University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science Reliable and Secure Systems	Spring semester, 2020 Open
Authors: Kristian Gingstad and Øyvind Jekteberg	
Programme coordinator: Krisztian Balog Supervisor(s): Krisztian Balog	
Title of master's thesis: ArXivDigest: A Living Lab for Personalized Scientific Literature Recommendation	
Credits: 30	
Keywords: Living Lab, Recommender systems, personalized recommendations, Online evaluation, Topic extraction	Number of pages: 125 + supplemental material/other: Code included in PDF Stavanger, 15/06/20

Abstract

The purpose of this thesis is to explore different methods for recommending scientific literature to scientists and to explore different methods for doing topic extraction. We will update and use the already existing arXivDigest platform, which uses feedback from real users to evaluate article recommendations, to evaluate and compare these methods.

We introduce scientific literature recommendation methods based on term-based scoring, query expansion, semantic similarity and similar authors. While on topic recommendation we explore the RAKE and TextRank algorithms for topic extraction and TF-IDF weighting for topic similarity matching. These methods are all running live on the arXivDigest platform where we collect user feedback on the recommendations they provide.

We were able to get some users to sign up and use our platform, but they were unfortunately not active enough to generate sufficient interaction data by the time of submission to draw any reliable conclusions about system performance. We can however see that the arXivDigest platform is performing as it should and recommendations are submitted daily.

Acknowledgements

We would like to thank Krisztian Balog, Professor at the Department of Electrical Engineering and Computer Science at the University of Stavanger, for being our supervisor. We greatly appreciate the invaluable feedback and guidance we have received during our weekly progress meetings. His passion for the project motivated us to challenge ourselves throughout our work on this thesis.

Contents

Abbreviations

IR	**I**nformation **R**etrieval
RAKE	**R**apid **A**automatic **K**eyword **E**xtraction
POS	**P**art **O**f **S**peech
BM25	**B**est **M**atching **25**
API	**A**pplication **P**rogramming **I**nterface
TF-IDF	**T**erm **F**requency - **I**nverse **D**ocument **F**requency
NLTK	**N**atural **L**anguage **T**ool **K**it
LM	**L**anguage **M**odel
UUID	**U**niversally **U**nique **ID**entifier
JSON	**J**ava**S**script **O**bject **N**otation
CTR	**C**lick-**T**hrough **R**ate
RSS	**R**ich **S**ite **S**ummary
URL	**U**niform **R**esource **L**ocator
SQL	**S**tructured **Q**uery **L**anguage
DOI	**D**igital **O**bject **I**dentifier
AJAX	**A**synchronous **J**avascript **A**nd **X**ML
DB	**D**ata**B**ase
XML	E**X**tensible **M**arkup **L**anguage
CSV	**C**omma-**S**eparated **V**alues
HIN	**H**eterogeneous **I**nformation **N**etwork
URI	**U**niform **R**esource **I**dentififer
PDF	**P**ortable **D**ocument **F**ormat

Chapter 1

Introduction

Research papers, reports, experiments, and many other forms of scientific literature are created and released every day. For an every-day scientist or science enthusiast, it is quite a lot of work finding these new publications and keep up with the newest information available. Fortunately, there exists a service called arXiv [1] which is an open access archive and free distributor of scientific literature. The problem with this service, is that a scientist must filter though a lot of non-relevant papers each day to find the papers that are of interest to them. There are a lot of new publications released each day and this filtering process takes a long time to do by hand. Different platforms have therefore emerged to try fix this problem. CiteSeerX [2], Semantic Scholar [3], ArnetMiner [4] and arXiv Sanity Presever [5] are some examples of platforms that help their users sort trough scientific literature and find literature that are relevant for them. Some of these platforms also provide scientific literature recommendations to their users. Services like these helps scientists use their valuable time reading relevant scientific literature instead of wasting it on finding relevant literature to read.

Creating good recommendations is not always easy however. Many different algorithms and approaches exist to serve this purpose and newer algorithms or modifications to older algorithms are created frequently. Testing the different algorithms and getting accurate real world performance measurements is often difficult without testing the algorithm in a live setting because of the many factors that must be accounted for [6]. It would therefore be beneficial to have a platform where real users can interact with recommendations from a lot of different recommendation algorithms at once. This way, one could create statistics about each algorithm based on the users interactions with the recommendations and use these statistics for improving the algorithms. This would both benefit users, which would get better recommendations, and researchers who would get a service to test their new algorithm ideas in a live setting. Taking this a step further, it would also be useful having explanations for the recommendations. It has been shown that having explanations increases the persuasiveness of a recommendation, the users trust in the system and makes the user more forgiving towards bad recommendations [7, 8]. It would also be useful to recommend topics of interests to users based on their publications or reading history as having more topics gives the recommendation algorithms more information to work with. This would include topics that the users might forget to add to their profiles or topics that specifies their interests further. These are the problems we would like to address in this project.

1.1 Project History

Two years ago, in 2018, we created the arXivDigest platform [9]. This platform already performs many of the tasks that were described in the previous section. First, it provides a web interface for scientists and others with interest for scientific literature to use. Here, they can create their own profiles and receive scientific literature recommendations based on what personal information they provide. Secondly, arXivDigest also provides an API for connecting experimental recommender systems. This API has endpoints for fetching user information, information about the newest released scientific literature and for submitting personalized article recommendations. Recommendations submitted through the API are combined from multiple recommender systems in a way that makes it possible to compare user interaction with different recommender systems and use this to evaluate the performance of the recommender systems. The API is secured by API-keys, but anyone with the intent of creating and evaluating real recommender systems may apply for an API-key. This is the state we left the project at after the bachelor thesis.

1.2 Objectives

There are three main objectives in this thesis. First, we want to upgrade the arXivDigest platform infrastructure to support explanations for article recommendations, to allow for topic recommendation and some quality-of-life features for users. While doing this, we will also need to fix and improve some of the old features which could have been implemented better. Secondly, we want to research and develop novel recommender algorithms to generate scientific literature suggestions to our users. In addition, we will need to be able to create explanations for why the algorithms recommend each piece of literature. Lastly, we want to research and develop some algorithms to automatically recommend different topics that our users can add to their profiles.

1.2.1 Infrastructure Development

The application infrastructure from our bachelor thesis need to be upgraded and expanded upon to support the new features that we want to implement. The novel recommender systems that are going to interact with our application will from now on provide an explanation on the recommendations they make. We therefore need to extend the frontend web application, the API and the digest email to handle and show these explanations along with the recommendations. Since the application is going to go live, we want users to be able to leave feedback through a feedback form. The users should be able to use this feedback form to report problems or address issues with certain recommendations they were given. Other frontend changes we want are email verification of users on signup, the ability for users to unsubscribe from the digest emails and some more plots that show the performance of the different experimental recommender systems for admins and users that own recommender systems. The last major change we want to implement is to automatically recommend topics to the users. This include some new endpoints for the new topic recommender systems to interact with on the API side and a way of showing these topic recommendations to the users on the frontend side.

In short, our infrastructure objectives are:

- Support for explanations on recommendations.
- Feedback form.
- Verification of users on signup.
- Unsubscribe from digest emails.
- Extend API for topic support.
- Support for topics on the web interface.
- More statistic plots on experimental recommender systems performance.
- Code quality improvements, including modularity and robustness.

More infrastructure changes are most likely going to be added as we work on getting the system up an running and discover other changes that needs to be done.

1.2.2 Article Recommendation

Since the application is going to go live, we need to create some recommendation algorithms to provide recommendations to our users. These algorithms will run outside the arXivDigest platform and will connect with the arXivDigest API. The new algorithms must also be able to provide simple explanations on why they recommended each article. The goal is to create three or more of these recommendation algorithms that all use different approaches or techniques. At the end of the project, we can use user feedback to check each systems' performance and see how they compare to each other.

In short, our article recommendation objectives are:

- Create three or more recommendation algorithms.
- Provide an explanation with each recommendation.
- Use user feedback to compare system performance.

1.2.3 Topic Recommendation

The last main objective is to recommend topics to our users. Since article recommendations are often based on users' topics, it is important for the quality of the article recommendations that the users have many good topics on their profile. We want to create some recommendation algorithms for the topics, in the same spirit as we will do for the articles. Information on users for the creation of these topic recommendations will be fetched from the user profiles on arXivDigest and other web pages. These topics will then be shown to the users and they will have the choice of rejecting them or adding them to their profiles. The goal here is also to create two or three topic recommendation algorithms. Then, at the end of the project, we can use the user feedback to check which of the systems recommends the best topics.

In short, our topic recommendation objectives are:

- Create two or three topic recommendation algorithms.
- Fetch information about the users from other web pages.
- Use user feedback to compare the algorithms' performances.

1.3 Main Contributions

The arXivDigest infrastructure has been extended with new functionality for accommodating explanations for article recommendations and an API and user interface for topic recommendations from recommender systems. In addition to many small changes like email verification on sign-up, a feedback form, option to unsubscribe from the digest mail and more. There were also several technical improvements for example changing the platform to be an installable package instead of folders with separate script files. This made it much easier to import files and functions from other directories in our application structure. A connector was also made that is installed with the arXivDigest package and can be used to make connection to the arXivDigest API easier. It provides functions for creating a connection and easier fetching and sending of data without needing complicated code in the specific recommender system files.

For the article recommendation objective we created four different experimental recommender systems. All based on different methods for recommending the articles.

- A system that uses term-based ranking for scoring articles, using Elasticsearch.
- A System that performs query expansion on the users liked articles before scoring with Elasticsearch.
- A system that uses word2vec to semantically rerank articles.
- A system that recommends articles based on author citations.

The topic recommender systems were created in a similar manner to the article recommender systems. We ended up making three different topic recommendation algorithms in total. The topic recommender systems also comes with functions to scrape external websites for information about the users. This user information is mainly the titles of their previously published articles. The systems we created are listed below.

- A system based on the RAKE algorithm.
- A system based on the TextRank algorithm.
- A system that recommends already created topics using TF-IDF weights.

The service is running at `https://arxivdigest.org/` and the source code is available under a license at `https://github.com/iai-group/arXivDigest`.

1.4 Outline

The remainder of this thesis is structured as follows:

Chapter 2 Introduces different information retrieval concepts and other work related to scientific literature recommendation and topic extraction.

Chapter 3 goes more in depth on the infrastructure development we did to the arXivDigest platform to accommodate the new features and a smoother overall experience.

Chapter 4 presents the work done on the article recommender systems and algorithms.

Chapter 5 goes into detail about the topic recommendation systems and algorithms.

Chapter 6 presents the statistics we have collected on the experimental recommender systems that we created and discusses the results.

Chapter 7 concludes and presents suggestions for further work.

Chapter 2

Related Work

2.1 Information Retrieval

Information retrieval is the field of study concerning retrieving relevant information that satisfies an information need from a large collection. This can be elements such as images, videos, text documents and other types of information [10, 11]. In this project we are only concerned with text based information retrieval, more specifically scientific literature retrieval, and will thus only focus on the techniques relevant for this.

2.1.1 Text Preprocessing

Before any information retrieval techniques can be applied to any given query and document corpus, it is important to preprocess the text. The purpose of preprocessing text is to standardize the format of the input text and removing inconsistent and irrelevant information. This will increase the precision of the information retrieval techniques that will be applied later [12]. Preprocessing can be as simple as just lowercasing the text, but more advanced techniques may also be applied such as trying to reduce different forms of a word into a base form. We will go over the most common techniques of text preprocessing in the following sections.

Tokenization

Tokenization is the process of breaking the text up into lexical units, named tokens. The tokens may be words, numbers, symbols or sometimes more advanced units such as "New York" [12]. A naive way of tokenizing a text may be to simply split tokens on spaces. However, this may miss tokens such as hyphenated words. Splitting on symbols may fix this issue, but will again introduce new problems like splitting URIs and emails into multiple tokens. Tokenization is thus not as trivial as it first may seem, because of edge cases like these, and the fact that each language has different rules for how tokens are divided [11, 12].

Stopwords

Stopwords are words in a text that add so little value differentiating documents that they can be safely excluded without affecting the end result too much. Most of them are unimportant because they appear so often that most texts will include them regardless of content, but it can also be words that hold little semantic meaning [11]. Common examples of stopwords are words like 'the', 'a', 'and', 'is' etc. One common way to identify stopwords is by the frequency of the words in the text. The most common words are often the words with less semantic meaning and can for this reason be removed. This may be combined with manual filtering of a stopword list for the best results [11].

One may choose to use a predefined stopword list or to create a corpus specific stopword list. The advantage of a corpus specific stopword list is that different corpus may have different word frequencies and different words may be important. Different tasks may require different amounts of stopwords, but the general trend in IR seems to be going for smaller stopword lists [10, 11].

Stemming and Lemmatization

In texts, the same word may appear in many different forms, verbs have different tenses, nouns may be singular or plural etc. When retrieving documents, it will often be beneficial to also look for documents containing other forms of a queried word. In this situation it is therefore useful to apply stemming or lemmatization to the text. The goal of both stemming and lemmatization is to reduce the different forms of a word into one common form, often called the stem or the root [12]. They achieve this using different methods. Stemming usually works by cutting of the ends of the words according to certain rules in a hope of achieving a common form. Meanwhile, lemmatization often uses a vocabulary and differentiates based on the part of speech(POS) of a word to more accurately determine the base form of a word [11].

2.1.2 Indexing

When looking for a document containing a specific term, it will be quite slow to scan through all the terms of all the documents just to find the documents containing the one specific term. This is where an index may be useful. An index is a mapping from terms to documents containing each term. In information retrieval this structure is often referred to as an inverted index. The index is built in advance such that on search-time one may just find the term in the index to get all the documents containing that term. By building an index, the runtime cost of finding documents with a term has been replaced by an upfront cost of building the index [11].

2.1.3 Term Importance Weighting

Many information retrieval techniques also weights terms rather than just checking whether a term is present in a document or not. TF-IDF is a common weighing scheme in IR. In TF-IDF, we do not care about the ordering of the terms in the document, but only look at the number of occurrences of each term. This is also known as a bag of words model [10, 11].

TF stands for term frequency and is as the name imply the a measure of how frequently a term appears in a document. The reasoning behind this is that the more often a term appears in document, the more relevant this term is for the document. TF may also sometimes be normalized by document length such that long documents do not get an unfair advantage over shorter documents just because they contain more terms in general [11, 13]. Equation 2.1 details how document length normalized TF can be calculated by taking the frequency $f_{t,d}$ of a term t in a document d, then dividing it by the total number of terms in the document.

$$tf_{t,d} = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{2.1}$$

IDF is an abbreviation for inverse document frequency and is meant to reward terms that appear less frequently in the corpus. IDF builds on some of the same motivation as stopwords, being that terms that appear in most documents are almost useless when determining relevance. This intuitively makes sense when searching for e.g. "the Beatles". Here the common word "the" will match most documents, while the more uncommon word "Beatles" will be far more likely to find relevant documents about the Beatles. As the name implies the IDF of a term t is calculated by dividing the total number of documents N by the number of documents containing the term n_t. Then the logarithm is applied to the result to provide a dampening effect [11, 13]. This can be seen in Equation 2.2. Note that there exists different variations.

$$idf_t = \log(\frac{N}{n_t}) \tag{2.2}$$

TF and IDF are then combined into the final TF-IDF measure by using the formula in Equation 2.3. Here, the TF-IDF score is calculated for a single term in a single document in the collection of documents by multiplying the TF score with the IDF score. This will give a measure that takes into consideration both the isolated term frequency of the document and the IDF term that considers the whole document collection [14].

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t \tag{2.3}$$

2.1.4 Retrieval Models

Retrieval models define the notion of relevance of documents for queries and a retrieval function scores documents based on a relevance model. This makes it possible to rank documents by relevance and is at the core of information retrieval [10].

Reranking First-Pass Retrieval Results

For certain retrieval functions, it would not be feasible to apply the function to all documents in a corpus because this would be too inefficient. A normal solution for this is to use a more efficient ranker to retrieve a first-pass result which can then be reranked using the more expensive function. A ranker may be any system or algorithm for ranking documents based on some criteria, like relevance. This approach is extremely effective for improving the efficiency of the ranking, but it is not without drawbacks. It is impossible for a reranking algorithm to retrieve a document not available in the first-pass result. This essentially means that the first-pass ranker acts as a filter. Because of this, a bad first-pass ranker may be detrimental to a retrieval model [15].

BM25

BM25 is a popular retrieval function for scoring documents or texts with respect to an input query [10, 16]. This retrieval function is based on TF, IDF and document text length normalization [17]. BM25 uses a bag-of-words representation of text which means that each term is scored independently and the order of terms are not accounted for [10]. There are two parameters which we can tune in BM25. The $k1$ parameter which limits how much a single query term can contribute to the score of a document and is achieved by letting the score approach an asymptote. The notion for this is that the frequency of a term's appearance has diminishing returns on the relevance of a document and should therefore also have diminishing returns on the score of a document [10, 16, 17]. The b

parameter controls the amount of length normalization applied to a document. When b is set to 0 there is no length normalization, but when b is set to a value greater than 0 the shorter documents are rewarded while longer documents are penalized [10].

In Equation 2.4 we can see the formula for calculating BM25 score for a document-query pair [10, 17].

$$score(d,q) = \sum_{t \in q} \frac{f_{t,d} \cdot (1 + k1)}{f_{t,d} + k1(1 - b + b\frac{|d|}{avgDocLen})} \cdot idf_t \tag{2.4}$$

LM

Another popular retrieval model is the query likelihood model. This model takes a probabilistic approach to retrieval, also known as language modeling. In this model, the documents are ranked by the probability that a user would use a particular query to find a particular document. As with BM25, it is assumed that terms are independent and order is not preserved [10].

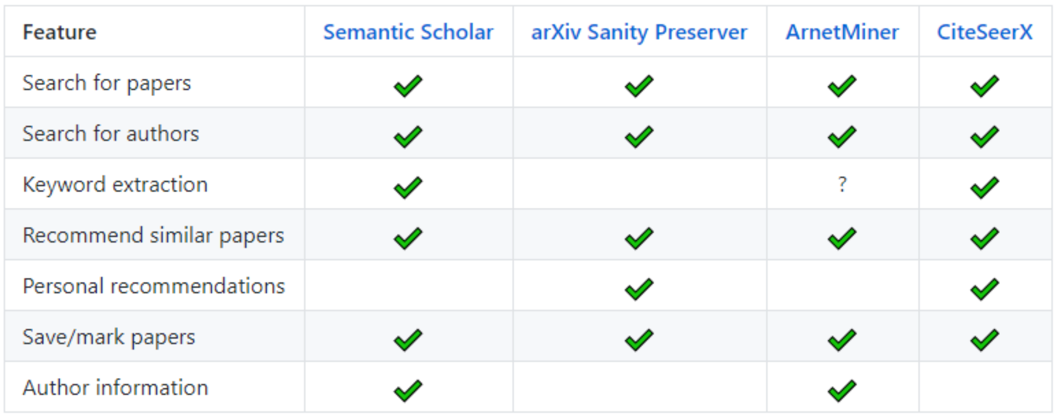
The probability of a term appearing in a document can be calculated as seen in Equation 2.5, but this probability is almost always smoothed with collection statistics before use. This is because one term with a zero probability will give the entire query a probability of zero, as the probabilities are multiplied. Jelinik-Mercer smoothing is one such form of smoothing, and can be seen in Equation 2.6 [10].

$$P(t|d) = \frac{f_{t,d}}{|d|} \tag{2.5}$$

$$P(t|\theta_d) = (1 - \lambda)P(t|d) + \lambda \frac{\sum_{d'} f_{t,d'}}{\sum_{d'} |d'|} \tag{2.6}$$

The final score is calculated by multiplying all the term probabilities. Multiplying small probabilities together will often lead to very small numbers, which could lead to a arithmetic underflow. Because of this, the probability is most often transformed into log-space, as can be seen in Equation 2.7 [10].

$$score(d,q) = \sum_{t \in q} \log P(t|\theta_d) \cdot f_{t,q} \tag{2.7}$$



Feature	Semantic Scholar	arXiv Sanity Preserver	ArnetMiner	CiteSeerX
Search for papers	✓	✓	✓	✓
Search for authors	✓	✓	✓	✓
Keyword extraction	✓		?	✓
Recommend similar papers	✓	✓	✓	✓
Personal recommendations		✓		✓
Save/mark papers	✓	✓	✓	✓
Author information	✓		✓	

Figure 2.1: Different literature services and what they provide.

2.2 Academic Literature Search

As outlined in the introduction, the main goal of this project is to facilitate easier discovery of relevant academic literature. Tackling this problem using IR techniques gives us two choices for how to deliver this information to the users, push or pull [11].

A system in pull mode lets users take the initiative to find the information, typically by issuing a query to a search engine or by browsing through articles within a specific field of study. This mode of information retrieval is best suited for a temporary, ad hoc information need that typically will be resolved after finding something specific [10].

Push mode however, is initiated by the system. Examples of this may be a movie site recommending movies based on a users watch history on the main page or a news site sending out email notifications on news that matches a users interest profile. This mode of finding relevant information is typically more useful in fulfilling a long term information need [10].

2.2.1 Ad Hoc Scientific Document Retrieval

Ad hoc document retrieval addresses a temporary information need and is thus most often best solved by pull-based techniques like querying and browsing. There exists several services that provides features like this for finding scientific literature. Semantic scholar [3], arXiv Sanity Preserver [5], ArnetMiner [4] and CiteSeerX [2] all have features for searching for articles and authors, browsing similar articles and saving articles. Semantic scholar [3] also provides more advanced features like showing the influence of articles and authors. An overview of features from the different sites can be seen in Figure 2.1.

2.2.2 Scientific Literature Recommendation

As recommendations usually focus more on the long term information needs of the users, it is more natural to employ push based techniques in our scenario. For example, if the goal is to recommend content from a continuous stream of articles, it is not natural to expect users to continuously watch or query the system with the same query. Seeing that the users' long term information needs usually stay relatively constant, it is possible to build an interest profile for each of the users. Then this can be used for determining the relevance of each article and alert the user when an article matches their profile.

There exists two main approaches for recommending content to users. The first one is content based filtering. Content based approaches bases the recommendations on the users profile and the contents metadata. This approach may also exploit information about what a user has previously shown interest for. However, this requires rich and accurate metadata for the content to provide reliable recommendations [18].

The other approach is collaborative filtering. In this approach, we do not need metadata for the items nor a profile for the users. Instead, content relevance is modeled based on user interaction. The recommendations are then based on what like-minded users show interest for [18].

Of the services we looked into, only CiteSeerX [2] and arXiv Sanity Preserver [5] provides personal recommendations.

2.2.3 Explainable Recommendations

The goal of explanations on recommendations is to help users understand why the item was recommended to them by the system. It also makes it easier for system designers to debug the systems [7]. Explanations make the system more transparent to the user, makes the recommendations more persuasive and increases users' trust in the system [7, 8]. It has been shown that users are more forgiving towards recommendations they disagree with, if served together with an explanation [8].

There are two approaches for explaining recommendations. The first approach is to develop interpretable models. As the models inner workings are transparent, it is easy to see which decisions lead to the recommendations and this can therefore be converted directly into an explanation [7]. BM25 is an example of an interpretable model. The other approach is a model-agnostic approach, sometimes called post-hoc explanations. With this approach, the recommendation may be created first and then the system attempts to find a fitting explanation for the recommendation afterwards. This is useful for models that are hard to explain or inherently unexplainable [7].

2.3 Extracting Metadata

Extracting data and knowledge from documents is a common objective for applications that collect and present scientific literature. This is done not only for displaying the extracted data to the users along with the documents, but also for use in building knowledge databases. A knowledge database is a structure used to store information that is used by a computer system. These knowledge databases can help to connect different documents, explore related documents, search for documents and discover other statistical properties of the document collection. The services we mentioned earlier in Figure 2.1 all use some form of metadata extraction to populate their knowledge databases. The document PDF files they collect sometimes comes with correctly formated metadata, but many PDFs provide incomplete metadata or no metadata at all. All the literature services therefore have different ways of dealing with this problem.

Semantic Scholar for example, uses the ScienceParse system to predict the missing metadata from a PDF with incorrect metadata. This metadata is needed to complete the knowledge database that Semantic Scholar uses for their scientific literature. The ScienceParse system predicts the missing paper title, list of authors and list of references for each paper using recurrent neural networks(RNN). The PDF is split into each individual word before then being fed into the RNN along with some constants such as page number and numbers detailing if the letters are uppercase or not. The RNN uses this information to predict the mentioned metadata [19].

Another way of getting document metadata is by using another service that provides document with already extracted metadata available. This is provided by both ArXiv [1] and Semantic Scholar [3] though their APIs.

2.3.1 Topic Extraction

The paper title, authors and references etc. are not the only metadata that can be extracted from a document. Another type of extractable metadata are entities or topics for a specific document. These topics can be used in the knowledge databases to link documents together and to help find related documents.

Some of the earlier approaches to extracting topics from documents were to use statistics about single words. One could then select the most statistically discriminating words from a vocabulary of unique words extracted from all the documents in the corpus. Later this evolved to also compare against a vocabulary of unique words from a standardized reference corpus. These methods are called corpus-oriented methods for topic extraction [20]. However, there are some downsides to this type of approach. First, these methods only operated on single-word topics and not topics consisting of multiple words. Secondly, topics that occur in many documents in the corpus will not be selected as they are not statistically discriminating for a single document [20]. However, this might not be a a downside at all unless one wish to assign one specific topic to many documents in the corpus.

To overcome some of the problems with the corpus-oriented approach, there exists another type of topic extraction called document-oriented methods. These methods focus on words in one individual document only and do not take in consideration the other documents in the corpus at the same time. This has the consequence of allowing these methods to select the same topics for multiple documents in the same corpus, avoiding the second mentioned drawback of the corpus-oriented methods. Previous works on these methods include selecting topics using POS tags, calculating word co-occurrences using a chi-square measure [21], TextRank [22] and RAKE [20].

In later times, neural networks have also been used to extract topics from a document. The neural networks can be trained on a set of reference documents with manually defined topics, where the input to the network is the documents PDFs and other available information. The result is a trained neural network that can predict the most useful topics or categories for any given document. This has been explored by Semantic Scholar [19] and in the paper 'Domain-Independent Extraction of Scientific Concepts from Research Articles' [23].

2.4 Evaluation

Part of the goal with arXivDigest is to evaluate experimental recommender systems. ArXivDigest can support several experimental recommender systems running at once and we need a way of measuring the performance of these recommender systems. We also need to ensure that all systems get their fair share of exposure to users in an unbiased way and that measurement of performance and the comparisons between the systems are fair. In this section we will therefore look into the different evaluation methodologies available.

2.4.1 Offline vs. Online Evaluation

There are three main ways to evaluate the quality of recommendation algorithms, also known as rankers. These main ways are user studies, offline evaluation and online evaluation.

User studies are often carried out in a lab setting on recruited users. They have some advantages over other evaluation methods in that it is possible to measure unique data such as the users' eye movement or brain activity in these controlled environments. At the same time they are often expensive, do not scale well and might not be generalizable to the userbase of a platform [24].

For offline evaluation methods, it is common to have experts create data sets and queries with relevance judgements. This makes it easy to compare systems against each other. At the same time it is expensive to obtain these relevance judgments, and these judgments may not always reflect real users' opinions [24].

