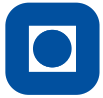
NTNU – Trondheim
Norwegian University of
Science and Technology

Clustering User Behavior in Scientific Collections

Øystein Hoel Blixhavn

Master of Science in Computer Science
Submission date: July 2014
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science



Clustering User Behavior in Scientific Collections

Øystein Blixhavn

Submission date: July 2014

Supervisor: Heri Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This master thesis looks at how clustering techniques can be applied to a collection of scientific documents. Approximately one year of server logs from the CERN Document Server (CDS) are analyzed and preprocessed. Based on the findings of this analysis, and a review of the current state of the art, three different clustering methods are selected for further work: Simple k-Means, Hierarchical Agglomerative Clustering (HAC) and Graph Partitioning. In addition, a custom, agglomerative clustering algorithm is made in an attempt to tackle some of the problems encountered during the experiments with k-Means and HAC. The results from k-Means and HAC are poor, but the graph partitioning method yields some promising results.

The main conclusion of this thesis is that the inherent clusters within the user-record relationship of a scientific collection are nebulous, but existing. Furthermore, the most common clustering algorithms are not suitable for this type of clustering.

Preface

This thesis is the concluding work of the degree of Master of Science in Computer Science for Øystein Blixhavn. It is a part of a larger effort to improve the usability of the open source digital library Invenio, where the main focus is on creating a specialized recommender system using collaborative filtering.

Acknowledgements

First of all, I would like to thank my dear Camilla, for being a huge help these past months. I would also like to thank my advisor Heri, for helping me get started, and showing me the ropes of how to write a thesis. A big thank you to Jean-Yves at CERN, who was very cooperative with providing access logs for the CDS. Lastly, I would like to thank all those who helped with proofreading and other input: Fredrik, Geir, Marius, Svein Thore and Erik. Your inputs were invaluable!

Contents

List of Figures xi

List of Tables xiii

Glossary xv

List of Acronyms xvii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Invenio and CERN Document Server . . . 2
1.3 Digital Libraries . . . 2
1.4 Scope and limitations . . . 3
1.5 Problem definition . . . 3
1.5.1 Objectives . . . 4
1.6 Document structure . . . 4

2 Background and preliminary study 5
2.1 Similarity models . . . 5
2.1.1 Boolean model . . . 5
2.1.2 Vector Space Model . . . 6
2.2 Recommender systems . . . 7
2.2.1 Content-based recommendation . . . 7
2.2.2 Collaborative filtering . . . 9
2.2.3 Explicit feedback . . . 10
2.2.4 Implicit feedback . . . 10
2.2.5 More Like This (MLT) . . . 10
2.3 Clustering techniques . . . 11
2.3.1 K-means clustering . . . 11
2.3.2 Hierarchical Agglomerative Clustering . . . 12
2.4 Dimensionality reduction . . . 15
2.5 Graph partitioning . . . 15
2.5.1 Multi-level graph partitioning . . . 15

3 State of the art **19**
3.1 Co-clustering 19
3.2 User behavior logging 20
3.3 Citation based ranking 21
3.4 Graph based collaborative filtering 22
3.5 Reviewing the state of the art 23
3.5.1 Conclusion 24

4 Approach **25**
4.1 Experimental setup 25
4.2 Hardware 26
4.3 Data pre-processing 26
4.4 Clustering algorithms 30
4.4.1 Simple K-means Clustering 31
4.4.2 Hierarchical Agglomerative Clustering 31
4.4.3 Custom clustering algorithm 31
4.4.4 Metis graph partitioning 32
4.5 Experimental procedure 36
4.5.1 Simple K-means 36
4.5.2 Hierarchical Agglomerative Clustering 37
4.5.3 Custom clustering algorithm 37
4.5.4 Graph partitioning 38
4.5.5 Evaluating clusters 40
4.6 Experimental results 40

5 Discussion **45**
5.1 Simple K-means 45
5.2 Hierarchical Agglomerative Clustering 45
5.3 Custom Clustering Algorithm 45
5.4 Graph partitioning 46
5.5 General observations and discussion 47

6 Conclusion **49**
6.1 Future work 50

References **53**

Appendices

A Source code **57**
A.1 Custom clustering algorithm 57
A.2 Scripts 64
A.2.1 Bot removal 64

A.2.2 CSV converter 66
A.2.3 Extract IP → records 68
A.2.4 Extract Record → IPs 69
A.2.5 Extract IPs with over 500 downloads 70
A.2.6 Plot histogram of record accesses 71
A.2.7 IP pruning 72
A.2.8 Record pruning 74
A.2.9 Remove duplicates 75
A.2.10 Convert to relevance matrix 77
A.2.11 Generate Metis graph 79
A.2.12 Extract score from graph partitions 82
A.2.13 Map records to partitions 83
A.2.14 Plot edge cut values 85

B Results **87**
B.1 Output from k-Means, $\frac{1}{100}$ of full matrix (k=2) 87
B.2 Output from custom developed algorithm 89
B.3 Output from Metis, $N = 25$ 96

List of Figures

2.1 RSS-graph for use in K-means cluster evaluation 13
2.2 Example of dendrogram 14
2.3 Graph coarsening 16
2.4 Multi-level graph bisection 17

4.1 Pie chart of log composition 27
4.2 Bar chart of different types of user activity 28
4.3 Records accessed per user 29
4.4 Number of IPs accessing each record 30
4.5 Example of 2-way partitioning 39
4.6 3-way partitioning of the same graph 39
4.7 4-way partitioning of the same graph 40
4.8 Dendrogram of the hierarchical clustering with Euclidean distance . . . 41
4.9 Dendrogram of the hierarchical clustering with Jaccard similarity 42
4.10 Edge cut for 2-300 partitions 42
4.11 Edge cut, magnified 43

List of Tables

2.1 Index 8
2.2 Inverted index 8

3.1 Example of relevance matrix 20
3.2 Example of co-clustered relevance matrix 20

Glossary

Clique	A term from graph theory which denotes a fully connected subgraph (i.e. every vertex in the subgraph is connected to all the others).
Degree	A term used in graph theory as the connectivity of a vertex. A vertex connected to 4 other vertices has a degree of 4.
Dendrogram	A graph showing the hierarchical nature of a tree. Used in hierarchical clustering to display the similarity values of the clustering.
Edge cut	A term in graph partitioning. Denotes the number of edges "cut off" by the partitioning, i.e. all edges between partitions.
Hypergraph	A graph in which an edge can connect several vertices.
Invenio	Digital Library system developed at CERN..
IP address	Internet Protocol address. Often abbreviated IP.
Metis	A graph partitioning tool, developed by Karypis, George and Kumar, Vipin at the University of Minnesota, USA.

Recommendation system	A recommendation system is an automatic search that uses information such as user behaviour and personal information to predict interesting elements.
Serendipity	A term used within recommender systems theory. Describes the concept of a user finding interesting items which are not necessarily popular.
Stemming	The process of reducing a word to it's stem ("Stemming" $\Rightarrow$ "Stem").
urllib	Web scraping library for Python.
Weka	A knowledge analysis tool, developed at the University of Waikato, New Zealand.

List of Acronyms

BM Boolean model.

BPA basic probability assignments.

CDS CERN Document Server.

CERN Conseil Européen pour la Recherche Nucléaire.

DL Digital Library.

GAAC Group-Average Agglomerative Clustering.

GGGP Greedy Graph Growing Partitioning algorithm.

GGP Graph Growing Partitioning algorithm.

HAC Hierarchical Agglomerative Clustering.

HCM Heavy Clique Matching.

HEM Heavy-Edge Matching.

HEP High Energy Physics.

IDI Department of Computer and Information Science.

IP Internet Protocol.

IR Information Retrieval.

LEM Light Edge Matching.

LSI Latent Semantic Indexing.

MAE Mean Absolute Error.

MLT More Like This.

NAT Network Address Translation.

NTNU Norwegian University of Science and Technology.

PDF Portable Document Format.

RAM Random Access Memory.

RM Random Matching.

ROCK Robust Clustering Algorithm for Categorical Attributes.

RSS Residual Sum of Squares.

SB Spectral Bisection.

SVD Single Value Decomposition.

TF-IDF Term Frequency - Inverse Document Frequency.

VSM Vector Space Model.

Chapter 1
Introduction

1.1 Motivation

The motivation for this thesis starts out in the current situation of open source systems within search and cataloging. Even though there are several popular systems in this area, there are none that has prioritized implementing a recommender system that utilizes the extensive research done in the field. Recommender systems are common in commercial systems, such as Amazon, eBay and Netflix, as they are valuable for improving the usability of the system. By the use of several different techniques – some derived from Information Retrieval (IR) techniques, and some specifically developed for recommender systems – recommendations can be made with background in previous interests and behavior.

This master thesis subsequently focuses on taking the next step towards creating a recommender system for Invenio. Collaborative filtering is an area of research where the algorithms and approach *can be* highly specific, possibly yielding superior results over a generic, one-size-fits-all algorithm. By analyzing the access logs of Invenio's biggest user – the CERN Document Server (CDS) – the hope is to show that there exists a solid basis for creating clusters that can be used in a collaborative filtering algorithm.

A user study among the physicists in the High Energy Physics (HEP) field shows that currently, only 6.5 to 10.9% consider collaborative tools and personalization as a very important feature, compared to search accuracy and depth of coverage, which are at 59.5% and 72.7% respectively [1]. However, 15.4% consider "Recommendation of documents of potential interest" a very important feature for the ideal scientific information system. In addition, 45% of the respondents have used HEP search engines for over 10 years, and there is a clear trend among the younger physicists for using Google and Google Scholar, both of which utilize more advanced ranking and recommendation than the current HEP repositories. This suggests that smarter recommender systems are becoming more and more commonly used, also within

scientific circles.

On a more general note, Huang et al. [2] states that "Recommendations comprise a valuable service for users of digital libraries". As such, Invenio needs to make an effort in improving the user experience and increasing the usability, and a recommender system is an important tool in achieving this goal.

1.2 Invenio and CERN Document Server

Invenio is a digital library system developed by CERN[1] to run the CERN Document Server (CDS). It was launched as a stand-alone open source software July 2006, while still being maintained by a core team of developers at CERN [3]. Invenio is a full featured digital library, capable of handling both text documents, audio, video, photos and several other formats. The records are organized in collections, and can be navigated through the use of keywords, categories and other metadata. User registration is also supported, providing access control for restricted records, as well as some personalization options. Since its release, it has been adopted by many large institutions, such as the École polytechnique fédérale de Lausanne (EPFL) and the Deutsches Elektronen-Synchrotron (DESY) [4].

CDS is the main library of CERN, and is the original reason Invenio was developed. With over 1.3 million scientific articles and preprints [5], it is one of the top institutional repositories[6]. It has over 30 thousand visitors each month. CDS contains not only articles but also images, video and other texts. It has approximately 8000 registered users, most of which are researchers of HEP [4].

1.3 Digital Libraries

Although an intuitive term, it is hard to properly define Digital Libraries (DLs). Several approaches to the area exist, and each with their own main focus. A well-covering definition suitable for the context of this thesis states [7, p. 712]:

> Complex information systems that help satisfy information needs of users, provide information services, and organize, present, and communicate information with users in usable ways.

Hence, the key features of a Digital Library (DL) are searching and browsing capabilities, preservation of content (assuring satisfactory, permanent access regardless of content type), and tools for maintaining quality control. In addition, they are

[1] http://home.web.cern.ch/

often specialized within a certain field, such as scientific areas like HEP, medicine and biology, or to maintain collections of data within enterprises or other organizations.

1.4 Scope and limitations

Because of the time constraints for this thesis, the scope is narrowed to collaborative filtering in scientific collections, a very specific application of recommender systems. Despite these limitations, the ultimate goal of this thesis is to improve recommender systems for open source systems in general, and Invenio in particular. This thesis thus serves a sort of first step in this effort, attempting to perform effective clustering on usage logs of the CDS. The limitations of this thesis are explained below.

Only scientific collections considered

The thesis will only consider collections of scientific papers. This decision was made to impose a limitation on the workload, but also because the thesis tries to show how recommender systems can be tailored to specific applications. While this thesis choses to focus on scientific collections in general, it is part of a larger effort to improve the usability of Invenio. There is reason to believe that scientific institutions will be the largest user base of Invenio in the future, which will increase the value of such a specialized tool. In addition, the original idea for this thesis was based on the hypothesis that academic collections have special attributes with regards to usage, which allows for specialized recommendation algorithms. This hypothesis will be further elaborated in Chapter 4.

Only collaborative filtering considered

While there exist numerous directions within the research field of recommender systems, a choice had to be made on the construction of the algorithm. In the preparatory work done for this work, several different recommender techniques were selected for a final, distributed recommendation system. When selecting a path for further work, collaborative filtering in combination with scientific collections stood out as the one method with most potential.

1.5 Problem definition

With basis in the above-mentioned introduction, the problem definition has been condensed in the following main research question:

> *How can clustering methods utilize the document collection and access logs of CDS to generate recommendations?*

1.5.1 Objectives

The objective of this thesis is to investigate the current state of the art in the field of collaborative filtering recommender systems. More specifically, it will focus on methods of analyzing usage data used by recommender systems. The state of the art research will be evaluated in terms of how it can be used for clustering collections of academic papers. In addition, an overview of the CDS usage will be presented, based on statistical analysis of access logs over a 13 month period. Based on this research, clustering methods will be applied to the processed logs, and the results will be evaluated.

The thesis' sub-objectives are formulated into the following three questions:

- **Q1:** How can the usage logs of CDS best be processed to allow for effective analysis of usage patterns?
- **Q2:** How can clustering methods be adapted to identify user groups from automatic analysis of usage logs?
- **Q3:** How can the the resulting clusters of the usage logs be utilized in a way that mitigates information bubbles?

1.6 Document structure

The thesis is organized as follows:

Chapter 2: Background presents the relevant background information and theory, as well as surrounding theory that might improve the understanding of the state of the art.

Chapter 3: State of the Art provides an overview of the current state of the art, with focus on methods in recommender systems theory.

Chapter 4: Approach contains a documentation of the approach, together with intermediate results and reflections made during the process. It starts with an overview of the process leading up to the research, followed by a step by step report on the work.

Chapter 5: Results and discussion presents the results, discussing them against each other and highlighting strengths and weaknesses.

Chapter 6: Conclusion rounds off the thesis by presenting the final results, lessons learned, and the answers to the objectives found in section 1.5.1. In addition, the chapter gives suggestions to further work on the experiments, and looks at possible future applications of the results.

Chapter 2
Background and preliminary study

In this chapter, we will present the basic theoretical concepts mentioned in the thesis. Most of it is later referenced in State of the Art (Chapter 3) and Approach (Chapter 4), and serve as a further explanation of the concepts mentioned there. The most relevant background sections are Clustering techniques (2.3), Graph partitioning (2.5 and Recommender systems (2.2). The other sections mainly serve to give background theory for the introduced concepts previously used within recommender systems and collaborative filtering.

2.1 Similarity models

Similarity models are used in information retrieval systems to find relevant results based on a query. The goal of a similarity model is to present results with high *recall* and high *precision*. This essentially means that the results should include as many of the relevant records as possible (recall), while excluding as many of the irrelevant records as possible (precision). There exist several different similarity models, each utilizing different principles which have different strengths and weaknesses. Those presented in this section are the basis for current recommender system implemented in the CDS.

2.1.1 Boolean model

The boolean model within information retrieval is the most basic similarity model used for retrieval of results in search systems. Systems using the boolean model uses boolean algebra as the query language, and the records are returned if they match the given query [8]. The boolean model has no inherent concept of ranking; the records either match the query, or they do not. As an example, lets look at the query [q = t_1 OR (t_2 AND t_3)]. This will return any document that either contains the term t_1, the terms t_2 and t_3, or all three terms.

While this model is both fast and simple, it has several drawbacks. It does not consider term weighting, has no concept of gradual relevance, and the queries have a tendency to either return too many, or too few results [7]. As an attempt to overcome these limitations, an extended boolean model has been created. Similarly to the Vector Space Model, this model represents documents as term vectors, and is a step towards more advanced similarity models.

2.1.2 Vector Space Model

The vector space model seeks to extend the limitations imposed by the Boolean model, and introduces a way of partial matching. It represents documents as n-dimensional vectors, where n is the number of indexed terms in the IR system. Documents are compared to the query string using cosine similarity, which is a similarity measure that compensates for differences in vector length (i.e. the number of terms present in the document and query).

$$\mathrm{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_1)|} \tag{2.1}$$

where $\vec{V}(d_1)$ and $\vec{V}(d_2)$ are the vector representations of the two documents.

Ranking is often further done by applying the TF-IDF similarity measure, as explained below. Here, *term frequency* represents a weight measuring the occurrence of a term in a given document, while *inverse document frequency* acts as a sort of counterbalance, penalizing terms that are common throughout the collection. There exists several variants of both measures, and two of them are presented below:

$$\mathrm{TF}_{t,d} = 0.5 + \frac{0.5 \times f_{t,d}}{\max\{f(_w, d) : w \in d\}} \tag{2.2}$$

where $f_{t,d}$ is the frequency of term f in document d and $max\{f(_w, d) : w \in d\}$ denotes the frequency of the most common word in the given document.

$$\mathrm{IDF}_t = \log \frac{N}{n_t} \tag{2.3}$$

where N is the number of documents in the collection, and n_t is the document frequency of term t.

The TF-IDF score of a term in a document is then given by

$$\text{TF-IDF}_{t,d} = \mathrm{TF}_{t,d} \times \mathrm{IDF}_t \tag{2.4}$$

2.2 Recommender systems

A recommender system is a type of information retrieval system that gives item suggestions to a user based on previously associated items (bought, downloaded, viewed etc), the explicitly stated preferences by the user, or by utilizing known information about the user, i.e. location, profession, age etc [9, p. 1]. It is often deployed in commercial systems with a large amount of items (e.g. music, news, movies etc), and aims to help the user or customer to find interesting items with little to no effort on their part.

The motivation for recommender systems comes from the observation that humans often rely on peer recommendation for the choices in daily life [9, p. 13]: we eat at the restaurants recommended by friends, we buy brands recommended by friends, and we read books and watch movies based on the reviews we trust. With the growth of Internet, the information available in many areas has increased to a size where it is not possible to make informed choices based solely on browsing. By implementing recommendation in information retrieval systems, we introduce an improved browsing experience that helps reduce the amount of options, which leads to a better usability for the system.

