyperparameters set to their fixed default values as defined in Table 6.1.

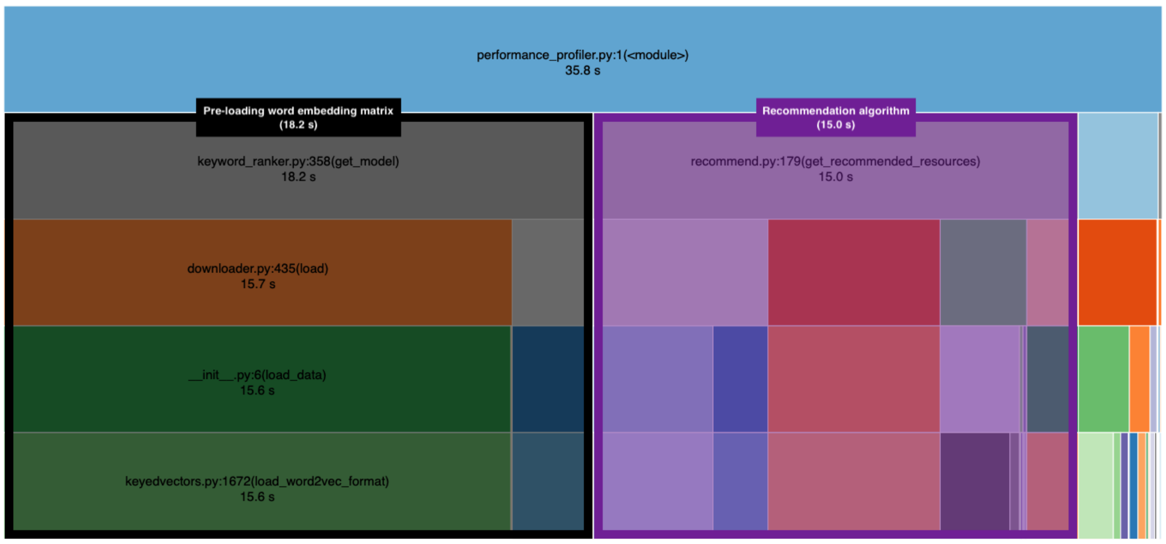


Figure B.1: A visualisation of the execution time allocation of the (top-level) main profiling function. Pre-loading the word embedding matrix uses 50.8% and processing the single recommendation request uses 49.2% of the execution time respectively.

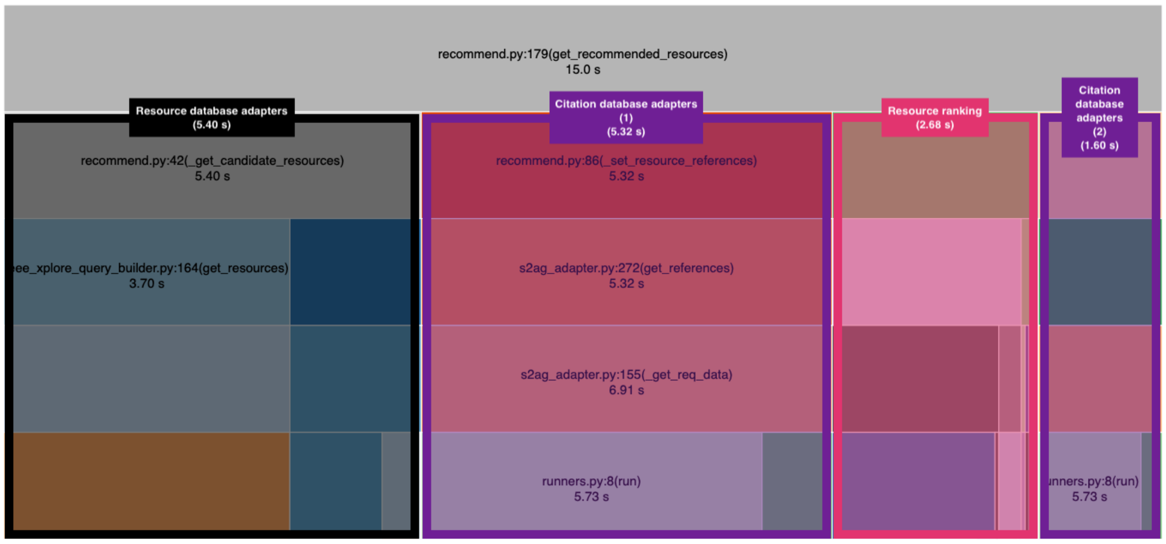


Figure B.2: A visualisation of the execution time allocation of the recommendation algorithm. Gathering candidates via resource databases uses 36.0%, gathering citation data via citation databases uses 46.1%, and resource ranking uses 17.9% of the execution time respectively.

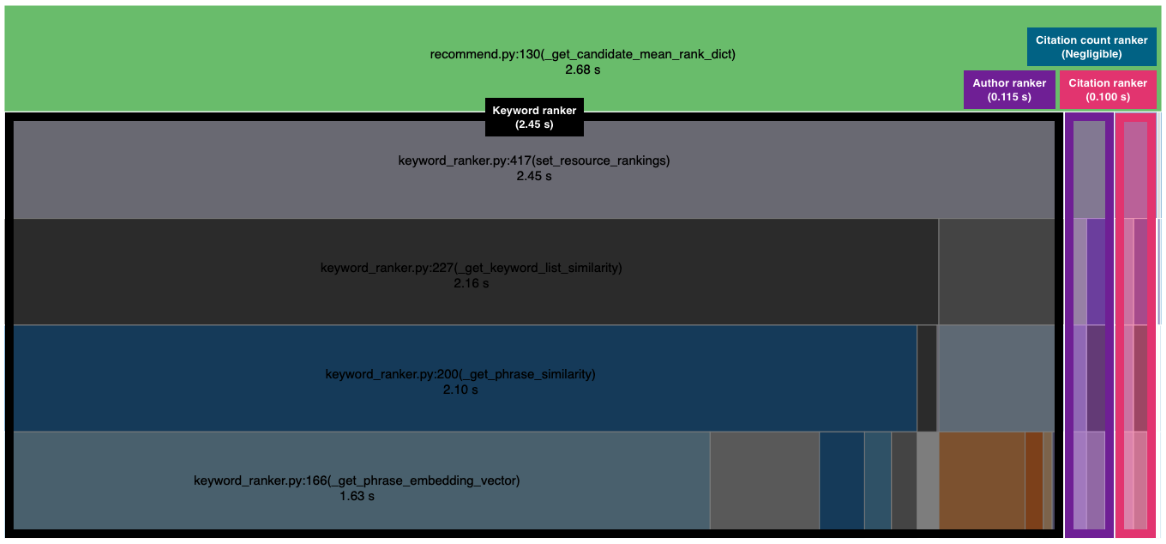


Figure B.3: A visualisation of the execution time allocation of the resource ranking algorithm. The keyword ranker uses 91.4%, the author ranker uses 4.29%, and the citation ranker uses 3.73% of the execution time respectively. The citation count ranker uses a negligible amount of time.