Online evaluation uses real user interactions in a real system to evaluate the performance of a ranker. Both implicit and explicit interaction data are collected and used for evaluation. Explicit interaction data is when the user performs explicit actions like marking a document as relevant, for example through liking documents or a score system. This gives easy to interpret data, but often has the downside of disturbing the users' normal workflow. Meanwhile, implicit interaction data is for example actions like query reformulation and mouse movements. Implicit interaction data is much more abundant, as it is generated by many different user interactions, but is also harder to evaluate [24].

Joeran Beel et al. [25] compare the effectiveness of online vs. offline evaluation on recommendation algorithms. They measure the recommendation algorithms based on click-through rate (CTR), i.e. the ratio of clicked recommendations. For instance, if a system displays 10,000 recommendations and 120 of them are clicked, the CTR is 1.2%. In their paper they state that offline evaluation often does not reliably predict an algorithms' CTR compared to an online evaluation. One of the reasons for this is the influence of human factors. Humans might not always make the 'correct' answers when it comes to selecting recommendations and interests might shift over time. Another reason for the offline evaluations' worse performance comes from imperfections in the datasets they use. There are many different reasons for why a dataset might be bad but the consequences are often the same. In offline evaluation, the ranker algorithms are limited by the dataset they have been trained against, so having a bad dataset will lead to worse performing ranker algorithms. ArXivDigest is an online evaluation service and we will thus focus on online evaluation from this point on.

2.4.2 A/B Testing

One of the simplest, yet very popular methods of online evaluation is A/B testing. With this method, users are divided into random groups, where one group is shown results from

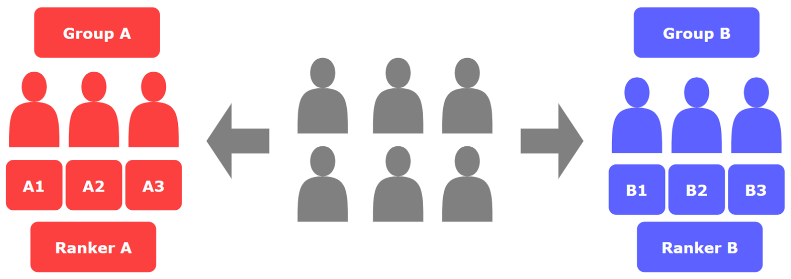


Figure 2.2: A/B testing illustration

one system and another group is shown results from another system. Evaluation is then performed by measuring differences in user interactions with the different systems [24, 26].

2.4.3 Interleaving

Because user behavior can vary much from user to user, A/B testing typically require a large amount of observations and users. Interleaving is one of the methods that as been proposed for combating this problem. Interleaving is performed by giving each user results from two rankers instead of just one. It has been shown that this significantly reduces the variance in measurements and the required sample size [24].

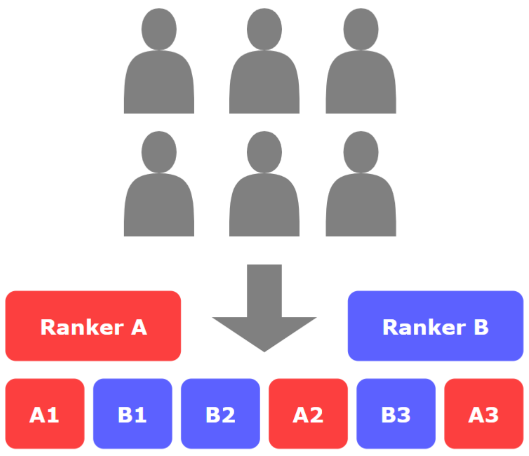


Figure 2.3: Team Draft Interleaving illustration

There exists several algorithms for interleaving. Algorithm 2.1 shows the pseudo code for one such method called Team Draft Interleave. The first step of Team Draft Interleave is to add the common prefix between the rankers list to the result. For this common prefix, no ranker is given credit. After this, the rankers add their best items to the result in turns until the result reaches a specific length or the rankers are out of items. The order of each turn is randomized to give all rankers a fair chance of getting results early in the result list. The rankers are given credit for the items they commit during these rounds.

Algorithm 2.1 Team Draft Interleave [24]

```
Input: Rankings $A = (a_1, a_2, ...)$ and $B = (b_1, b_2, ...)$
 1: Init: $L \leftarrow ()$; $TeamA \leftarrow \emptyset$; $TeamB \leftarrow \emptyset$; $i \leftarrow 1$
 2: while $A[i] = B[i]$ do
 3:     $L \leftarrow L + A[i]$
 4:     $i \leftarrow i + 1$
 5: end while
 6: while $(\exists i : A[i] \notin L) \wedge (\exists j : B[j] \notin L)$ do
 7:     if $(|TeamA| < |TeamB|) \vee ((|TeamA| = |TeamB|) \wedge (RandBit() = 1))$ then
 8:         $k \leftarrow min_i\{i : A[i] \notin L\}$
 9:         $L \leftarrow L + A[k]$
10:         $TeamA \leftarrow TeamA \cup \{A[k]\}$
11:     else
12:         $k \leftarrow min_i\{i : b[i] \notin L\}$
13:         $L \leftarrow L + B[k]$
14:         $TeamB \leftarrow TeamB \cup \{B[k]\}$
15:     end if
16: end while
17: return Interleaved ranking $L, TeamA, TeamB$
```

This credit is used later in the evaluation stage. An example of a Team Draft Interleave result list can be seen in Figure 2.3.

2.4.4 Multileaving

The last evaluation method we discuss is called multileaving, which is an extension of interleaving that makes it possible to evaluate more than just two rankers at the same time. Multileaving is designed to more quickly compare many rankers again each other [24]. Another advantage of multileaving is that it lessens the effect of the presence of a bad ranker. In A/B testing, having a bad ranker will lead to half of the users getting

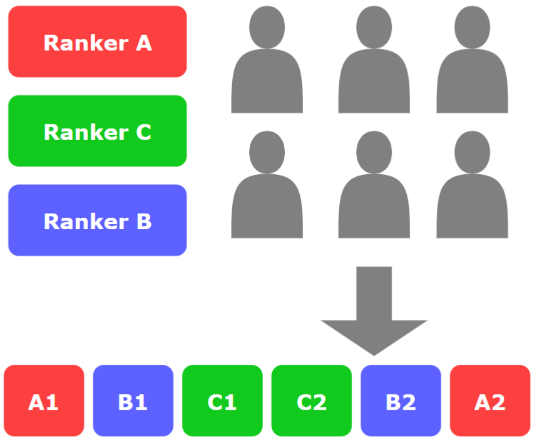


Figure 2.4: Team Draft Multileaving illustration

bad results. In interleaving, it will only lead to half of each users results being bad. Finally, in multileaving only $1/n$ of each users results will be bad, where n is the number of rankers [24].

As with interleaving, there also exist several algorithms for multileaving. In algorithm 2.4 we can see a version of Team Draft Multileave proposed by Schuth et al. [24] extended for use in the arXivDigest platform [9]. This version multileaves multiple systems, gives no credit for common prefixes and also limits the number of systems in each users results to a set number N systems. The algorithm uses the same working principles as Team Draft Interleaving, except for the listed changes. An example of a Team Draft Multileave result list can be seen in Figure 2.4.

Algorithm 2.2 Commonprefix [9]

```
Input: set of rankings $R$
 1: $cp \leftarrow []$
 2: for $i \leftarrow 0$ to $|R_0|$ do
 3:     for each $R_x$ in $R$ do
 4:         if $i >= |R_x|$ then
 5:             return cp
 6:         end if
 7:         if $R_0[i] \neq R_x[i]$ then
 8:             return cp
 9:         end if
10:     end for
11:     $L \leftarrow L + [R_0[p]]$
12: end for
13: return cp
```

Algorithm 2.4 Team Draft Multileave limit number of systems per multileaving [9]

```
1: SAVED STATE: impressions $\leftarrow$ $dictionary/map$ - default value: 0
Input: set of rankings $R$, multileaving length $k$, systems per list $s$
2: $lists \leftarrow []$
3: while $|lists| < s$ do
4:     select $R_x$ randomly s.t $|impressions_x|$ is minimized
5:     $lists \leftarrow lists + [R_x]$
6:     $impressions_x \leftarrow impressions_x + 1$
7: end while
8: return Team Draft Multileave ignore common prefix(lists,k)
```

Algorithm 2.3 Team Draft Multileave ignoring common prefix [9]

```
Input: set of rankings $R$, multileaving length $k$
 1: cp ← Commonprefix(R)
 2: $L \leftarrow cp$
 3: $\forall R_x \in R : T_x \leftarrow \emptyset$
 4: while $|L| < k$ do
 5:   select $R_x$ randomly s.t $|T_x|$ is minimized
 6:   $p \leftarrow 0$
 7:   while $R_x[p] \in L$ and $p < k - 1$ do
 8:     $p \leftarrow p + 1$
 9:   end while
10:   if $R_x[p] \notin L$ then
11:     $L \leftarrow L + [R_x[p]]$
12:     $T_x \leftarrow T_x \cup R_x[p]$
13:   end if
14: end while
15: return $L, T$
```

2.4.5 Evaluating Performance

Creating interleaved rankings serves little purpose if we do not have any metrics to evaluate the user preference for the different rankers. One way of evaluating interleavings is by counting wins, losses and ties for each interleaving. A win is given to the best performing system in a interleaving, a loss to the worst performing system and a tie is given when they performed equally. We score the systems based on the amount of user interaction, where some types of user interactions contributes more to the score than other. The best performing system is the system with the highest score in the interleaving and the worst performing system is the one with the lowest score. This is then used to calculate the *outcome*, which is the metric that we can compare systems by. *Outcome* is calculated as $\#Wins/(\#Wins + \#Losses)$. Another important metric is the number of impressions a system has. Impressions is the the total number of unique interleavings a system has been part of, or can alternatively be defined as the sum of wins, ties and losses for a system. It is useful to know, as it tells us about the sample size when checking the significance of the results [24].

2.4.6 Living Labs

The idea behind the "living lab" concept is to let researches test their ideas directly on real users (without their knowledge) [6]. Testing methods on real users is not a new idea by itself as this is the foundation of online evaluation, which we discussed earlier. In fact, all major search engines can be described as living labs [27]. The problem with these living labs is that access is usually limited to those who work at the organization hosting these search engines. Which again means most academic researchers have to resort to simulated users or other offline methods [27]. This also affect the industry negatively as it takes a longer time before many of the ideas though of in academia become available to the industry [6]. The argument for living labs is that by giving academic researchers access to real users, it will lead to better algorithms and approaches, which can then be used by the industry to provide better services for the users [6, 27, 28]. Also by letting research groups share a common, well maintained service for evaluation, they can gather a larger user base and focus more on the research than they would be able to with the overhead of maintaining a service themselves [27].

There exists several implementations of living labs. TREC OpenSearch implements a system that lets third party research groups interleave their search results with the production system in an academic literature search engine [27]. The CLEF NewsREEL challenge provided a living lab with potentially millions of users for the development of news recommendation algorithms [6]. The Living Labs for Information Retrieval (LL4IR) CLEF lab is a platform that acts as a middleman between commercial organizations and experimental systems for two use cases, product search and web search. This platform facilitates data exchange and comparisons between participating systems [6].

Chapter 3

Infrastructure Development

The original arXivDigest application needed some updates to support the objectives we set in Section 1.2.1. Work had to be done on all parts of the application and we also used this opportunity to improve some of the already existing code.

3.1 Overview

First let us introduce arXivDigest. ArxivDigest is an application that we created for our bachelor thesis in 2018 [9]. The purpose of the application is to provide a platform for evaluation and development of new recommendation algorithms for scientific literature. It also serves as a service for scientists and science enthusiasts where they can receive personal recommendations on newly published scientific literature. The application is structured as several different modules that interact together trough a shared MySQL database. In Figure 3.1 we show an overview of what the old application structure looked like.

ArXivDigest uses a website as an interface between the users and the rest of the application. Here users can check their personal recommendations and create new experimental recommender systems for recommending scientific literature to other users. This scientific literature is harvested from the arXiv [1] stream each day through a separate script. To access the user information and the scientific literature available for recommendation, the experimental recommender systems can connect to the arXivDigest application through the arXivDigest API. This API is also used by the experimental recommender systems for submitting the scientific literature recommendations. The recommendations must be submitted during a fixed time slot each day.

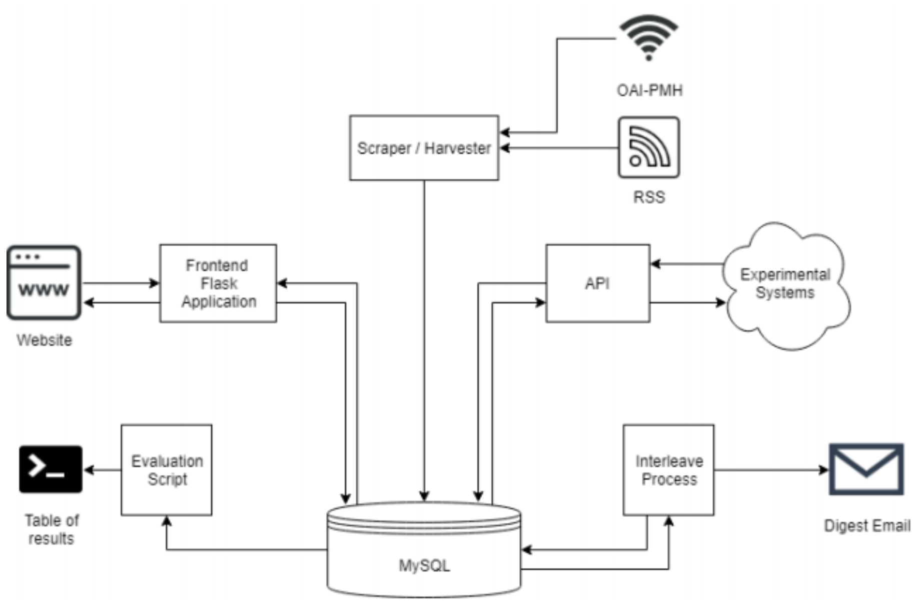


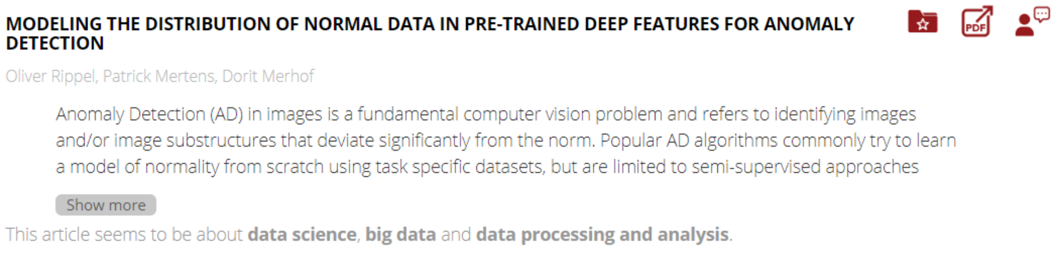
Figure 3.1: Overview of the original arXivDigest platform.

After this fixed time slot, an interleaver script will be executed. This script uses the Team Draft Multileaving method, as discussed in Section 2.4.4, to create interleaved recommendations from many experimental recommender systems for each user. After this the interleaving script will also send out an email to all users with a digest of their newly generated recommendations. The final part of the application is a script that can be run to evaluate the performance of the different systems over specified time periods. This script uses the users' feedback and interactions with their recommendation to score each experimental recommender system. A list of all the systems and their final scores are displayed at the end.

This is a summary of how the original implementation of arXivDigest worked and was structured. Now we will detail the changes and upgrades we did to the original implementation during this thesis.

3.2 Explanations

In the old infrastructure there was no feature for letting recommender systems explain their recommendations. As one of the features we wanted to provide to our users were explanations for the recommendations, we had to update the platform to accommodate this. We felt that by giving explanations to the users, the systems would be more transparent to the user and it would be easier to understand why a recommended article might be relevant for that specific user.

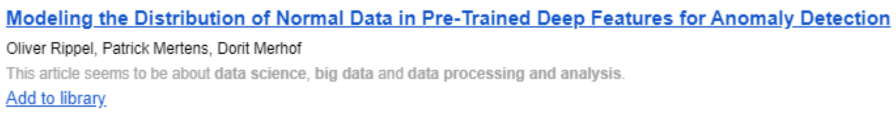

MODELING THE DISTRIBUTION OF NORMAL DATA IN PRE-TRAINED DEEP FEATURES FOR ANOMALY DETECTION

Oliver Rippel, Patrick Mertens, Dorit Merhof

Anomaly Detection (AD) in images is a fundamental computer vision problem and refers to identifying images and/or image substructures that deviate significantly from the norm. Popular AD algorithms commonly try to learn a model of normality from scratch using task specific datasets, but are limited to semi-supervised approaches

This article seems to be about **data science**, **big data** and **data processing and analysis**.

Figure 3.2: Explanation shown beneath an article recommendation.


Modeling the Distribution of Normal Data in Pre-Trained Deep Features for Anomaly Detection

Oliver Rippel, Patrick Mertens, Dorit Merhof

This article seems to be about data science, big data and data processing and analysis.

Figure 3.3: Explanation shown beneath an article recommendation on the digest email.

First we had to update the API to require an explanation field for each recommended article in the endpoint for submitting recommendations. We added a new configuration option to the main configuration file, which controlled the maximum allowed length of the explanations. This way we could adjust the maximum length of the explanations to fit our database and web frontend fields. The maximum explanation length is enforced by the API before it accepts any new recommendations. If the explanation is too long or if the explanation is missing, the API will return an error message. We also had to update the database tables related to article recommendations with a new field to store this new explanation information.

We added explanations to the web interface below the abstracts of each article as shown in Figure 3.2 and to the digest mail below the author names as seen in Figure 3.3. Both of these are generated through the Jinja2 templating language. Using Jinja2, the new explanation values could easily be inserted into the template by fetching the explanation together with the article data and referring to it directly in the template file. We also wanted to let systems use bold text in the explanations to make it easier to see the important parts of the explanation. Jinja2 automatically escapes any HTML tags in inserted text and this is a security feature we want to keep as we do not want the systems to be able to perform HTML injections. At the same time this security feature limits us from letting systems use the `<b>` tag directly. Our solution to this was to let systems mark text that they want to boldface with asterisks as `**this**` markdown style. Then we could convert it to `<b>` tags ourselves via Jinja filters. A Jinja filter simply applies some function to text when converting it to HTML. In Listing 3.1 it is shown how we replace the asterisks with opening and closing tags and registers it as a filter in the frontend app.

```
@app.template_filter('md_bold')
def md_bold(text):
    text = str(escape(text))
    text = re.sub('\*\*(.*?)\*\*', r'<b>\1</b>', text)
    return Markup(text)
```

Listing 3.1: Jinja boldface filter.

3.3 Topics

Article recommendations from the experimental recommender systems are often based on the topics the users have listed on their profiles. The experimental recommender systems are therefore dependent on the users having good topics on their user-profile to give good and accurate recommendations. Originally, we had created a text area for the users to manually input topics they wanted on their profile. One problem with users manually inputting topics is that it is tedious, error prone and hard to accurately capture ones interest this way. This may cause users to not put much effort into adding many topics to their profiles or they may be unable to come up with good topics. Also, the users might not bother to modify their profiles to add new topics after the initial profile creation. There is also the problem of users misspelling the topics they manually input, which will possibly lead to erroneous recommendations. To fix all these problems we wanted a better system in place which would make it easier and quicker for the users to add topics to their profiles.

Since we already had implemented support for experimental recommender systems that recommended articles to the users, we figured that we could easily extend the API to also accept topic recommendations the same way as article recommendations. This meant that the experimental recommender systems would also be able to recommend topics to the users and the users would have another way of enriching their profile with topics by accepting or rejecting these topic recommendations. This functionality involved extending the API with new endpoints mostly mirroring the existing article recommendation API, adding tables for topics in the database and creating a user interface for interacting with the topic recommendations on the website.

3.3.1 Topic API Endpoints

Below are detailed information about the newly created endpoints related to topics in the arXivDigest API.

GET /recommendations/topics

This endpoint returns the previous topic recommendations for one or several users.

- Parameters
  - **user_id**: User ID, or a list of up to 100 user IDs, separated by commas
- Fields returned for each user:
  - **topic**: The topic that was recommended.
    - **score**: Score of the topic for this user.
    - **date**: Date this recommendation was given.
    - **system_id**: The system which gave this recommendation.

- Request:

```
'GET /recommendations/topics?user_id=123'
'header':{"api_key": "355b36dc-7863-4c4a-a088-b3c5e297f04f"}
```

Listing 3.2: Get topic recommendations request.

- Response:

```
    {
      "users": {
        "123": {
          "Information Retrieval":[
              {"system_id":2,
              "score": 3,
              "date": "2020-01-17 17:06:23"},
              {"system_id":33,
              "score": 2,
              "date": "2020-01-17 17:06:23"}
          ],...
        }
      }
    }
```

Listing 3.3: Response to get topic recommendations request

POST /recommendations/topics

This endpoint is used for inserting recommendations of topics to users. Each recommendation should have a score describing how well it matched the users information.

- JSON
  - **user_id**: List of recommendations for user with this ID
    - **topic**: Topic to recommend, containing only a..z, 0..9, space and dash
    - **score**: Score of the recommendation
- Fields returned
  - **success**: True if the insertion was successful
  - **error**: Describing the problem if something went wrong.

- Request:

```
'POST /api/recommendations'
'header':{
        "Content-Type": "application/json",
        "api_key": "355b36dc-7863-4c4a-a088-b3c5e297f04f"
}
'JSON':{
        "recommendations": {
                user_id: [
        {"topic": "Information Retrieval", "score": 2},
        {"topic": "Entity Oriented Search", "score": 3},
        {"topic": "Retrieval models", "score": 2}
        ],...
        }
}
```

Listing 3.4: Insert topic recommendations request

- Response:

```
{
        "success": True,
        "error" : "Some error"
}
```

Listing 3.5: Response to insert topic recommendations request

GET /user_feedback/topics

This endpoint returns the feedback on topic recommendations recorded for a given user (or list of users).

- Parameters
  - **user_id**: User ID, or a list of up to 100 user IDs, separated by commas
- Fields returned for each user:
  - **topic**: The topic recommended to the user.
    - **seen**: Datetime of when topic was seen or null if not seen.
    - **clicked**: Datetime of when topic was clicked or null if not clicked.
    - **state**: What the user did with the topic recommendation.
    - **recommendation_date**: Datetime of when the topic was recommended.
    - **interleaving_order**: The order the topic got in the interleaving.

- Request:

```
'GET /user_feedback/topics?user_id=1,2,3'
'header':{"api_key": "355b36dc-7863-4c4a-a088-b3c5e297f04f"}
```

Listing 3.6: Get user topics feedback request

- Response:

```
{
"user_feedback": {
  "1": {
    {
      "higher education and career education": {
        "clicked": "2020-03-17 18:12:45",
        "seen": "2020-03-17 17:13:53",
        "state": "SYSTEM_RECOMMENDED_ACCEPTED",
        "interaction_time": "2020-03-17 18:12:45",
        "recommendation_time": "2020-03-15 11:16:53"
        "interleaving_order": 8
      }
    },
    {
      "transportation planning": {
        "clicked": null,
        "seen": "2020-03-17 17:13:53",
        "state": "REFRESHED",
        "recommendation_time": "2020-03-15 11:16:53"
        "interleaving_order": 4
      }
    }
  },
  "2": {
    {
        "transportation planning": {
          "interaction_date": "2020-03-23 22:27:43",
          "state": "USER_ADDED"
        }
      }
    },
    "3": {}
  }
}
```

Listing 3.7: Response to user topic feedback request

GET /topics

This endpoint returns a list of all the topics currently stored in the arXivDigest database.

- Return value:
  - **topics**: List of all topics.

- Request:

```
'GET /api/topics'
'header':{"api_key": "355b36dc-7863-4c4a-a088-b3c5e297f04f"}
```

Listing 3.8: Get articles from date request

- Response:

```
{
        "topics": [
            topic1, topic2, topic3, ....
        ]
}
```

Listing 3.9: Response to get topics request

3.3.2 Submitting a Topic Recommendation

To submit a topic recommendation to arXivDigest, one first needs to have an API key. This can be acquired by visiting https://arxivdigest.org/livinglab and registering a new system. An API key is then provided on the web page and sent by email once your system has been approved by an administrator. After one has acquired an API key, one may follow these steps to submit topic recommendations.

1. Call `GET /` to get the settings of the API.
2. Call `GET /users?from=0` to get a batch of user IDs. Increment the offset to get new batches.
3. Call `GET /user_info?ids=...` with the user IDs as a parameter to get information about the users.
4. Call `GET /user_feedback/topics?ids=...` with the user IDs as a query parameter to get information about the users interaction with previous topic recommendations. These previous topic recommendations can not be submitted again, so make sure to filter these recommendations out of your own recommendations.
5. One can also call `GET /topics` to get the list of topics that already exist in the arXivDigest database if one wish to recommend already existing topics to new users.
6. Use the available data about the users and topics to create topic recommendations for each user.
7. Submit the generated topic recommendations to `POST /recommendations/topics` in batches of the maximum number of users which is specified by the API.
8. Repeat step 2 to 6 until all user batches have been given recommendations.

3.3.3 Topics in the Database

These topic recommendations from the API also needed to be stored in the database. We added some more tables that stored the topics, the topic recommendations and the users' interactions with the topics. This ended up being three different tables. We have the `topics` table that stores each topic string along with their unique ID. The `topics` table also has a filtered column that can be used in the future to filter out explicit topics. Then we have the `user_topics` table which stores the topics related to each user and what action the user has taken with that topic. The different actions are described more closely in Section 3.3.7. The `user_id` and `topic_id` fields are combined to a primary key for this table. Lastly, we have the `topic_recommendations` table where we store the topic recommendations for each user and which system made the recommendation. Here we also store the scores of that recommendation, the interleaving order if the topic recommendation was interleaved and also some feedback flags telling us if the user has seen or clicked the topic recommendation. The last `interleaving_batch` field is used to keep track of which interleaving a recommendation is part of if any as a date and time value. These tables are displayed in Figure 3.4

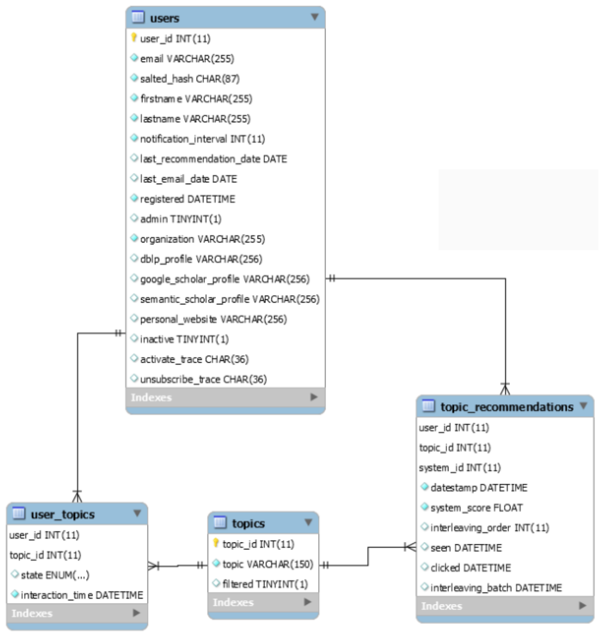


Figure 3.4: Diagram of new topic database tables.