Recommender systems are often specialized for one particular implementation, and employs different methods developed for recommender systems that utilizes the special properties of the problem at hand. For example, the recommender system implemented at Amazon.com uses an item-to-item collaborative filtering. This helps keeping the calculations needed for the recommendations local, and helps with the scaling in a system with massive datasets and sparse user data [10].

There are two main approaches to recommender system [11]: *content-based recommendation* promotes items similar to those items the user have previously bought, viewed, downloaded or otherwise shown interest in; and *collaborative recommendation* analyzes usage patterns and identifies users similar to the given user, and recommends items these users have liked.

2.2.1 Content-based recommendation

As stated in the previous paragraph, content-based recommendation is based on presenting items similar to what the user has previously liked, viewed or bought [9, p. 74]. There are several advantages of content-based recommendation. Most notably, it works well for new items; item similarity can be calculated without any usage statistics. In addition, the recommendations are based solely on the behavior of the current user, which allows for good recommendations in systems with fewer users. On the other hand there are also some noteworthy drawbacks: content-based recommendation is prone to over-specialization, where the metadata and information

used to find recommendation is too specific, producing recommendations with very specific relevance.

Exactly how to calculate the similarity between items is what separates the different methods within content-based recommendation. There are two main approaches to this type of recommendation.

Word similarity

Word similarity is a technique derived from the widely known TF-IDF concept, and uses regular and inverted indices. These indices are created by analyzing each document in the collection. After removing stop words (and, but, while, etc), the remaining terms of the documents are *stemmed* and added to an index (see Table 2.1). Similarity, an entry is made in the inverted index of how many times the term is used (see Table 2.2). An inverted index may also contain the location of where the term is used, to add support for phrase search.
The word similarity uses a simple principle to retrieve relevant results. The n most frequently used terms in the source document are selected from the regular index. The inverted index is then consulted using these terms as index keys, and the resulting documents are then retrieved and ranked (e.g. based on TF-IDF).

Document	**Terms**
d1	search(2), filter(3)
d2	library(1)
d3	filter(2)
d4	filter(1), library(3), search(2)

Table 2.1: Index

Terms	**d1**	**d2**	**d3**	**d4**
filter	3	-	2	1
library	-	1	-	3
search	2	-	-	2

Table 2.2: Inverted index

Metadata utilization

In searchable collections, like eCommerce systems and libraries, metadata can be utilized to create collections of elements. A typical way of employing this would be to recommend books by authors you have read, or items of similar function or theme (like recommending gardening books to a user that has bought gardening tools)[10]. The challenge of this method is to obtain the user profile, i.e. gathering

the user information needed to create useful recommendations. This is most often done through some type of relevance feedback — either gathered explicitly through feedback given by the user (ratings, like/dislike etc) or through implicit feedback. Implicit feedback methods are based on looking at the users consumption history (view/download/buy etc) and other significant user actions, and using this data for profiling algorithms [9, p. 150].

2.2.2 Collaborative filtering

Collaborative filtering is the principle of utilizing the behavior of the users in a system to predict recommendations for specific users. This approach overcomes some of the limitations of content-based recommendation. Most notably, while content-based recommendation relies heavily on being able to identify items based on content, collaborative filtering has no such inherent requirement, which makes it suitable for information systems containing several types of information objects (i.e. books, images and video) [9, p. 111]. On the other hand, collaborative filtering requires a lot of data on user activity to be able to produce good recommendations. Collaborative filtering methods can be classified as either neighbor based or model based [11].

Neighborhood based collaborative filtering

In neighborhood based methods, the recommended items are calculated by identifying users with a similar usage pattern as the current user. A list of recommendation candidates are then selected from the items these users have consumed, omitting any items already known to the current user. This approach is intuitively understandable, and relatively easy to implement.

Model based collaborative filtering

Model based collaborative filtering analyzes how the system is used, and creates models to use as a basis for recommendation, where users are categorized by how they match the models. This approach has shown to yield more precise recommendations than neighborhood based collaborative filtering [9, p. 112]. In addition, the main part of the work is done creating and adjusting the models, a task that can be done asynchronously.

The main drawback of model based collaborative filtering is regarding the nature of the recommendations. Although the recommendations are precise, they lack *serendipity* [9, p. 112]. Serendipity introduces the concept of "lucky encounters", in this case meaning how the recommendation system can present elements that are not necessarily ranked highest, but might still be interesting to the user. More important, it means providing the user with alternatives she might not have discovered otherwise. To illustrate, let us say the system in question is a music streaming service. If our

user listens to the genres *jazz* and *country*. A model based recommender system would identify these distinct interests and recommend music within either, while a neighborhood based recommender system would look at what other similar users have listened to, and most likely recommend something within a similar genre, for example *blues*.

2.2.3 Explicit feedback

Explicit feedback is a method where the user of a system gives input, either by rearranging the results from a query or click on some predefined button that says "this is relevant to my query". The feedback is defined as explicit only when the user know that that the feedback provided is treated as relevance judgements. The user can indicate the relevance either as a binary value, relevant or not relevant, or by using a graded system like relevant, somewhat relevant or very relevant.

The feedback information can be used to expand the original query to improve the performance of the search, or to create user profiles utilized by collaborative filtering.

2.2.4 Implicit feedback

Where explicit feedback collects information from users deliberate actions of feedback, implicit feedback is given by the users implicit actions when using the system. For instance, which documents the user do and do not select for viewing, the duration of time spent viewing a document or interaction like scrolling or browsing actions. The main difference between explicit and implicit feedback is that implicit feedback is given for all users, and is thus a more reliable source of input. The quality of explicit feedback is however often higher [12, p. 172].

2.2.5 More Like This (MLT)

More Like This (MLT) is a type of recommender system where the recommendations are made on basis of a record, providing the user with similar records. While there are several approaches to this functionality, they all have in common that they use some information from the source record to produce search results. A typical implementation of More Like This (MLT) is the "People who bought this also bought" list presented in most eCommerce web sites.

Search-based recommendation

Search-based recommendation utilizes metadata of the given record to produce recommendations. The systems search engine is utilized, providing a cheap implementation

of MLT with often satisfactory results. Search-based recommendation can be implemented using for example word similarity, where the terms of the source document is used to create a query, providing similar documents through word similarity search; or by performing a search on some of the metadata, e.g. author, journal, collection etc.

2.3 Clustering techniques

Clustering is a type of unsupervised classification which groups elements in a data set by similarity. This grouping is done through the use of distance measures, which are mathematical tools that calculate the distance between two elements, or between an element and some coordinate. There are several different distance measures, each tailored to different types of data. Clustering methods can broadly be categorized into flat, and hierarchical clustering. Flat clustering groups the data in separate partitions, while hierarchical clustering produces a tree of nested clusters, where the higher level clusters contains smaller subclusters [9, p. 61].

2.3.1 K-means clustering

Among flat clustering methods, K-means is the most widely used, and is considered the most important algorithm[8]. The algorithm tries to classify all elements into k clusters, by minimizing the average squared Euclidean distance to the closest centroid. This measure effectively calculates the distance between two elements in Euclidean space, by computing an absolute difference between the elements for every dimension:

$$|\vec{x} - \vec{y}| = \sqrt{\sum_{i=1}^{M} (x_i - y_i)^2} \tag{2.5}$$

Initially, the k centroids are placed at random, or according to some rule of separation. Through several iterations, each element is then assigned to a centroid, or cluster, and a new position for each centroid is calculated. The algorithm is complete either after N iterations, at the point where the centroids does not move, or when the algorithm falls below a certain threshold of improvement. As a measure for evaluating the position of centroids, Residual Sum of Squares (RSS) is used. This method calculates the squared distance of each element to its assigned cluster, and the total RSS score is the sum of this distance over all clusters:

$$\text{RSS}_k = \sum_{\vec{x} \in \omega_k} |\vec{x} - \vec{\mu}(\omega_k)|^2 \tag{2.6}$$

$$\text{RSS} = \sum_{k=1}^{K} \text{RSS}_k \tag{2.7}$$

where ω_k is the cluster k, $\vec{\mu}(\omega_k)$ is the centroid of cluster k, and $\vec{x}$ is the elements in cluster k.

As the algorithm requires a pre-specified number of clusters, it can often be difficult to pinpoint the optimal clustering. The solution for this is to select the range in which the optimal number of clusters is suspected to lie, and apply the algorithm for each integer in that range. The total RSS score is then calculated for each resulting clustering, and graphed, showing how the clustering improves as k increases. When graphing these values, a certain "knee" can often be observed, pinpointing the number of clusters where the RSS decrease flattens, i.e. where the increase in clusters loses its effectivity (see Figure 2.1). If several similar knees are present, a selection must be made on basis of other available information.

The complexity of the K-means clustering algorithm is $O(IKNM)$, where I is the fixed number of iterations, K the number of clusters, N the number of elements, and M the dimensionality of the data.

2.3.2 Hierarchical Agglomerative Clustering

HAC is a clustering technique that produces a hierarchy of clusters. The algorithm starts by considering every element as a leaf-level cluster, and proceeds by merging pairs of clusters by comparing the similarity, or distance, between them. In contrast to K-means clustering, HAC does not need to predefine the amount of clusters, and will continue to merge clusters until there is only one left, producing a *dendrogram* of clusters (see Figure 2.2 for example). The clustering can also be stopped when a certain numbers of clusters have been reached, or when the similarity needed for merging clusters is below a certain threshold. A common method of finding the optimal number of clusters is to compare these similarities on a graph. If the similarity requirement for merging suddenly drops, it is likely that the algorithm has passed the number of natural clusters in the data. This can be observed in the dendrogram as a large gap between consecutive merges, see for example at $\textit{similarity} = 0.4$ on the y-axis in Figure 2.2. This is comparable to finding the "knee" when graphing performance of K-means clustering (as mentioned in section 2.3.1).

The similarity of elements can be computed using several different similarity measures. The two most important for the context of this thesis are Euclidean distance and Jaccard distance. The Euclidean distance is explained in section 2.3.1.

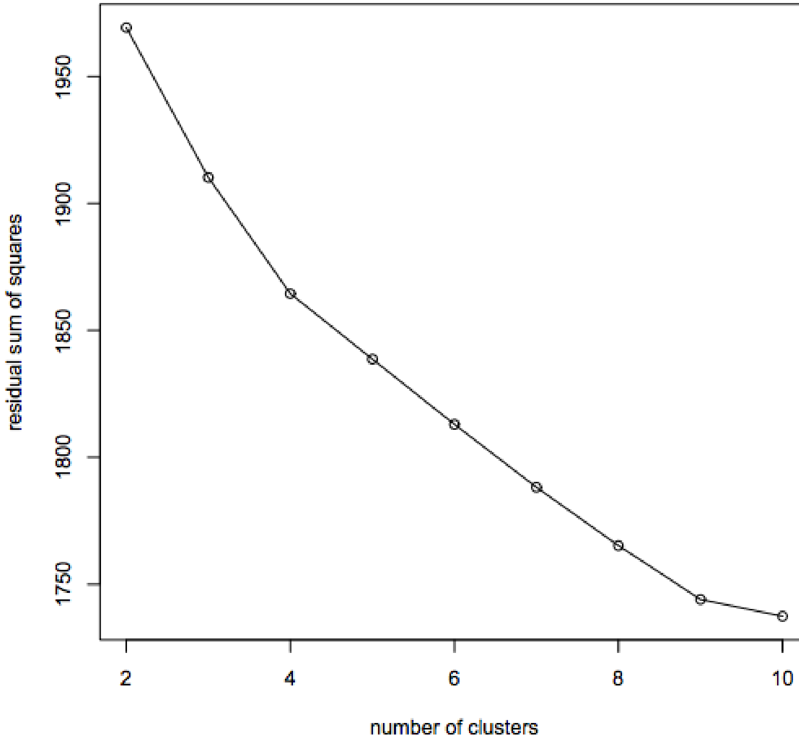


Figure 2.1: Graph of RSS as a function of the numbers of clusters in K-means. A knee can be observed at $k = 4$ and $k = 9$. The collection represented in this graph have four categories, so $k = 4$ should be selected. Figure from [8]

The Jaccard similarity measure was initially developed for comparing biological species, and is designed for sets of binary attributes. It computes the similarity by dividing the intersection by the union of the two elements [12, p. 65]. The Jaccard distance J_δ is simply $1 - J(A, B)$.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.8}$$

$$J_\delta = 1 - J(A, B) \tag{2.9}$$

For comparing clusters, there are three principles of calculating the similarities: single-link clustering, complete-link clustering and average-link clustering. Single-link clustering compares clusters by looking at the minimum distance between them, i.e.

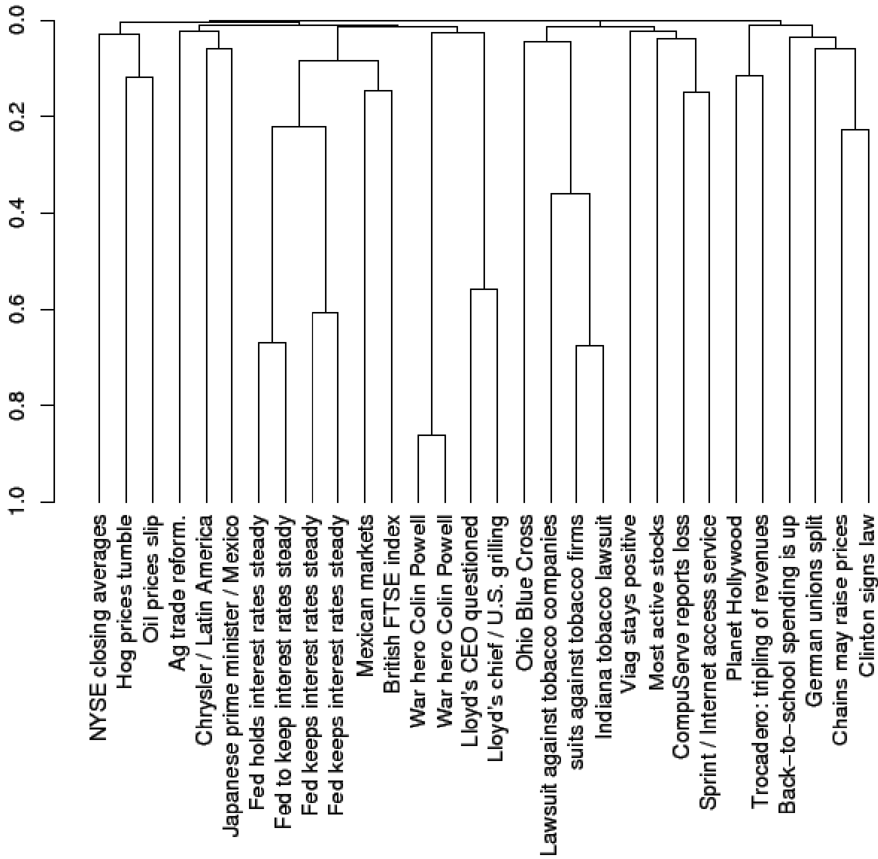


Figure 2.2: Example of dendrogram produced through hierarchical agglomerative clustering. The y-axis show the similarity score at which the clusters are merged. Figure from [8]

finding the elements within each cluster that is closest to the other cluster. Complete-link compares the clusters by looking at the distance between the two furthest elements in the clusters. Average link computes the distance between every element of the two clusters, and produces the average of these values as the resulting distance. Each principle have their strengths and weaknesses. The local evaluation of distance in single-link makes the method prone to "chaining", where the cluster expands by adding long tails of single elements. The complete-link clustering prefers compact clusters, but is more sensitive to outliers. Average-link clustering, or Group-Average Agglomerative Clustering (GAAC), avoids the pitfalls of the previous methods, but is more computationally expensive [8]. The complexity of HAC is $O(N^2 logN)$, making it magnitudes slower than K-means clustering.

2.4 Dimensionality reduction

For large sets of multi-dimensional data, dimensionality reduction can be an effective optimization Through these methods, the data is analyzed for subsets of similar attributes, which are gathered in feature attributes. This process is called *feature selection*. The feature vectors reduce the total dimensionality of the data, and serve to reduce the efficiency of clustering and classification. In addition, they also help to reduce overfitting; in adding a generalizing step to the clustering process, new documents are more easily added to the clusters, because the dimensionality reduction makes it more similar to the pre-existing documents [7]. The dimensionality reduction can be done through several methods, including latent semantic indexing.

Latent Semantic Indexing (LSI) is a type of dimensionality reduction where the relevance matrix is decomposed using a mathematical technique called Single Value Decomposition (SVD)

$$\mathbf{M} = \mathbf{K} \cdot \mathbf{S} \cdot \mathbf{D}^T \tag{2.10}$$

where $\mathbf{K}$ is a matrix of eigenvectors from the term-term correlation matrix given by $\mathbf{M} \cdot \mathbf{M}^T$, $\mathbf{S}$ is a diagonal matrix containing the singular values of $\mathbf{M}$, and $\mathbf{D}^T$ is a matrix of eigenvectors from the document-document correlation matrix given by $\mathbf{M}^T \cdot \mathbf{M}$. SVD is often used in feature extraction or matrix approximations. By excluding the singular values with the lowest values, the resulting matrix approximation $\mathbf{M}_S$ can contain the most important features of $\mathbf{M}$, while still having reduced size and complexity. The resulting matrix $\mathbf{K}_S$ which contain the term-term eigenvectors of the reduced feature set, can be used to find neighboring terms, by containing information on every terms relationship with all other terms.

2.5 Graph partitioning

Graph partitioning is a concept not often used within information retrieval. The primary areas of focus for these methods include distributing workloads for parallel computation [13], network traffic routing and similar cases, where the goal is to minimize network, or graph traffic. Graph partitioning seeks to split a graph into N equally sized partitions, such that the edge cut (number of edges running between partitions) is minimized. For graphs with weighted edges, the accumulated edge weight across partitions is minimized.