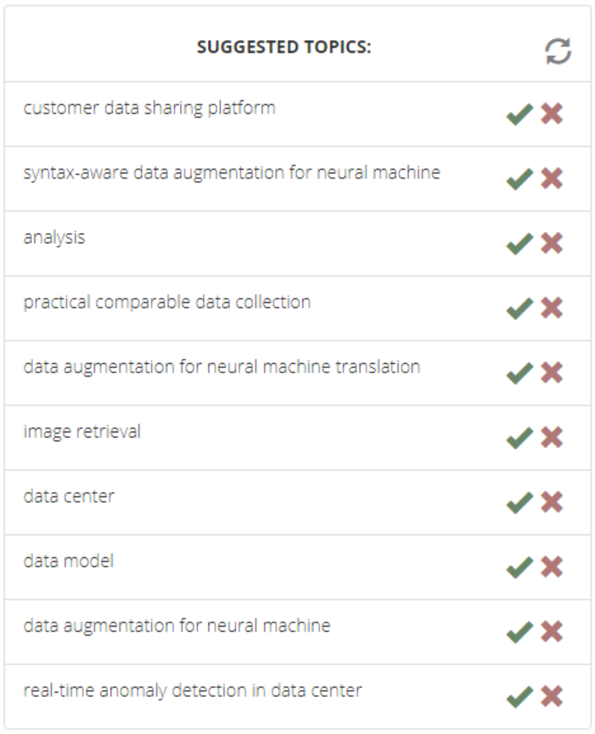
3.3.4 Frontend Topic Implementation

Now that we could accept and store topic recommendations, we needed a way to show these recommendations to the users. On the frontend, we created a section on the main page where we could list these topic recommendations right alongside the article recommendations. In this list, the users can reject topics to make them disappear, or accept the topics which adds them to their profile. There is also an option to refresh the list of topics which will remove all the current topic recommendations and replace them with new ones. This is possible as the topic interleaving is done on demand until there are no more new recommendations available. Topics are also automatically refreshed once every day if the user has logged in, such that the user regularly sees new topics even whiteout interacting with the list. The topic recommendation interface is shown in Figure 3.5.

We also replaced the text area for adding topics to the users profiles on the signup and modify profile pages with an input field that has an auto-complete feature. Users can now start typing a topic in the topic input box and the auto-complete feature will automatically suggest topics from the topics table in the database that are similar to what the user has written. Example of this can be seen in Figure 3.6. This is achieved by using the SQL LIKE operator as shown in Listing 3.10.

```
sql = '''SELECT topic FROM topics WHERE topic
LIKE CONCAT(LOWER(topic_search_string), '%') LIMIT max_results'''
```

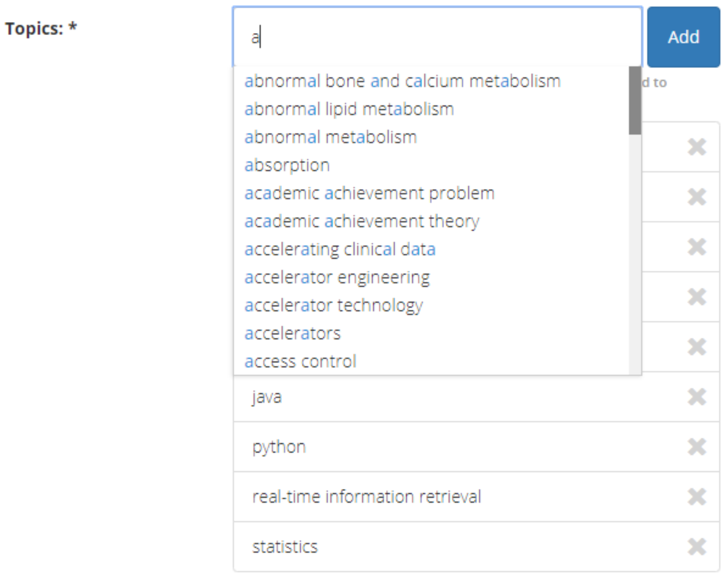
Listing 3.10: MySQL query for topics auto-complete.



SUGGESTED TOPICS:

customer data sharing platform	✓ ✗
syntax-aware data augmentation for neural machine	✓ ✗
analysis	✓ ✗
practical comparable data collection	✓ ✗
data augmentation for neural machine translation	✓ ✗
image retrieval	✓ ✗
data center	✓ ✗
data model	✓ ✗
data augmentation for neural machine	✓ ✗
real-time anomaly detection in data center	✓ ✗

Figure 3.5: The list of recommended topics on the main web page.



Topics: *

a

Add

abnormal bone and calcium metabolism
abnormal lipid metabolism
abnormal metabolism
absorption
academic achievement problem
academic achievement theory
accelerating clinical data
accelerator engineering
accelerator technology
accelerators
access control

d to

java
python
real-time information retrieval
statistics

Figure 3.6: Auto-complete list of suggested topics in the profile forms.

3.3.5 Topic Interleaving

In contrast to the article interleaving that runs once a day, the topic interleaving is only run when we need it to run. It might be once every several days or several times in one day. This depends on the users' activity, as the user has more control over the topic interleaving and is even able to create new interleavings whenever they desire new topic suggestions. Except for the different running intervals, the topic interleaver works in the same way as the article interleaver. It uses topic recommendations from different systems that each have a score and then creates one new interleaved list of topic recommendations with a new set of scores. More on how interleaving works can be found in Section 2.4.4 and in Section 3.7.1.

When a user loads the main arXivDigest web page, we check how old the latest topic suggestions are. If they are older than 24 hours, we interleave the next topic recommendations so that we can present new suggested topics for the users. If they are not older than 24 hours, we just show the newest suggested topics already in the database. The age of the last interleaving can be found in the `interleaving_batch` field in the `topic_recommendations` table in the database. This is a `datetime` field which holds the date and time of when the topic recommendations were last interleaved. Another way of interleaving new topic suggestions is to press the refresh button in the top right corner of the suggested topics list shown in Figure 3.5. This will start a new interleaving of topics and give the user a new set of suggested topics.

3.3.6 Initial List of Topics

When the arXivDigest application is started from scratch, there are no topics stored in the database and the auto-complete topics feature on the signup form will therefore not suggest anything to the users. To fix this, we created a list of pre-made topics that is bundled with the application. The topics on this list are taken from a website that has a selection of many different topics for a lot of different research fields [29]. We scrape this page for all the topics to use them as our initial topic list. For this purpose, we use the Python requests library [30] and the Beautifulsoup library. Beautifulsoup is a tool used to parse HTML or XML structured text into a data structure that is easier to work with [31]. Beautifulsoup also allows us to search the data structure for different types of elements and text. First we request the website that has the list of topics [29]. The way they organize their topics is with a table structure where the row furthest to the right has the most specific topics. When we get the response, we can therefore parse it with Beautifulsoup and then select the last column in each row of the table to get all the specific topics. Finally, we can then store this list as a `.csv` file for easy file modification and reading in the future.

This `.csv` file is then the file that is bundled with the arXivDigest application. It can be directly inserted into the database after the database has been created by executing the `init_topic_list.py` script that is located in the `scripts` folder the the arXivDigest repository.

```
python scripts/init_topic_list.py
```

Listing 3.11: Initialize pre-made topics

3.3.7 Types of Topic Feedback

As we keep track of different user interactions with article recommendations, we also store information about the different ways users interact with the topic recommendations. We track if the user has seen or clicked a topic recommendation and these flags are stored in the `topic_recommendations` table. In addition, we also wanted to store which type of action a user performed for a topic. Since some of these interactions are performed against topics that are not recommended to a user, we decided to store this in the `user_topics` table, which dictates the users' relationship to a given topic. Here we store the action performed in the `state` field, and the time of the interaction in the `interaction_time` field. The state field is an enum, and the possible values and their meaning is listed below.

- `USER_ADDED`: The topic was added to the users profile manually by the user.
- `USER_REJECTED`: The topic was added to the users profile manually by the user but then removed again by the user.
- `SYSTEM_RECOMMENDATION_ACCEPTED`: The topic was recommended by a system, was accepted by the user and added to their profile.
- `SYSTEM_RECOMMENDATION_REJECTED`: The topic was recommended by a system but was rejected by the user.
- `REFRESHED`: The topic was recommended by a system but the user refreshed the suggested topic list without interacting with it.
- `EXPIRED`: The topic was recommended by a system but the user did not interact with it within 24 hours.

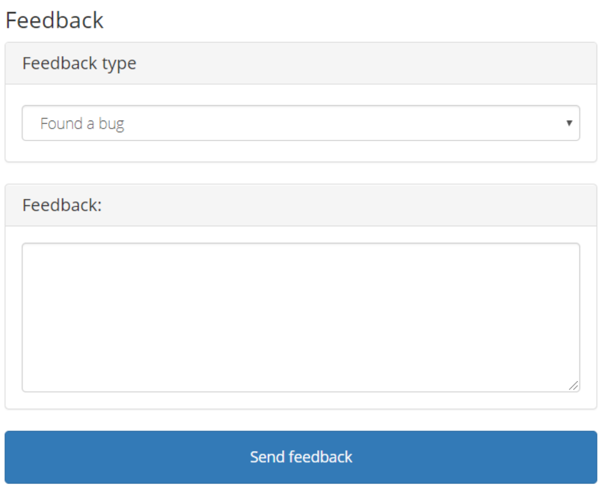


Figure 3.7: The basic feedback page.

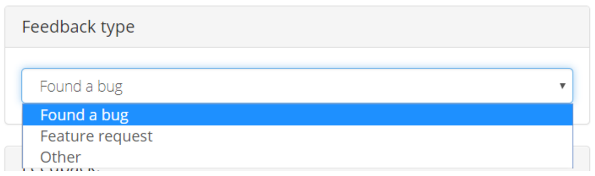
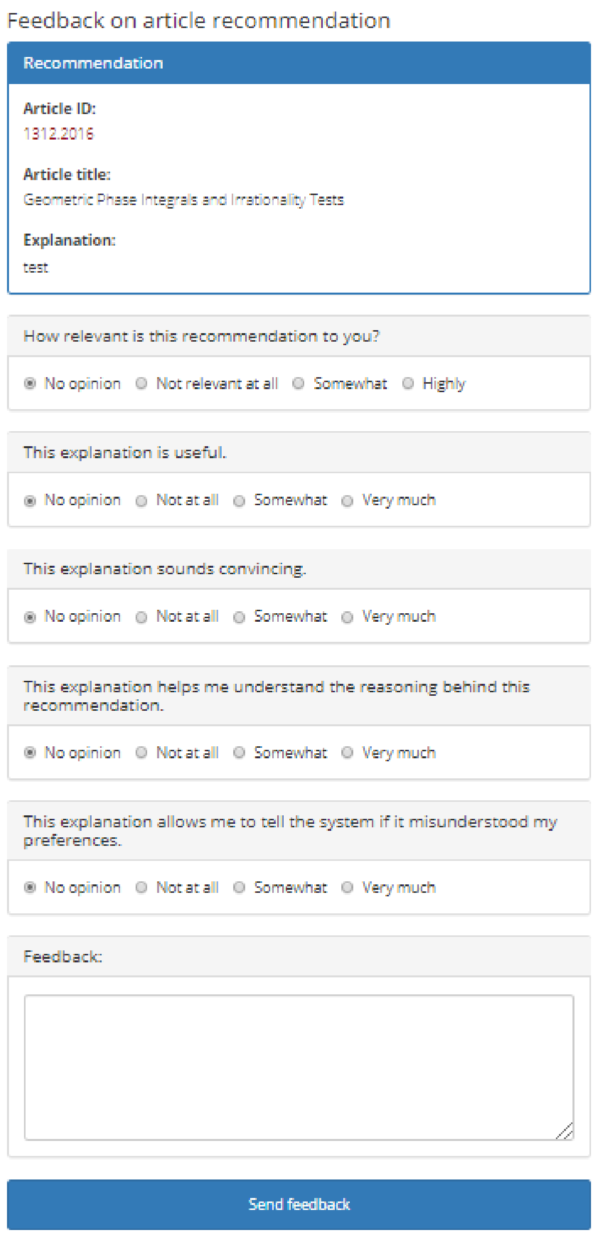


Figure 3.8: The basic feedback page options.

Some of these values are used directly by the platform to keep track of which topics are part of a users profile, like `USER_ADDED`, `USER_REJECTED`, `SYSTEM_RECOMMENDATION_ACCEPTED` and `SYSTEM_RECOMMENDATION_REJECTED`. The value `SYSTEM_RECOMMENDATION_ACCEPTED` is also used when evaluating the performance of a system. Meanwhile, `refreshed` and `expired` were implemented just to get a more detailed view into why users choose not to accept topics when generating statistics and are not used for anything specific at the moment.

3.4 User Feedback

When the application went live, we wanted a place where the users could leave feedback on different issues they might be having. We therefore created a new feedback page where the users can leave feedback on the service in general, report bugs or suggest features. The basic feedback page can be accessed through the link in the footer of the arXivDigest web page and an image of what this page looks like can be seen in Figure 3.7. In this form, users can choose between different feedback types from a drop down menu. The default value here is 'Found a bug' and the other possible values can be seen in Figure 3.8.



Feedback on article recommendation

Recommendation

Article ID:
1312.2016

Article title:
Geometric Phase Integrals and Irrationality Tests

Explanation:
test

How relevant is this recommendation to you?

[✓ No opinion] Not relevant at all Somewhat Highly

This explanation is useful.

[✓ No opinion] Not at all Somewhat Very much

This explanation sounds convincing.

[✓ No opinion] Not at all Somewhat Very much

This explanation helps me understand the reasoning behind this recommendation.

[✓ No opinion] Not at all Somewhat Very much

This explanation allows me to tell the system if it misunderstood my preferences.

[✓ No opinion] Not at all Somewhat Very much

Feedback:

Figure 3.9: The article feedback page.

In addition to this basic feedback page, we created another page specifically for users to leave feedback on article recommendations they get. It is easier for both the user and us if this page is used for feedback on specific recommendations since the resulting feedback will be connected with a specific article recommendation. This also made it possible to create feedback options specifically tailored for article recommendations. In this form we added several multiple choice questions about the quality of the recommendation. These are easy for the user to answer, and they make it easy for us to get a general idea about the users opinion since it is in a standardized format. The user is required to answer at least one of the questions or give a free-text answer before being able to submit. This article feedback page can be seen in Figure 3.9.

To store the feedback, we added an extra table called `feedback` to the database. The feedback will have a foreign key to a user if the user is logged in while leaving feedback, if not this field may be null. All feedback has one of four types, which are shown in the list below.

- `Recommendation`: This is feedback on an article recommendation.
- `Bug`: This is a bug report.
- `Feature`: This is a feature request.
- `Other`: This is feedback that does not fit into any of the other categories.

The `Recommendation` type will always have a foreign key to an article, this field is null for the other feedback types. There is also a `feedback_text` field. This field holds the free-text feedback if the user left any. The last field is the `feedback_values` field. In this field we store structured data in key-value pairs. This is the field that holds the answers for the multiple choice questions in the feedback form. An example of this can be seen in Listing 3.12.

```
relevance:0,
expl_satisfaction:1,
expl_persuasiveness:0,
expl_transparency:2,
expl_scrutability:1
```

Listing 3.12: Values stored for the multiple choice article recommendation feedback

3.5 ArXivDigest Package

In the old arXivDigest platform architecture, we used relative imports to import functions from other files and directories. For the most part this worked fine for development and even deployment, however sharing code between different parts of the application became increasingly harder and more fragile as time went on, especially when we started deploying the application. For example, when we ran the application from the University of Stavanger's Unix servers, we discovered that the relative imports there worked differently from how they worked in our Windows workspaces. We could easily fix this by adding a few extra lines of code, telling the system where to look, but we felt this would just make the problem even harder to deal with in the future if a new problem occurred. Instead we opted to solve the problem in a more robust way by using installable python packages.

We created a package called **arxivdigest** where we put all the code for the platform except for the scripts responsible for launching individual parts. These scripts import the launch function for whatever part of the platform they are responsible for launching and run this function. By having these lightweight launch scripts we are able to update the entire platform without having to reconfigure the scheduling and execution of different tasks, as long as the interface of the launch functions stays the same. This makes both installing and updating the platform much easier, as most of this can be handled by the setup script. For example, earlier we needed to make sure the dependencies of the platform were installed before running anything. Now this installation is handled automatically by the setup script. The dependencies are loaded from a **requirements.txt** file in the arXivDigest directory and installed automatically before the arXivDigest package itself is installed. The setup for the package installation can be seen in Listing 3.13. As a result of converting arXivDigest into a package, sharing code between the different parts of the platform is much easier. It is also stable and easy to use both for Windows and Unix. Static files are also handled by the package, which makes sharing of these easier as well.

```
from setuptools import find_packages
from setuptools import setup
with open('requirements.txt') as f:
    requirements = f.read().splitlines()
setup(
    name='arxivdigest',
    version='1.0',
    packages=find_packages(),
    package_data={'arxivdigest.core.mail': ['templates/*.tmpl'],
                  'arxivdigest.frontend': ['templates/*.html',
                                           'templates/Macros/*',
                                           'uncompiled_assets/css/*',
                                           'uncompiled_assets/javascript/*',
                                           'static/*',
                                           'static/icons/*',
                                           ],
                  },
    url='https://github.com/iai-group/arXivDigest',
    author='Oyvind Jekteberg and Kristian Gingstad',
    install_requires=requirements
)
```

Listing 3.13: Setup file for installing the arXivDigest package

In the package, we include the arXivDigest frontend application, the arXivDigest API, the arXivDigest connector which is described in Section 3.7.3 below and other core functionality. This core functionality include the code for interleaving article- and topic recommendations, the code for scraping the arXiv feed and the code for email services. See Figure 3.10 for the folder structure in the arXivDigest package and what's included.

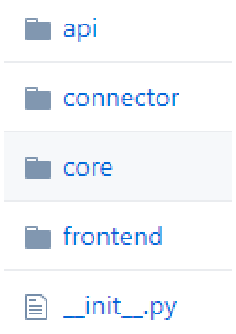


Figure 3.10: Folder overview of the arXivDigest package.

The package can be installed in two different ways. One can either install the package as a normal package in the Python path with the command found in Listing 3.14. The other option is to install the package in the current directory for development purposes with the command found in Listing 3.15. For both these commands, one must be in the repository path of arXivDigest when executing them.

```
pip install .
```

Listing 3.14: Install the arXivDigest package

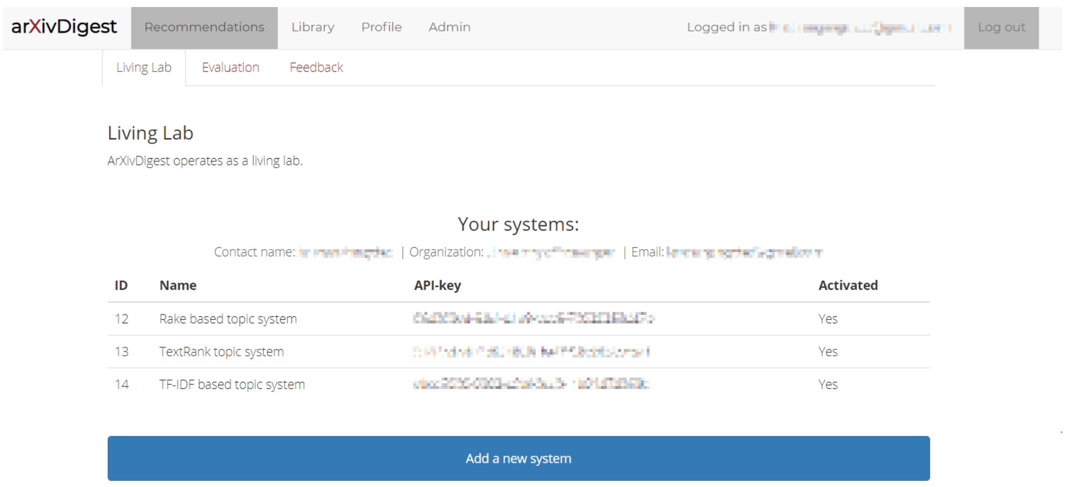
```
pip install -e .
```

Listing 3.15: Install the arXivDigest package for development purposes

3.6 Living Labs and Systems

Experimental recommender systems have always been an important part of arXivDigest, but the original implementation felt a bit lacking on the system owners side. Earlier the platform only provided system owners with the option of registering systems and access to the API. While most of the statistics can be calculated by using data from the API, it feels clunky, and each system owner has to create their own code to transform the data into something useful. There was also no easy way to see which systems one owns and the API-keys, except for in the original mail sent by arXivDigest. To fix these problems we decided to make several changes to how the experimental recommender systems were handled.

Earlier, the experimental recommender systems were completely separated from the users. While there may be benefits for not tying users and systems together, it also made it a lot harder to implement web interfaces for systems where owners can manage their systems and see system statistics. In the end, we could not find any good reasons for keeping this separation and decided that each system would belong to a user. We could



Living Lab

ArXivDigest operates as a living lab.

Your systems:

Contact name: [redacted] | Organization: [redacted] | Email: [redacted]

ID	Name	API-key	Activated
12	Rake based topic system	[redacted]	Yes
13	TextRank topic system	[redacted]	Yes
14	TF-IDF based topic system	[redacted]	Yes

Figure 3.11: Living labs web page.

then delete the registration form for recommender systems, and start from scratch with the recommender systems web interface. How systems interacts with the API stayed the same as before. We could also remove the `contact_name`, `organization` and `email` from the experimental recommender system, as we would instead take this information from the user that creates and owns the system.

Although the recommender system concept always fit the description of a living lab [6], which we discuss more in Section 2.4.6, we wanted to make it more evident that this is part of what our service offer. This would also entail making a more unified experience for the system owners using our service. The first step of this process was to create a living labs page where users can create new systems and see their existing systems, API-keys and statistics about their systems. The main living labs web page can be seen in Figure 3.11.

3.6.1 Evaluation

An important part of a living lab is being able to evaluate system performance. In the old version of arXivDigest [9], we used *outcome* as a metric for comparing performance of systems. One problem we noticed when applying this metric to multileaved comparisons was that it produced unreasonably many ties. We felt that much valuable information was lost when such a low amount of impressions ended up as a win or a loss. Because of this, we propose a new metric that we call *reward*. The intuition behind *reward* is that instead of having a winner takes it all, loser loses all situation, it will be more fair to divide the points equally based on the amount of user interaction with each system in the interleaving.

The reward of a system in an interleaving is defined as the weighted sum of actions towards that system. E.g. if a click is worth 2 points and a like is worth 5 points. The example system has 4 clicked articles and 2 saved articles in an interleaving, the reward of this system would be $2 * 4 + 5 * 2 = 18$.

$$normalized\_reward_s = \frac{reward_s}{\sum_{i \in I} reward_i} \tag{3.1}$$

In Equation 3.1 it is shown how the normalized reward of a system s in an interleaving I can be calculated. The score is normalized by each interleaving such that interleavings with more interaction do not get more points than interleavings with fewer interactions. This ensures each interleaving having exactly one point to divide among the participating systems.

We can then calculate the mean normalized reward for a system over a set time period by taking the mean of the normalized reward accumulated over the given period. This is the same as the sum of normalized reward divided by the number of impressions. We should be able to compare systems performance in relation to each other by comparing this metric. This comparison is dependent on all the multileavings having the same expected rewards and being sufficiently random. These are properties we try to ensure in our multileaver, given enough recommendations to choose from.

3.6.2 Evaluation Web Interface

In the old application, all evaluation was handled by a Python script which would access the database to get the statistics and print them to the console. This was problematic on several levels. Firstly, this makes the statistics hard to share with system owners. It is also far less elegant to run a script and read the console output than having it available on a web interface. In addition, it was much harder to include detailed information in a console than it is to provide the same information with graphs and plots on a web interface. Because of these reasons and the fact that we wanted to make a more unified experience for system owners, we decided to replace the evaluation script with a web interface.

We show the number of impressions and the mean normalized reward in a plot on the web interface. The plot has several options to choose from. The user can toggle between article and topic statistics, choose a start and end date for the evaluation and choose whether to calculate mean normalized reward over a period with length of days, weeks or months. This evaluation interface is shown in Figure 3.12. The accurate numbers for the plots can be seen by hovering the mouse pointer over the bars and blue points. A

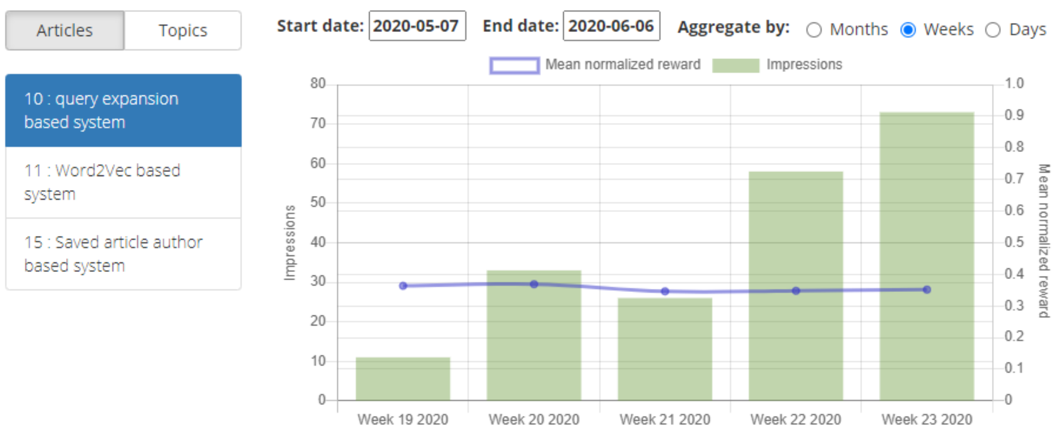


Figure 3.12: Plot for evaluation statistics about a system.

user can access the evaluation interface for their systems through the Living Labs page. The same interface is also available for admins through the admin page. The admins are able to see statistics for all systems registered on arXivDigest.

3.6.3 Feedback Web Interface

We also created plots that shows the amount and types of feedback received for each system over a period. These plots use the exact same interface and has the same options as the evaluation interface and they are also accessible from the living lab page and the admin page. On the admin interface it is also possible to show the total number of each feedback type for all systems in one plot to see the overall activity on the platform.

The plot showing feedback on article recommendation is shown in Figure 3.13. This plot shows how many of the article recommendations from an experimental recommender system has been seen, clicked and saved by the users. In Figure 3.14 we can see the feedback on the topic recommender systems. This plot shows how many of the topic recommendations from a experimental recommender system has been accepted, rejected, refreshed or expired by the users actions.

3.7 Miscellaneous Other New Features

While working on the recommender systems and the other parts of the infrastructure development, we made some other smaller changes to the platform code. Some of these changes were necessary to make other features work properly and some of these were purely quality of life improvements either for the users of the service or us the developers.