2.5.1 Multi-level graph partitioning

A multi-level graph partitioning algorithm consists of three phases, where the original graph is first reduced to a coarse set of vertices. The coarsening process constructs

several smaller graphs $G_l = (V_l, E_l)$ based on the original graph $G_0 = (V_0, E_0)$. Each graph is constructed from the previous by finding maximal matches within the graph, and merging these vertices. The best matching algorithms are also the slowest, making this choice a tradeoff between performance and quality. All the methods essentially try to find *locally strong* connection between vertices, that is $w_{i,j}$ is comparable to $\min\{\max_k, w_{ik}, \max_k, w_{kj}\}$, where w_{ij} is the strength between vertex i and j [14].

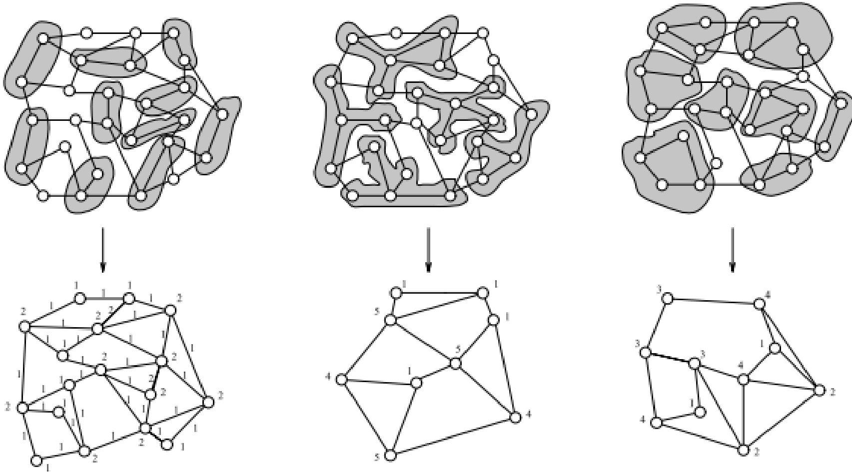


Figure 2.3: Illustration of graph coarsening. The shaded figures shows which vertices are combined for the coarsening, producing the graphs below. Figure from [15].

When the coarsening process is done, the small, resulting graph is then bisected in a way that minimizes edge cut. Several methods are available for this step as well, including spectral bisection, geometric bisection and combinatorial methods [16]. In the last phase, the graph is iteratively uncoarsened back to the original graph. For each step, the coarsened vertices of graph G_{k+1} are split back to the original vertices and edges of G_k. If the content of a coarsened vertex is split by the bisection, the bisection of this subgraph is refined, further reducing edge cut [16]. An illustration can be seen in Figure 2.4.

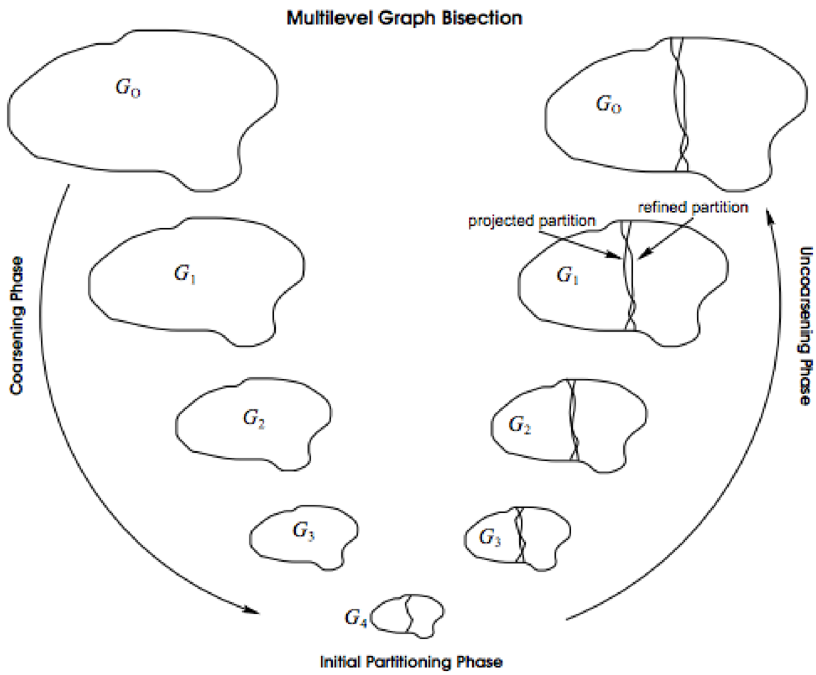


Figure 2.4: Illustration of Multi-level Graph Bisection. Figure from [16]

Chapter 3
State of the art

In this chapter, we will present the most interesting result of the preliminary literature study, and try to give an image of the current research situation in the specific area of collaborative filtering. The findings will be broadly grouped by theoretical field. Lastly, we will be reviewing the current state of the art, and discuss it in light of the research questions of this thesis.

3.1 Co-clustering

A common and intuitive way of addressing recommendation systems is to create a relevance matrix, as seen in Table 3.1. Recommendations are then created by computing the similarities between the given user and the rest of the users. The similarities are computed using a distance function, for example cosine similarity or Euclidean distance[9, p. 309]. In principle, this method does not scale well, as the relevance matrix increases as a product of $users \times records$. To address this problem, several methods of dimensionality reduction have been applied, attempting to cluster the relevance matrix along both axes (see Table 3.2). Banerjee et al. [17] propose a co-clustering method where Bregman divergences are used to create reduced matrix approximations, reducing the dimensionality, or size, of the matrix. Long et al. [18] utilize block value decomposition to discover block structures (co-clusters) in the relevance matrix effectively reducing the computational load of calculating recommendations. Entries of new records and users into clusters is briefly mentioned by George et al. [19], who propose a method of incremental training which assigns the new entities to an intermediate cluster while waiting for the next evaluation of co-clusters. Zhang et al. [20] mentions the concept of "cold start", which concerns the challenges related to new users and documents, and claims having addressed these by allowing for reevaluation of sub clusters separately from the complete relevance matrix. Co-clustering then seems like an effective method for extracting subgroups in large collections of user-item relationships, both considering scalability and performance.

	u1	u2	u3	u4	u5	u6	u7	u8	u9
d1	**X**			**X**			**X**		
d2		**X**				**X**			
d3			**X**		**X**			**X**	
d4	**X**						**X**		
d5					**X**			**X**	
d6	**X**			**X**			**X**		
d7		**X**							**X**
d8	**X**				**X**			**X**	
d9		**X**				**X**			**X**
d10	**X**			**X**					
d11			**X**		**X**				
d12		**X**				**X**			**X**

Table 3.1: Example of relevance matrix. Each row represents a document, while each column represent a user. An **X** in the matrix represents a connection between the two. The relevance matrix can also be populated by relevance weights.

	u8	u5	u3	u9	u2	u6	u7	u1	u4
d1							**X**	**X**	**X**
d10								**X**	**X**
d6							**X**	**X**	**X**
d4							**X**	**X**	
d5	**X**	**X**							
d3	**X**	**X**	**X**						
d11		**X**	**X**						
d8	**X**	**X**							
d9				**X**	**X**	**X**			
d2					**X**	**X**			
d7				**X**	**X**				
d12				**X**	**X**	**X**			

Table 3.2: The relevance matrix from Table 3.1, co-clustered by document and user

3.2 User behavior logging

Collaborative filtering can be based on both explicit and implicit feedback. For implicit feedback however, methods must be implemented to collect information on

user behavior. Using Invenio and the CDS as a basis, Gvianishvili et al. [4] explores different ways of collecting user behavior data, both by analyzing server access logs and through Invenios implemented logging abilities. The server logs are analyzed, and data related to the access patterns to each record is saved. They state:

> Four types of counts are extracted from the logs:
>
> - Number of detailed page views: for each record we count the occurrences of record abstract being viewed
> - Number of downloads: for each record we count the occurrences of the associated full-text being downloaded
> - Number of displays: for each record we count the occurrences of the record being listed on the results pages
> - Number of seens: for each record we count the occurrences of record being seen. We mark all records seen from the first up to the one on which an action has been performed (download/view). For example, if a user downloads record #6 we mark all records from #1 up to #6 as seen by the user. This count provides us with an approximate result of records seen, since there is no guarantee that the user has really seen those records.

Their conclusion states that user behavior and record access logging is a good foundation for further development of ranking methods, recommender systems and other extensions of the current search engine.

A more advanced approach to user behavior analysis is done by Xie et al. [21]. In addition to collecting information from web server logs, they propose to use a belief function to assign users to clusters, by analyzing the basic probability assignments (BPA) associated with each user. The algorithm displays satisfactory results on experimental data, and their conclusion emphasizes the importance of being able to create proper recommendation solely by implicit feedback. They do however point out that the algorithm needs improvement with regard to scalability.

3.3 Citation based ranking

In the area of scientific research, an evident feature for ranking methods is citations. Previous methods exploiting this feature counted the number of citations of a given record, and assigned this as a weight for the ranking of search results. Several problems has been identified with this approach. The two most prominent are that it does not prioritize citations from important/popular papers, and it does not

differentiate between old and new citations. In addition, there are problems with phantom citations, deflation and inflation[22].

To improve on this situation, Marian et al. [23] presents a citation graph based ranking method. By extracting citations from each record in the collection, a citation graph can be constructed – linking related records to each other. This graph can then be scored by an algorithm similar to the PageRank algorithm used by Google to score websites. In addition, they apply a time decay function to the records to promote new entries. To account for missing citations internally in the experimental framework, an external citation authority was added to increase the completeness of the citation graph. The proposed method performs better when compared to citation count or basic PageRank.

The problem with incomplete citation graphs is also addressed by Sugiyama et al. [24], who propose to extend the graph by identifying "potential citation papers" through collaborative filtering. By doing this, they claim to address the cold start problem of cutting edge research. The experimental results are significant, and potential citation papers proves to be an effective way of dealing with sparse citation graphs.

Another approach to citation based ranking is taken by Caragea et al. [25]. Here, the mathematical method SVD (see Section 2.4) is used to extract latent semantic factors in the citation graph, which is then used to predict interesting records for the recommendation. The results presented in this paper are positive, and the authors argue that the method is quite versatile: “SVD allows for easy incorporation of additional information [...] (e.g. textual information, author, venue)”

3.4 Graph based collaborative filtering

A different approach to collaborative filtering can be done by looking at access data as a bipartite graph, connecting users U to items I. Similarities can then be calculated by traversing this graph (a very simple example could be to collect all users that share items with a given user). Huang et al. [2] propose a graph based method where a graph is constructed linking related users and items, as well as connecting similar users and items respectively. Recommendation is then done by graph traversal, collecting items related to similar users and ranking them by the degree of association.

O’Connor and Herlocker [26] propose to mitigate the problems of collaborative filtering in large, heterogenous collections (movies within genres) by performing a preliminary clustering. This divides the collection into smaller, more homogenous collections with lower dimension. Four clustering methods were applied: hierarchical

clustering using average link evaluation; Robust Clustering Algorithm for Categorical Attributes (ROCK) – a specialized clustering algorithm believed to have increased performance on categorical data; and kMetis and hMetis[16] – two graph partitioning algorithms developed at the University of Minnesota. Their results showed poor results for the hierarchical clustering; a low k resulted in one large cluster containing almost all the elements, while a high k produced small, useless clusters. The ROCK algorithm performed even worse. The Metis algorithms showed promising results, producing similar sized clusters with good coverage and reasonable Mean Absolute Error (MAE). Recommendations based on the resulting partitions did not consistently outperform unpartitioned recommendation. The authors suggest one reason for this is the fact that none of the applied clustering techniques support soft clustering, where one item may belong to several clusters.

3.5 Reviewing the state of the art

From the research on previous accomplishments within clustering for recommender systems, it is apparent that there are several paths to the target. In this section we discuss the state of the art in light of the research questions of this thesis.

User behavior logging is very relevant for this thesis. The groundwork done by Gvianishvili et al. [4] shows the possibilities present in current digital libraries, and underpins the importance of utilizing advanced user statistics in future recommender systems. While Xie et al. [21] mention problems related to scalability for their method, they do emphasize the importance of being able to rely solely on implicit feedback methods in recommender systems. Further reasoning upon this idea is done in Chapter 5.

The co-clustering methods presented in section 3.1 seems like a very interesting approach to the current problem, and effort should be made to evaluate these methods on the scientific collections. Dimensionality reduction provide an effective method of reducing the size and complexity of the recommendation calculation, although not without problems. Unfortunately, the time restrictions of this thesis prevented elaborate studies of a field with the complexity of co-clustering.

The concept of citation based ranking is not discussed in this thesis, but a review has been done in the area because of its importance in the field of recommender systems within scientific collections. However, a few elements can be picked up from this approach. Organizing records in a graph where related documents are linked to each other, is a different approach than using relevance matrices. This is something that should be examined. In the same area of graph based recommendation, work has been done to connect records based on similarity measures, rather than citations. This avoids the problems of connectivity and completeness related to citation based

graphs, but introduces new ones. The choice of similarity measure is important; classical, continuous ones like Euclidean distance will not work unless a cut-off similarity value is set (lest the graph will have edges from and to every vertex). A few measures could be shared citations, shared users (user has downloaded/accessed both records) and shared tags or other metadata.

3.5.1 Conclusion

Through this review, we can see that there are several different approaches to clustering for collaborative filtering. There is one important difference separating the methods, namely how similarity between elements is calculated. Not only are several similarity measures implemented, but there are different approaches to what makes two elements similar. Marian et al. [23], while not applying clustering techniques, uses citations as a way of connecting records, while Huang et al. [2] uses word similarity.

What has not been found is an approach where the similarities between elements is calculated from implicit user behavior. The closest research found is from O'Connor and Herlocker [26]. They review clustering methods on the MovieLens dataset, which contains explicit movie ratings by the users.

Our approach then, will attempt to fill this gap, by applying clustering algorithms to the dataset containing user-record relationships, creating clusters of both users and records As no concrete conclusion can be drawn as to what clustering method to use, three different variants will be applied to the dataset: K-means, HAC and graph partitioning. There are some suspicions as to how the different algorithms will behave: The K-means algorithm should be the fastest of the two clustering methods, while the HAC should better illustrate the natural clustering occurring in the dataset. The graph partitioning should be able to create similarly sized partitions.

Chapter 4

Approach

As mentioned in Chapter 1, the hypothesis on which this thesis is based is that the user groups of scientific collections have quite distinct usage patterns, creating a good foundation for using clustering techniques in recommender systems. The objective of this master thesis is to lay the foundation for a recommender system based on collaborative filtering, specialized for collections of academic papers. More specifically, the focus is to successfully create a clustering from the chosen dataset, such that extraction of recommended records is possible.

4.1 Experimental setup

The CERN Document Server (CDS) collection was chosen as the basis for the experiments performed in this thesis, because it is one of the largest, publicly available collections of academic papers. As mentioned in section 3.5.1, the approach of this thesis will attempt to perform a dual clustering on both records and users, where the user and record clusters would be pairwise related. The idea is that recommendations can then be made by looking at records in the corresponding record cluster for users in a user cluster.

The collection can be broadly grouped into the following collections, giving an idea of the number of clusters to expect:

- CERN Accelerators (29,400)
  - All CERN Accelerators documents (22,033)
  - PS Complex (10,719)
  - SPS and CNGS (6,365)
  - LHC (3,868)
  - Projects, Upgrade and Consolidation Projects (71)

  - R&D and Studies (25)
  - EU Funded Projects (905)
  - Decommissioned Facilities (2,769)
  - Accelerators and Facilities outside CERN (12)
  - Technologies, Systems, Accelerator Physics and Processes (4,651)

- CERN Experiments (53,005)
  - LHC Experiments (43,578)
  - Fixed Target Experiments (2,810)
  - Recognized Experiments (748)
  - LEP Experiments (5,569)
  - PS Experiments (313)

- CERN R&D Projects (2,962)
  - CERN Accelerator R&D Projects (1,440)
  - CERN Detector R&D Projects (28)
  - EU Projects (2,105

The methodology of the research approach is based on the one proposed by Kazanidis et al. [27], and consists of thee main parts: Data pre-processing, clustering and evaluation. During the pre-processing, analysis of the data will be done, along with an evaluation of what data to keep. In the clustering process we will attempt to apply different clustering algorithms on the dataset, and evaluate the results consecutively. Lastly, we present the full set of results, and provide a more thorough evaluation of the research.

4.2 Hardware

The data pre-processing was done on a Macbook Air mid 2013, with a dual core 1,3 GHz Intel Core i5 processor and 4 GB RAM. Clustering algorithms were run on NTNUs calcfarm – a server installation with two 6-core 2,66 GHz Intel Xeon 5650 processors and 192 GB RAM.

4.3 Data pre-processing

The initial raw log file spanned from 2010-10-04 to 2012-03-15, with a size of 40.11 gigabytes, containing over 184 million log entries. Before the pre-processing could start, a decision had to be made about what type of data to retrieve from the logs.

Given the size of the initial data, the decision was made to exclude all but the strongest connection between an IP and a record. The first step on pre-processing then was to isolate the entries of IPs downloading a Portable Document Format (PDF) file. This reduced the file size by 95,8% to 1.7 gigabytes – around 16 million log entries.

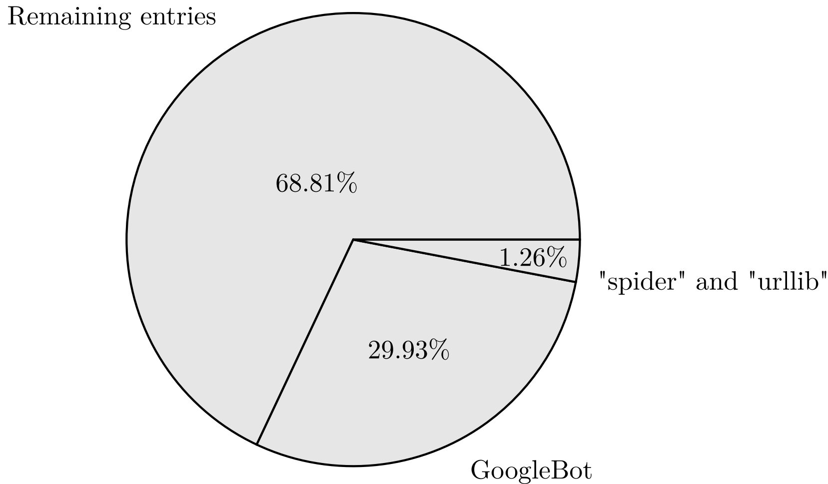


Figure 4.1: Pie chart showing the composition of log entries for downloaded documents. As can be seen, bot activity accounts for a large part of the total traffic for the CDS

Furthermore, a basic bot removal script was applied, excluding all access from user agents containing the words "GoogleBot", "spider" and "urllib". A pie chart showing the composition of the log entries can be seen in Figure 4.1. At this point, the log was condensed to a comma separated file, containing IP, record ID, and timestamp (see appendix A.2.2). Further analysis showed however, that until Febuary 2011, the logging was sporadic and did not represent a proper usage pattern. For better log integrity, the log was therefore cropped to the period from 2011-02-17 to 2012-03-15.

To help with giving an overview of the data, two scripts were made to create a sorted list of each record with their corresponding accessing IPs, and a list of each IP with their accessed records. These scripts can be seen in appendix A.2.3 and A.2.4. The resulting files also served a purpose in later scripts, by allowing quick access to this otherwise expensive sorted list.

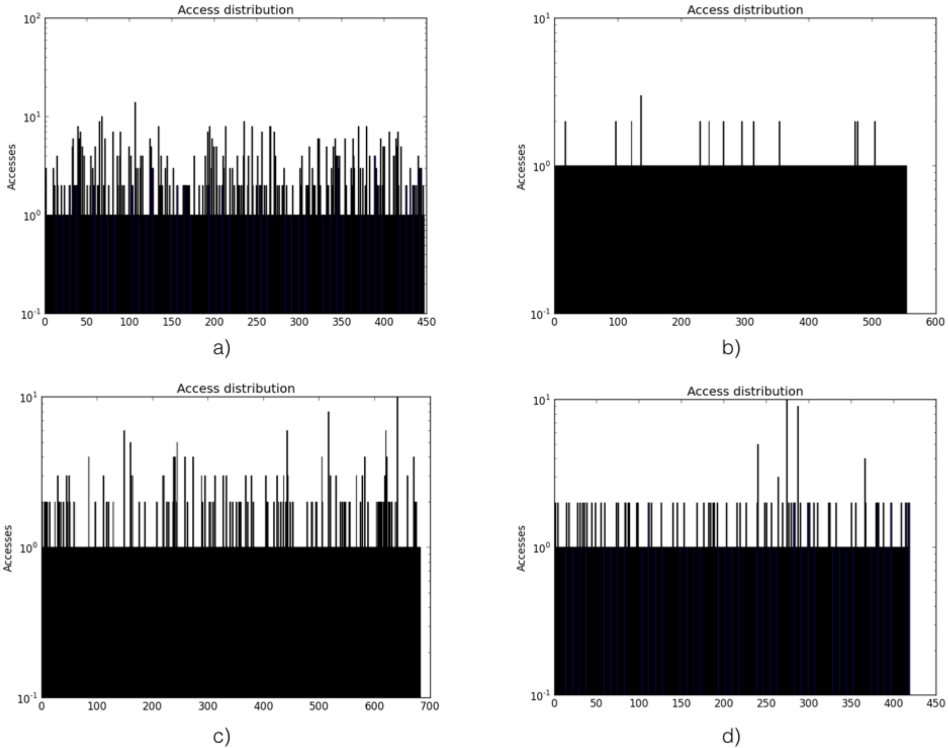


Figure 4.2: Bar charts presenting the usage patterns of potential bots. a) and c represents suspected users, while b) and d) represents bots. Records not downloaded by the IP is not included in the histogram.

Having removed the most obvious bots, a deeper dive was now done to assure the integrity and quality of the resulting log. A more manual approach was taken, and the most active IPs (those with more than 500 downloads) were extracted to separate log files, which were then analyzed by histogram (appendix A.2.5 and A.2.6). Figure 4.2 show the different types of charts represented by these IPs. Subfigure *a*) was believed to be a real user, which was further substantiated by looking at the corresponding server log. Similarly, subfigure *b*) was confirmed as a bot. This evaluation was done by looking at the distribution of entries; the IP would be classified as a bot if the usage pattern showed a very high activity over a short timespan (i.e. 100 downloads within a few minutes).

By analyzing these histograms, the immediate thought was that users and bots might perhaps be separated by the total number of log entries. However, further

analysis showed potential users with greater activity than confirmed bots (subfigure *c*) vs. *b*) in Figure 4.2), as well as bots with less activity than recognized users (subfigure *d*) vs. *a*) in Figure 4.2). This removed any direct distinction of users and bots based on activity, and resulted in the development of a more advanced bot detection algorithm that removed every IP downloading more than 60 documents within one hour (see appendix A.2.1). A new histogram analysis confirmed the efficiency of this algorithm, as no bots were found among the IPs with high activity. At this point, the log file had been reduced to 86 megabytes and 1779292 lines, with 444825 IPs accessing 159794 records.

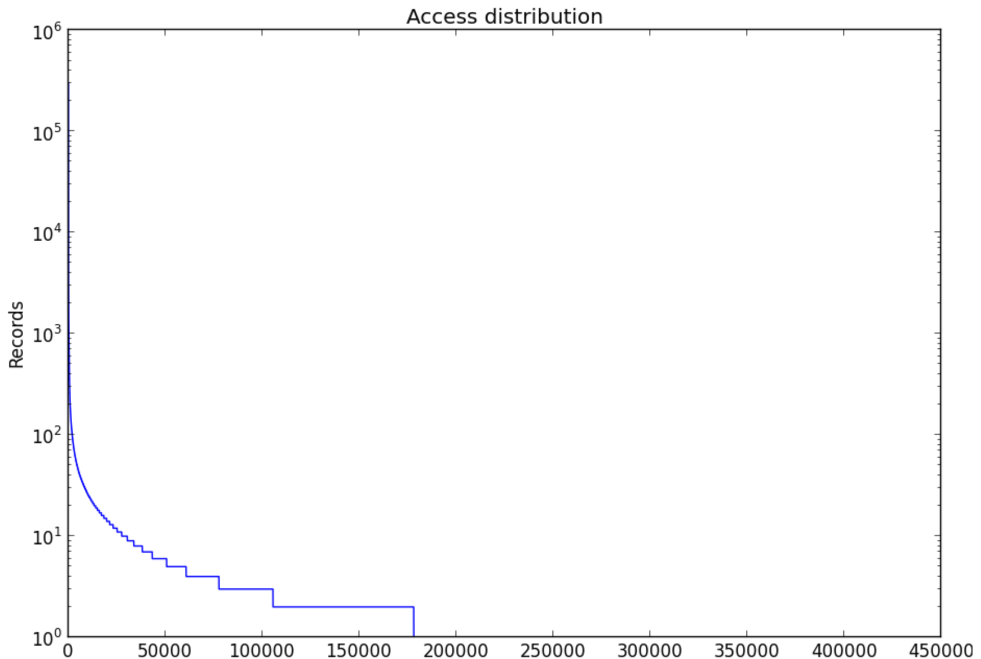


Figure 4.3: Records accessed per user, sorted from active to inactive. The graph shows that about 270000 (60%) of the users have accessed only one record.

The next step of the log pruning was removing inactive users and records. The threshold for user contribution was set to having downloaded 3 records. As can be seen in Figure 4.3, this removes around 76% of the IPs, while being a reasonable requirement for participating (one will most likely have downloaded more than two documents before noticing the need for a recommender system). Based on the graph presented in Figure 4.4, the threshold for record popularity was set to 3. This removes around 55% of the records, increasing quality of the dataset, while still being

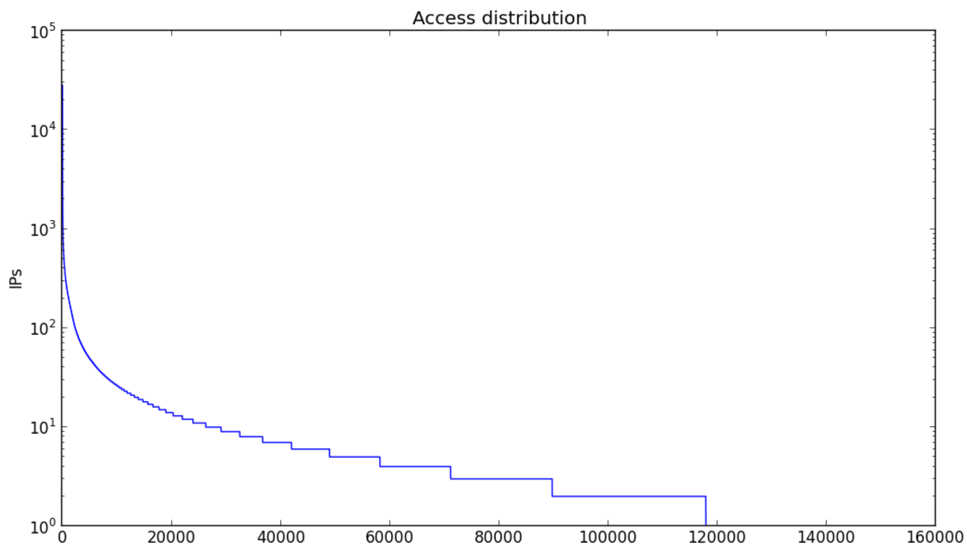


Figure 4.4: Number of IPs accessing each record, sorted from popular to unpopular. The graph shows that about 40000 (25%) of the records have been accessed by only one IP.

a reasonably low threshold for participating. The scripts for performing this pruning can be seen in appendix A.2.7 and A.2.8.

The final step was removing duplicates. The proposed clustering pays no attention to how often a user has downloaded a document, so this was a convenient simplification. See appendix A.2.9 for source code of this script.

The resulting log file was 24 megabytes and 495221 lines, with 70006 IPs accessing 65754 records.

4.4 Clustering algorithms

The scientific field of clustering is large and complex, and there exists hundreds of clustering and partitioning algorithms which might be suitable for this problem. To do an exhaustive study of all possible clustering algorithms is not within the scope of this thesis. In total, four methods were attempted, presenting a somewhat progressive learning approach:

- Simple K-means Clustering

- Hierarchical Agglomerative Clustering (HAC)
- A custom designed clustering algorithm
- Metis k-Way Graph Partitioning

4.4.1 Simple K-means Clustering

As mentioned in section 2.3.1, K-means clustering is widely considered to be the most important flat clustering algorithm. As such, it seemed a reasonable method for the initial experiments. Furthermore, its complexity is low, which would indicate that it could handle the large dataset from the CDS.

4.4.2 Hierarchical Agglomerative Clustering

A clustering method often mentioned alongside K-means is HAC, which made it a natural choice for this thesis. Additionally, it was selected partly because of its coverage: by design, every element is part of a cluster. The idea was also to use the resulting dendrogram to illustrate how the records in the dataset are related on different levels of similarity.

The HAC implementation used in this experiment was supplied in Matlab. This uses *pdist* as a similarity function, which among other includes the two similarity measures Euclidean and Jaccard distance.

4.4.3 Custom clustering algorithm

Although the HAC had several promising qualities, the performance issues hindered the use of this algorithm directly. As an attempt to overcome these issues, a custom, greedy, and agglomerative algorithm was created, where thoroughness was sacrificed for reduced complexity (see appendix A.1 for source code). The algorithm was developed to cluster IPs on the basis of shared records. Had the clustering been successful, an additional opposite version would have been made, clustering records on the basis of shared IPs.

Where a regular HAC initially would consider every IP as a separate cluster, our algorithm starts by creating pairs of IPs along with their shared records. Pairs were only made for IPs that shared three or more records, and one IP could be assigned to several pairs. To avoid expensive similarity computation and sorting, the assignment of IP pairs to clusters was done in a very greedy manner. Clustering started by iterating through the list of IP pairs. For every pair, the algorithm searched existing clusters for both IPs, and checking for four possible cases (see algorithm 0):

- If both IPs were found within the same cluster, the records of the current pair would be added to that cluster, and the cluster variable *internal_connections* would increase by one.
- If only one of the IPs were found in a cluster, the other IP would be added along with their shared records.
- If the IPs were found in different clusters, the cluster link between the two clusters would be increased by one, and the shared records would be added to this link.
  - If the link between the two clusters was stronger than five, the clusters would be merged, and the records in the link added to the resulting cluster.
- If none of the IPs were found in a cluster, a new cluster would be created, containing the two IPs and their shared records.

The algorithm completed when the iteration of the IP pairs list was done. The pseudocode for the most important parts of the algorithm can be seen in Algorithm 4.1 and 4.2.

Parameter selection

There are two important parameters in the custom clustering algorithm: threshold for IP pairing, and the limit for merging clusters. The initial placement of these values were somewhat arbitrary, and several values were tested. Changes in the threshold for IP pairing did not produce any observable changes, but changing the merge limit showed the same types of results as observed by O'Connor and Herlocker [26] when applying the HAC algorithm: for low limits of merging, the result was one large cluster and several small, but useless clusters. For stricter limits, the resulting clusters were of better shape (up to 326 IPs per cluster), but the coverage 1314 of 25304 IPs – an alarming 5.2%. This was due to the cut off value set for including clusters in the result. For this experiment, it was placed at 25 IPs per cluster – again an arbitrary value, but it still illustrates the low quality of the majority of the clusters.

4.4.4 Metis graph partitioning

Through the research done by O'Connor and Herlocker [26], the concept of graph partitioning was introduced to the thesis. More specifically, the Metis graph partitioning tool was discovered. In their report, two variants of the graph partition tool Metis was applied: kMetis (or Metis) and hMetis – the latter being a hypergraph variant. Additionally, the hMetis algorithm has an unbalance parameter, which allows for the

Algorithm 4.1 Cluster assignment

```
Input: ip1, ip2, records
  for all clusters do
      if ip1 ∈ cluster then
          ip1_cluster ← cluster
          if ip2 ∈ cluster then                    ▷ Both IPs found in the same cluster
              cluster ← records
              cluster.internal_connections += 1
  return
          end if
      end if
      if ip1 ∈ cluster then
          ip2_cluster ← cluster
      end if
  end for
  if ip1_cluster and ip2_cluster then              ▷ IPs found in separate clusters
      Create link between cluster 1 and 2
      limit ← 5 MERGE_CLUSTERS(ip1_cluster, ip2_cluster, link, limit)
  else if ip1_cluster then                         ▷ One of the IPs are found
      Add ip2 and records to cluster 1
  else if ip2_cluster then
      Add ip1 and records to cluster 2
  else                                             ▷ None of the IPs are found
      clusters.add(new Cluster(ip1, ip2, records))
  end if
```

Algorithm 4.2 Merge clusters

```
  function MERGE_CLUSTERS(cluster1, cluster2, link, limit)
      ratio = (no. of records in link) / (link.strength)
      if ratio < limit then return
      end if
      cluster1 ← cluster2.ips, cluster2.records
      cluster1 ← link.records
      cluster1.int_connections += cluster2.int_connections + link.strength
      cluster2.deleted = True
  CLUSTER.UPDATE_LINKS(cluster1, cluster2)
  end function
```

creation of partitions of unequal size. In their experiments, both algorithms perform similarly, and they recommended using kMetis because of its faster run time. Given that the input formats of the algorithms are different, the regular Metis algorithm was chosen for this experiment.

The Metis tool was initially developed in 1995 at the University of Minnesota, by Karypis, George and Kumar, Vipin [16]. It points to better performance than comparable algorithms, both regarding running time and resulting partitioning. Metis is a k-way graph partitioning tool, where the input graph is separated into k partitions of similar size, minimizing the *edge cut*. It uses multi-level graph partition, which is explained in detail in section 2.5.1. Four coarsening schemes, as well as four partitioning schemes are implemented in Metis [15]. A brief summary of these techniques follow below.

Coarsening schemes

Random Matching (RM). RM selects vertices from the graph in random order, and compares them to the neighboring vertices. If these vertices have not yet been visited, they are matched with the randomly selected vertex. It is a greedy algorithm that does no computation to find the optimal matching.

Heavy-Edge Matching (HEM). HEM has a more advanced approach. The vertices are still selected from the graph in random order, but is only matched with the vertex connected through the heaviest edge of the randomly selected vertex. While being more expensive than RM, they still has the same complexity.

Light Edge Matching (LEM). LEM works opposite of HEM, by matching vertices connected by the lightest edge. While this may seem counter-intuitive, the idea is that the resulting graph G_{k+1} has an average *degree* significantly higher than that of G_k. This is optimal for some partitioning schemes, such as the KL algorithm (explained below).

Heavy Clique Matching (HCM). HCM finds *cliques* within the graph, and collapses them into a coarsened vertex. The cliques are found by analyzing subgraphs $G_U = (U, E_U)$ and comparing the cardinality (i.e. size or amount) of the vertices U and edges E_U therein. The ratio between these values is calculated by

$$R = \frac{2|E_U|}{|U|\,(|U|-1)} \tag{4.1}$$

If the subgraph is a clique, R will be equal to 1, and the value will decrease the looser the connection in the subgraph.

Partitioning schemes

Spectral Bisection (SB). SB uses the spectral information of the graph to create a partitioning. The graph is organized into an adjacency matrix A, where $a_{i,j}$ denotes the edge between vertex i and j; and a diagonal degree matrix D, where the values $d_{i,i}$ denotes the degree of vertex i.

$$a_{i,j} = \begin{cases} ew(v_i, v_j) & \text{if } (v_i, v_j) \in E_m, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

$$d_{i,i} = \sum_{(v_i,v_j)\in E_m} ew(v_i, v_j) \tag{4.3}$$

The algorithm then finds the second smallest eigenvector y from the Laplacian matrix $Q = D - A$. This eigenvector (called the Fiedler vector) contains a value for each of the vertices in the coarsened graph. From their value in this vector, vertices are assigned to two partitions: Let r be the weighted median of the y values. Each value of y is then compared to r; if $y_j \leq r$, the corresponding vertex is assigned to one partition, else is assigned to the other.