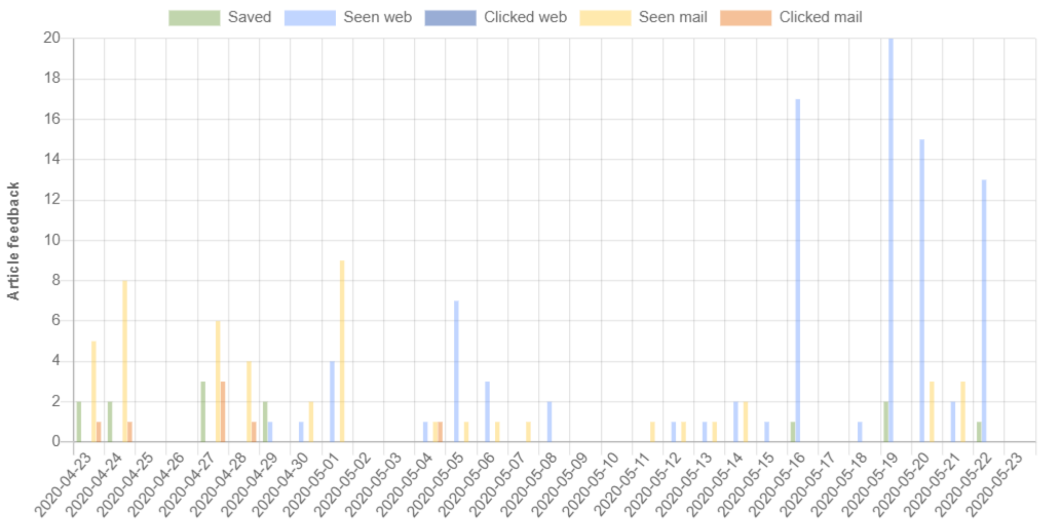


Figure 3.13: Plot for feedback on article recommender systems.

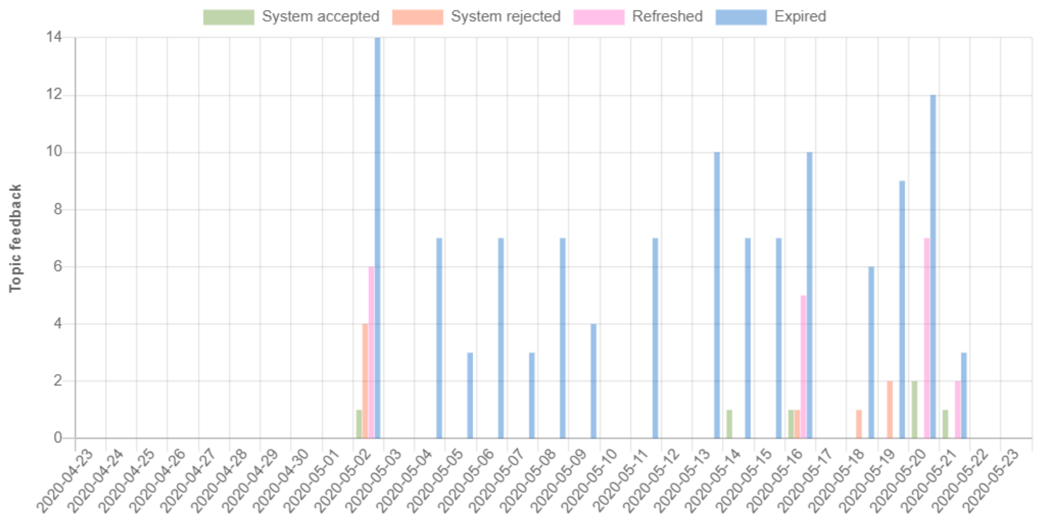


Figure 3.14: Plot for feedback on topic recommender systems.

3.7.1 Interleaving Update

After creating the topic recommender systems, we also needed to interleave these recommendations the same way we interleaved the articles. However, the original multileaver was a bit too tightly connected to the article recommendations. We therefore had to separate the multileaving logic into its own class, allowing it to work with all types of recommendations. An added bonus was that we were able to easily create some unit tests for the logic, so that we could be more confident about the multileavings being fair. The end result was a multileaver that can multileave any type of items and we can have more confidence in it multileaving correctly.

The article recommendations are interleaved once a day as they originally were in the previous implementation. However we did decide that systems would be allowed to recommend articles from the last week, not just the current day. We felt that this would let the systems be a bit more flexible in their recommendations and that days with few good recommendations would be more rare if the recommender systems had a bigger pool of articles to recommend from. This also meant that we could remove the time slot limit when accepting recommendations which gives the systems more time each day to create recommendations.

When separating the multileaving logic from the rest of the interleaver script, we also ended up separating out the email sending code into a separate script. This made the code easier to read and manage and it also made it possible to schedule the email sending independently of the interleaving process.

3.7.2 API Settings

The API has settings for a lot of variables that directly affects the recommender systems. Examples of these settings are the maximum number of users per request or the max number of recommendations that can be submitted per user. Originally, the experimental recommender systems developers had to visit the arXivDigest documentation to get these values and then directly code them into their experimental recommender systems. When we started to develop our own experimental recommender systems, we realised that it would be easier for the developers of recommender systems if the API returned these API settings through one of the endpoints. Since we already had the `https://api.arxivdigest.org/` endpoint, that is used to check if the API is online, we decided to send the API settings variables in the response from this endpoint. Now the experimental recommender systems can automatically request these API settings and and also update them automatically if the settings change in the future.

3.7.3 ArXivDigest Connector

When we started working on the individual experimental recommender systems, we realised that it might be beneficial to create a separate module for easier connection with the arXivDigest API. This would reduce the amount of code in each recommender system and also make the API connection cleaner and easier in the recommender systems code.

What we ended up creating is an `arXivDigestConnector` class that is installed together with the main arXivDigest package. This makes the connector easily shareable between all recommender systems. The connector is easy to initialize and use and an example of fetching user info by using the connector can be seen in Listing 3.16.

```
from arxivdigest.connector import ArxivdigestConnector

arxivdigest_connector = ArxivdigestConnector(api_key,
                                             'https://api.arxivdigest.org/')

#Example of use: fetching information about users 1, 2 and 3 from the API
user_info = arxivdigest_connector.get_user_info([1,2,3]])
```

Listing 3.16: Initialize an arXivDigestConnector object.

When initializing the `arXivDigestConnector` object, it automatically tests the connection with the arXivDigest API to make sure it has been configured with the correct URL and a valid API key. The connector also parses the API settings mentioned in Section 3.7.2 above. These settings are saved in the `arXivDigestConnector` object and are used by the connector to comply with the API settings. The settings can also be accessed from the connector object at at any time.

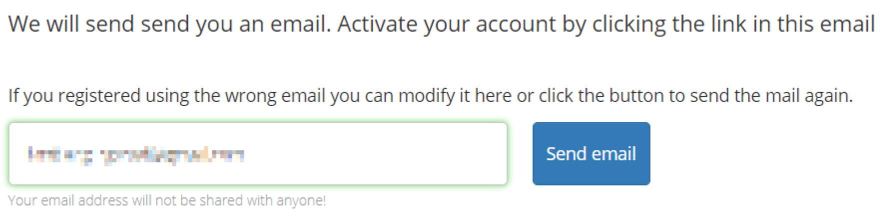
The full list of functions available for the arXivDigestConnector object is available below.

- `get_numbers_of_users()`: Returns total number of arXivDigest users.
- `get_user_ids(offset)`: Returns user ids to the one hundred first users starting at the offset.
- `get_user_info(user_ids)`: Returns the user data for the users in the `user_ids` list.
- `get_article_ids()`: Returns a list of article ids that are candidate for recommendation from arXivDigest.
- `get_article_data(article_ids)`: Returns the article data for the articles in the `article_ids` list.
- `get_article_feedback(user_ids)`: Returns a dictionary with the users feedback on earlier article recommendations for each user in the `user_ids` list.
- `get_interleaved_articles(user_ids)`: Returns the earlier interleaved articles for each user in the `user_ids` list.
- `send_article_recommendations(recommendations)`: Sends article recommendations to the arXivDigest API.
- `get_article_recommendations(user_ids)`: Returns the earlier article recommendations to a user for each user in the `user_ids` list.
- `get_topics()`: Returns a list of all topics in arXivDigest.

- `get_topic_feedback(user_ids)`: Returns a dictionary with the users feedback on earlier topic recommendations for each user in the `user_ids` list.
- `send_topic_recommendations(recommendations)`: Sends topic recommendations to the arXivDigest API.
- `get_topic_recommendations`: Returns the earlier topic recommendations to a user for each user in the `user_ids` list.

3.7.4 Email Verification

In the original application, the new users were not verified in any way. As a minimum, we wanted to verify that the user has access to the email they sign up with. This helps the user to keep control of their account by ensuring that they have access to the signup mail and also makes it a bit more tedious to create non-serious accounts. When a user signs up, they get sent a mail with a unique link that activates their account. Before the account is activated, the user is unable to use most features of arXivDigest, except for changing their email and resending the verification mail. The interface for this can be seen in Figure 3.15 and the verification email can be seen in figure Figure 3.16. To ensure that the activation link is unique and also not guessable, we generate a UUID for each user that is used as part of the verification link. This UUID is also stored in the database together with the users information, thereby making it easy to know which account to activate based on the UUID sent to the activation endpoint.

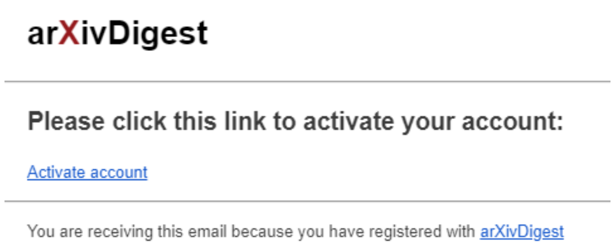

We will send send you an email. Activate your account by clicking the link in this email

If you registered using the wrong email you can modify it here or click the button to send the mail again.

[redacted]

Your email address will not be shared with anyone!

Figure 3.15: Web page for waiting on user verification.

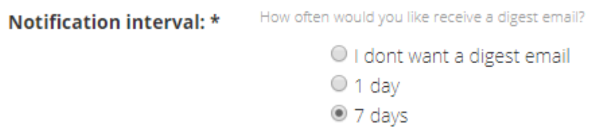

arXivDigest

Please click this link to activate your account:

Activate account

You are receiving this email because you have registered with arXivDigest

Figure 3.16: Verification email.


Notification interval: *

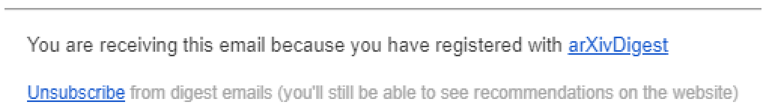
How often would you like receive a digest email?

I dont want a digest email [✓ 7 days] 1 day

Figure 3.17: Unsubscribe option in the modify user web page.

3.7.5 Unsubscribe from Digest Email

Another thing we realised was that some users might not want to receive the digest emails at all. Originally, there only existed options for receiving digest emails daily or digest emails weekly. We decided to add the option of never receiving a digest email for the users that might not want their email inbox full of arXivDigest emails. However, this option is not available on signup. If a user does not want to receive digest emails at all, he or she can choose this by selecting this option through modifying their profile. The new option on the profile modification page can be seen in figure Figure 3.17. Another way of unsubscribing from the digest emails is to click the unsubscribe link at the bottom of each digest email as seen in figure Figure 3.18. This is implemented in the same way as the email verification which is by using an UUID to keep track of which user wants to unsubscribe and including this UUID in the unsubscribe link in the email.


You are receiving this email because you have registered with arXivDigest

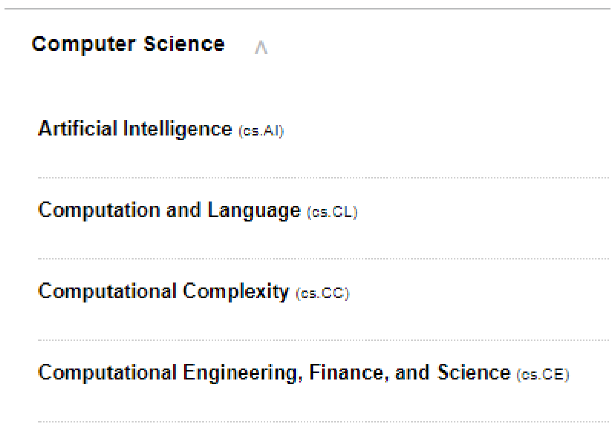
Unsubscribe from digest emails (you'll still be able to see recommendations on the website)

Figure 3.18: Unsubscribe link on the bottom of the digest email.

3.7.6 ArXiv Scraper Update

We also ended up making a couple of changes to the process that fetches article information from arXiv [1]. In the original implementation, the scraper would only fetch articles from the previous day, but we found out that it could also be useful to fetch articles from a configurable time period. Most of the functionality needed for this was already present, but we had to make some structural changes and create some configuration options to make it easy choose which period to download articles for. We also stopped using the RSS stream for finding articles, as it seemed to bring little value over just using the OAI-PMH API directly for finding and fetching articles.

The other change we made was how category names are added to the application. In the original implementation we had to find and update category names manually as the arXiv APIs only provide category IDs. This is not a big problem as the categories are rarely updated, but the process was tedious and if we do not notice and update it, it may take



Computer Science ^

Artificial Intelligence (cs.AI)

Computation and Language (cs.CL)

Computational Complexity (cs.CC)

Computational Engineering, Finance, and Science (cs.CE)

Figure 3.19: Selection from the ArXiv category translation list.

some time for a category to get a name. ArXiv does have a page [32] that provides these category names and the reason why we did not originally use this list is that it is made for humans and not machines. Therefore it is harder to parse and might break with an update. Even with these problems, we felt it would be worth it to automatically scrape categories from this page. The categories are written in a format that is reasonably easy to parse and an excerpt of the category list is shown in Figure 3.19. To not be affected too much by an update of this page, we decided to leave the old system as a fallback option. This way it is easier for us as long as the page stays the same and if they change the page it is not any harder than it was earlier.

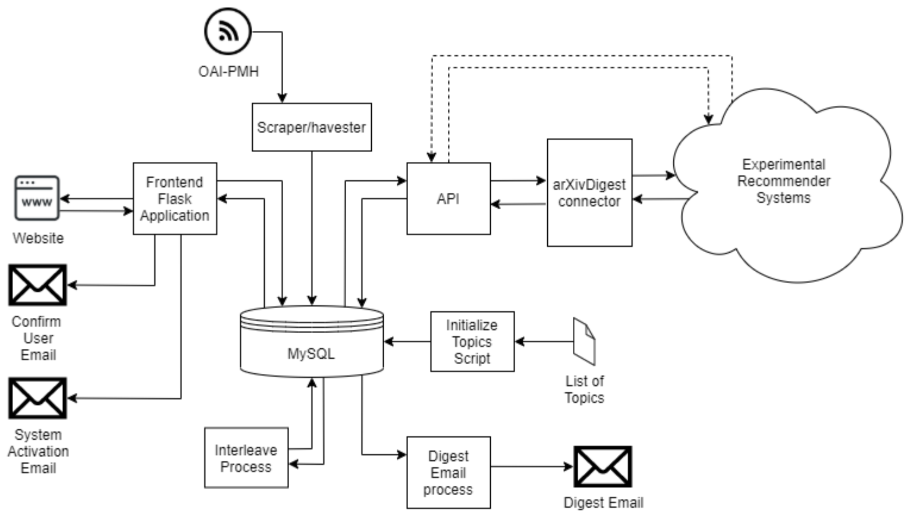


Figure 3.20: Updated overview of the arXivDigest platform.

3.8 Final Architecture

In Figure 3.20 we can see the final arXivDigest overview. If we compare it with the old overview in Figure 3.1, we can notice some of the differences we mentioned in this chapter. The RSS feed is missing from the scraping part, the new process for sending digest emails, the arXivDigest connector and the new script for initializing the topics list are some of the notable differences. We also updated the frontend with what emails they send out. Note the possibility of connecting through the API either directly or through the arXivDigest connector.

3.8.1 Submitting an Article Recommendation

The steps for submitting article recommendations have also had some changes because of the updates. The process of registering a system and acquiring an API key has been moved to the living labs interface, together with the topic recommender systems. All the same features that are provided to the topic recommender systems are available for the article recommender systems through this interface. After one has acquired an API key, these are the steps for submitting article recommendations.

1. Call `GET /` to get the settings of the API.
2. Call `GET /users?from=0` to get a batch of user IDs, the offset may be incremented to get new batches.
3. Call `GET /user_info?ids=...` with the user IDs as a query parameter, to get information about the users.
4. Call `GET /articles` To get the IDs of articles that are candidates for recommendation.
5. Call `GET /article_data?article_id=...` with the article as a query parameter, to get information about the articles.
6. Call GET /user_feedback/articles?user_id=... with the user IDs as a query parameter to get information about what recommendations has already been shown to a user. These articles should be filtered out as they will be ignored by the platform.
7. Use the available data about users and articles to create personalized recommendations with explanations for each user. Important parts of the explanations may be boldfaced by surrounding it by asterisks like: **text**.

8. Submit the generated article recommendations to `POST recommendations/articles` in batches of the size defined by the API settings.

9. Repeat step 2 to 6 until all user batches have been given recommendations.

3.8.2 API Endpoints Overview

There has been quite a few updates to the API during the update of arXivDigest. In Table 3.1 we can see a summary of the new endpoints added to the API. Table 3.2 shows a summary of the original arXivDigest API endpoints where changes to the endpoints are highlighted.

Endpoint	**Description**	**Input**	**Return Value**
GET /topics	Endpoint that returns all the topics available at arXivDigest	None	List of arXivDigest topics.
POST /recommendations/topics	Endpoint for recommending topics to users.	JSON of user IDs to lists of dictionaries with topics and scores.	Error message if something went wrong.
GET /recommendations/topics	Endpoint that returns information about topic recommendations for a batch of users.	List of user IDs as query parameter.	Scores, date and system ID for each topic recommendation to each user.
GET /user_feedback/topics	Endpoint that returns the feedback on all topics a user has interacted with for batch of users.	List of user IDs as query parameter.	Interaction data with dates for each topic for each user.

Table 3.1: Table of new API endpoints.

Endpoint	**Description**	**Input**	**Return Value**
GET /	Endpoint that returns a welcome message and settings that should be used when interacting with the API.	None	Welcome message and API settings.
GET /users	Endpoint that returns batches of user IDs from a given starting point.	'from' as query parameter.	List of user IDs.
GET /user_info	Endpoint that returns detailed information about users.	List of user IDs as query parameter.	Details about the users. Updated with external profiles and topics.
GET /articles	Endpoint that returns the IDs of articles that are eligible for recommendation.	None	List of article IDs.
GET /article_data	Endpoint that returns detailed information about articles.	List of article IDs as query parameter.	Detailed information about each article.
POST /recommendations/articles	Endpoint for recommending articles to users.	JSON of user IDs to lists of dictionaries with article_ids, score and explanation.	Error message if something went wrong.
GET /recommendations/articles	Endpoint that that returns information about article recommendations for a batch of users.	List of user IDs as query parameter.	Scores, date and system ID for each article recommendation to each user.
GET /user_feedback/articles	Endpoint that returns the feedback for all multileaved article recommendations for batch of users.	List of user IDs as query parameter.	Interaction data with dates for each multileaving for each user.

Table 3.2: Updated table of original API endpoints.

Chapter 4

Article Recommendation

This chapter starts by giving an overview of our work with recommender systems. Afterwards we explain the reasoning and implementation of the four different recommender systems we created.

4.1 Overview

The arXivDigest platform in itself brings little value to neither information retrieval researchers nor users. The platform is dependent on external experimental recommender systems that recommends articles to users based on their profiles. We wanted to create a few different systems that would be providing recommendations to users when we launched the platform. This was to ensure a good user experience at launch, before any recommender systems from independent researchers or organizations are added. The platform also needs several systems to actually make use of the multileaving features that are implemented. It also allows us to gather data on how users interact with the different systems from the start, which will let us compare the performance of systems over time.

We decided to write all our systems in Python, in part because it is the main language for all the other parts of the platform, and in part because Python has many existing packages that would help in developing the systems. All recommender systems, including our own, are required to perform all communication with the platform through the arXivDigest API. This can be done either by using pure http request directly to the API, or by using the arXivDigest Connector that we included with the platform. As we were using Python, it was an obvious choice for us to use the arXivDigest Connector (Section 3.7.3) to handle the communication.

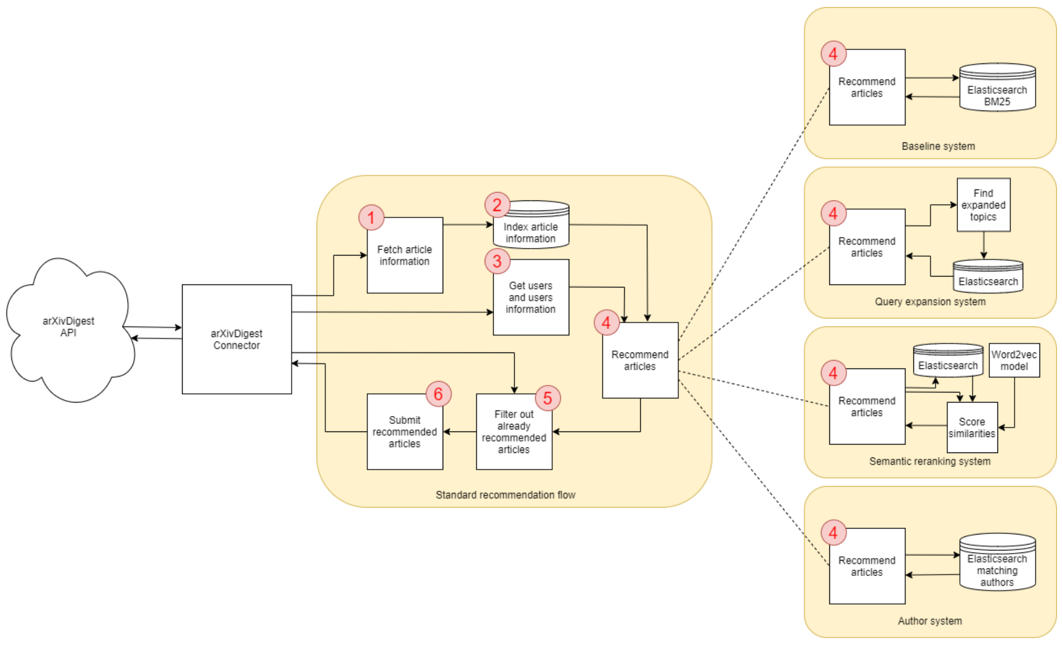


Figure 4.1: Overview of article recommender systems structure.

Most systems will follow the process below when recommending articles, where step 4 will be the main difference between systems. Step 4 is therefore vaguely defined, letting each system decide the best approach.

1. Query arXivDigest for articles eligible articles for recommendations.
2. Index the available data for the articles.
   - Additional data may optionally be collected for articles from external services, like Semantic Scholar [3].
3. Query arXivDigest for information about the users.
   - The system may optionally gather additional data based on the available user profiles at external services.
4. Use the available data about users and articles to create recommendations with explanations for each user, the system may choose to ignore users missing information vital for its recommendation method.
5. Query arXivDigest for recommendations already shown to each user and filter these out.
6. Submit the final recommendations to arXivDigest.

4.2 Baseline System

The main goal when developing the our first recommender system was to create a simple system, that could be shipped with the platform code, as an example of how to a system could be implemented. Because of this, we wanted to keep the recommendation logic quite simple as to not make it any more confusing than it needed to be. Still, we wanted the recommendations given by the system to be useful to users and therefore we would have to apply some minimum of IR techniques when selecting recommendations.

As previously stated, all recommendations submitted to the arXivDigest platform need to have an attached explanation that describes why the article is relevant to the user. We felt that the simplest way of achieving this was by recommending articles using an explainable model, in contrast to post hoc explanations which would be a bit too complicated for such a simple system.

With these requirements in mind, we needed to find a method that would be easy to set up, easy to understand and that uses an explainable IR model. We decided on using Elasticsearch [33] in conjunction with the users' topics to achieve this. Elasticsearch is an open source, distributed search and analytics engine. It has many useful features, but for our purpose we are most interested in the indexing and search features. An Elasticsearch index stores a collection of documents, in our case the documents are scientific articles. All the stored documents are added to an inverted index, which makes Elasticsearch really fast for full-text searches. After adding the documents to the index, we are able to search for documents using Elasticsearch's powerful query language. The resulting documents retrieved for these queries are by default ranked by BM25, which we discussed in Section 2.1.4. We felt that Elasticsearch met our requirements as it is easy to setup and use, it uses a well established IR technique for ranking, includes many text preprocessing techniques and gives results that can be explained. The results may be explained by seeing which topics contributes most to the final score, as BM25 may be seen as an explainable model.

Implementation

The baseline system is implemented as a simple Python script with one external dependency. Namely, it expects that it has access to a running Elasticsearch instance. The first step the system takes is checking that there exist an index with the correct attributes, if not it creates such an index. We configure the Elasticsearch index to apply standard English preprocessing to the text, as we currently only target scientific literature published in English. By doing this, we do not have to worry about preprocessing the text in our system since the index handles this for us. The next action it performs is to fill this index with articles that are candidates for recommendation. These articles and their metadata are fetched from the arXivDigest API using the arXivDigest connector. Users are then retrieved in batches by the arXivDigest connector. These steps roughly corresponds to step 1-3 in Section 4.1. Then, the recommendations and explanations are created for the users in these batches like stated in step 4. We will look more into how this is done in the next paragraphs as this is the most important part of the system.

This system is supposed to score each article by the BM25 algorithm against the users' topics using Elasticsearch. The simplest method of achieving this is by just concatenating all the users topics into a string and use this string as the query for the Elasticsearch index. However, this method has one problem. Even though we get the most relevant articles, we do not know which topics contributed to the score for the retrieved articles. To get this information, it is actually possible to search the index for each topic individually and combining the scores afterwards. This works because the score of a query is actually just the sum of the score of each term in BM25.

Now that we have the score of each topic for every relevant article, we can sort the articles by the sum of their scores and select the top scoring ones as our recommendations. To generate the explanations we can add the top scoring topics to a sentence template with some simple rules for commas and 'and' to create a natural sounding explanation, like shown in Listing 4.1.

```
def create_explanation(sorted_topics, number_of_topics_in_explanation=3):
    topics = sorted_topics[:number_of_topics_in_explanation]
    last = topics.pop()
    topic_sting = ', '.join(topics)
    topic_string += ' and ' + last if topic_string else last
    explanation = 'This article seems to be about {}.'.format(topic_string)
    return explanation

create_explanation(['Topic1', 'Topic2', 'Topic3', 'Topic4'] )
Output: "This article seems to be about Topic1, Topic2 and Topic3."
```

Listing 4.1: Example of explanation generation.

Lastly, previously shown recommendations are filtered out before they are submitted using the arXivDigest connector as described in step 5 and 6 in Section 4.1. Since the system works in batches, we need to repeat step 3-6 until all users have been processed.

4.3 Shared Article Recommender System Code

When we started working with the next system we realized that most of the code in the different systems, except for the actual recommendation method, would be almost identical. To avoid having to copy-paste too much code when creating a system, we decided to refactor the shared code into an abstract base class. This class handles things such as removing already recommended articles from the recommendations, creating explanations, getting user batches and sending recommendations. We are not using this class for the baseline system for two main reasons. Firstly, we wanted to keep the baseline system as simple as possible and not clutter it with any additional functionality from the base class. The second reason is that the other new systems will be in a different repository than the base system, making it harder to share code with the current setup. Having this base class available makes creating new systems much cleaner and easier as we can avoid writing duplicate code. It also makes updating the systems easier as we can apply changes to every system in one place.

To implement a new system we can just extend the `BaseArticleRecommenderSystem` abstract class. Before the new system can be run we need to implement the `_create_recommendations` function and the `run` function. Setup should be added in the `run` function, while the article recommendation logic needs to be added to the `_create_recommendations` function.

We also found out that most of our systems would probably use an Elasticsearch index in some way, so we decided to create an article index class with the most commonly needed functionality. This index class contains functionality for creating the index, adding the articles to the index, searching articles by topics or authors and other similar functions. Combined with the abstract base system it becomes very easy to setup a simple system, letting us instead focus on implementing advanced techniques instead of repeating code.

4.4 Query Expansion Based System

After completing the sample system and abstracting away the boilerplate code, we could finally start focus on more interesting recommendation techniques. The first method we decided to implement is a type of query modeling technique.

4.4.1 Background

When users specifies a query, in our case topics of interest, they try to describe a representation of their information need. Even when trying their best, most users will be unable to provide a complete representation of this information need. There will always be more synonyms, topics that they forget to add, related topics that they have not heard about yet, etc. This is the problem query modeling tries to solve by using methods such as weighting the the terms in the query and expanding the query with terms that fill these gaps [34]. There exist many different methods for achieving this, including using collection co-occurrence data to calculate similarity between terms and then adding the terms that are most similar to the query-terms to the query. Other methods have used dictionaries and grammar rules for finding similar terms that can be added to the query, or finding new terms by extracting them from known relevant documents [34, 35].

Since all our users have a library of saved articles they have shown interest in, we have a golden opportunity for using known relevant articles as our basis for the query expansion. The idea being that by expanding the query with terms from articles in the users library, we will be able to recommend articles similar to what the users already like. We now have the known relevant documents, so the next thing we need is a method of extracting and weighing the terms such that we can use them for building an expanded query. Balog et al. [34] propose a method for expanding queries with weighted terms from sample documents with promising results. The paper originally describes the method in conjunction with language modeling, but the concept and formulas are easily generalized, letting us use the expanded query directly with Elasticsearch. We decided to implement this method in our system as it seemed like it would fit our need for extracting and weighing terms from the users library.

First we need to decide which terms to expand our query with. This is done by selecting the K terms with the highest sampling probability, $P(t|S)$, where K is the number of terms to expand with. $P(t|S)$ can be calculated using using Equation 4.1, where $P(t|D)$ is the probability of sampling term t from document D and $P(D|S)$ is the importance of document D in the sample collection S [34]. If we assume the documents to be equally important we can calculate $P(D|S)$ as $P(D|S) = 1/|S|$.

$$P(t|S) = \sum_{D \in S} P(t|D) \cdot P(D|S) \tag{4.1}$$

We can calculate $P(t|D)$ by taking the maximum likelihood estimate of a term using Equation 4.2, where $n(t, D)$ is the number of occurrences of term t in document D. The other option is to calculate it by smoothing the maximum likelihood estimate of the term in document D with the maximum likelihood estimate of the term in the collection C, like in Equation 4.3 [34].

$$P(t|D) = P_{ML}(t|D) = \frac{n(t,D)}{\sum_{t'} n(t',D)} \tag{4.2}$$

$$P(t|D) = P(t|\theta_D) = (1-\lambda) \cdot P_{ML}(t|D) + \lambda \cdot P_{ML}(t|C) \tag{4.3}$$

Now that we know which terms to expand the query with, we need to calculate the weight of all the terms in the new query. This also includes reweighing the terms in the original query. We can calculate the weight to assign to each term by using Equation 4.4, where we use $P(t|\theta_Q)$ as the weight. μ is the weighting of the original query compared to the expanded terms, $P(t|Q)$ and $P(t|\hat{Q})$ are the probability of sampling term t from the query Q and expanded query $\hat{Q}$) [34].

$$P(t|\theta_Q) = (1-\mu) \cdot P(t|\hat{Q}) + \mu \cdot P(t|Q) \tag{4.4}$$

$P(t|Q)$ can be calculated using Equation 4.5 and is just based on the frequencies of terms in the original query [34].

$$P(t|Q) = \frac{n(t,Q)}{\sum_{t'} n(t',Q)} \tag{4.5}$$

$P(t|\hat{Q})$ can be calculated using Equation 4.6 [34], which normalizes the sampling probability of term t from sample collection S for the K terms selected for expansion, such that the sum of their probabilities becomes 1.

$$P(t|\hat{Q}) = \sum_{t \in K} \frac{P(t|S)}{\sum_{t'} P(t'|S)} \tag{4.6}$$

4.4.2 Implementation

As earlier stated, we had most of the basic functionality for a system already implemented in the system base class, article index class and the arXivDigest connector. That meant we could start directly on implementing the query expansion method. The only information we needed to calculate which terms to use for expansion and their weights was the term counts of all the terms in the saved articles and topics. Both of which are easy enough to get as the arXivDigest API exposes endpoints for fetching topics, saved article ids, and the content of the articles.

Figure 4.2: First 10 expanded terms and their weights

quantum (0.077), computers (0.068), we (0.063), from (0.054), states (0.054), our (0.047), classical (0.043), can (0.039), provides (0.039), which (0.035)

However, when doing this we discovered a problem. When counting terms we first want to stem the terms in a way such that `house` and `houses` are treated as the same term, as they would be when we search using them anyway. The problem is that when we stem a word we are not guaranteed to get a real word back, in this case the result of the stemming would be `hous`. This is not ideal seeing that we are going to include them in the human readable explanations later. Our solution for this was to count how many of each term gets stemmed to each stem. After we are done counting, we select the most common unstemmed version of each term as the human readable version. This is not a perfect solution as the most common form is not guaranteed to be the best form to use in an explanation, but we believe it is the most fair since this is the form that contributed the most to the result.

We can then input the term counts directly in the formulas from the previous section to find the expanded topics and their weights. The next step is to query the Elasticsearch index with the weighted topics and the expanded topics. We can apply weights to an Elasticsearch query by using the following syntax: `(term)^weight`. While implementing the system, we created a fictional user to test the query expansion method on. This user had an interest for quantum computing and had therefore liked several articles about this topic. The top 10 expanded terms can be seen in Figure 4.2.

One of the first things we noticed was that many of the expanded terms hold very little discriminatory value, like `we`, `from`, `our` and `can`. This is despite the fact that Elasticsearch has removed some stopwords already. Having this many words of little discriminatory value in the expanded query is problematic as instead of adding value to the query, it just adds noise.

To try to combat this problem we created our own domain specific stopword list. Elasticsearch saves document frequency for all terms, so it is not to hard to check if a term is frequent or not. However, Elasticsearch does not provide an easy method of getting all terms present in an index. Turns out this is not huge problem as we are only interested in the common terms anyway. We can get a list of most common terms appearing in the index by finding all the terms appearing in a large sample of random documents, which we can get. Then we can get the document frequency of a term by dividing the number of documents the term appears in by the total number of documents in the index, both of which we can find by using the term vectors API of Elasticsearch. The final step in selecting the stopwords is just to select a threshold for the document frequency of a term. Normally we would not set this threshold too low, as many frequent terms may still hold some value to a query. However, we are not afraid of losing some common but useful terms in this case since we are more interested in the terms that are more common in the saved articles and less common in the collection overall.

In Figure 4.3 we can see the top 10 expanded terms after filtering with the domain specific stopword list. Now most of the terms are more descriptive of the topics of the saved articles, even though there still exist some terms that do not tell us too much like `become` and `even`.

Figure 4.3: First 10 expanded terms and their weights after applying stopwords

quantum (0.103), classical (0.057), protocol (0.047), information (0.047), interacting (0.046), security (0.046), become (0.043), even (0.041), implementations (0.041), expected (0.040)

4.5 Semantic Reranking Based System

4.5.1 Background

Up until now we have treated words as atomic units. The words 'king' and 'queen' are two completely separate entities with no relation, even though both are a form of monarch. Another method of representing words which has some interesting properties, is to build low dimensional vector representations of words also known as word embeddings [36]. It has been shown that by learning vector representations of words from large amount of unstructured text data, one can get vectors with an understanding of relationships between words. An example of this is by subtracting the vector of 'Spain' from the vector of 'Madrid' and adding the vector of 'France' gives a resulting vector that is closer to the vector of 'Paris' than to any other word [37]. The similarity between words vectors can be measured by calculating the cosine similarities of the vectors [36]. Cosine similarity is a measure of how much the direction of two vectors differ and can be calculated by taking the cosine of the angle between the vectors [38].

It is also possible to use word embeddings to compare documents instead of just single words. The simplest method of creating a vector representation of a document is to sum the embeddings of each word in the document and divide it by the number of words. The resulting document vectors can then be compared to each other or the vector of a query by using cosine similarity [39]. Having word embeddings for documents and queries makes it possible to give high retrieval scores to documents that have few or none common terms with the query because they have a similar semantics [39].

As we discussed in the query expansion system, there will almost always exist gaps in the users query either though missing synonyms, missing closely related topics and many other reasons. Because of this, we thought word embedding seemed like an interesting method of trying to fill the gap between the query and the information need by focusing more on the semantics of the query instead of the exact wording.

4.5.2 Implementation

We decided that it would be best to rerank a first-pass result, instead of creating and comparing word embeddings for all documents available for recommendation. This first pass result is just based on the regular user topics. The first-pass results are then reranked based on the similarity of the word embeddings of the result's titles against the word embeddings of the user's topics. There are two reasons for not reranking all documents. The first reason being that this could get very computationally demanding with increasing number of documents, while the second reason is that as most users will have many topics, most relevant documents will share at least one common topic. The reason we chose to rerank the documents based on just the title instead of the whole document is that we are not quite sure how well this simple method of combining word embeddings scales with longer documents. For the word embedding library to use, We chose Word2vec through Gensim [40]. Word2vec is one of the most popular word embedding approaches, which have shown promising results in earlier works [37]. This library provides several high quality pre-trained models, but also provides an easy to use interface for training models on our own data.

```
def calculate_sentence_vectors(sentences):
    vectors = []
    for sentence in sentences:
        tokens  = gensim.utils.simple_preprocess(sentence)
        tokens =  [t for t in tokens if t in model.vocab]
        vectors.append(sum([model[token] for token in tokens]))
    return vectors
```

Listing 4.2: Functions for calculating sentence embedding.

In Listing 4.2 we can see the the function we use for calculating sentence embeddings. To do this we first preprocess the sentence using a preprocessing method provided by Gensim for better compatibility with the pre-trained models. Then we sum the word vector for every word in the sentence.

```
def calculate_ranking(titles, topics, threshold):
    topic_vecs = calculate_sentence_vectors(topics)
    title_vecs = calculate_sentence_vectors(titles)
    title_scores = defaultdict(list)
    for topic, vector in zip(topics, topic_vecs):
        scores = model.cosine_similarities(vector, title_vecs)
        for title, score in zip(title, scores):
            if score > threshold:
                title_scores[title].append((score, topic))
    return title_scores
```

Listing 4.3: Functions for ranking titles by topics using word embedding.

We can then calculate the score of every topic to every title like shown in Listing 4.3 by computing the cosine similarity. The final score of each title is calculated by summing together the scores of each topic. When implementing this function, we had to incorporate a threshold for how similar two embeddings must be to contribute to the score. This is because even very different vectors will have some similarity when measuring cosine similarity. By thresholding the scores, we can make sure that only topics that match the title by a meaningful amount gets to contribute. Not thresholding scores would create very misleading explanations for titles that have a low similarity to all topics. This would especially be a problem if there are few relevant articles for a user one day. Then one of these misleading explanations might show up in their recommendations on arXivDigest.

Verifying the Implementation

Table 4.1: Sample titles

Fauna diversity in equatorial habitats.
How forests affect climate change.
Frogs live in the swamp, near water.
Computers are built from electronics and transistors.
Cultures around the world.
Black holes are formed when stars collapses.

Table 4.2: Sample topics

Rainforest climate
Tropical species
Climate change and endangered species

Before we tried to incorporate this method into a system, we wanted to test it with some sample titles and queries. This was done because we wanted to get a better understanding of how the method works, as this will be more difficult to do when it is implemented as a system. The experiments we performed were in no way comprehensive enough to give accurate performance results for the method, but they were designed to help us understand the method and check if our implementation showed some potential. In Table 4.1 and Table 4.2 we can see the titles and topics created for one of these experiments. The topics are centered around climate change and species in the rainforest. If the method works, we expect high scores for the first two titles. The third title is in the grey zone, so while it should be above the threshold we would expect it to have a lower score. We do not expect the last three titles to score above the threshold at all, as they are not related to the topics in any way. In the experiment we used the google-news-300 pre-trained embeddings and a threshold of 0.4. We found that this threshold value varied much from embedding to embedding, but a threshold between 0.35 and 0.45 seemed to work good for our purpose using this model.

In Figure 4.4 we can see the result of the sample ranking. The ranking matched our initial expectations, and seemed quite promising. Because of this, we decided to move forward with implementing this as a recommender system.

Figure 4.4: Final ranking of sample titles and topics with scores

- Fauna diversity in equatorial habitats. (2.03)
  - Tropical species (0.72)
  - Climate change and endangered species (0.66)
  - Rainforest climate (0.64)
- How forests affect climate change. (1.75)
  - Climate change and endangered species (0.70)
  - Rainforest climate (0.65)
  - Tropical species (0.41)
- Frogs live in the swamp, near water. (0.91)
  - Tropical species (0.47)
  - Climate change and endangered species (0.44)
- Computers are built from electronics and transistors. (0)
- Cultures around the world. (0)
- Black holes are formed when stars collapses. (0)

Training Our Own Model

The google-news-300 embeddings are trained on a large corpus of news articles. While this embedding delivers impressive performance, we find it reasonable to assume that the terms and topics varies greatly between news articles and scientific literature. Because of this, we decided to train our own embedding. Getting the data to train an embedding is not a big problem, as we have access to almost two million articles from arXiv [1]. We also have the needed functionality for downloading them through the scraping part of arXivDigest. Downloading all the articles was quite slow, so we decided to implement some simple caching in case we wanted to rebuild the model with updated data. The cache is just a collection of files sorted by dates and that way we can keep downloading articles from the point where we stopped the previous time the script was run. For simpler training, we merge all the titles and abstract of the articles into a single file with one line per article. Using Gensim makes the actual training part very easy and the code we used for training the embedding can be seen in Listing 4.4. The embedding training is a standalone script which should be run at regular intervals, to accommodate the evolving vocabulary of scientific literature.

```
class SentenceIterator:
"""Preprocesses and yields line by line  for input file."""
def __init__(self, filepath):
    self.filepath = filepath

def __iter__(self):
    with open(self.filepath) as file:
        for line in file:
            yield gensim.utils.simple_preprocess(line)

model = Word2Vec(SentenceIterator(CORPUS_FILE))
```

Listing 4.4: Code for training a word2vec embedding using gensim.

The embedding seemed to pickup some language patterns from the dataset, one example of this can be seen in Figure 4.5, where the top 10 similar terms to the term AES is shown. AES is an encryption standard, so it makes sense that most of the terms are either related to cryptography or cryptography methods. We will have to wait for results from the living lab for further validation of the method and model, as we do not have any labeled data to compare against.

Implementing the Recommender System

To finish the system we just need to combine the base system class, the article index class, the trained embedding and the ranking function defined in Listing 4.3. The first

Figure 4.5: Top 10 similar terms AES.

decryption (0.703), cipher (0.695), encryption, (0.687), paillier (0.680), ecdsa (0.672), mceliece (0.669), cryptosystem (0.668), rijndael (0.666), elgamal (0.666), encryptions (0.653)

thing we have to do when we initialize the system is to load the word2vec embedding, which should already be trained. Then the system queries the article index for a first-pass result based on each users' topics. This first-pass result is scored against each topic like in Listing 4.3. The articles in the first-pass result are then sorted by the sum of the topic scores and the top articles are selected as recommendations. Explanations are generated as in the other systems, where the topics contributing the most gets included in the explanations.

4.6 Author Based System

4.6.1 Background

For the sake of variety we also wanted to provide the users with recommendations that are not based on topics in any way. We considered techniques like implementing a system based on similarities of paths between objects in a heterogeneous information network (HIN) as shown in [41]. A HIN is a network of objects and their connections. This can be objects like authors, venues, citations and connections like authored by, cited by and published at. Similarity between objects can be measured by by evaluating the number of paths connecting them [41]. In the end, because of time constraints, we decided to go for a simpler approach that works based on the same premise.

We chose to implement a system that recommends articles based on the authors of the users earlier saved articles. The reason being that an author often write articles concerning the same field of study and it is therefore not unreasonable to assume that the user may be interested in other works by the same author.

4.6.2 Implementation

The system is built using the base system class and the article index class, same as the other systems. We now needed to add the article authors to the article index, preferably as a list of authors for each article. However, this leads to a problem if we were to use the normal Elasticsearch data types to store and search for authors. Due to how Elastic search handles inner objects, the association between values are lost. This means that if we add the authors Alice Jones and John Smith to an article, it would actually match Alice Jones, Alice Smith, John Smith and John Jones. Fortunately, Elasticsearch provides a solution to this problem, which is using nested objects and queries [42]. By using these we can query authors independently. We add functions both for adding nested authors to articles while indexing articles, and for searching for authors to the article index class.

We can then get all the authors of articles saved by a user through looking up which articles a user has has saved using the arXivDigest connector. Which we can then get the author information for by using the arXivDigest connector again. The system searches for articles matching these authors using the author search function from the article index for each author. Explanations are then generated using the same approach as in Listing 4.1, only this time we replace the explanation template with: 'You have previously saved articles authored by .'.

Chapter 5

Topic Recommendation

5.1 Overview

As we have seen in Chapter 4, many of the article recommendation systems are based on the topics the users have listed on their profiles. Therefore, both the quality and accuracy of article recommendations from these systems are dependent on the quality and accuracy of the users' topics. The better users' topics fit their interest profile, the better article recommendations provided by these systems will be. To increase the quality and quantity of topics on the users' profiles, we decided to implement the possibility of recommending topics to the users in the same way we recommend articles. This would provide better topic auto-complete on sign-up and also suggest topics to users whenever they visit the web interface. The process of actually recommending these topics were left to external topic recommender systems, much in the same way as recommending articles are left to external article recommender systems. These systems will have access to all the same user information as the article recommender systems as well as the users feedback on previous topics.

5.2 Common Functions

The different topic recommender systems will have a bit of shared code between them. This includes the fetching of user information, the text preprocessing they use and the sending of topic recommendations back to the arXivDigest API. We describe these shared components in this section before going on to explain each topic recommendation method we implemented.

5.2.1 Fetching User Information

Before we could recommend any topics to users, we needed some user information to base the topic recommendations on. The arXivDigest API provides user information like topic interaction data, article interaction data, the users' interest for arXiv categories and links to the users' profiles at external services. This is the starting point for most topic recommendation. We consider two main ways of using this information for recommending topics. The first method we consider is to extract topics from text that is relevant to the user, which in our case will most likely be scientific literature they are interested in. The other method is to match topics from a pre-existing list of topics to a user profile based on some similarity metric.

There were two main sources for relevant scientific literature for a user that we could think of. We find it reasonable to assume that the library of saved articles each user has on arXivDigest, contains literature relevant to the user. For the users that have links to services containing published papers, we can also find it reasonable to assume that the user is interested in the topics mentioned in their published papers. The arXivDigest platform lets users add links to four of these services. These four services are their personal website, their DBLP profile, their Google Scholar profile and their Semantic Scholar profile. We did not bother with using the personal websites for this project, as these can come in many different and possibly unique formats. We also had to skip Google Scholar because it lacked a good API for fetching the data we wanted and the website was created dynamically with extra requests which made it difficult to scrape the titles directly. However, both DBLP and Semantic Scholar provides easy to use APIs for fetching the titles of a user's publications, which should be enough to extract some relevant topics from.

DBLP

DBLP has an API in place to fetch all the data for one user as an XML file. However, to use this API and get the data, we need the user's persistent profile URL. Through the arXivDigest's API we only have the users general DBLP profile page link. So before we could download the user data, we needed to use the general DBLP profile page link to get the users persistent profile URL. This persistent profile URL can be found in a drop-down menu on the users DBLP profile as seen in Figure 5.1. We get this URL by using the Beautifulsoup library [31] to parse the profile page of the user and then search for the persistent URL. Once we had the persistent URL, we could add `.xml` to the end of this URL to get the link to a XML file containing all the user's DBLP profile data. We used Beautifulsoup again to search for all the `title` tags in the XML file to get all

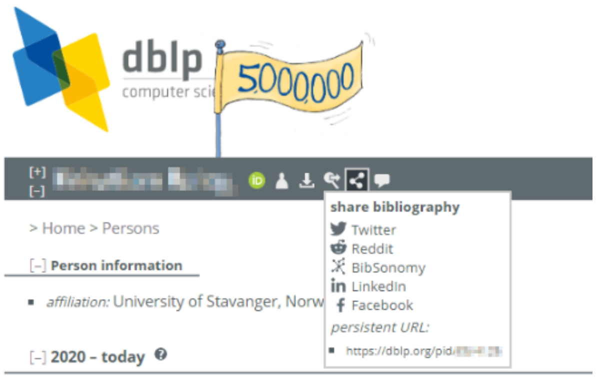


Figure 5.1: Location of persistent URL on DBLP.

the titles of the user's publications. An example of how to find these titles can be seen in Listing 5.1.

```
resp = requests.get(persistent_profile_URL + '.xml')

soup = BeautifulSoup(resp.content, 'lxml')

titles = soup.find_all('title')
```

Listing 5.1: Using Beautifulsoup to find user titles.

Semantic Scholar

Semantic Scholar also has an API in place for providing data about its authors. However, the Semantic Scholar API only needs the user's Semantic Scholar author ID to complete its request. This ID can be found as a part of the user's Semantic Scholar profile URL that we have from the user's profile on arXivDigest. We only need to split the Semantic Scholar profile URL on '/' and we have the Semantic Scholar author ID. This ID is then used in a request to the Semantic Scholar API using the URL `https://api.semanticscholar.org/v1/author/[S2AuthorId]`. Semantic Scholar then returns a JSON object with the user's information. We can search for the `title` tag in the JSON object to fetch the titles of the users publications. Example of how to find these titles can be seen in Listing 5.2.

```
API_URL = 'https://api.semanticscholar.org/v1/author/'
resp = requests.get(API_URL + semantic_scholar_author_id)

author_info = json.loads(resp.content)
titles = []

for paper in author_info['papers']:
    titles.append(paper['title'])
```

Listing 5.2: Fetching publication titles from Semantic Scholar API.

5.2.2 Text Preprocessing

We use the same text preprocessing in all of the topic extraction algorithms so we will also cover the preprocessing before we look into the specific extraction algorithms. We decided to make the preprocessing the same for each of the extraction methods to give them an equal starting point for a more fair comparison of the results. The preprocessing method we went for in the topic recommender systems is heavily inspired by Jishnu Ray Chowdhury's implementation of TextRank where he also included some preprocessing steps in his implementation [43].

Tokenization

There are several steps involved in our text preprocessing for the topic extraction systems. The first step is to clean the provided text for all non-printable characters. We do this with the `printable` constant from the `string` library. In this step we also tokenize the text with the `word_tokenize` function from the NLTK library [44] and convert all the characters to lowercase. An example of the `word_tokenize` function and how we lowercase the characters can be seen in Listing 5.3. (See Section 2.1.1 for an explanation on what tokenization is and why we do it.)

```
from nltk import word_tokenize

tokenized_text = word_tokenize(text.lower())
```

Listing 5.3: Word_tokenize example

Lemmatization

Next, we lemmatize each words in the cleaned text using the `wordnetlemmatizer` from the NLTK library. For more on lemmatization see Section 2.1.1. We also use the `pos_tags` function from NLTK to get the POS tags of a word. Using these POS tags, we can figure out if the word is a noun, verb, adjective etc. and we can supply this information along with the word to the `wordnetlemmatizer`. This will give us a more accurate lemmatization for each word. The lemmatzation process is shown in Listing 5.4.