KL algorithm. This algorithm starts with a bisection of the graph, and improves this partitioning by doing local changes, moving a vertex from one partition to the other, improving edge cut. The initial bisection can be random, or found through some other method. If the bisection is random, the KL algorithm is often run several times, and the best resulting partitioning is chosen. The gain from moving a vertex is calculated by

$$g_v = \sum_{(v,u)\in E \wedge P[v]\neq P[u]} w(v, u) - \sum_{(v,u)\in E \wedge P[v]=P[u]} w(v, u) \tag{4.4}$$

where $w(v, u)$ is the weight of the edge between vertex v and u, and P is the set of partitions. If g_v is positive, the edge cut of the partition can be reduced by g_v, and the vertex should change partition. However, after moving a vertex from one partition to the other, the edge cut of adjacent vertices may have changed. Therefore, these vertices should be reevaluated.

Graph Growing Partitioning algorithm (GGP). GGP bisects the graph by starting at a random vertex, and grow a selection of vertices by increasing one at the time in a breadth-first manner, where all of the adjacent vertices are included before moving further. The bisection is complete when this selection contains half of the

vertices, or half of the total vertex weight. As for the KL algorithm, this method is sensitive to poorly chosen starting vertex. It is therefore often run several times to reduce the risk of suboptimal partitions.

Greedy Graph Growing Partitioning algorithm (GGGP). GGGP works similarly to GGP, but increases its selection in a greedy fashion, including the vertices that provides the smallest increase in edge cut. It is less sensitive to starting vertex, and performs better, both in regard to running time and resulting partition.

4.5 Experimental procedure

In this section, the experimental process will be elaborated. Some results will be mentioned, but the full result report will be saved for Section 4.6.

4.5.1 Simple K-means

After the logs were pre-processed and ready for clustering, the first attempt was done using Simple K-means Clustering in the knowledge analysis tool Weka Explorer. To prepare for this, the log was converted to a relevance matrix using the script in appendix A.2.10, which increased the file size from 24 megabytes to 9.2 gigabytes. Record IDs were prefixed with "rec_", i.e. converting the attribute from numerical to nominal, thus allowing clustering on this class.

Importing a file of this size to Weka proved counterproductive, as the import process still had not completed after two days. This did not bode well for the actual clustering that was to follow, so a selection of the original matrix had to be made. By using the *awk*[1] program, a selection of every 10th line was written to a new file:

```
awk '!(NR%10)' matrix_full.csv > e10th_matrix_full.csv
```

This file was then imported to Weka without problems. K-means clustering was then applied, with $k = 2$. The procedure completed after about an hour, but yielded no usable results. It was apparent that the numbers of clusters had to be increased. Using the collections presented in section 4.1 as a guide, one can see that the realistic number of clusters would lie in the range $15 < k < 25$, effectively increasing the runtime to a full day.

To speed things up, a further reduction was done, reducing the filesize and number of records to $\frac{1}{100}$ of the original matrix. The resulting matrix was easily imported to Weka, and a test run with $k = 2$ was done in around one minute (see appendix B.1). The algorithm was then run with $k = 25$. After five hours however, the process was

[1] http://www.grymoire.com/Unix/Awk.html

cancelled. According to the complexity formula of K-means, an increase in k from 2 to 25 should only increase the runtime with a factor of 12,5 (linearly). At a factor of 150, it seemed that further experiments using K-means in Weka was not worthwhile. A contributing factor to this decision was also that the similarity model of K-means is not optimal for binary relevance data: the centroids computed from the Euclidean distance function does not fully match any elements, and so the difference between elements belonging to a cluster and the rest is too small for effectively assigning clusters.

4.5.2 Hierarchical Agglomerative Clustering

The application of the HAC algorithm was done in Matlab. The $\frac{1}{10}$-matrix was imported, and the clustering initiated. This first attempt stalled, because the memory usage had exceeded the allowed limit (30 gigabytes). The $\frac{1}{100}$-matrix was then imported, and the clustering successfully completed. The resulting dendrogram showed no sign of any strong clusters however. This might be because of the downsampling, but most likely the reasons are related to the similarity function. The default similarity function used in Matlabs HAC is the Euclidean distance, and as mentioned in section 4.5.1, this is not optimal for binary data.

The algorithm was run again, using *Jaccard* similarity – a similarity measure designed for binary values. Unfortunately, the results were just as bad, with no natural clustering appearing. This is further mentioned in Section discussed in Section 5.2.

4.5.3 Custom clustering algorithm

As neither K-means nor HAC could handle the large dataset, a custom clustering algorithm was created. This algorithm used the agglomerative nature of HAC, but sought to reduce the complexity by its very greedy nature. The algorithm was not especially quick, and gave results similar to those presented in [26]: Strict parameters produced several, useless clusters; while looser parameters produced one, giant cluster and some additional, smaller ones. The development of this algorithm stopped after a few test runs, as it became apparent that fixing the drawbacks meant developing something that became more and more like traditional HAC. More specifically, the results shows that the greedy assignment to clusters does not work satisfactory. A more refined prioritization of which IP pairs to cluster could have made the algorithm work better, but the increased complexity this would cause makes it unfit for the Python language (which is several times slower than C for heavy algorithms).

Reevaluating the initial design

At this point, the cost of clustering was apparent. A reevaluation of the initial design was therefore done, to see if any improvements could be done reducing the practical cost of implementation. The decision was made to omit the user clustering. The reasoning behind this was that cluster affiliations could most likely just as effectively be calculated by analyzing a user's accessed documents directly.

4.5.4 Graph partitioning

The idea of swapping the classical clustering techniques with graph partitioning came from O'Connor and Herlocker [26], who faced many of the same problems as those appearing in this chapter. Before being able to apply graph partitioning to the data, it first had to be converted to a graph. This was done by creating weighted edges between every record based on how many common users they share. In essence, if three users have read the same two documents, an edge with weight 3 is created between them. The Metis graph partition tool was used, and the very compact graph notation format required by this tool resulted in a graph file of 730 megabytes.

At this point, we had to find the optimal number of partitions. When increasing the number of partitions, the edge cut will most likely also increase. However, the increase in edge cut is likely to slow down when the number of partitions come close to the optimal number. For partitionings above the optimal number, the increase in edge cut will speed up again. The principle is illustrated in Figure 4.5, 4.6 and 4.7. Based on this assessment, the edge cut for $N = [2, \ldots, 300]$ partitions was graphed. This graph showed a "knee", and the optimal partitioning could be found in a manner similar to finding the optimal number of clusters for K-means clustering. The size of the "knee" also says something about how prominent the natural partitions in the collection are; if the edge cut decreases when the partition number increases, a well fitting partitioning has been found.

The resulting output was saved for each iteration. Additionally, a text file was created with partition assignment for each vertex/record. From the graph produced by plotting the edge cuts for the different partitionings, a proposed number of 25 partitions was found, which is close to the number of collections presented in Section 4.1 The output of this partitioning can be seen in Appendix B.3, and the graph in Figure 4.10.

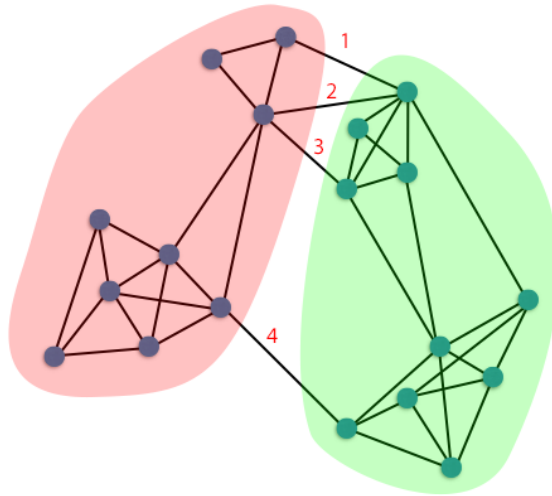


Figure 4.5: 2-way partitioning of a graph that has three natural partitions.

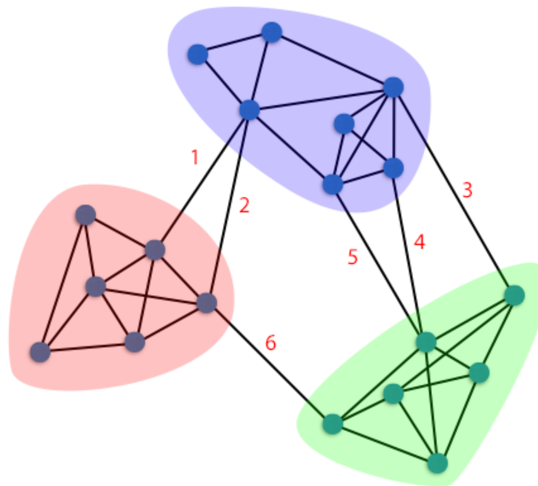


Figure 4.6: 3-way partitioning of the same graph. The edge cut is has increased from the 2-way partitioning, but it is apparent that this is the correct partitioning.

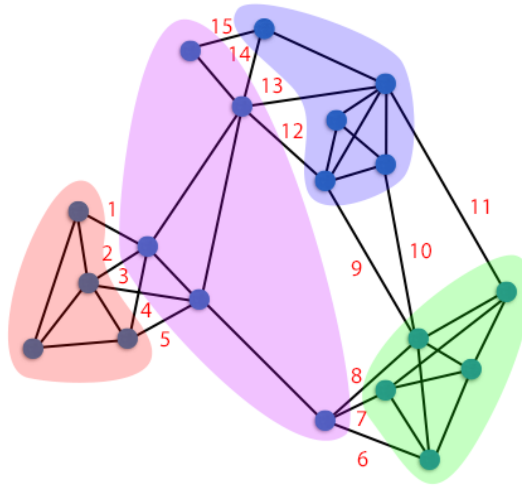


Figure 4.7: 4-way partitioning of the same graph. As the partitioning needs to split up natural partitions, the edge cut increases drastically. While this is probably not the optimal 4-way partitioning, it illustrates the point.

4.5.5 Evaluating clusters

As the Metis algorithm produces potentially useful partitions, an evaluation method had to be devised to assess the quality of the results. One of the most common methods for evaluating clusters is the purity function. It is defined as follows:

$$\text{purity}(\Omega, \mathbb{C}) = \frac{1}{N} \sum_k \max_j |\omega_k \cap c_j| \tag{4.5}$$

where $\Omega = \{\omega_1, \omega_2, \ldots, \omega_K\}$ is the set of clusters and $\mathbb{C} = \{c_1, c_2, \ldots, c_j\}$ is the set of classes[8]. However, this method needs a blueprint – a definitive set of right and wrong classification. This does not exist for the CDS, and the creation of this collection is a major task. Still, if the work on clustering user behavior in CDS is to continue, a way of evaluating the clusters must be made possible.

4.6 Experimental results

No interesting results were produced by the K-means algorithm, mostly because the relevance matrix for the full log was too large to handle for the Weka tool. Still, the research suggests that K-means is not the optimal clustering method to user for this

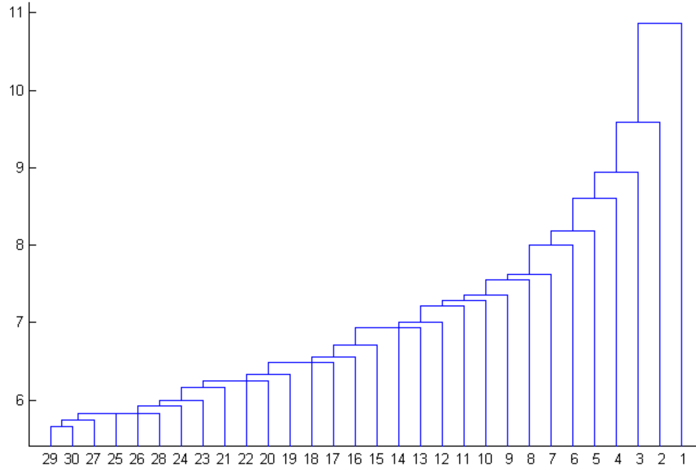


Figure 4.8: Resulting dendrogram from the HAC clustering of every 100th record. Using Euclidean distance.

type of data. The output of the 2-Means clustering of the $\frac{1}{100}$-matrix can be found in appendix B.1.

As can be seen in Figure 4.9, no similarity gap can be observed in the dendrogram for the HAC algorithm, neither by using Euclidean nor Jaccard similarity. Even though the result from Euclidean similarity is poor, there is at least some prioritization as to which clusters to merge first. Using Jaccard similarity, every cluster is evaluated at a distance of 1.0 to every other cluster, which suggests a failure in the application of the method.

The graph partitioning did produce some interesting results. As can be seen in Figure 4.10 and 4.11, an optimal number of partitions can be guessed from the bend at $N \approx 25$. The appearance of this bend suggests promising results from graph partitioning.

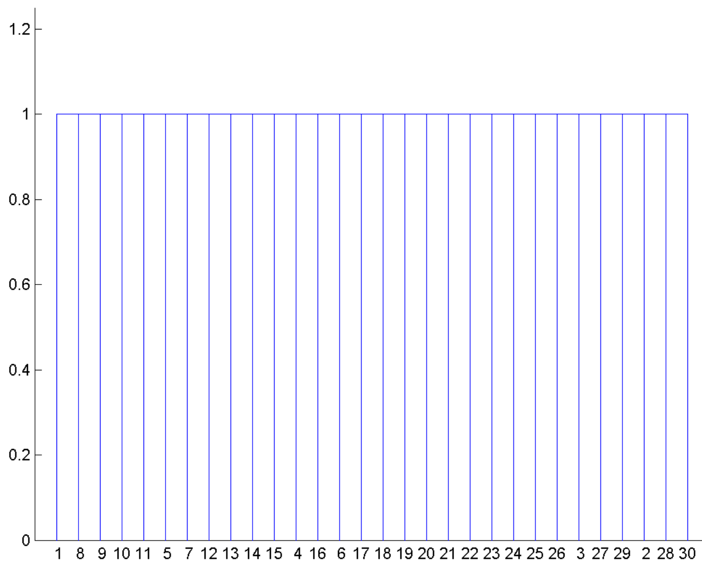


Figure 4.9: Resulting dendrogram from the HAC clustering of every 100th record. Using Jaccard-similarity.

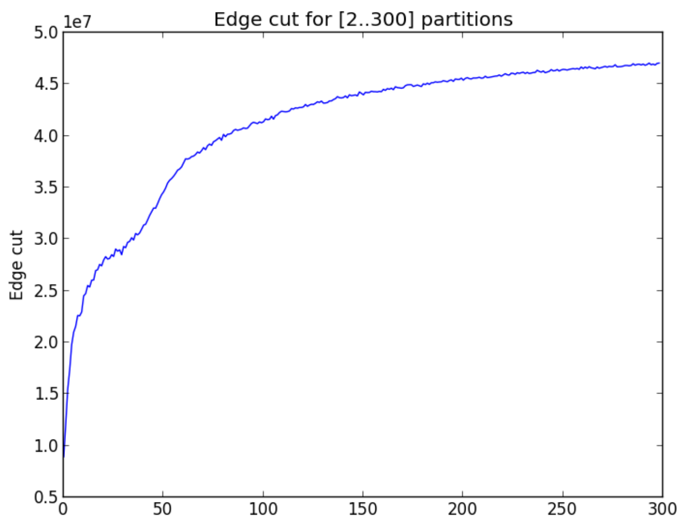


Figure 4.10: The edge cut for $[2, \ldots, 300]$ partitions. Note the bend at $N \approx 25$.

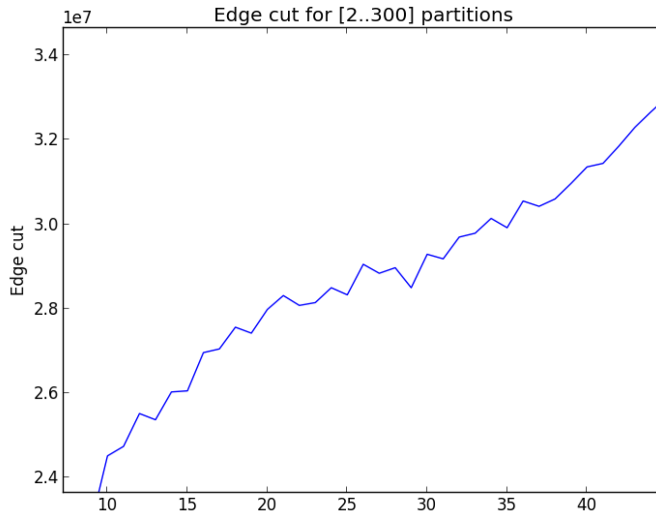


Figure 4.11: A closer look at the area of the bend.

Chapter 5
Discussion

5.1 Simple K-means

Experimentation with the K-means algorithm presented several problems. The input format increased the size from 24 megabytes to 9,2 gigabytes, or by a factor of around 383. The resulting relevance matrix is thus far too sparse to justify this approach. It was however discovered that the Weka tool does support a format specialized for sparse data, but unfortunately too late to allow repeating the experiments. Through our experiments, we also discovered that the similarity function used in K-means (Euclidean distance) was suboptimal for binary attribute values. Additionally, it became apparent that the Weka Explorer tool (a GUI front-end for the Weka library) uses an unnecessary amount of memory. Thus, a better approach would have been to use the Weka clustering libraries directly in a custom application.

5.2 Hierarchical Agglomerative Clustering

As suspected, this method proved to be too complex and expensive for the full dataset. Although both Euclidean and Jaccard distance was tested, the results of clustering with reduced dataset showed no signs of natural clustering. HAC using Jaccard distance showed puzzling results, apparently calculating a similarity of 1.0 between every record. This suggests that the method did not work properly. One reason why the method did not work might be because of lost information in the dataset reduction, but a more likely explanation is that the inherent clusters in the dataset are too vague for the algorithm to pick up.

5.3 Custom Clustering Algorithm

Although an exciting way to learn about the quirks and pitfalls of clustering, the results of this approach were not especially successful. It is apparent that the way this algorithm greedily assigns objects to clusters does not work satisfactory.

Two major problems were found: Due to its greedy nature, the algorithm might "steal" users from its true cluster. If there are N documents which are read by almost all of the users in the collection, and these documents are connected to a cluster, this cluster will act as a magnet – attracting users that belong to other user groups. In addition, these documents might create artificial links between clusters, increasing the connection strength between this magnet cluster and other clusters. This may ultimately lead to the merging of clusters which have no real connection. One way of mitigating this effect would be to exclude documents read by more than X users, however to pinpoint the value of X might prove difficult.