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

POS_group = get_word_POS(word)
lemmatized_word = lemmatized.lemmatize(word, POS_group)
```

Listing 5.4: Wordnetlemmatizer example

One problem that this lemmatization creates is that while it helps making the algorithms more accurate, the resulting lemmatized topics may not always make sense grammatically. To solve this, we need to be able to translate the lemmatized topic recommendations back into non-lemmatized words for the final topics. We try to tackle this problem much in the same way that we tackled it for stemming in Section 4.4.2. As with stemming, one lemmatized word might come from several different forms of the unlemmatized word and we do not know which of these forms to translate it back into. Again, we use a translation dictionary to keep track of which lemmatized word stems from which original word and a counter of how many times the original word was lemmatized into that specific lemmatized word. Then, when we translate the lemmatized word back, we can choose the translation back to the original form of the word with the highest counter. This ensures that we have a higher chance of making a translation into a topic recommendation that makes more sense grammatically.

Stopwords

As a final step of the preprocessing, we create our stopword list. We chose to implement a hybrid version of a stopword list which uses both a predefined list of stopwords and also a generated list of stopwords. The predefined stopword list contain most of the basic stopwords one would find elsewhere, but no advanced words or phrases. We use this predefined list to guarantee that all these basic words will be filtered out. To get an even better final stopword list, we also generate our own stopwords based on the provided text. The way we generate this stopword list is by looking at the words' POS tags. Usual topics are often noun, adjectives or foreign words [43]. We therefore add all words that are not noun, adjectives or foreign words to the already existing predefined stopword list. This gives us a stopword list tailored for the texts we extract topics from. The advantage of this self-generated stopword list is that the stopwords change to fit the texts of each user. Because of this, we are able to remove words that are less likely to create good topics based on the words' properties. This will be important later, especially because one of the methods we will try using for extracting topics uses stopwords as a separator between topics [20].

5.2.3 Base Topic Recommender Class

When we implemented the different topic recommendation algorithms, we needed a framework to support the algorithms and provide them with the necessary information such as information about the users and existing topics. Most of this code would be identical for all topic systems, like fetching user data and topic data from the arXivDigest

API or submitting topic recommendations. Making a shared framework structure for all the topic recommender systems therefore seemed like the most efficient approach. This would reduce a lot of duplicate code between the topic recommender systems and also make it easier to implement new systems in the future.

We created a `BaseTopicRecommenderSystem` class that is responsible for handling all of the duplicate code between the topic recommender systems. This included fetching user information from the arXivDigest API, scraping the profile of the users' external services for text to use in the topic extraction algorithms and sending the finished topic recommendations back to the arXivDigest API. The only unfinished function is the `recommend_topics` function that creates the topics recommendations. The `BaseTopicRecommenderSystem` also includes a `run` function that is used to execute the topic recommender system.

To create a new topic recommender system, we have to create a new class for that specific topic recommender system. This new class should extend the `BaseTopicRecommenderSystem` class, which gives the topic recommender system access to all the functions from the `BaseTopicRecommenderSystem` that we mentioned above. The only function we need to implement in the new topic recommender class is the abstract `recommend_topics` function. Here we implement the specific topic extraction method that we are going to use for that topic recommender system.

Most topic recommender systems will follow the steps outlined in the list below, these steps are also shown in Figure 5.2 with numbers corresponding to the list. These are the steps that we have tried following when implementing the base system to make it as generalized as possible. Step 3 is where the actual recommendation happens and it is thus up to the system developers to define. This is reflected in the base class, where the unfinished function `recommend_topics` may be compared to step 3.

1. Query arXivDigest for users and their information.
2. The system may optionally collect extra user data from external services like Semantic Scholar or DBLP, if the user has provided links to any services like this.
3. Use the collected user information to recommend topic to each user with a method of choice.
4. Clean all the topics so that they conform to arXivDigest's standard. This include removing illegal symbols, too long topics and topics that a user already has seen. Data for this standard is provided by the arXivDigest connector.
5. Send the top scoring topics for each user back to arXivDigest.

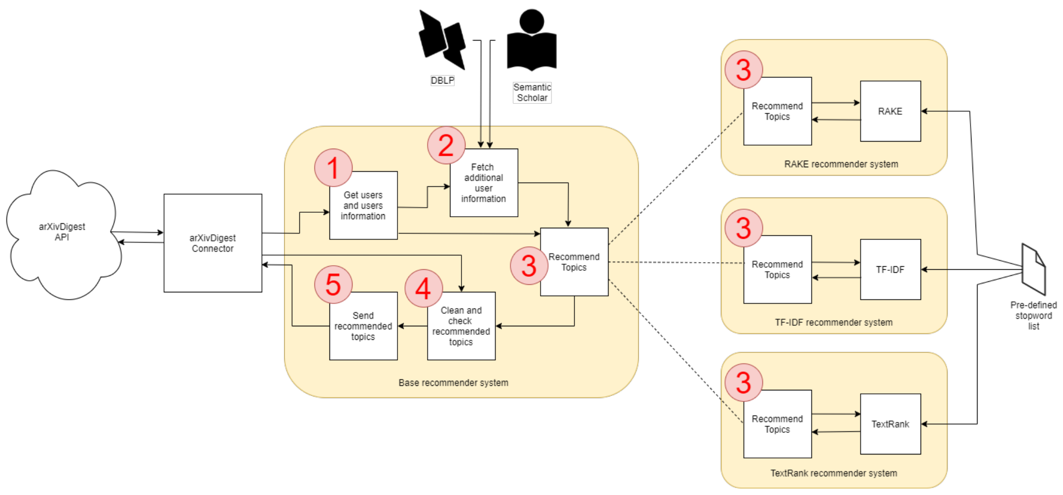


Figure 5.2: Overview of topic recommender systems structure.

5.3 Topic Recommendation Algorithms

We decided to implement a selection of two different topic extraction algorithms and one method for recommending topics from a list of existing topics. The two topic extraction algorithms we selected are from the document-oriented methods that we talked about earlier in Section 2.3.1. We selected RAKE [20] and TextRank [22] because the algorithms are both able to extract phrases instead of only single words, and because they have shown promising results earlier.

5.3.1 RAKE

The first topic extraction method we implemented was the RAKE algorithm proposed by Stuart Rose et al. [20]. This algorithm was suggested to us by our supervisor and after reading the paper we chose to implement this algorithm because it seemed to suit our needs well. According to the source paper RAKE works well on different domains and also operates well on multiple different types of documents, particularly those that do not follow specific grammar conventions [20]. This is good for us since we were going to use titles of scientific articles as the input texts for this algorithm and article titles are often written with different structure and wording than normal sentences.

Algorithm

RAKE begins with creating candidate topics from the preprocessed text. These are unique phrases that are considered as allowed topics. To do this, RAKE takes the preprocessed text list and uses phrase delimiters (punctuations) and stopwords to divide the text into shorter phrases which will be our candidate topics [20, 45]. After this first step, we have a list of unique candidate topics with varying lengths. One problem with using delimiters and stopwords to separate candidate topics is that the resulting candidate topics can not contain an interior stopword. This was described as the 'axis of evil' problem in the original paper [20], since 'axis of evil' would never be selected as a candidate topic because of the 'of' stopword in the middle. To solve this problem, we also keep track of all the possible adjoining topic combinations and the number of times they appear in the text. Then, after we have fully processed all the texts, we select the adjoining topics from all the possible adjoining topic combinations that satisfy our requirements. These adjoining topic requirements include appearing in the document a certain number of times, being under a certain maximum length and neither starting or ending with a stopword. The adjoining topics are then added to the previously made list of candidate topics [20].

After we have the list of candidate topics it is time to calculate the score of each candidate. To do this, we have the choice between three scoring metrics. These metrics are word frequency, word degree and word degree divided by word frequency [20].

- Word frequency is simply the number of times a single word occurs in all the candidate topics. This metric favors words that frequently occur regardless of which candidate topic they come from.
- Word degree also takes into consideration the neighboring words that often occur together. This is calculated as a matrix with all the single words from all candidate topics along each axis. We then count what words occur together in a text. The final degree score for a word is achieved by adding together the numbers for one word in one row in the degree matrix. Degree favors the words that occur often and also often in longer candidate topics.
- Word degree divided by word frequency is as simple as it sounds. To calculate this, simply divide the word's degree score on the word's frequency score. This metric favors the words that mostly only occur in longer candidate topics.

Finally we can compute the scores of the candidate topics by taking the sum of the score of each word in the candidate topic. This will give a final score for the candidate topic that is the sum of all the scores of the individual words it contains. We then sort the candidate topics based on their final scores and that is the output of the RAKE algorithm [20].

Implementation

We decided to implement RAKE as its own class. This allow us to create a RAKE object with specific parameters that we can keep reusing for each user in our recommendation system. It also keeps the RAKE functions isolated in the class, away from the rest of our recommender system. Since we were going to work with the titles of the users' saved and published articles, we chose to let the input value of the RAKE algorithm be a list of sentences. In our case, this would be the list of titles of relevant articles to a user. In Listing 5.5, we show how we create a RAKE object and how we execute the algorithm with a list of titles by using the `run_rake` function. The extracted topics and their scores can be fetched with the `get_keywords_with_score` function.

```
from rake import Rake, Metric

scoring_metric = Metric.WORD_FREQUENCY
r = Rake(max_length,
         stopwords_file,
         scoring_metric,
         punctuations,
         min_occurrences)

r.run_rake(title_list)

sorted_final_scores  = r.get_keywords_with_score()
```

Listing 5.5: Initialize and use RAKE

As we explained in Section 5.3.1, RAKE can use three different scoring metrics to score the extracted topics. We implemented this as an enum as seen in Listing 5.6. This way we could use the names of the metrics to make the code more understandable while also reduce the risks of crashing the program due to spelling errors.

```
class Metric(Enum):
    """Different metrics that can be used for scoring the extracted topics."""
    DEGREE_TO_FREQUENCY_RATIO = 0
    WORD_DEGREE = 1
    WORD_FREQUENCY = 2
```

Listing 5.6: Creating the metric enum

We can use the text in Figure 5.3 as an example. This is what the RAKE algorithm will use to create the candidate topics. In this example, the candidate topics created by RAKE can be seen in Figure 5.4. The top five final scoring topics from our RAKE implementation can be seen in figure Figure 5.5. This result was calculated using the word degree divided by word frequency metric described above in Section 5.3.1. We can confirm our results by looking at one specific topic and the score. Lets look at the topic `linear diophantine equations` which got a `9.0` in score. If we use the single

Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal-supporting set of solutions and algorithms of construction of minimal generating sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal-supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.

Figure 5.3: Example sentence for topic extraction.

'criterion', 'compatibility', 'system', 'linear diophantine equation', 'strict inequations', 'nonstrict inequations', 'upper bound', 'component', 'minimal-supporting set', 'solution', 'algorithm', 'construction', 'minimal generate set', 'type', 'give', 'construct', 'solve', 'mixed type', 'criterion of compatibility', 'system of linear', 'bound for component', 'set of solution', 'solution and algorithm', 'algorithm of construction', 'construction of minimal', 'type of system', 'system be give', 'algorithm for construct', 'construct a minimal-supporting', 'system and system', 'system of mixed'

Figure 5.4: RAKE candidate topics from example sentence.

linear diophantine equations (9.0), minimal generating set (8.667), systems of linear (7.667), set of solutions (7.667), construction of minimal (7.0)

Figure 5.5: Top five final topics and scores from RAKE example.

'linear' (1), 'diophantine' (1), 'equation' (1)

Figure 5.6: Words frequency scores from RAKE.

'linear' (3), 'diophantine' (3), 'equation' (3)

Figure 5.7: Words degree scores from RAKE.

words frequency scores found in Figure 5.6 and the single words degree scores found in Figure 5.7, we can calculate the full topics score using Equation 5.1. We observe that this score matches the output score of the RAKE algorithm we implemented.

$$final\ score = \frac{3}{1} + \frac{3}{1} + \frac{3}{1} = 9 \tag{5.1}$$

5.3.2 TextRank

The next algorithm we decided to implement was the TextRank algorithm. This algorithm is specified in the paper 'TextRank: Bringing Order into Texts' by Rada Mihalcea and Paul Tarau [22]. We chose this algorithm for the same reasons we chose to implement RAKE. These reasons include working well on multiple types of documents and also supporting multi-word topics. The TextRank algorithm is based of a graph-based algorithm called PageRank. This makes it work in a very different way from how the RAKE algorithm works and we also thought it would be interesting to see how these two totally different algorithms would compare against each other.

Algorithm

As we mentioned above, the TextRank topic extraction algorithm is based on another algorithm called PageRank [46]. Knowing how this PageRank algorithm works is essential in also understanding how TextRank works. So first lets see how the PageRank algorithm works.

PageRank is an algorithm for measuring the relative importance of web pages based on the links leading to and from them [46]. This algorithm represents each website as a node in a graph, while the incoming and outgoing links between nodes are represented as directed edges. The intuition behind PageRank is that pages with more in-links are more popular and therefore more important than pages with fewer in-links. In addition, the links coming from more important pages are worth more than the links coming from less important pages [46].

The PageRank score of the pages are calculated iteratively until the scores converge. If iterated until convergence, the initial scores of the nodes does not really matter too much. However, more accurate initial scores will cause faster convergence [46]. For each iteration, every node is scored according to Equation 5.2 [47] where V_i is a node and d is a dampening factor for the case of nodes with no outgoing links. The iterative process is repeated until the changes in the scores of the nodes are below a certain threshold. The scores of each node at this point is the final importance of the web pages and the output of the algorithm [46]. In essence, PageRank lets web-pages vote for the importance of other web-pages, but votes from important web-pages are given more weight. As the importance of web-pages changes after a vote, the process must be repeated until the result stabilizes.

$$S(V_i) = (1 - d) + d \sum_{j \in In(V_i)} \frac{S(V_j)}{|Out(V_j)|} \tag{5.2}$$

Now that we know how PageRank works, let us continue with the TextRank algorithm. This works in a way that is very similar to PageRank, except the nodes in the graph are the words in the preprocessed text. The edges between the word nodes are formed when the words are within a certain distance of each other in the text. This distance is called the window size. We do not use directional edges in TextRank, instead all of the edges are considered both an in-link and an out-link [22]. To determine which words builds up the graph we start by looking at the already preprocessed text. We iterate through the text and all the words that are not stopwords are added into a vocabulary. If the word is already in the vocabulary, it is ignored. The final vocabulary is the set of unique words that appears in the preprocessed text. We use this set of words as the nodes in the graph we create from the text. To form the edges between these nodes, we use the preprocessed text to figure out which words are within the window size distance of each other. An edge is then formed between these two word nodes in the graph. Figure 5.9 shows an example of a TextRank graph.

After constructing the graph, we can start the TextRank scoring algorithm. Unlike PageRank, where each node can have a different initial weight, in TextRank each word node is assigned an initial weight of one. The algorithm then iterates through through all the nodes in several rounds, scoring them based on the weights and edges just like the PageRank algorithm [22]. The scoring function for TextRank can be seen in Equation 5.3. The iteration cycle stops when the changes in the scores are less than a certain threshold, or a maximum number of iterations has been reached. The score of a word node at the last iteration is then the final score for that word.

$$WS(V_i) = (1 - d) + d \sum_{j \in In(V_i)} \frac{w_j i}{\sum_{V_k \in Out(V_j)} w_j k} WS(V_j) \tag{5.3}$$

After the scores of the words in the vocabulary have been calculated, we can score our candidate topics. To get these candidate topics, we use almost the same technique as in RAKE where we divide the preprocessed text into candidate topics delimited by the stopwords and punctuations. The result is a set of unique candidate topics with varying lengths. To score these candidate topics we use the scores of each word from the vocabulary graph. The candidate topic is scored by the sum of the score of its individual words from the TextRank graph nodes. The resulting candidate topics and final scores are then sorted based on their scores which is then the final result of the algorithm.

Implementation

To match the RAKE implementation, we also implemented TextRank as its own class with a list of sentences as the input. This way we could use the same infrastructure code for executing both of these two algorithms. A TextRank object is created in very much the same way as a RAKE object as seen in Listing 5.7. We then run the TextRank algorithm with the `run_textrank` function with a list of article titles for one user. The final topics and their scores can be fetched using the `get_sorted_scores` function.

```
from textrank import TextRank

t = TextRank(windowsize,
             stopwords_file,
             max_iterations,
             d,
             threshold)

t.run_textrank(title_list)

sorted_final_scores  = t.get_sorted_scores()
```

Listing 5.7: Initialize and use TextRank

Originally we created an implementation heavily inspired by Jishnu Ray Chowdhury's implementation on Github [43], but we soon realized that we had to optimize this implementation. We ran into two main problems. Firstly, the algorithm was extremely slow which would become a problem when recommending topics to many users. The second problem was that for big vocabularies we would start to get memory errors. There were multiple reasons for these problems, one of them being that the text was iterated over excessively many times. This was solved by iterating through it in a single pass, storing data in hash-based structures such as dictionaries and sets for fast lookup. The memory errors was caused by the Python math library `Numpy` struggling to allocate huge two-dimensional arrays for storing the graph. The solution for this was to instead store the graph as a sparse representation of edges, as most of the matrix was only zeros anyway. This indirectly also improved the speed, as earlier when calculating the scores, much of the time was spent iterating over the zero values that were just skipped. By just iterating over the actual edges, the scoring became much faster. After doing these optimizations we were sufficiently content with the speed an memory usage for our use-case.

We will now show an example of TextRank results using the same text from Figure 5.3 as we tested RAKE on. First TextRank creates a graph from the text with words being nodes, and edges are created between words within close proximity. The graph created nonstrict (0.81523347), algorithm (1.2645329), compatibility (0.58448005), minimal-supporting (1.2524251), equation (0.8203648), minimal (0.71050864), linear (0.7854648), criterion (0.99296856), construct (0.68472433), diophantine (0.79357594), generate (0.7015791), system (2.388677), upper (0.80486965), bound (0.7717413), construction (0.7079529), component (0.7447296), strict (0.8255826), type (1.4630581), set (1.7876948), solution (1.7439471), give (0.6762841), inequations (1.5000348), mixed (0.5281559), solve (0.65141493)

Figure 5.8: Vocabulary and scores created by TextRank from example.

from this text can be seen in Figure 5.9. The vocabulary and resulting scores of the words in the vocabulary can be seen in Figure 5.8.

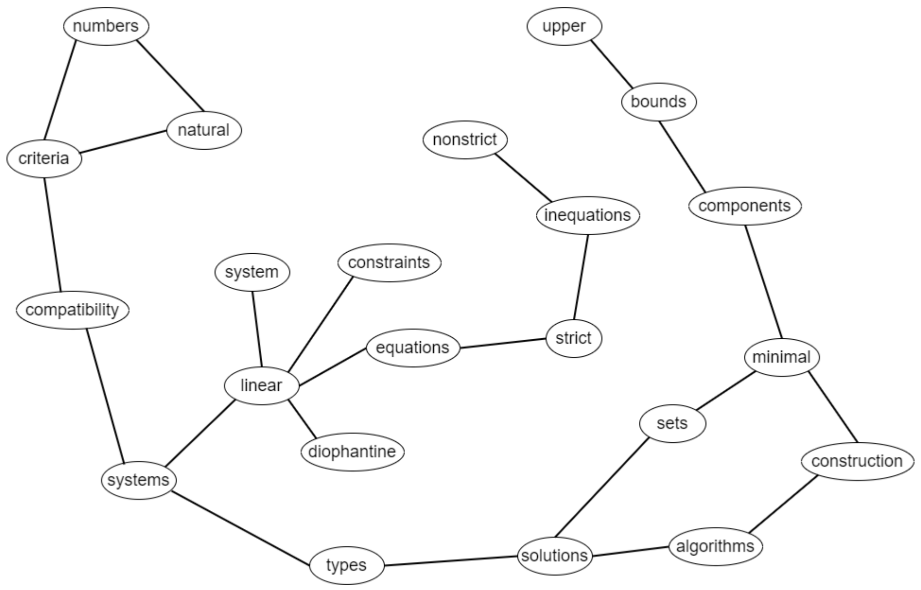


Figure 5.9: Example of a TextRank graph.

('algorithm',), ('linear', 'diophantine', 'equation'), ('upper', 'bound'), ('construction',), ('give',), ('strict', 'inequations'), ('solution',), ('system',), ('nonstrict', 'inequations'), ('compatibility',), ('solve',), ('mixed', 'type'), ('construct',), ('type',), ('minimal-supporting', 'set'), ('component',), ('criterion',), ('minimal', 'generate', 'set')

Figure 5.10: TextRank's unique phrases fom example.

We can then use the sum of the scores of the individual words in the candidate topics, that are shown in Figure 5.10, as the final score of a topic. The top five final scoring topics from TextRank can be seen in Figure 5.11,

The score for the top scoring topic `minimal generating set` can be calculated using the vocabulary scores as shown in Equation 5.4 using the values from Figure 5.8. This result is the same as we see for the same topic in Figure 5.11.

minimal generating set (3.199), minimal-supporting set (3.041), linear diophantine equations (2.399), systems (2.389), strict inequations (2.326)

Figure 5.11: Top five final topics and scores from TextRank example.

Here one can also observe the lemmatization translation that we mention in Section 5.2.2. In the vocabulary we have `generate` but the final topic is `generating` which fits much better for the topic `minimal generating set`. This shows that our lemmatization to original text translation works in adding a better suited form of the words to the final extracted topics.

$$final\ score = 0.71050864 + 0.7015791 + 1.7876948 \approx 3.199 \tag{5.4}$$

5.3.3 TF-IDF Weighting

The last algorithm we wanted to implement is not a topic extraction method, but rather a method that matches existing topics to user profiles based on some similarity measure. This gives us a way of reusing already recommended topics from our database for other users. For this approach, we decided to simply use a TF-IDF measure to find the most similar topics to a user profile, specifically the topics similarity to the titles of relevant articles for the user. We described the workings of the TF-IDF measure in further detail in Section 2.1.3.

Algorithm

We base our solution on an algorithm in a paper by Juan Ramos [48] where he uses TF-IDF to determine the word relevance of documents queries. In his approach he has a query and a collection of documents and the goal is to find the documents most relevant to the query. This is done by going though each document and calculate the TF and IDF scores for each word in the query. This calculation is detailed in Section 2.1.3. The most relevant documents are then calculated by summarizing the TF-IDF scores for all words in the query for each document and return the top scoring documents.

In our system we reverse the approach, by replacing the query with the preprocessed texts from relevant articles of a user and replacing the documents with the pre-extracted topics from arXivDigest. We then follow the approach detailed above, where we go through each of the topics and calculate the TF-IDF score for each word in each of the preprocessed texts. The most relevant topics are then calculated the same way, by taking the sum of the TF-IDF score for all words in each of the preprocessed texts. We can then select the top scoring topics as our recommendations.

Implementation

To match the RAKE and TextRank implementations, the TF-IDF algorithm was also implemented as its own class. As with the other two algorithms, the TF-IDF uses a list of titles the users has published as the input parameter. In addition to the users' titles, this approach also needs some topics to match to the users. We use topics provided by arXivDigest as the candidates for recommendations. These can be fetched from the arXivDigest API's `topics` endpoint. The TF-IDF algorithm can be initialized as seen in Listing 5.8. The algorithm is executed with the `run_tfidf` function. The final topic scores can then be fetched using the `get_topic_scores` function.

To get better word matching between the preprocessed texts and the topics, we also apply the same preprocessing steps we used on the texts to the topics. This way the topics would better match words in the texts and we would get a more accurate TF-IDF score. When calculating the final topic scores, we also divided the full topic score by the number of words in the topic. This prevented longer topics from automatically getting a better score and gave a more fair chance to shorter topics. The resulting topics and scores are then sorted based on their scores and this is the output of the algorithm.

Function for calculating TF-IDF for all topics in all sentences can be seen in Listing 5.11. This uses the `tf` function from Listing 5.9 and the `idf` function from Listing 5.10.

```
from tf_idf import TFIDF

t = TFIDF(stopwords_file)

t.run_tfidf(text_sentences, topics)

sorted_final_scores  = t.get_topic_scores()
```

Listing 5.8: Initialize and use TF-IDF

```
def tf(self, text, topic):
    """Calculates the tf of a topic in a sentence."""
    return text.count(topic) / len(text)
```

Listing 5.9: Calculate TF

```
def idf(self, topic, sentences):
    """Calculates idf for a topic in the collection of sentences."""
    count = 0
    for sentence in sentences:
        if topic in sentence:
            count += 1
    return math.log(len(sentences) / (1 + count))
```

Listing 5.10: Calculate IDF