Another problem is that the merge function will either merge way too few clusters, producing a large number of small, useless clusters; or to often, which essentially produces one, huge cluster containing almost all the documents. A way of mitigating this could be to do one iteration of all the clusters with a given merge limit, and then decrease this limit for each subsequent iteration. However, given that this would be even less effective than regular HAC, the development was stopped at this point.

5.4 Graph partitioning

This method was the only one producing interesting results. In addition, it spawned an unexpected contribution, presented in detail in Section 4.5.4. The reason why this method of deciding the optimal number of partitions is not widespread is probably because graph partitioning is not often used in situations where the number of partitions is unknown. The most common usage for graph partitioning is within network separation and distribution of workloads in a distributed system — all situations where the number of partitions is mostly known beforehand.

The graph in Figure 4.10 shows an optimal partitioning at $N \approx 25$, which is close to the number of collections listed in Section 4.1. While it is tempting to make the connection between these two, the size of the listed collections vary greatly, while the graph partitioning inherently produces partitions of similar size. As can be seen, this might not always reflect the nature of the collection. A way to mitigate this could be to apply the hMetis tool mentioned in Section 4.4.4, experimenting with the unbalance parameter.

Exactly how accurate the partitions created by this method are, is hard to say without doing a proper evaluation, which unfortunately was not possible. In addition, it is difficult to do a coarse evaluation by manually comparing the most important documents within the partitions, because the partitions have no measure of internal degree of affiliation. For use in a recommender system, the records would therefore have to be ranked by some external ranking method.

5.5 General observations and discussion

While this thesis concerns the clustering user-record relations, the experiments are performed on a server log where the users are represented by IPs. This is of course not ideal, and may be a source of error for three reasons. Firstly, a user may access the collection from several locations, increasing their associated IPs. In addition, re-leasing of dynamically allocated IP addresses might cause a single location user to change IP. Secondly, the same DHCP issue might cause an IP to be shared by two or more users, causing the users to merge. Thirdly, several users might be hidden behind the same Network Address Translation (NAT) hub (e.g. a router), presenting them with a common IP address to the CDS server. Unfortunately, data containing actual user profiles does not exist. However, when asked about this problem, the CDS chief engineer replied that the relationship between users and IP was sufficiently permanent to avoid significant errors.

When pre-processing the server logs to retrieve a dataset for the clustering, the decision was made to focus on downloaded PDFs, stripping the log of any type of softer connection. As mentioned in Section 3.2, these types of relationships can provide valuable data for use in recommender systems. However, the clustering approach of this thesis uses a binary relationship between users and records. Thus, to ensure the quality of the dataset, only the strongest connections between users and records were selected.

Through our experiments, it was discovered that clustering both users and records was unnecessary. A connection from the users to the record clusters could be found in cheaper ways – for example by looking at the clusters associated with the user's download history. This also allows the user to be connected to more than one cluster, supporting the user with diverse interests.

The lack of a proper tool for evaluating the resulting partitions is evident. The consequence is that no definitive conclusion can be drawn on the results from the graph partitioning.

One might argue that soft, or fuzzy, clustering should be tested. While all of the clustering methods used in this experiment are in principle "hard" (in that they separate the elements into distinct clusters where an element is contained in only one cluster), there exists fuzzy, or soft, variants of each algorithm. These would allow records to belong to several clusters. The vague nature of the partitions found during the experiments might suggest that fuzzy clustering should be used. These algorithms were not included in this thesis, but should be kept in mind for further work.

Although a large number of established recommender systems are based on

explicit feedback, we have in this thesis decided to focus on implicit feedback. Even though explicit feedback often has a higher information value than implicit, it is often sporadic. This is especially true for areas where the users are not used to or expecting the need to give feedback. We believe that this applies to scientific collections, and that utilization of the existing implicit feedback is the best approach to recommender systems in this area.

Chapter 6
Conclusion

In this thesis, we have looked at three different clustering methods, and evaluated their application on a collection of scientific papers. Approximately one year of server logs from the CERN Document Server (CDS) was acquired and analyzed. The idea was to apply different clustering techniques, to see if any latent usage patterns could be discovered from the data. Three clustering methods were chosen: K-means, Hierarchical Agglomerative Clustering (HAC) and graph partitioning. In addition, a custom, agglomerative clustering algorithm was made in an attempt to tackle some of the problems encountered during the experiments with K-means and HAC.

The results from K-means clustering were poor, mostly due to a poor choice of input format, and the use of the WEKA Explorer. In addition, it was discovered that the similarity model used in K-means was not optimal for the binary type of data used in this experiment. Before running the HAC method, the data was down-sampled to $\frac{1}{100}$ of the original size. The resulting dendrograms showed no recognizable patterns, neither with the default Euclidean distance nor the Jaccard similarity measure. The custom algorithm showed interesting trends, but was ultimately too greedy to produce any satisfactory results.

As we look back on the process described in this thesis, we note a few elements that should have been handled differently. The most important element is the lacking tools of evaluation. Ideally, a test set should be crafted from the CDS collection – or a similar, large collection of scientific documents. This tool should be present before any further in-depth research on recommender systems in this area. Furthermore, the thesis would most likely have benefited from a more thorough theoretical research on clustering tools; the procedure presented shows signs of a progressive learning approach, where better knowledge of the systems used would have streamlined the research somewhat.

As a final conclusion of the thesis, the research questions from section 1.5.1 will be repeated and replied in order:

***Q1**: How can the usage logs of CDS best be processed to allow for effective analysis of usage patterns?*

The main objective of the pre-processing was to create a representation of user-record relationships that gave the best result for clustering. Given the large amount of input data given, the decision was made to exclude all but the strongest connection type, just including the entries of downloaded records. With this approach, the idea was to concentrate the dataset to a more manageable size, while maintaining the quality of the data.

***Q2**: How can clustering methods be adapted to identify user groups from automatic analysis of usage logs?*

While K-means and HAC proved not to be optimal for the clustering problem of this thesis, the graph partitioning did provide promising results. This shows the presence of at least loose natural clusters within the dataset, and suggests further experiments using fuzzy alternatives, as mentioned in Section 5.5.

***Q3**: How can the the resulting clusters of the usage logs be utilized in a way that mitigates information bubbles?*

The partitions provided by the graph partition algorithm needs an external ranking before being useful as recommendations. An intuitive way of proceeding would be to use the partitions as a document selection from which the recommendations are done, as suggested by O'Connor and Herlocker [26]. However, this introduces the problem of information bubbles, where documents are excluded from consideration based on affiliation. A better approach would be to combine the cluster basted recommendation with other, less discriminating ranking methods. In this way, new documents are more easily included in the clusters, and the *serendipity* of the system increases.

Another way of producing generalized recommendations could be by applying a collaborative ranking method on the partitions, and collecting the top ranked records from each of them. This would produce a diverse set of highly attractive records, which would help in mitigating information bubbles.

6.1 Future work

As the thesis has concluded, there are still some loose threads that might prove useful to pursue. The clustering methods chosen for this thesis are not an exhaustive representation of the available methods. As such, a further survey should be done to evaluate even more clustering methods.

One specific clustering tool that should be evaluated in further work is Cluto. It is specialized for datasets from areas such as information retrieval and customer purchasing transactions, which indeed looks promising. This was not included in this thesis, simply because it came to our attention just a few days before delivery. It is created by the same developers as the Metis tool. If further work is done on finding better clustering methods, both fuzzy clustering techniques and co-clustering should also be evaluated.

The most important task left for future work is the creation of a test set, either from the CDS, or another similar collection. With this tool, the graph partitioning method can be evaluated properly, and a decision can be made about whether to pursue this concept or not. Still, creating such a test set is a tedious task, and might in some cases not provide an absolute definitive classification.

The idea to an alternative evaluation method arose when looking at the graph in Figure 4.10. Further analysis of the shape of this graph might produce some metric describing the quality of the partitioning, based on the graph. This metric might be the degree of which the graph decreases its growth. For rapid declines, it is reasonable to believe that the resulting partition has found a natural number of clusters, where the number of edges between the clusters are low.

Furthermore, the concept of utilizing graph partitioning methods on user-content relations should be extended to other types of collections, to test whether it can be applied to digital libraries in general.

Future applications

As stated in the introduction of this thesis, the work done is considered a step on the way to implementing specialized recommender systems for open source systems. As such, this section contains some ideas for further approaching this goal. The resulting partitions of the experiments conducted in this thesis provide an unranked, unnamed classification of records. The possible applications for these partitions can be broadly grouped into two types.

Personalized hit sets The partitions can be used to generate personalized hit sets for users, based on their download history. For example, if 40% of the users downloaded documents are in cluster A, and 60% in cluster B, recommendations could be made from ranking elements from these clusters, with a default score of 0.4 and 0.6 respectively. These hit sets can be used both within recommender systems, and as additional ranking boosts within ordinary search and MLT systems. This is quite similar to the procedure of O'Connor and Herlocker [26], where clustering is used to minimize the load for applied recommender systems.

Automatic classification If evaluation of the partitions is made possible and the clusters prove good enough, it can be used as a tool for creating automatic collections or categories based on usage. This can again either be combined with tools for automatic naming of collections, or be proposed to a site administrator as collections, requiring a manual input of names.

References

[1] Gentil-Beccot A, Mele S, Holtkamp A, O'Connell HB, Brooks TC. Information Resources in High-Energy Physics: Surveying the Present Landscape and Charting the Future Course. Journal of the American Society for Information Science and Technology. 2008;60(1):150–160.

[2] Huang Z, Chung W, Ong TH, Chen H. A Graph-based Recommender System for Digital Library. In: Proceedings of the 2Nd ACM/IEEE-CS Joint Conference on Digital Libraries. JCDL '02. New York, NY, USA: ACM; 2002. p. 65–73.

[3] Invenio 0.9 release notes; 2006. Available from: https://raw.githubusercontent.com/inveniosoftware/invenio/v0.90.0/RELEASE-NOTES.

[4] Gvianishvili G, Šimko T, Caffaro J, Rajman M, Le Meur JY, Kaplun S, et al. Capturing and Analyzing User Behavior in Large Digital Libraries. In: 3rd Workshop on Very Large Digital Libraries. Glasgow; 2010. p. 24–31.

[5] CERN Document Server; 2014. Available from: http://cds.cern.ch.

[6] Top Institutional Repositories; 2014. Available from: http://repositories.webometrics.info/en/top_Inst.

[7] Baeza-Yates R, Ribeiro-Neto B. Modern Information Retrieval. 2nd ed. Edinburgh: Pearson Education Limited; 2011.

[8] Manning CD, Raghavan P, Schütze H. Introduction to Information Retrieval. New York: Cambridge University Press; 2008.

[9] Ricci, Francesco and Rokach, Lior and Shapira B. Introduction to Recommender Systems Handbook. New York: Springer US; 2011. p. 842. Available from: http://dx.doi.org/10.1007/978-0-387-85820-3_1.

[10] Linden G, Smith B, York J. Amazon.com recommendations: item-to-item collaborative filtering. Internet Computing, IEEE. 2003 Jan;7(1):76–80.

[11] Adomavicius G, Tuzhilin A. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering. 2005;17(6):734–749.

[12] Tan PN, Steinbach M, Kumar V. Introduction to Data Mining. 1st ed. Pearson international Edition. Pearson Addison Wesley; 2005.

[13] Chamberlain BL. Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations; 1998.

[14] Chevalier C, Safro I. Comparison of Coarsening Schemes for Multilevel Graph Partitioning. In: Stützle T, editor. Learning and Intelligent Optimization. Berlin, Heidelberg: Springer-Verlag; 2009. p. 191–205.

[15] Karypis G, Kumar V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing. 1998;20(1):359–392.

[16] Karypis G, Kumar V. METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0; 1995.

[17] Banerjee A, Dhillon I, Ghosh J, Merugu S, Modha DS. A Generalized Maximum Entropy Approach to Bregman Co-clustering and Matrix Approximation. The Journal of Machine Learning Research. 2007;8:1919–1986.

[18] Long B, Zhang ZM, Yu PS. Co-clustering by block value decomposition. Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining KDD 05. 2005;p. 635.

[19] George T, Merugu S. A scalable collaborative filtering framework based on co-clustering. Fifth IEEE International Conference on Data Mining (ICDM'05). 2005;.

[20] Zhang Y, Zhang M, Liu Y, Ma S. Improve Collaborative Filtering Through Bordered Block Diagonal Form Matrices. In: Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR '13. New York, NY, USA: ACM; 2013. p. 313–322.

[21] Xie Y, Phoha VV. Web User Clustering from Access Log Using Belief Function. In: Proceedings of the 1st International Conference on Knowledge Capture. K-CAP '01. New York, NY, USA: ACM; 2001. p. 202–208.

[22] Jacso P. Deflated, inflated and phantom citation counts. Online Information Review. 2006;30:297–309.

[23] Marian L, Vesely M, Rajman M, Le Meur JY. Citation graph based ranking in Invenio. Lect Notes Comput Sci. 2010 Sep;6273:236–247.

[24] Sugiyama K, Kan MY. Exploiting Potential Citation Papers in Scholarly Paper Recommendation. In: Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries; 2013. p. 153–162.

[25] Caragea C, Silvescu A, Mitra P, Giles CL. Can't See the Forest for the Trees?: A Citation Recommendation System. In: 13th ACM/IEEE-CS Joint Conference on Digital Libraries. JCDL '13. New York: ACM; 2013. p. 111–114.

[26] O'Connor M, Herlocker J. Clustering items for collaborative filtering. the Proceedings of SIGIR-2001 Workshop on. 2001;.

[27] Kazanidis I, Valsamidis S, Theodosiou T, Kontogiannis S. Proposed framework for data mining in e-learning: The case of open e-class. In: IADIS AC (2); 2009. p. 254–258.

Appendix A Source code

A.1 Custom clustering algorithm

```
from collections import defaultdict
from recordtype import recordtype
import copy
import sys
import csv
import threading

ClusterLink = recordtype('Link', 'count␣records')

class Cluster():

    # Static class variable for cluster links
    links = defaultdict(lambda: defaultdict(lambda:
        ClusterLink(0, set([]))))
    _next_id = 0
    _id_lock = threading.RLock()

    @classmethod
    def _new_id(cls):
        with cls._id_lock:
            new_id = cls._next_id
            cls._next_id += 1
        return new_id

    def __init__(self, ips, internal_connections):
        self.id = self._new_id()
        self.ips = set(ips)
        self.internal_connections = internal_connections
```

```
        self.records = defaultdict(int)
        self.deleted = False

    def __str__(self):
        return_str = "\n\nCluster "+str(self.id) +\
                     "\n=============\n"\
                     + str(len(self.ips))+" users:\n"

        return_str += "\n\n" + str(self.internal_connections
            ) + " internal connections"
        self_links = Cluster.get_links(self.id)
        if len(self_links) > 0:
            return_str += "\n\nConnected clusters: "
            for id, strength in self_links.iteritems():
                return_str += str(id) + " (" + str(strength)
                    + "), "
        return_str += "\n\n"+str(len(self.records))+" 
            records"
        return_str += "\nMost popular:\n"
        self_sorted_records = sorted(self.records.items(),
            key=lambda x: x[1], reverse=True)
        counter = 0
        for record_id, count in self_sorted_records:
            if counter > 14:
                break
            return_str += str(record_id) + " (" + str(count)
                + "), "
            counter += 1

        return return_str

    def add_ip(self, new_ip):
        self.ips.append(new_ip)

    @classmethod
    def add_link(cls, cl1, cl2, records):
        cls.links[cl1][cl2].count += 1
        cls.links[cl1][cl2].records.update(records)

    @classmethod
    def get_links(cls, start_id):
```

```
        links_dict = defaultdict(lambda: ClusterLink(0, set
            ([])))

        if start_id in cls.links:
            for linked_id, cluster_link in cls.links[
                start_id].iteritems():
                links_dict[linked_id] = cluster_link

        for link_id, linked_ids in cls.links.iteritems():
            if start_id in linked_ids:
                links_dict[link_id] = cls.links[link_id][
                    start_id]

        return links_dict

    @classmethod
    def update_links(cls, from_id, to_id):
        new_link_dict = copy.deepcopy(cls.links)

        for id_1, ids in cls.links.iteritems():

            if id_1 == from_id:
                new_link_dict[to_id].update(cls.links[
                    from_id])
                del new_link_dict[from_id]
                continue

            for id_2, count in ids.iteritems():
                if id_2 == from_id:
                    new_link_dict[to_id][id_1].count += cls.
                        links[id_1][id_2].count
                    new_link_dict[to_id][id_1].records.
                        update(cls.links[id_1][id_2].records)

                    del new_link_dict[id_1][id_2]

        cls.links = new_link_dict


class defaultlist(list):
    def __init__(self, fx):
```

```
        self._fx = fx

    def __setitem__(self, index, value):
        while len(self) <= index:
            self.append(self._fx())
        list.__setitem__(self, index, value)


def assign_to_cluster(cluster_objects, ip1, ip2, records):
    # Check every cluster to see if they contain the IP
    ip1_cluster = -1
    ip2_cluster = -1
    for cluster_object in cluster_objects:
        # Any of the IPs in the cluster? If not, continue
        if not ip1 in cluster_object.ips and not ip2 in
            cluster_object.ips:
            continue

        if ip1 in cluster_object.ips:
            ip1_cluster = cluster_object.id

            # If both are already in cluster, add records,
                +1 to internal connections and return
            if ip2 in cluster_object.ips:
                for record in records:
                    cluster_object.records[record] += 1

                cluster_object.internal_connections += 1
                return

        if ip2 in cluster_object.ips:
            ip2_cluster = cluster_object.id

    # IPs are found in separate clusters. Add a link
    if ip1_cluster >= 0 and ip2_cluster >= 0:
        Cluster.add_link(ip1_cluster, ip2_cluster, records)

        # Add records to both clusters
        for record in records:
            cluster_objects[ip1_cluster].records[record] +=
                1
```

```
                cluster_objects[ip2_cluster].records[record] +=
                    1

            # Also, merge clusters if the link is stronger than
                five
            merge_clusters(cluster_objects, ip1_cluster,
                ip2_cluster, 5)

        # IP1 found. Add IP2 to its cluster
        elif ip1_cluster >= 0:
            cluster_objects[ip1_cluster].add_ip(ip2)
            for record in records:
                cluster_objects[ip1_cluster].records[record] +=
                    1
            cluster_objects[ip1_cluster].internal_connections +=
                1

        # IP2 found. Add IP1 to its cluster
        elif ip2_cluster >= 0:
            cluster_objects[ip2_cluster].add_ip(ip1)
            for record in records:
                cluster_objects[ip2_cluster].records[record] +=
                    1
            cluster_objects[ip2_cluster].internal_connections +=
                1

        # None of the IPs are found. Create new cluster
        else:
            new_cluster = Cluster([ip1, ip2], 1)
            for record in records:
                new_cluster.records[record] += 1
            cluster_objects[new_cluster.id] = new_cluster


def merge_clusters(cluster_objects, ip1_cluster, ip2_cluster
    , limit):
    ip1_ip2_links = Cluster.get_links(ip1_cluster)[
        ip2_cluster]

    ratio = float(len(ip1_ip2_links.records) / float(
        ip1_ip2_links.count))
```

```
    if ip1_ip2_links.count < 3 or ratio < float(limit):
        print "\nRatio "+str(ratio)+" < "+str(limit)+". Not 
            merging"
        return False

    cluster1 = cluster_objects[ip1_cluster]
    cluster2 = cluster_objects[ip2_cluster]

    # Add all IPs from cluster 2 to cluster 1
    cluster1.ips.update(cluster2.ips)

    # Add all records from cluster 2 to cluster 1
    for record in cluster2.records:
        cluster1.records[record] += 1

    # Add the common records to cluster 1
    for record in ip1_ip2_links.records:
        cluster1.records[record] += 1

    # Increase the number of internal connections to the sum
         plus the number of links
    # between the clusters
    cluster1.internal_connections += (cluster2.
        internal_connections + ip1_ip2_links.count)

    # Change all cluster-links to reflect the merge
    Cluster.update_links(cluster2.id, cluster1.id)

    return True


def create_clusters(input_file):
    try:
        log = open(input_file, 'r')
    except IOError:
        print "Error: can\'t find file or read data: "+
            input_file+"\n"
        sys.exit()
    log_reader = csv.DictReader(log, delimiter=",")
```

```
# Create the data structures
ip_recs = defaultdict(set)
rec_ips = defaultdict(set)
ip_pairs = defaultdict(lambda: defaultdict(set))
clusters = defaultlist(lambda: Cluster)

print "Creating sets..."
for line in log_reader:
    ip_recs[line['IP']].add(line['Record'])
    rec_ips[line['Record']].add(line['IP'])

print "\nBuilding IP relationships..."
counter = 0
for ip, records in ip_recs.iteritems():
    for record in records:
        if ip in rec_ips[record]:
            for rel_ip in rec_ips[record]:
                if ip != rel_ip:
                    ip_pairs[ip][rel_ip].add(record)

    counter += 1
    sys.stdout.write("\rProcessed " + str(counter) + " 
        of " + str(len(ip_recs)) + " ips")
    sys.stdout.flush()


print "\n\nCreating clusters..."
pair_length = len(ip_pairs)
counter = 0
for ip, rel_ips in ip_pairs.iteritems():
    for rel_ip, records in rel_ips.iteritems():

        # If the IP <-> IP relationship has less than 3
            shared records, don't cluster them
        if len(records) < 3:
            break
        # Check every cluster to see if they contain the
             IP
        assign_to_cluster(clusters, ip, rel_ip, records)

    counter += 1
```

```
        sys.stdout.write("\rProcessed␣" + str(counter) + "␣
            of␣" + str(pair_length) + "␣ip␣pairs")
        sys.stdout.flush()

    print "\n\nFound␣%d␣clusters" % len(clusters)

    # Remove clusters with less than 25 ips
    pruned_clusters = []
    ip_lm = 25
    for id, cluster in enumerate(clusters):
        if len(cluster.ips) >= ip_lm:
            pruned_clusters.append(cluster)

    print "Removed␣" + str(len(clusters) - len(
        pruned_clusters)) + \
          "␣clusters␣(less␣than␣" + str(ip_lm) + "␣users)"

    # Print the resulting clusters
    sorted_pruned_clusters = sorted(pruned_clusters, key=
        lambda x: len(x.ips), reverse=True)
    return sorted_pruned_clusters


def main():
    for cluster in create_clusters(sys.argv[1]):
        print cluster

if __name__ == "__main__": main()
```

A.2 Scripts

In this section, all the scripts used for log pre-processing are pasted. The most interesting scripts are shown first, while smaller utility scripts follow. Please note that some inconsistencies between files might exist, as the files has changed somewhat during the process.

A.2.1 Bot removal

```
from collections import defaultdict
import csv
import sys
from utils import date_converter as dc
```

```
filename = sys.argv[1]
print "Removing␣bots␣from␣"+filename

try:
    log = open(filename, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+filename
    sys.exit()
reader = csv.DictReader(log, delimiter=',')

# count the number of lines in the input file, for use in
   progress reporting
with open(filename, "r") as f:
        file_lines = sum(1 for _ in f)


# List containing all IPs we want to exclude
blacklist = []

# per_ip[IP] = [timestamps]
per_ip = defaultdict(list)


print "Separating␣timestamps␣by␣IP..."
counter = 0
for line in reader:
    per_ip[line['IP']].append(line['Timestamp'])
    counter += 1
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()


print "\nChecking␣every␣IP␣log␣for␣100␣entries␣within␣1␣
   hours.␣Blacklist␣if␣found."
ip_count = len(per_ip)
counter = 0
for ip, timestamps in per_ip.iteritems():
```

```
    counter = 0
    if len(timestamps) > 100:
        while len(timestamps) > 100+counter:
            if dc.sec_diff(timestamps[100+counter],
                timestamps[counter]) < 3600:
                 blacklist.append(ip)
                break
    counter += 1
    percentage = int(round(100*float(counter)/float(ip_count
        )))
    sys.stdout.write("\r" + str(percentage) + "%")
    sys.stdout.flush()

# Reset the log reader
log.seek(0)
reader.__init__(log, delimiter=',')


print "\nWriting␣output␣file..."

new_file = open("sans_bots_" + filename, 'w')
new_file.writelines("Timestamp,Request,IP\n")

counter = 0
for line in reader:
    if line['IP'] not in blacklist:
        new_file.writelines(line['Timestamp'] + "," + line['
            Request'] + "," + line['IP'] + "\n")
    counter += 1
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()

print "Blacklisted:␣" + str(len(blacklist))
for ip in blacklist:
    print ip
```

A.2.2 CSV converter

```
import apachelog
```

```
import sys
import csv

# Format copied and pasted from Apache conf - use raw string
    + single quotes
log_format = r'%h␣%l␣%u␣%t␣\"%r\"␣%>s␣%B␣\"%{Referer}i\"␣
    \"%{User-Agent}i\"'


p = apachelog.parser(log_format)

for filename in sys.argv:
    if filename == "converter.py":
        continue

    print "Converting␣"+filename

    try:
        log = open(filename, 'r')
    except IOError:
        print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+
            filename+"\nContinuing..."
        continue

    new_filename = filename.split(".")[0]+".csv"

    with open(filename, "r") as f:
            file_lines = sum(1 for _ in f)
    new_file = open(new_filename, 'w')
    writer = csv.writer(new_file, delimiter=',')
    counter = 0
    fails = 0

    # First line determines attribute names
    writer.writerow(['Timestamp', 'Request', 'IP'])

    print file_lines

    for line in log:
        counter += 1
        line = line.replace(",", "␣")
```

```
        line = line.replace("%", "")
        try:
            data = p.parse(line)
        except apachelog.ApacheLogParserError:
            print "Failed␣at␣line␣" + str(counter)
            fails += 1
            continue

        writer.writerow([data['%t'], data['%r'], data['%h'
            ]])
        if counter % 10000 == 0:
            percentage = int(round(100*float(counter)/float(
                file_lines)))
            sys.stdout.write("\r" + str(percentage) + "%")
            sys.stdout.flush()


    print "Failed␣" + str(fails) + "␣out␣of␣" + str(counter)
        + "␣lines"
```

A.2.3 Extract IP → records

```
from collections import defaultdict
import sys
import csv

filename = sys.argv[1]

print "Generating␣statistics␣for␣"+filename

try:
    log = open(filename, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+filename
    sys.exit()

new_filename = "ip_recs_" + filename

ip_dict = defaultdict(set)

new_file = open(new_filename, 'w')
reader = csv.DictReader(log, delimiter=',')
```

```
for line in reader:
    ip_dict[line['IP']].add(line['Record'])


new_file.writelines("IP,Records\n")
for ip in sorted(ip_dict, key=lambda x: len(ip_dict[x]),
    reverse=True):
    line = ip + "," + ";".join(ip_dict[ip])
    line += "\n"
    new_file.writelines(line)
```

A.2.4 Extract Record → IPs

```
from collections import defaultdict
import sys
import csv


for filename in sys.argv:
    if filename == "record_hits.py":
        continue

    print "Generating statistics for "+filename

    try:
        log = open(filename, 'r')
    except IOError:
        print "Error: can\'t find file or read data: "+
            filename+"\nContinuing..."
        continue

    new_filename = "record_stats_" + filename

    record_list = defaultdict(lambda: defaultdict(int))

    new_file = open(new_filename, 'w')
    reader = csv.DictReader(log, delimiter=',')

    for line in reader:
        if not isinstance(record_list[line['Record']], dict)
            :
```

```
            record_list[line['Record']] = {line['IP'], 1}
        else:
            record_list[line['Record']][line['IP']] += 1

    for record, ips in sorted(record_list.viewitems(), key=
       lambda x: len(x[1]), reverse=True):
        line = record
        for ip, amount in ips.items():
            line += "," + ip + "," + str(amount)
        line += "\n"
        new_file.writelines(line)
```

A.2.5 Extract IPs with over 500 downloads

```
import sys
import csv


logfile = sys.argv[1]
statfile = sys.argv[2]
print "Extracting␣active␣ips␣from␣"+logfile+"␣based␣on␣"+
   statfile

try:
    stat = open(statfile, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+statfile+
       "\nContinuing..."
    sys.exit()
stat_reader = csv.DictReader(stat, delimiter=",")

try:
    log = open(logfile, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+logfile+"
       \nContinuing..."
    sys.exit()
log_reader = csv.DictReader(log, delimiter=",")


interesting_ips = []
```

```
for line in stat_reader:
    if int(line['amount']) > 500:
        interesting_ips.append(line['IP'])
    else:
        break

with open(logfile, "r") as f:
        file_lines = sum(1 for _ in f)
lines = 0

for line in log_reader:
    lines += 1
    if line['IP'] in interesting_ips:
        newfile = open(line['IP']+"_extracted_log.csv", "a")
        newfile.write(line['Timestamp']+","+line['Request']+
            ","+line['IP']+"\n")
        newfile.close()
    if lines % 1000 == 0:
        percentage = int(round(100*float(lines)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
```

A.2.6 Plot histogram of record accesses

```
from collections import defaultdict
import matplotlib.pyplot as Plt
from os import listdir
import csv
import re


import sys

arg_length = len(sys.argv) - 1
counter = -1
requests = defaultdict(int)
existing_img = []
imgdir = listdir('img')
for file in imgdir:
    if re.search(".png", file):
        file = file.replace(".png", "")
```

```
        existing_img.append(file)

for filename in sys.argv:
    counter += 1
    requests.clear()
    access_counter = 0

    if filename == "bar_chart.py":
        continue

    ip = filename.split("_")[0]
    if ip in existing_img:
        print "Image␣already␣exists"
        continue

    print "\rCreating␣plot␣" + str(counter) + "␣of␣" + str(
       arg_length) + ":␣" + ip

    try:
        log = open(filename, 'r')
    except IOError:
        print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+log+"
           \nContinuing..."
        sys.exit()
    log_reader = csv.reader(log, delimiter=",")

    for line in log_reader:
        access_counter += 1
        requests[line[1]] += 1

    print "\rDrawing␣plot"

    Plt.ylabel("Accesses")
    Plt.title("Access␣distribution")
    Plt.bar(range(len(requests)), requests.values(), log=
       True)
    Plt.savefig('img/'+ip+'.png', bbox_inches='tight')

    sys.stdout.flush()
```

A.2.7 IP pruning

```
import sys
import csv


logfile = sys.argv[1]
statfile = sys.argv[2]
print "Removing IPs from "+logfile+" based on "+statfile

try:
    stat = open(statfile, 'r')
except IOError:
    print "Error: can\'t find file or read data: "+statfile+
        "\nContinuing..."
    sys.exit()
stat_reader = csv.DictReader(stat, delimiter=",")

try:
    log = open(logfile, 'r')
except IOError:
    print "Error: can\'t find file or read data: "+logfile+"
        \nContinuing..."
    sys.exit()
log_reader = csv.DictReader(log, delimiter=",")

new_file = open("gt2ip_"+logfile, "w")

ignored_ips = set([])


with open(statfile, "r") as f:
        statfile_lines = sum(1 for _ in f)

counter = 0
print "Reading statfile"
for line in stat_reader:
    if int(line['amount']) < 3:
        ignored_ips.add(line['IP'])

    if counter % 1000 == 0:
        percentage = int(round(100*float(counter)/float(
            statfile_lines)))
```

```
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
    counter += 1

with open(logfile, "r") as f:
        logfile_lines = sum(1 for _ in f)

new_file.write("Timestamp,Record,IP\n")

counter = 0
print "\nWriting␣new␣file"
for line in log_reader:
    counter += 1
    if line['IP'] not in ignored_ips:
        new_file.write(line['Timestamp']+","+line['Record']+
            ","+line['IP']+"\n")
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            logfile_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
```

A.2.8 Record pruning

```
from collections import defaultdict
import csv
import sys


filename = sys.argv[1]
rec_ips = defaultdict(int)
count_list = []
print "Removing␣records␣from␣"+filename

try:
    file = open(filename, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+file
    sys.exit()
log_reader = csv.DictReader(file, delimiter=",")
```

```
new_file = open("gt2rec_"+filename, "w")
# count the number of lines in the input file, for use in
# progress reporting
with open(filename, "r") as f:
        file_lines = sum(1 for _ in f)


print "Reading records"
counter = 0
for line in log_reader:
    rec_ips[line['Record']] += 1

    counter += 1
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()


new_file.write("Timestamp,Record,IP\n")

file.seek(0)
log_reader.__init__(file, delimiter=",")

counter = 0
print "\nWriting new file"
for line in log_reader:
    counter += 1
    if rec_ips[line['Record']] > 2:
        new_file.write(line['Timestamp']+","+line['Record']+
            ","+line['IP']+"\n")
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
```

A.2.9 Remove duplicates

```
from collections import defaultdict
import csv
```

```
import sys


filename = sys.argv[1]
entries = defaultdict(int)
print "Removing␣duplicates␣from␣"+filename

try:
    file = open(filename, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+file
    sys.exit()
log_reader = csv.DictReader(file, delimiter=",")


new_file = open("nodup_"+filename, "w")

# count the number of lines in the input file, for use in
# progress reporting
with open(filename, "r") as f:
        file_lines = sum(1 for _ in f)


print "Reading␣records"
counter = 0
for line in log_reader:
    entries[(line['Record'], line['IP'])] += 1

    counter += 1
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()


new_file.write("Timestamp,Record,IP\n")

file.seek(0)
```

```
log_reader.__init__(file, delimiter=",")

counter = 0
print "\nWriting␣new␣file"
for line in log_reader:
    counter += 1
    if entries[(line['Record'], line['IP'])] == 1:
        new_file.write(line['Timestamp']+","+line['Record']+
            ","+line['IP']+"\n")
    if counter % 10000 == 0:
        percentage = int(round(100*float(counter)/float(
            file_lines)))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
```

A.2.10 Convert to relevance matrix

```
import sys
import csv


record_stats = sys.argv[1]
ip_stats = sys.argv[2]
output_file = sys.argv[3]

try:
    log = open(record_stats, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+
        record_stats+"\n"
    sys.exit()
rec_reader = csv.reader(log, delimiter=",")


try:
    log = open(ip_stats, 'r')
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+ip_stats+
        "\n"
    sys.exit()
ip_reader = csv.DictReader(log, delimiter=",")
```

```
print "Counting␣records..."
with open(record_stats, "r") as f:
        records_count = sum(1 for _ in f)
print "Done.\n"


print "Creating␣array␣of␣IPs..."
ip_array = []
for line in ip_reader:
    ip_array.append(line['IP'])
print "Done.\n"

print "Writing␣legend␣first␣line..."
ofh = open(output_file, "w")
first_line = "Record"
for ip in ip_array:
    first_line += "," + ip
ofh.write(first_line+"\n")
print "Done.\n"


print "Writing␣output␣file..."
counter = 0
for line in rec_reader:

    # The new line starts with record ID
    newline = "rec_"+line[0]

    # Create a set of IPs having read the record
    record_ips = set()
    i = 1
    while i < len(line):
        # We only want odd elements
        if i % 2 == 1:
            record_ips.add(line[i])
        i += 1

    # Iterate through master ip array and match for current
       record. Add 0's and 1's respectively
    for ip in ip_array:
        if ip in record_ips:
```

```
            newline += ",1"
        else:
            newline += ",0"

    ofh.write(newline+"\n")
    counter += 1

    # Print progress
    sys.stdout.write("\r" + str(counter) + "␣records␣
        processed")
    sys.stdout.flush()
```