```
topic_stats = {}
for topic in topics:
    topic_stats[topic] = []
    idf = self.idf(topic, sentences)
    for sentence in sentences:
        tf = self.tf(sentence, topic)
        topic_stats[topic].append(tf * idf)
```

Listing 5.11: Calculate TF-IDF statistics

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

A goal of this project was to update and launch arXivDigest as a living lab focused on the recommendation of scientific literature. The data collected through this living lab will be the the main method of evaluating the performance of the article and topic recommendation methods we implemented as the other goal of this thesis.

6.1.1 Evaluation Methodology

ArXivDigest is an online evaluation platform which builds on the concept of living labs discussed in Section 2.4.6. This means that we test our recommendation techniques on real users on a live platform. In Section 2.4 we discuss several strategies for comparing systems using online evaluation. For arXivDigest, we settled on using multileaving as our strategy, as this is a technique specifically designed to handle comparisons of many experimental recommender systems at once. Both interleaving and multileaving attempts to solve the problem of user variance by showing results from several systems to each user [24]. This is a very useful property, as having results from many systems to show to the users leads to less users seeing results from each system. By reducing the variance of the measurements it should decrease the needed amount of users for getting reliable evaluation results.

We collect various user interaction data that we use to gauge the users interest in the recommendations. This interaction data is discussed in more detail in Section 6.1.5. Each system gets a score for each multileaving based on the amount of interaction it gets from the users. This score has different weights depending on the type of interaction. For example the ’saved’ interaction gives a higher score than the ’clicked’ interaction. This

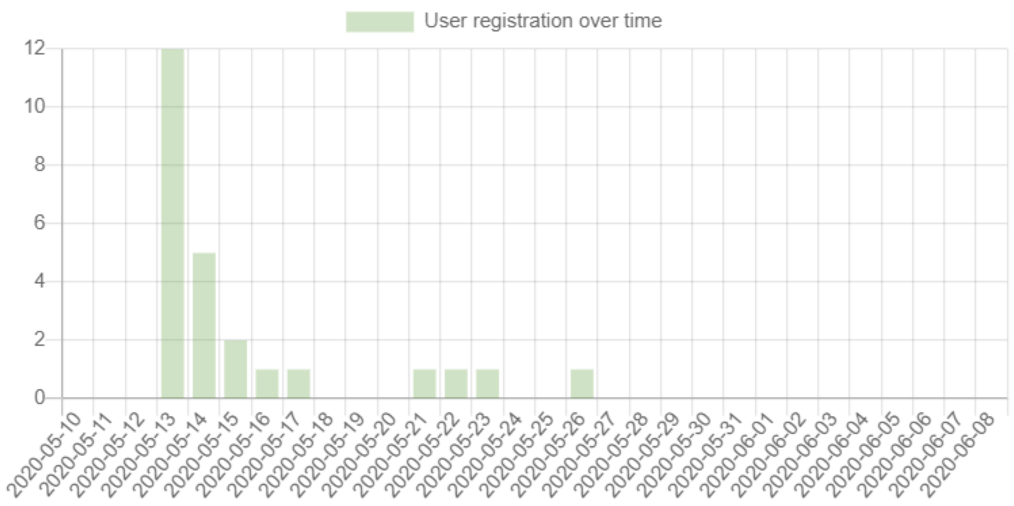


Figure 6.1: Plot over new users over time.

score is then used for calculating our performance metric, which we call mean normalized reward. We discuss how this metric work and the intuition behind it in Section 3.6.1. With this metric, the number of impressions a system gets and the feedback amount a system gets are all easily available through the web interface we described in Section 3.6.2.

6.1.2 Users

As arXivDigest is an online evaluation platform, we will not be able to collect any data without real users. We therefore got help from our supervisor to reach out to his connections and tell them about the arXivDigest platform. The first group of users to use the application were some of these connections, which agreed to trying the application at an earlier stage. This were around ten people who signed up before all of the parts of the application were up and running. We did not collect any usable data from this period, however they did help uncover some bugs we needed to fix. Later, when the application was fully operational, our supervisor advertised arXivDigest on the social media platform Twitter to get some more people to try the application. We can see this Twitter advertisement in Figure 6.1 as the date with the highest bar. The ten earlier users signed up outside of this graphs range.

6.1.3 Articles

As we mentioned in section 3.1, we get our articles from arXiv [1] every weekday. This includes all of the available information about the articles as well as their authors. In Figure 6.2 it shows how we get articles Monday to Friday and then a two day break during the weekend.

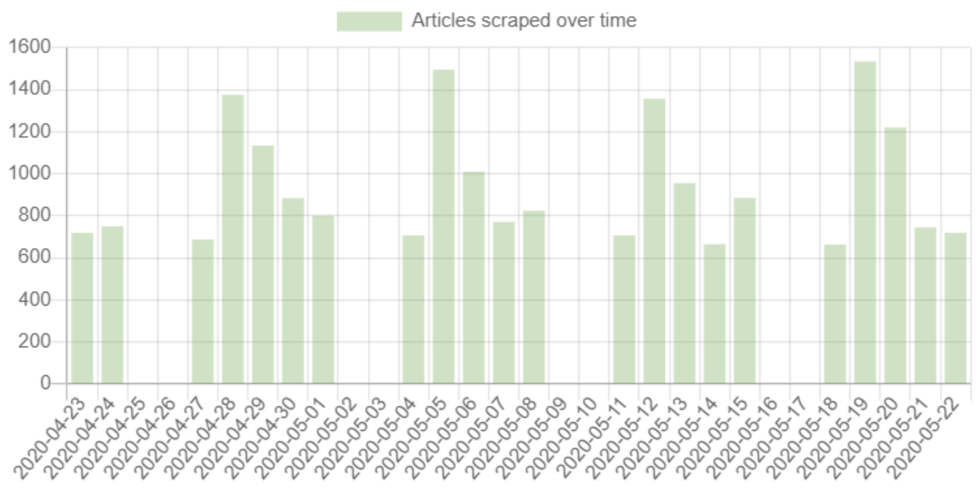
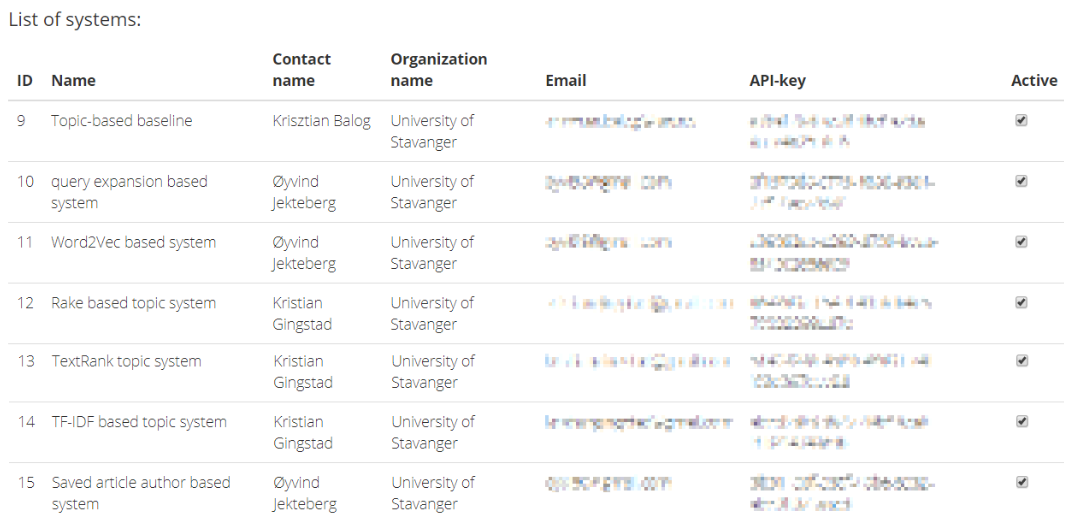


Figure 6.2: Plot over new articles over time.


List of systems:

ID	Name	Contact name	Organization name	Email	API-key	Active
9	Topic-based baseline	Krisztian Balog	University of Stavanger	[redacted]	[redacted]	✓
10	query expansion based system	Øyvind Jekteberg	University of Stavanger	[redacted]	[redacted]	✓
11	Word2Vec based system	Øyvind Jekteberg	University of Stavanger	[redacted]	[redacted]	✓
12	Rake based topic system	Kristian Gingstad	University of Stavanger	[redacted]	[redacted]	✓
13	TextRank topic system	Kristian Gingstad	University of Stavanger	[redacted]	[redacted]	✓
14	TF-IDF based topic system	Kristian Gingstad	University of Stavanger	[redacted]	[redacted]	✓
15	Saved article author based system	Øyvind Jekteberg	University of Stavanger	[redacted]	[redacted]	✓

Figure 6.3: List of current systems on arXivDigest.

6.1.4 Experimental Recommender Systems

ArXivDigest allows for all our users to create their own experimental recommender systems, but for the purpose of this thesis we created our own experimental recommender systems to evaluate. These systems are detailed in Chapter 4 and Chapter 5. We can see a list of these systems in Figure 6.3. Additional systems will hopefully be added by other researchers in the future.

6.1.5 User Feedback

To evaluate the different experimental recommender systems, we record many different types of interactions that users can perform on the recommendations. This include both implicit and explicit actions. We will describe the different interactions we track in more detail below. This information is used for evaluating systems and for general usage statistics concerning the arXivDigest platform. We also give users the option of giving even more detailed feedback through a feedback form and this type of feedback is described in more detail in Section 3.4.

We track five types of user interaction with article recommendations, these interaction types are listed below. Clicking and saving the recommendation are the interactions we score article recommendations by.

- User has seen the recommendation on email.
- User clicked on the recommendation in email.
- User has seen the recommendation on web.
- User clicked on the recommendation in web.
- User saved the recommendation.

We only track three types of interaction for topic recommendations, however one of these interaction types is further subdivided based on different interaction scenarios. These interaction types are listed below. At the moment, only topic recommendations accepted by a user are used for evaluation as this is the only positive interaction relevant for a system. By extending the performance metric to handle negative scores it could be possible to give negative weights to actions like rejecting topics in the future.

- User clicked on the recommendation in the suggested topic list.
- User has seen the recommendation in the suggested topic list.
- User interacted with the recommendation, we differentiate this interaction by several different scenarios:
  - Topic was recommended by a system but was rejected by the user.
  - Topic was recommended by a system an was accepted by the user.
  - Topic was added to the users profile manually by the user.
  - Topic was added to the users profile manually by the user but then removed again.
  - Topic was recommended by a system but the user refreshed the suggested topic list without interacting with it.
  - Topic was recommended by a system but the user did not interact with it within 24 hours.

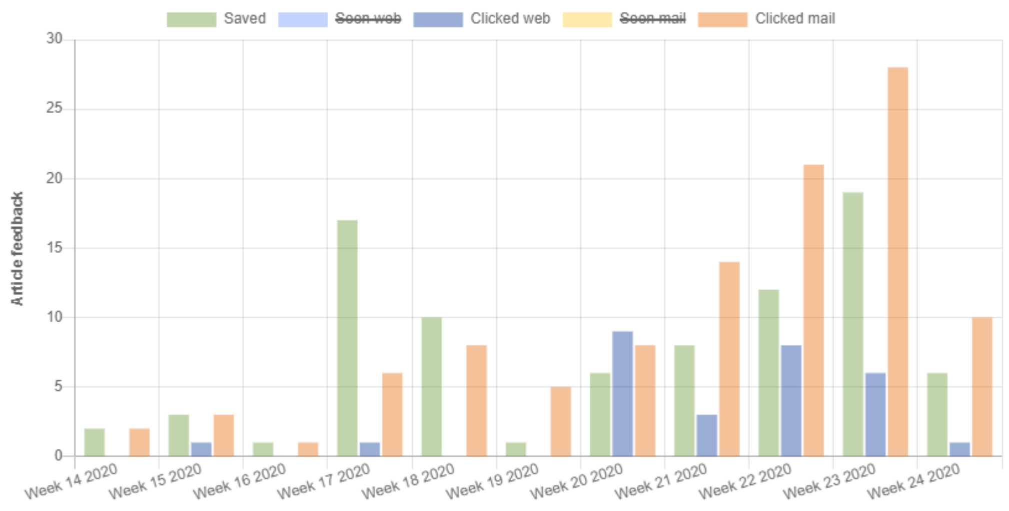


Figure 6.4: Total feedback for article recommendations.

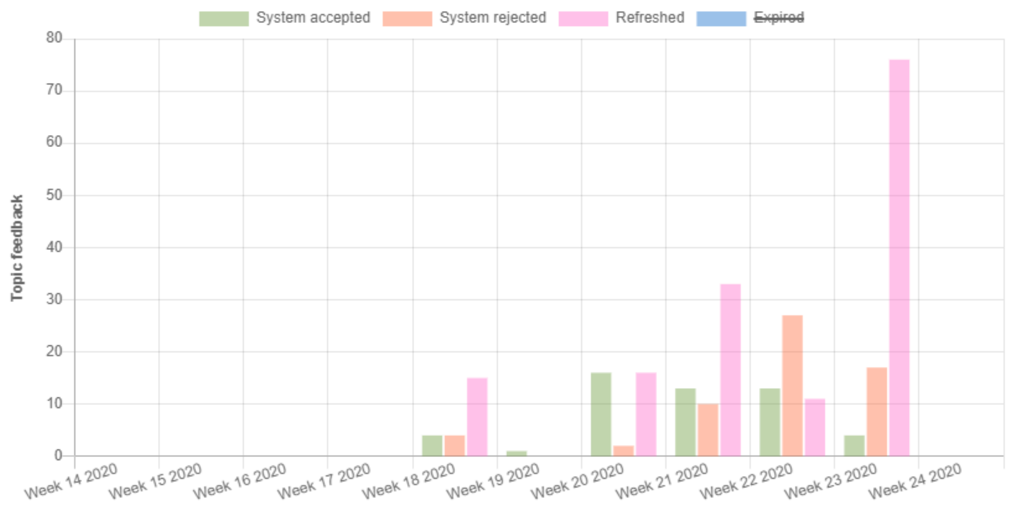


Figure 6.5: Total feedback for topic recommendations.

6.2 Results

On June 10, after the arXivDigest platform and all our experimental recommender systems had been up and running for over one and a half month, we fetched the results and feedback that had been created up to this point. At this point in time the arXivDigest platform had 37 registered users and 288 707 stored articles.

The users has been signing up throughout our testing period and the distribution of new users over time can be seen in Figure 6.1. Note the spike on May the 13. when we advertised the application on Twitter.

The articles has been collected each weekday from arXiv [1]. This can be observed in Figure 6.2 where we see that we collect no new articles on Saturdays and Sundays.

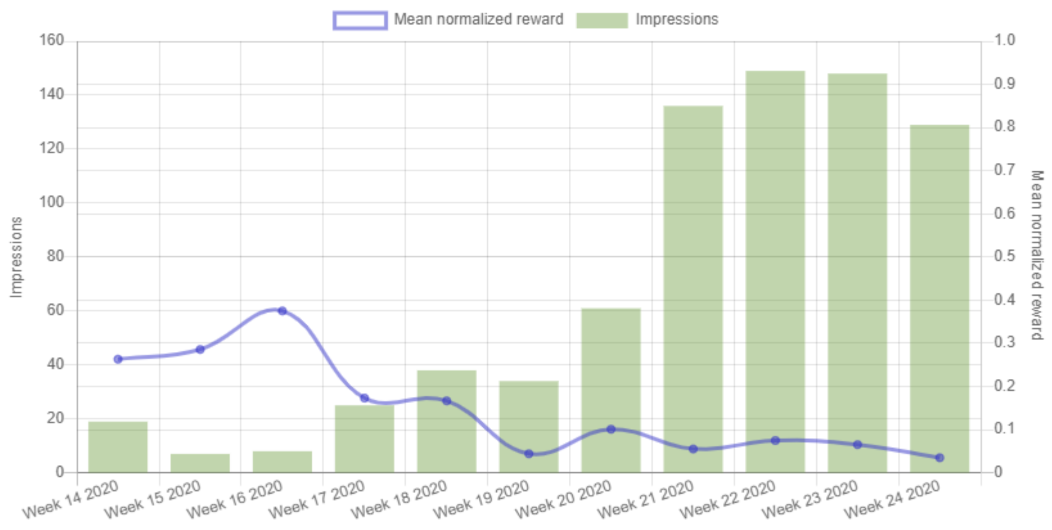


Figure 6.6: Evaluation results for the baseline system.

In Figure 6.4 we see the the number of articles that were clicked on the website, clicked in the email and saved for this period. Although we can clearly see that there has been some activity on the platform, it is much less than we had hoped for. With the current amount of gathered feedback it is highly doubtful we will be able to make any meaningful comparisons of the systems. Because of this, we will not draw any conclusions on system performance by comparing the mean normalized reward of systems. As with this little feedback, any difference may be caused by the high variance of the feedback data. In Figure 6.5 we see the accepted, rejected and refreshed topic recommendations for the same period. Here we have much the same problem as with article recommendations, as topics had a bit less interaction than article recommendations.

6.2.1 Article Recommendations

For the article recommender systems, we have one plot for evaluation of their performance and one plot for the user feedback on the article recommendations. As showing all these plots in this chapter would take too much space, we decided to only show the evaluation plot. Look in Appendix A for individual feedback counts for all systems. This is because with the limited feedback we received, there was too much variation in the data to say anything meaningful. The one feedback plot that stands out is the author based system, which has much less feedback than the others. But the author system also has much fewer impressions, that also contributes to the system having less feedback.

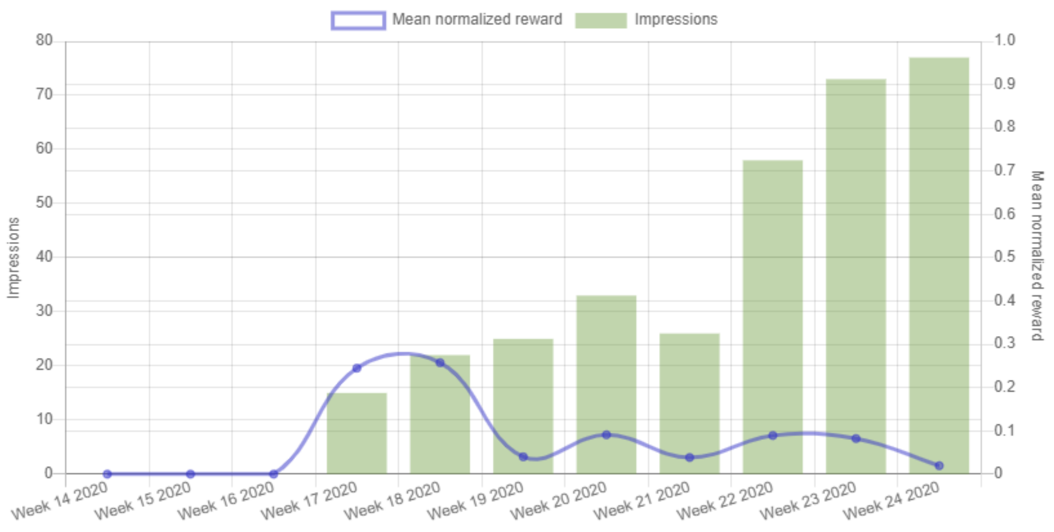


Figure 6.7: Evaluation results for the query expansion based system.

Topic-based Baseline System

Figure 6.6 shows the impressions and mean normalized reward for the baseline article recommender system. We can observe the number of impressions gradually increase over time. The reason for this is the increasing number of users, which in turn allows the system to gain more impressions. Confirmation of this can be seen from Figure 6.1. In this figure we can see the number of users sharply increase on the 13. of May. This corresponds to week 20 and the sharp increase in impressions around week 20 and week 21 in Figure 6.6. Increased impressions over time can also be seen in the results from the other article recommender systems later.

Another thing to note here is the high mean normalized reward of the base system in the early weeks. When we later look at the evaluations of the other article recommender systems, we can see that they do not start before week 17. This high mean normalized reward for the base line system in the early weeks is therefore a result of it being the only article recommender system operational at the time.

We also see that the there is quite a bit variation in the mean normalized reward even after week 19, when all article recommender systems became active. This is because, with the low feedback amounts we received, even a few interactions may amount to big changes in the results. This becomes even clearer in some of the later results.

Query Expansion Based System

Figure 6.7 shows the the impressions and mean normalized reward for the query expansion based article recommender system. We note the same trend here as we did for the baseline

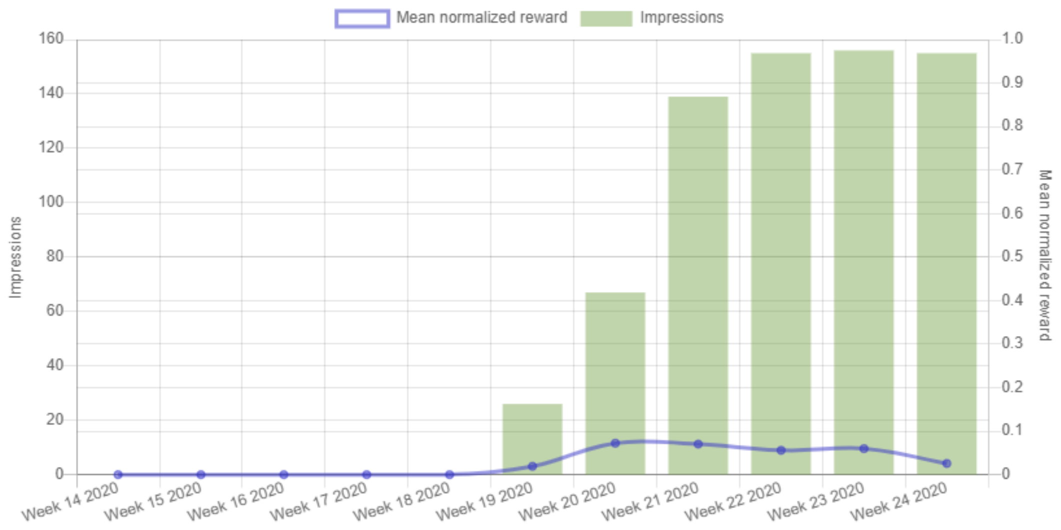


Figure 6.8: Evaluation results for the word2vec based system.

system, with rising impressions over time. However, the sharp increase in impressions between week 20 and 21 that we observed in the baseline system does not happen here. The reason for this becomes clear once we understand how the query expansion system works. This system can not provide recommendations unless the user has some previously saved articles on arXivDigest. The reason for this being that the system needs some saved articles to do query expansion on. The sharp increase in impressions is therefore moved one week to between week 21 and week 22 as the new users have had some time to save some articles. We can also see what we stated above, that this system does not start operating before week 17. There is a higher bump in the mean normalized reward for first two weeks and the reason for this is that there were now only two systems operational. In week 18, when the mean normalized reward decreased again, another system were initialized which we will look at next.

Word2Vec Based System

Figure 6.8 shows the the impressions and mean normalized reward for the word2vec based article recommender system. This system did not start operating before week 18 which is when the mean normalized reward for the base system and query expansion system decreased to share the reward with this new system as well.

Saved Article Author Based System

Figure 6.9 shows the the impressions and mean normalized reward for the saved article author based article recommender system. This system also did not start operating

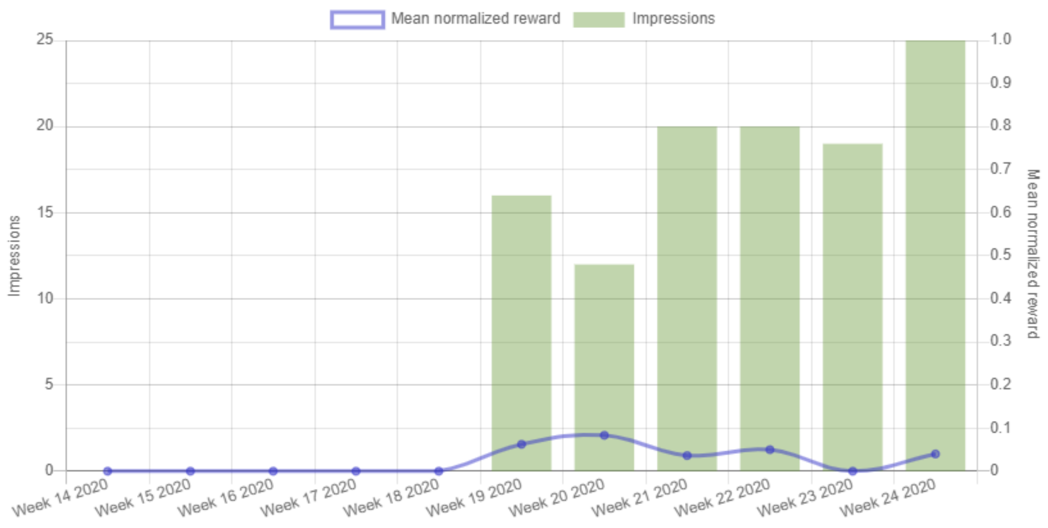


Figure 6.9: Evaluation results for the saved article author based system.

before week 18. Note the much lower number of impressions this system has compared to the others. This comes from the fact that this system recommends articles to users from the same authors they have previously saved. However, it is not often that an author publishes several articles right after each other. This system might therefore often not be able to recommend articles for every user.

Summary of Article Recommender Systems Results

In Table 6.1 we see the number of impressions and mean normalized rewards totals for week 18-23. We can see two things of interest here, with the the first one being that the query expansion based system and the author based system have far fewer impressions than the other two systems. This is actually a problem for the results as it is currently impossible that all the multileavings have three systems in them, which will affect the quality of the result. This was an oversight during the system development phase which we could easily have fixed. One way we could have fixed the problem was by just adding one extra system that always gives recommendations or making it so that at least one of these systems always give recommendations to each user. The other thing we notice is that the mean normalised reward is very low. We would expect a mean normalized reward of 0.33 if every system got the same amount of interaction and all interleavings were interacted with. Getting smaller numbers is not a problem in itself, however in our scenario it is caused by the fact that we got too little feedback, which is a problem.

System	Impressions	Mean Normalized Reward
Topic-based Baseline System	566	0.075
Query Expansion Based System	237	0.092
Word2Vec Based System	543	0.061
Saved Article Author Based System	87	0.043

Table 6.1: Aggregated article recommender system results for week 18-23.

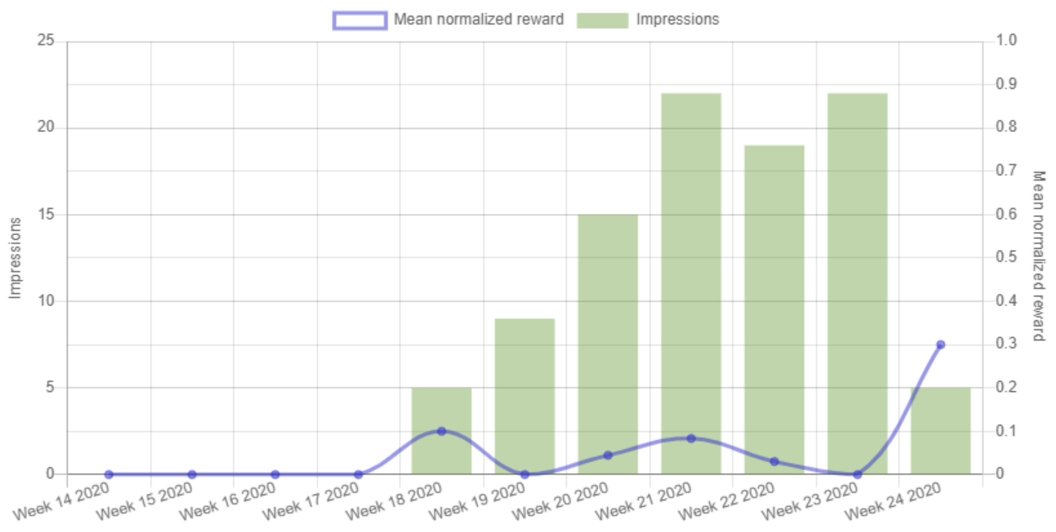


Figure 6.10: Evaluation results for the RAKE based topic system.

6.2.2 Topic Recommendations

For the topic recommender systems, we also have one plot for evaluation of their performance and one plot for the user feedback on the topic recommendations. We decided to only show the evaluation plots for these systems for the same reason as with the article recommender systems, which was because we had limited feedback with much variation. For individual feedback counts on each system look in Appendix A. However, we include a plot with the feedback from all topic recommendations in Figure 6.5.

Rake Based Topic System

In Figure 6.10 we can see the impressions and mean normalized reward for the RAKE based topic recommender system. From this figure we can observe that users are using the site and generating topic recommendations because the topic recommendations are generated on demand when users visit the website. We can also note the same increasing impressions trend as with the article recommender systems.

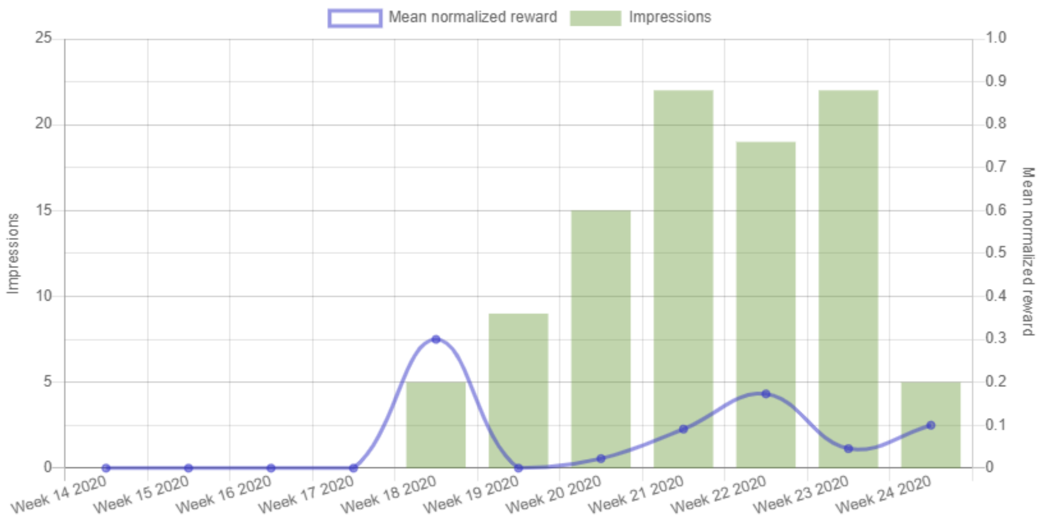


Figure 6.11: Evaluation results for the TextRank based topic system.

TextRank Based Topic System

In Figure 6.11 we can see the impressions and mean normalized reward for the TextRank based topic recommender system. We can observe that the impressions looks a lot like the impressions for RAKE in Figure 6.10. This is because the impressions of the systems are fairly equal thanks to the interleaving process trying to make the different recommender systems contribute evenly. The interleaving process multileaves recommendations from three systems at once and all the topic recommender systems we have created recommends topics to every user each day. Every system should therefore be present in every interleaving, leading to about the same amount of impressions for all the systems.

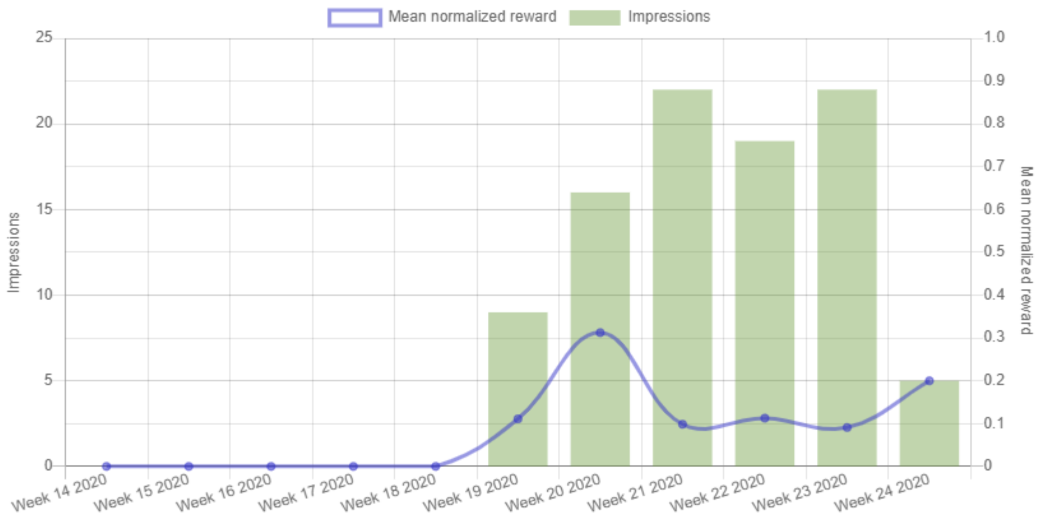


Figure 6.12: Evaluation results for the TF-IDF based topic system.

TF-IDF Based Topic System

The the impressions and mean normalized reward for the TF-IDF based topic recommender system can be seen in Figure 6.12. We can see that this starts working one week after the RAKE and TextRank systems.

Summary of Topic Recommender Systems Results

In Table 6.2 we see the number of impressions and mean normalized rewards totals for week 18-23. In this table we see that most interleavings actually have enough systems in contrast to the article recommendation interleavings. The mean normalised rewards are also low for the topic recommendations. This is caused by the same reason as with article recommendations which was too little user feedback.

System	**Impressions**	**Mean Normalized Reward**
Rake Based Topic System	92	0.039
TextRank Based Topic System	92	0.088
TF-IDF Based Topic System	88	0.140

Table 6.2: Aggregated article recommender system results for week 18-23.

Chapter 7

Conclusion