A.2.11 Generate Metis graph

```
from collections import defaultdict, OrderedDict, Callable
import csv


import sys


class DefaultOrderedDict(OrderedDict):
    def __init__(self, default_factory=None, *a, **kw):
        if (default_factory is not None and
            not isinstance(default_factory, Callable)):
            raise TypeError('first␣argument␣must␣be␣callable
                ')
        OrderedDict.__init__(self, *a, **kw)
        self.default_factory = default_factory

    def __getitem__(self, key):
        try:
            return OrderedDict.__getitem__(self, key)
        except KeyError:
            return self.__missing__(key)

    def __missing__(self, key):
        if self.default_factory is None:
            raise KeyError(key)
        self[key] = value = self.default_factory()
        return value
```

```
    def __reduce__(self):
        if self.default_factory is None:
            args = tuple()
        else:
            args = self.default_factory,
        return type(self), args, None, None, self.items()

    def copy(self):
        return self.__copy__()

    def __copy__(self):
        return type(self)(self.default_factory, self)

    def __deepcopy__(self, memo):
        import copy
        return type(self)(self.default_factory,
                          copy.deepcopy(self.items()))
    def __repr__(self):
        return 'OrderedDefaultDict(%s, %s)' % (self.
            default_factory,
                                               OrderedDict.__repr__
                                                   (self))

ip_statfile = sys.argv[1]
rec_statfile = sys.argv[2]
newfile = sys.argv[3]

print "Generating IP graph from "+ip_statfile+" and "+
    rec_statfile

try:
    file = open(ip_statfile, 'r')
except IOError:
    print "Error: can\'t find file or read data: "+
        ip_statfile
    sys.exit()
ip_reader = csv.DictReader(file, delimiter=",")


try:
    file = open(rec_statfile, 'r')
```

```
except IOError:
    print "Error:␣can\'t␣find␣file␣or␣read␣data:␣"+
        rec_statfile
    sys.exit()
rec_reader = csv.reader(file, delimiter=",")

ip_recs = DefaultOrderedDict(set)
rec_ips = defaultdict(set)

for line in ip_reader:
    ip_recs[line['IP']].update(line['Records'].split(";"))

for line in rec_reader:
    rec_ips[line[0]] = [line[i] for i in range(len(line)) if
         i % 2 != 0]


total_edges = 0
graph = [[]]
neighbors = defaultdict(int)
rec_num = defaultdict(int)
counter = 0

print "Creating␣graph␣list..."
for rec, ips in rec_ips.iteritems():
    counter += 1
    neighbors.clear()
    nodelist = []
    rec_num[rec] = counter
    for ip in ips:
        for n_rec in ip_recs[ip]:
            if n_rec == rec:
                continue
            neighbors[n_rec] += 1

    total_edges += len(neighbors)
    for rec, count in neighbors.iteritems():
        nodelist.append(rec)
        nodelist.append(count)

    graph.append(nodelist)
```

```
    if counter % 1000 == 0:
        percentage = int(round(100*float(counter)/float(len(
            ip_recs))))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()

print "\nReplacing␣records␣with␣line␣numbers..."
counter = 0
for x, nodelist in enumerate(graph):
    counter += 1
    for y, elem in enumerate(nodelist):
        if elem in rec_num:
            graph[x][y] = rec_num[elem]

    if counter % 1000 == 0:
        percentage = int(round(100*float(counter)/float(len(
            graph))))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()

new_file = open(newfile, "w")

new_file.write(str(len(graph)-1) + "␣" + str(total_edges/2)
    + "␣001")


counter = 0
print "\nWriting␣new␣file"
for node in graph:
    counter += 1

    new_file.write("␣".join(str(x) for x in node)+"\n")

    if counter % 1000 == 0:
        percentage = int(round(100*float(counter)/float(len(
            graph))))
        sys.stdout.write("\r" + str(percentage) + "%")
        sys.stdout.flush()
```

A.2.12 Extract score from graph partitions

```
import glob
import re
import sys

numbers = re.compile(r'(\d+)')

def numericalSort(value):
    parts = numbers.split(value)
    parts[1::2] = map(int, parts[1::2])
    return parts

new_file = open("partition_scores.csv", "w")

for outfile in sorted(glob.glob("*.out"), key=numericalSort)
    :
    partitions = int(outfile.split("-")[0])
    print "Finding␣edge␣cut␣for␣partition␣"+str(partitions)
    fh = open(outfile, "r")
    lines = fh.readlines()

    try:
        edge_cut = re.search("Edgecut:␣(\d+),", lines[14])
    except IndexError:
        print "Edge␣cut␣not␣found␣for␣"+outfile
        print lines
        continue

    try:
        part_size = re.search("desired:␣(\d+),", lines[20])
    except IndexError:
        print "Partition␣size␣not␣found␣for␣"+outfile
        print lines
        continue

    score = (int(edge_cut.group(1))/partitions)/int(
        part_size.group(1))
    new_file.write(str(partitions)+","+str(int(edge_cut.
        group(1)))+"\n")
```

A.2.13 Map records to partitions

```
from collections import defaultdict
```

```
import csv
import sys

num_partitions = sys.argv[1]
try:
    test = int(num_partitions)
except ValueError:
    print "Error:␣input␣must␣be␣number"
    sys.exit()

assignment_file = "full.graph.part."+num_partitions

# The record stats file holds the order of records
# corresponding to line numbers in the partition file
record_stats_file = "
   record_stats_gt2rec_gt2ip_b_rec_sorted_nobots_pdf_full.
   csv"

print "Mapping␣records␣to␣"+num_partitions+"␣partitions"

try:
    assignment_fh = open(assignment_file, 'r')
except IOError:
    print "Error:␣no␣partition␣file␣for␣"+num_partitions+"␣
       partitions."
    sys.exit()

try:
    record_stats_fh = open(record_stats_file, 'r')
except IOError:
    print "Error:␣record␣stats␣file␣not␣found"
    sys.exit()
record_reader = csv.reader(record_stats_fh, delimiter=",")

# Setting up variables
partition_dict = defaultdict(set)       # mapping all
   records to their respective partition
assignments = []                        # a list containing
   all partition assignments

# Populating the partitioning into a list
```

```
for line in assignment_fh:
    assignments.append(line.strip())


i = 0
for line in record_reader:

    # Add each record to the partitioned assigned
    partition_dict[assignments[i]].add(line[0])

    i += 1

# Write the partition dict to file
out_file = open("mapped_"+num_partitions+"_partitions.csv",
    "w")

for partition, records in partition_dict.iteritems():
    new_line = partition + "," + ",".join(records) + "\n"
    out_file.write(new_line)
```

A.2.14 Plot edge cut values

```
from collections import defaultdict
import matplotlib.pyplot as Plt
import csv


import sys


#filename = sys.argv[1]
filename = "partition_scores.csv"

print "Plotting scores from "+filename

try:
    file = open(filename, 'r')
except IOError:
    print "Error: can\'t find file or read data: "+file
    sys.exit()
log_reader = csv.reader(file, delimiter=",")
```

```
# count the number of lines in the input file, for use in
# progress reporting
with open(filename, "r") as f:
        file_lines = sum(1 for _ in f)

scores = []

print "Reading␣scores"
for line in log_reader:
    scores.append(int(line[1]))


fig = Plt.figure()
Plt.ylabel("Edge␣cut")
Plt.title("Edge␣cut␣for␣[2..300]␣partitions")
ax = fig.add_subplot(1,1,1)
ax.plot(scores)
Plt.show()
Plt.savefig('plot'+filename.split('.')[0]+'.png',
   bbox_inches='tight')
```

Appendix B
Results

B.1 Output from k-Means, $\frac{1}{100}$ of full matrix (k=2)

```
=== Run information ===

Scheme:         weka.clusterers.SimpleKMeans -N 2 -S 5
Relation:
    e100th_matrix_nodup_gt2rec_gt2ip_b_rec_sorted_nobots_pdf_full

Instances:      657
Attributes:     70006
                [list of attributes omitted]
Test mode:      Classes to clusters evaluation on training
    data
=== Model and evaluation on training set ===


kMeans
======


Number of iterations: 2
Within cluster sum of squared errors: 4733.471036585119

Cluster centroids:

Cluster 0
        Mean/Mode:    0.0274  0.0274  0.0213  0.029   0.0259  [...]
        Std Devs:     0.1635  0.1635  0.1446  0.1678  0.159   [...]
Cluster 1
        Mean/Mode:    0       0       0       0       0       0
            [...]
```

```
        Std Devs:     0          0          0          0          0          0
            [...]

Clustered Instances

0          656  (100%)
1            1  (  0%)


Class attribute: Record
Classes to Clusters:

 0 1  <-- assigned to cluster
 1 0 | rec_1343076
 1 0 | rec_1308178
 1 0 | rec_1360177
 1 0 | rec_1279383
 1 0 | rec_1361683
 1 0 | rec_1350835
 1 0 | rec_242313
 1 0 | rec_589989
 1 0 | rec_1129810
 1 0 | rec_1343460
 1 0 | rec_746295
 1 0 | rec_1384132
 1 0 | rec_1269075
 1 0 | rec_1358865
 1 0 | rec_1308703
 1 0 | rec_1309505
 1 0 | rec_1158505
 1 0 | rec_1177416
 1 0 | rec_1344086
 1 0 | rec_1347252
 1 0 | rec_1168026
 1 0 | rec_1277654
 1 0 | rec_1359224
 1 0 | rec_1374933
 1 0 | rec_1374136
 1 0 | rec_1298856

 [ ...................]
```

```
 1 0 | rec_389306
 1 0 | rec_343956
 1 0 | rec_990939
 1 0 | rec_456902
 1 0 | rec_1010320
 1 0 | rec_315607
 1 0 | rec_508256
 1 0 | rec_1309911
 1 0 | rec_328385
 1 0 | rec_478005
 1 0 | rec_436674
 1 0 | rec_379371
 1 0 | rec_471567
 1 0 | rec_560529
 1 0 | rec_490321
 1 0 | rec_268593
 1 0 | rec_1311877

Cluster 0 <-- rec_1343076
Cluster 1 <-- rec_1378097

Incorrectly clustered instances :       655.0     99.6956 %
```

B.2 Output from custom developed algorithm

```
Creating sets...
Pruning sets...

Building IP relationships...
Processed 25304 of 25304 ips

Creating clusters...
Processed 25258 of 25258 ip pairs

Found 99 clusters
Removed 87 clusters (less than 25 users)


Cluster 2
============
326 users
```

```
325 internal connections

Connected clusters: 11 (Link(count=1, records=set
    (['1344820',
'1322645', '1334838']))), 59 (Link(count=1, records=set(
['1330358', '1330342', '1334563']))), 10 (Link(count=1,
records=set(['1362007', '1361300', '1361687']))), 35
(Link(count=1, records=set(['1361670', '1341818',
    '1330327']))),

297 records
Most popular:
1361706 (18), 1330654 (17), 1330327 (16), 1325590 (13),
1340242 (13), 1325591 (12), 1345449 (12), 1334563 (12),
1329573 (11), 1336158 (11), 1331560 (11), 1330358 (11),
1313485 (11), 1345327 (11), 1363300 (11),


Cluster 10
============
258 users

257 internal connections

Connected clusters: 9 (Link(count=1, records=set(['1322424',
'1330366', '1355699', '1333398']))), 2 (Link(count=4,
records=set(['1281330', '1370067', '1345743', '632530',
'1301521', '451614', '1340242', '1308459', '1337087',
'1283470', '1355703', '1287902', '1286306', '1313485']))),
59 (Link(count=3, records=set(['1338579', '1362003',
'1343488','1338570', '1361669', '1334563', '1281367']))),
71 (Link(count=1, records=set(['1361771', '1110290',
    '1363355']))),

113 records
Most popular:
1287902 (82), 1322424 (57), 1323900 (55), 1340242 (49),
1331186 (46), 1330366 (43), 1345743 (42), 1337782 (36),
1335395 (34), 1356587 (32), 1338575 (32), 1355703 (30),
1350835 (28), 1332217 (25), 1380298 (25),
```

```
Cluster 14
==============
66 users

65 internal connections

Connected clusters: 42 (Link(count=1, records=set
    (['1033945',
'1323812', '1353579', '1358188', '1333091']))), 12
(Link(count=1, records=set(['1356235', '1367848',
'1370447']))), 23 (Link(count=2, records=set(['1357029',
'1288822', '1357323', '1349965',  '1297646', '1333667',
'1311236']))),

70 records
Most popular:
1357319 (11), 1311236 (11), 1288822 (11), 1333668 (9),
1367848 (9), 1361729 (8), 1333091 (8), 1297646 (8),
1348441 (8), 1357325 (8), 1355423 (7), 1344501 (7),
1357025 (7), 1357036 (7), 1370447 (7),


Cluster 3
==============
56 users

78 internal connections

89 records
Most popular:
277615 (33), 454178 (21), 400319 (20), 400320 (19),
1165534 (18), 1100537 (18), 630753 (17), 593687 (9),
865929 (7), 376642 (7), 691793 (7), 828987 (5),
532789 (5), 808372 (5), 866791 (5),


Cluster 1
==============
53 users
```

```
52 internal connections

Connected clusters: 9 (Link(count=2, records=set(['1356587',
'1356196', '1358623', '1356194', '1371903', '1369487']))),
26 (Link(count=1, records=set(['1361385', '1347788',
    '1327643',
'1353583']))), 59 (Link(count=1, records=set(['1330358',
'1329491', '1347747']))), 18 (Link(count=1, records=set(
['1366381', '1335111', '1365689', '1363019']))),

36 records
Most popular:
1356196 (25), 1356194 (24), 1355704 (21), 1356190 (21),
1356189 (18), 1347747 (11), 1365689 (11), 1323316 (9),
1351506 (7), 1356587 (7), 1366381 (7), 1363019 (6),
1335111 (5), 1337785 (5), 1269912 (3),


Cluster 7
=============
50 users

14 internal connections

50 records
Most popular:
1367064 (3), 1371903 (3), 1334560 (3), 1358178 (3), 1366095
    (2),
1329467 (2), 1344425 (2), 1350791 (2), 1333398 (2), 1329851
    (2),
1337288 (2), 1334563 (2), 1327968 (2), 1335399 (2), 1337015
    (2),


Cluster 9
=============
50 users

63 internal connections
```

```
Connected clusters: 1 (Link(count=2, records=set(['1326900',
'1356587', '1350488', '1353213', '1316163', '1347747',
'1354267']))), 10 (Link(count=15, records=set(['1328033',
'1287902', '1371897', '1316163', '1328968', '1350791',
'1369613', '1329397', '1331518','1313485', '1356196',
'1356587', '1336499', '1329888', '1299479', '1324522',
'1350215', '1335399', '1337784', '1357913', '1369572',
'1337270', '1373736', '1366639', '1330366', '1337072',
'1344425', '1332198', '1353221', '1371816', '1328280',
'1281333', '1358178', '1326900', '1322424']))), 59
(Link(count=2, records=set(['1337015', '1328275', '1334563',
'1373736', '1334560']))), 9 (Link(count=5, records=set(
['1368457', '1354189', '1329467','1329851', '1369836',
'1327968', '1351541', '1337015', '1350791', '1334563',
'1351101', '1353896', '1334560', '1328275', '1344079',
'1337288']))),

119 records
Most popular:
1316163 (10), 1337015 (9), 1334560 (9), 1369836 (9),
1331518 (8), 1314226 (7), 1326900 (7), 1329888 (7),
1334563 (7), 1287902 (7), 1329826 (6), 1329467 (6),
1330366 (6), 1337072 (6), 1322424 (6),


Cluster 5
==========
38 users

33 internal connections

20 records
Most popular:
603056 (23), 235242 (20), 425460 (19), 813710 (15), 923393
    (14),
1071486 (12), 212880 (8), 115976 (8), 306421 (8), 254420 (7)
    ,
211448 (6), 402784 (5), 181071 (4), 705845 (2), 1158462 (2),


Cluster 16
```

```
===============
38 users

47 internal connections

Connected clusters: 16 (Link(count=5, records=set(['254420',
'813710', '1071486', '402784', '603056', '211448', '212880',
'1158462', '923393', '425460', '181071', '235242', '115976',
'306421', '399425'])),

24 records
Most popular:
181071 (10), 603056 (7), 254420 (7), 1158462 (6), 115976 (6)
    ,
425460 (6), 235242 (6), 212880 (5), 923393 (4), 399425 (4),
813710 (4), 179307 (4), 1071486 (3), 306421 (3), 402784 (3),


Cluster 23
===============
29 users

29 internal connections

Connected clusters: 14 (Link(count=2, records=set
    (['1357029',
'1288822', '1357323', '1349965', '1297646', '1333667',
'1311236'])),

71 records
Most popular:
1313753 (5), 1298604 (5), 1369245 (4), 1100376 (4), 1288822
    (4),
1100261 (4), 1287107 (3), 1376436 (3), 1297646 (3), 1346063
    (2),
1333554 (2), 1297663 (2), 1319039 (2), 1183835 (2), 1278153
    (2),


Cluster 17
===============
```

```
28 users

27 internal connections

29 records
Most popular:
341617 (11), 421753 (10), 362934 (8), 580346 (7), 967216 (7)
    ,
677435 (6), 1009713 (5), 1207101 (4), 339248 (4), 358331 (4)
    ,
593231 (4), 259570 (4), 702811 (4), 802428 (3), 589989 (3),


Cluster 13
============
25 users

38 internal connections

9 records
Most popular:
1012969 (38), 940929 (33), 789872 (28), 856258 (24), 919852
    (24),
856012 (16), 226990 (7),
957584 (6), 952805 (3),
```

B.3 Output from Metis, $N = 25$

```
******************************************************
METIS 5.0 Copyright 1998-13, Regents of the University of
   Minnesota
(HEAD: , Built on: Jul  1 2014, 14:23:44)
 size of idx_t: 32bits, real_t: 32bits, idx_t *: 64bits

Graph Information ---------------------------------------------------
 Name: /Users/blixhavn/PycharmProjects/CollFilter/metis/full
    .graph, #Vertices: 69981, #Edges: 46584779, #Parts: 25

Options ---------------------------------------------------------
 ptype=kway, objtype=cut, ctype=shem, rtype=greedy, iptype=
    metisrb, dbglvl=0, ufactor=1.030, no2hop=NO, minconn=NO,
     contig=NO, nooutput=NO, seed=-1, niter=10, ncuts=1

Direct k-way Partitioning ------------------------------------------
 - Edgecut: 28143116, communication volume: 1384839.

 - Balance:
     constraint #0:  1.030 out of 0.000

 - Most overweight partition:
     pid: 1, actual: 2883, desired: 2799, ratio: 1.03.

 - Subdomain connectivity: max: 24, min: 24, avg: 24.00

 - The original graph had 1243 connected components and the
    resulting partitioning after removing the cut edges has
    1294 components.
Timing Information ---------------------------------------------
  I/O:                              9.435 sec
  Partitioning:                    23.113 sec   (METIS time)
  Reporting:                        2.820 sec

Memory Information ---------------------------------------------
  Max memory used:               1441.920 MB
******************************************************
```