We had three main objectives when starting our work on this thesis. The first objective was to update and maintain the arXivDigest platform so that we could use it for online evaluation of experimental recommender systems. The second objective we had was to explore some different methods information retrieval techniques, that we could then implement as experimental scientific literature recommender systems. Our last objective was to explore some techniques for topic extraction and similarity matching, that we could also then implement as experimental topic recommender systems.

7.1 Infrastructure Development

For the development of the arXivDigest infrastructure, we have completed all of the objectives we set in Section 1.2.1. The web page and the API has both been extended to support the new features and it is live at `https://arxivdigest.org/`. We have collected a small number of users over a period of about one and a half month, and the platform seems to be working as intended. Basic tasks like signing up, editing ones user profile, creating systems, etc. seems to be working and stable. Articles and topics are interleaved and shown to users as intended both through the web page and emails, and user interactions with the shown recommendations are tracked. We are also able to see usage and evaluation statistics both through the admin interface and system owner interface.

7.2 Article Recommendation

We created four article recommender systems that have all been up and running alongside the arXivDigest platform for about one and a half month. The systems provides users with recommendations each weekday and as we can tell from Section 6.2.1, there has been at least some user interaction with the article recommendations from each system. We feel like we succeeded in exploring several different solutions for recommending articles, where all seem to produce at least somewhat relevant results except for maybe the author based system. However, was expected from the start to perform the weakest because of its simplicity. Unfortunately, due to the nature of online evaluation, we were unable to do any more interesting comparisons between the systems because of too little user interaction.

7.3 Topic Extraction

We looked into two methods for topic extraction and one method for topic similarity matching, which were all implemented as topic recommender systems. The topic recommender systems have all been running for about the same period as the article recommender systems, providing recommendations of topics every weekday. We can see in Section 6.2.2 that there has been some user interaction with the topic recommendations and that the topic recommender systems provide new topics every day. In the end we feel we explored some different and interesting methods for topic recommendations. Unfortunately, we have the same problem with user interaction towards topic recommendations as towards article recommendations, which means we are also unable to draw any meaningful conclusions about the performance of these systems as well.

7.4 Inaccuracies and Improvements

The original plan for this project was to have two testing periods after we finished the infrastructure development parts of the project. First a month long test period where we could test the first implementation of our recommendation algorithms and collect feedback on them. Then we could make changes to these recommender systems accordingly before another month test period. After this we could then collect the final feedback and compare this with the earlier feedback to check if we had improvements. However this did not go as planned. There were a lot more to do on the infrastructure development part than we initially thought, so the arXivDigest application was not ready for the first test period we had planned. Therefore, we had to cancel this first test period and we were left with only a month long test period at the end of the project instead. This lead to fewer users than we anticipated and a shorter overall test period to collect feedback on our recommendations. We could also not make changes to our recommendation systems based on the feedback and results since this was towards the end of the project. The results and the recommender systems would positively benefit from a longer test period, and this is something to consider for the future.

Our results might also be affected by the small number of users that we got, and the users interacted with recommendations less frequently than expected. This meant much less interaction data than anticipated, which again makes the results susceptible to user variations in the feedback we get. Variation might come from irrelevant user behaviour that occur from very active users or users that might try to skew the results on purpose. The way we could improve this is by collecting more data over a larger period of time or by getting more users on our application.

Another thing that affects the results negatively is that the multileaving is dependent on having enough systems to interleave to create balanced and unbiased multileavings. Having few systems, and systems that might not always be able to give recommendations, may lead to multileavings containing too few systems. This again leads to a higher expected reward for that multileaving. It therefore makes it so you cannot directly compare the performance metric between these systems. Having enough systems providing recommendations should fix this problem and this is something that we hope will be improved in the future by researchers adding more systems to the application.

7.5 Future Directions

We believe the arXivDigest platform has a lot of potential. One part of realizing this potential is just to keep the application up and running to collect more users and scientific literature. Having the application up and running also brings the possibility of adding recommender systems from external research groups which could further improve the user experience by giving even better recommendations. We acknowledge that for realizing this potential, there are a number of improvements that should be made to improve user retention rate and generally to just make the platform better to use both for users and experimental system developers.

Even though we made some changes to the design of the website to make it look a bit better, this is something that can be further improved upon. The design of the web pages are still a bit dull and grey, so creating a more visually pleasing design for the website is something that should be considered.

We also improved the evaluation of the experimental recommender systems by adding evaluation plots to the web pages. These plots contain the basic statistics and evaluation results of the experimental recommender systems, but nothing more. However, as we have discussed, the arXivDigest application stores many different types of feedback. There are probably possibilities for other more advanced statistical analysis to be performed on this feedback data for more detailed evaluation of the experimental recommender systems. This is something that could be explored further in the future when the arXivDigest application has had the chance to collect even more user feedback.

Due to the modular nature of the platform it would also be possible to test out different methods of interleaving, multileaving or A/B testing in the future, to check if this provide better rankings than the multileaving we chose. As we store the users' interaction data, it is also possible to recalculate the performance scores for systems while experimenting with different performance metrics to find out which works best for the platform.

Lastly, a thing that is missing from arXivDigest, is a page where admins can view explicit user created feedback from the feedback forms. At the current time we need to access the database to read the explicit user feedback that has been submitted. In the future, a page showing this feedback should be created for easier access.

Appendix A

Additional Plots and Figures

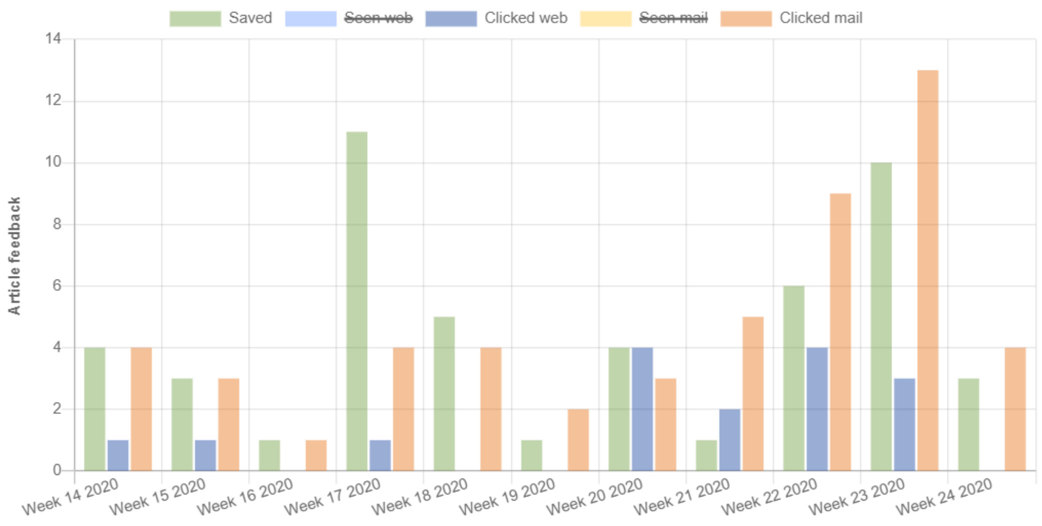


Figure A.1: Feedback for the baseline system.

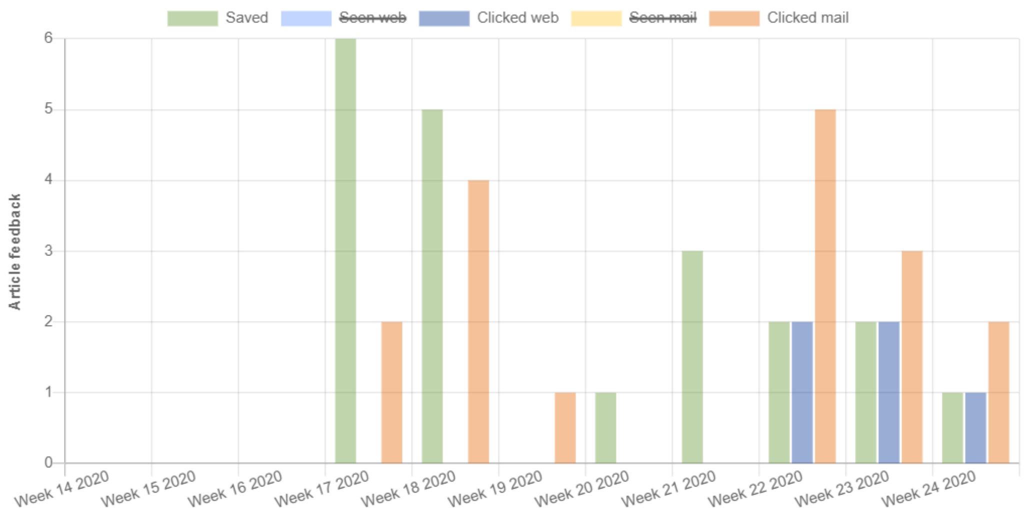


Figure A.2: Feedback for the query expansion based system.

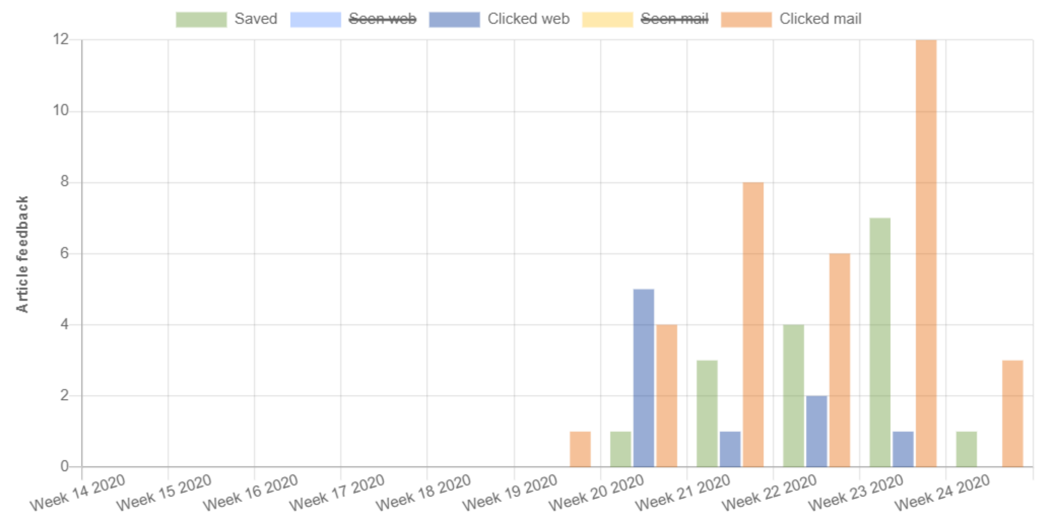


Figure A.3: Feedback for the word2vec based system.

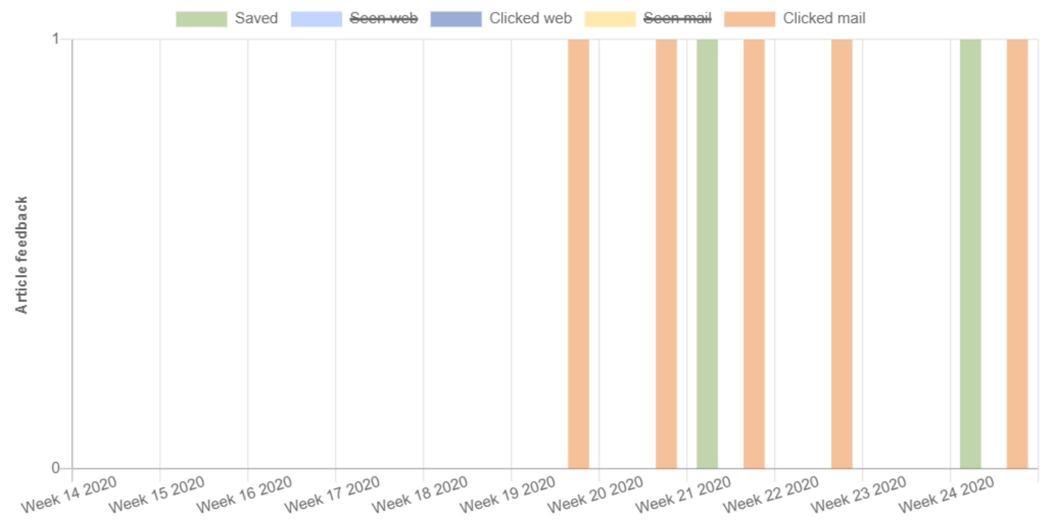


Figure A.4: Feedback for the saved article author based system.

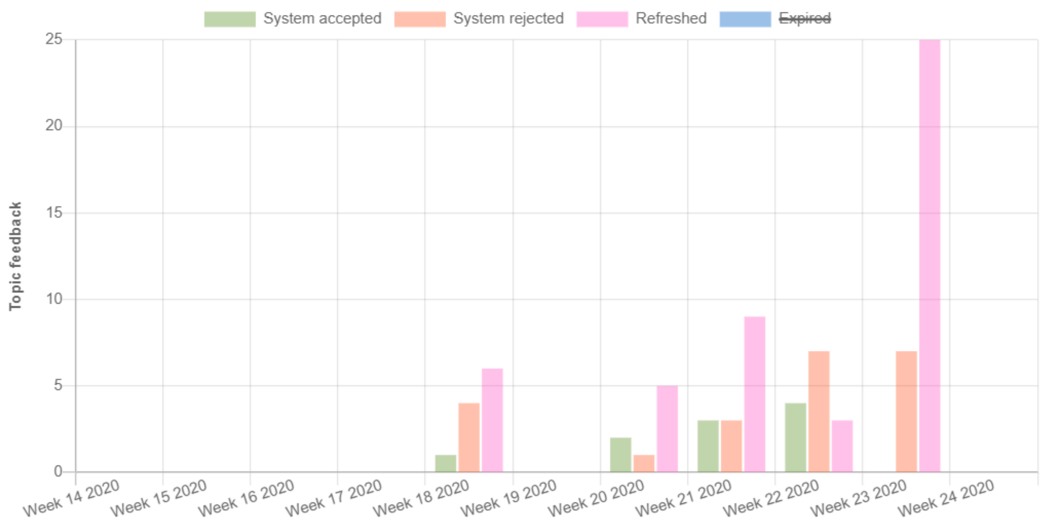


Figure A.5: Feedback for the RAKE based topic system.

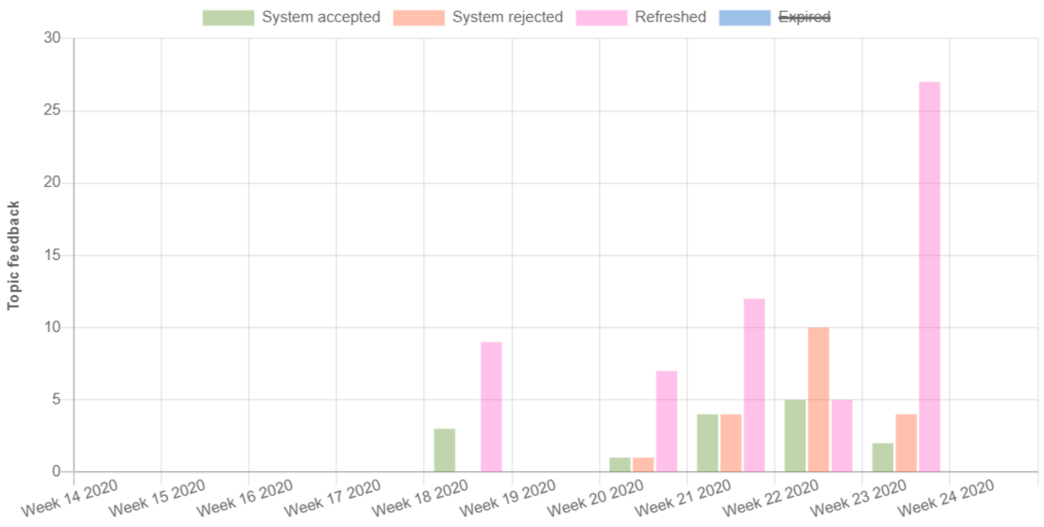


Figure A.6: Feedback for the rake based TextRank system.

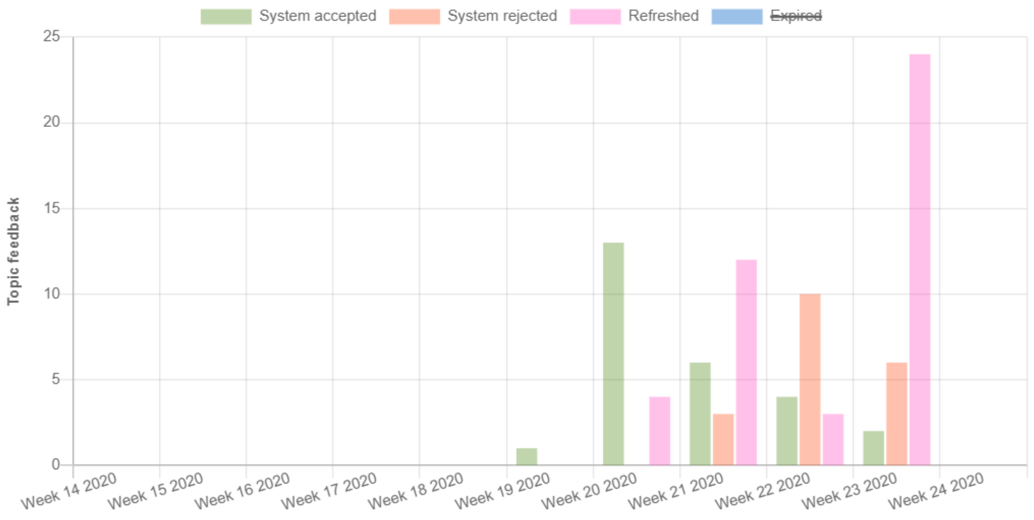


Figure A.7: Feedback for the TF-IDF based topic system.

Appendix B

Attachments

This appendix contains all the code created during this thesis.

- The implementation of the arXivDigest platform.
  - Embedded: arxivdigest.7z
  - Repository: https://github.com/iai-group/arXivDigest
- The implementation of the recommender systems:
  - Embedded: recommender_systems.7z

Bibliography

[1] Cornell University. arxiv, 2020. URL https://arxiv.org/.

[2] The College of Information Sciences and Technology. Citeseerx. URL https://citeseerx.ist.psu.edu/.

[3] AI2. Semantic scholar. URL Semanticscholar.orgl.

[4] Chinese National High tech R&D Program. Arnetminer. URL http://www.arnetminer.org/.

[5] Andrej Karpathy. Arxiv sanity preserver. URL http://www.arxiv-sanity.com/.

[6] Frank Hopfgartner, Krisztian Balog, Andreas Lommatzsch, Liadh Kelly, Benjamin Kille, Anne Schuth, and Martha Larson. Continuous evaluation of large-scale information access systems: A case for living labs. In Nicola Ferro and Carol Peters, editors, *Information Retrieval Evaluation in a Changing World - Lessons Learned from 20 Years of CLEF*, volume 41 of *The Information Retrieval Series*, pages 511–543. Springer, 2019. doi: 10.1007/978-3-030-22948-1_21. URL https://doi.org/10.1007/978-3-030-22948-1_21.

[7] Yongfeng Zhang and Xu Chen. Explainable recommendation: A survey and new perspectives. *ArXiv*, abs/1804.11192, 2020. URL https://arxiv.org/ftp/arxiv/papers/1804/1804.11192.pdf.

[8] Mohammed Taie. Explanations in recommender systems overview and research approaches. *International Arab Journal of Information Technology*, 09 2013. URL https://www.researchgate.net/publication/261062328_Explanations_in_Recommender_Systems_Overview_and_Research_Approaches.

[9] Øyvind Jekteberg and Kristian Gingstad. Arxivdigest: An online evaluation platform for personalized scientific literature recommendation. 2018.

[10] C.X. Zhai and S. Massung. *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining.* ACM Books. Association for Computing Machinery and Morgan & Claypool Publishers, 2016. ISBN 9781970001174. URL https://books.google.no/books?id=WoKkDAAAQBAJ.

[11] Hinrich Schütze Christopher D. Manning, Prabhakar Raghavan. *Introduction to Information Retrieval.* 2008. URL http://nlp.stanford.edu/IR-book/. ISBN: 0521865719.

[12] Gowri Shanmugam and Anandha Mala G S. Text preprocessing for the improvement of information retrieval in digital textual analysis. 01 2014. URL https://www.researchgate.net/publication/306538095_Text_Preprocessing_for_the_improvement_of_Information_Retrieval_in_Digital_Textual_Analysis.

[13] Shahzad Qaiser and Ramsha Ali. Text mining: Use of tf-idf to examine the relevance of words to documents. *International Journal of Computer Applications*, 181, 07 2018. doi: 10.5120/ijca2018917395.

[14] Marcel Caracilo. Machine learning with python: Meeting tf-idf for text mining. 2011. URL http://aimotion.blogspot.com/2011/12/machine-learning-with-python-meeting-tf.html.

[15] Hamed Zamani, Mostafa Dehghani, W. Bruce Croft, Erik Learned-Miller, and Jaap Kamps. From neural re-ranking to neural ranking: Learning a sparse representation for inverted indexing. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, page 497–506, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360142. doi: 10.1145/3269206.3271800. URL https://doi.org/10.1145/3269206.3271800.

[16] Yuanhua Lv and ChengXiang Zha. Adaptive term frequency normalization for bm25. URL http://sifaka.cs.uiuc.edu/~ylv2/pub/cikm11-adptTF.pdf.

[17] Elastic.co. The bm25 algorithm. 2020. URL https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables.

[18] Daniel Valcarce. Information retrieval models for recommneder systems. 2019. URL https://www.dc.fi.udc.es/~dvalcarce/thesis.pdf.

[19] Chandra Bhagavatula Iz Beltagy Miles Crawford Doug Downey † Jason Dunkelberger Ahmed Elgohary Sergey Feldman Vu Ha Rodney Kinney Sebastian Kohlmeier Kyle Lo Tyler Murray Hsu-Han Ooi Matthew Peters Joanna Power Sam Skjonsberg Lucy Lu Wang Chris Wilhelm Zheng Yuan † Madeleine van Zuylen Waleed Ammar, Dirk Groeneveld and Oren Etzioni. Construction of the literature graph in semantic scholar. 2018. URL https://arxiv.org/pdf/1805.02262.pdf.

[20] Nick Cramer Stuart Rose, Dave Engel and Wendy Cowley. Automatic keyword extraction from individual documents. 2010. URL https://www.researchgate.net/publication/227988510_Automatic_Keyword_Extraction_from_Individual_Documents.

[21] Y. MATSUO and M. Ishizuka. Keyword extraction from a single document using word co-occurrence statistical information. 2003. URL http://www.miv.t.u-tokyo.ac.jp/papers/matsuoIJAIT04.pdf.

[22] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into texts. 2004. URL https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf.

[23] Anett Hoppe Sören Auer Arthur Brack, Jennifer D'Souza and Ralph Ewerth. Domain-independent extraction of scientific concepts from research articles. 2019. URL https://link.springer.com/content/pdf/10.1007/978-3-030-45439-5_17.pdf?

[24] Anne Schuth. Search engines that learn from their users. 2016. URL http://anneschuth.nl/wp-content/uploads/thesis_anne-schuth_search-engines-that-learn-from-their-users.pdf.

[25] Bela Gipp Marcel Genzmehr Joeran Beel, Stefan Langer and Andreas Nürnberger. A comparative analysis of offline and online evaluations and discussion of research paper recommender system evaluation. 2013. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1031.973&rep=rep1&type=pdf.

[26] Katja Hofmann, Lihong Li, and Filip Radlinski. Online evaluation for information retrieval. *Foundations and Trends® in Information Retrieval*, 10(1):1–117, 2016. ISSN 1554-0669. doi: 10.1561/1500000051. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/ftir-online-evaluation-final-journal.pdf.

[27] Rolf Jagerman, Krisztian Balog, and Maarten De Rijke. Opensearch: Lessons learned from an online evaluation campaign. *J. Data and Information Quality*, 10(3):13:1–13:15, September 2018. ISSN 1936-1955. doi: 10.1145/3239575. URL http://doi.acm.org/10.1145/3239575.

[28] Frank Hopfgartner, Allan Hanbury, Henning Müller, Ivan Eggel, Krisztian Balog, Torben Brodt, Gordon V. Cormack, Jimmy Lin, Jayashree Kalpathy-Cramer, Noriko Kando, Makoto P. Kato, Anastasia Krithara, Tim Gollub, Martin Potthast, Evelyne Viegas, and Simon Mercer. Evaluation-as-a-service for the computational sciences: Overview and outlook. *J. Data and Information Quality*, 10(4):15:1–15:32, October 2018. ISSN 1936-1955. doi: 10.1145/3239570. URL http://doi.acm.org/10.1145/3239570.

[29] Table of keywords. URL https://www-kaken.jsps.go.jp/kaken1/keywordListEn.do#0010.

[30] Kenneth Reitz. Requests: Http for humans, 2020. URL https://requests.readthedocs.io/en/master/.

[31] Leonard Richardson. Beautiful soup documentation, 2020. URL https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[32] Cornell University. arxiv ategories, 2020. URL https://arxiv.org/category_taxonomy.

[33] Elasticsearch. Elasticsearch introduction. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html.

[34] Krisztian Balog, Wouter Weerkamp, and Maarten Rijke. A few examples go a long way constructing query models from elaborate query formulations. pages 371–378, 01 2008. doi: 10.1145/1390334.1390399.

[35] Yonggang Qiu and Hans-Peter Frei. Concept based query expansion. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '93, page 160–169, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897916050. doi: 10.1145/160688.160713. URL https://doi.org/10.1145/160688.160713.

[36] Marwa Naili, Anja Habacha Chaibi, and Henda Hajjami. Comparative study of word embedding methods in topic segmentation. *Procedia Computer Science*, 112:340 – 349, 2017. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2017.08.009. URL http://www.sciencedirect.com/science/article/pii/S1877050917313480.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *ArXiv*, abs/1310.4546, 2013. URL https://arxiv.org/pdf/1310.4546.pdf.

[38] Jiawei Han, Micheline Kamber, and Jian Pei. 2 - getting to know your data. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 39 – 82. Morgan Kaufmann, Boston, third edition edition, 2012. ISBN 978-0-12-381479-1. doi: https://doi.org/10.1016/B978-0-12-381479-1.00002-2. URL https://www.sciencedirect.com/topics/computer-science/cosine-similarity.

[39] Georgios-Ioannis Brokos, Prodromos Malakasiotis, and Ion Androutsopoulos. Using centroids of word embeddings and word mover's distance for biomedical document

retrieval in question answering. In *BioNLP@ACL*, 2016. URL http://www2.aueb.gr/users/ion/docs/BioNLP_2016.pdf.

[40] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[41] Chuan Shi, Zhiqiang Zhang, Ping Luo, Philip S. Yu, Yading Yue, and Bin Wu. Semantic path based personalized recommendation on weighted heterogeneous information networks. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, page 453–462, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337946. doi: 10.1145/2806416.2806528. URL https://doi.org/10.1145/2806416.2806528.

[42] Nested datatype, 2020. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/nested.html.

[43] Jishnu Ray Chowdhury, 2020. URL https://github.com/JRC1995/TextRank-Keyword-Extraction.

[44] Natural language toolkit. URL https://www.nltk.org/.

[45] CodeLingo. Keyword extraction using rake. 2017. URL https://codelingo.wordpress.com/2017/05/26/keyword-extraction-using-rake/.

[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

[47] Xu Liang. Understand textrank for keyword extraction by python. 2019. URL https://towardsdatascience.com/textrank-for-keyword-extraction-by-python-c0bae21bcec0.

[48] Juan Ramos. Using tf-idf to determine word relevance in document queries. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.1424&rep=rep1&type=pdf&fbclid=IwAR3mPHNjqPs1oW0Deh85gBANBosBv0b5tYpUWIibTKo0XYdWvQRq3WCJhPo.