USING SOCIAL GRAPHS IN ONE-CLASS COLLABORATIVE FILTERING PROBLEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HAMZA KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

USING SOCIAL GRAPHS IN ONE-CLASS COLLABORATIVE FILTERING PROBLEM

submitted by **HAMZA KAYA** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Ferda Nur Alpaslan _____
Supervisor, **Computer Engineering**

Examining Committee Members:

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering, METU _____

Assoc. Prof. Dr. Ferda Nur Alpaslan
Computer Engineering, METU _____

Assist. Prof. Dr. Tolga Can
Computer Engineering, METU _____

Dr. Ayşenur Birtürk
Computer Engineering, METU _____

Dr. Orkunt Sabuncu
ORBİM Information Technologies _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: HAMZA KAYA

Signature :

ABSTRACT

USING SOCIAL GRAPHS IN ONE-CLASS COLLABORATIVE FILTERING PROBLEM

Kaya, Hamza
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Ferda Nur Alpaslan

September 2009, 94 pages

One-class collaborative filtering is a special type of collaborative filtering methods that aims to deal with datasets that lack counter-examples. In this work, we introduced social networks as a new data source to the one-class collaborative filtering (OCCF) methods and sought ways to benefit from them when dealing with OCCF problems. We divided our research into two parts. In the first part, we proposed different weighting schemes based on social graphs for some well known OCCF algorithms. One of the weighting schemes we proposed outperformed our baselines for some of the datasets we used. In the second part, we focused on the dataset differences in order to find out why our algorithm performed better on some of the datasets. We compared social graphs with the graphs of users and their neighbors generated by the k-NN algorithm. Our research showed that social graphs generated from a specialized domain better improves the recommendation performance than the social graphs generated from a more generic domain.

Keywords: One-Class Collaborative Filtering, Recommendation Systems, Social Graphs

ÖZ

TEK SINIF KOLEKTİF FİLTRELEME PROBLEMİNDE SOSYAL ÇİZGELERİN KULLANIMLARI

Kaya, Hamza
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ferda Nur Alpaslan

Eylül 2009, 94 sayfa

Tek sınıf kolektif filtreleme, kolektif filtrelemenin karşıt örneklerin olmadığı veri kümeleri ile uğraşmayı hedefleyen özel bir durumudur. Bu çalışmada, sosyal çizgeleri var olan tek sınıf kolektif filtreleme yöntemlerine yeni bir veri kaynağı türü olarak sunarak onlardan yararlanmanın yollarını aradık. İki bölüme ayırdığımız çalışmamızın ilk kısmında bazı popüler tek sınıf kolektif filtreleme algoritmaları için sosyal çizgeleri esas alan ağırlıklandırma şemaları oluşturduk. Bu şemalardan birinin bazı test veri kümeleri için referanslarımızdan daha iyi sonuç verdiğini gözlemledik. Araştırmamızın ikinci kısmında veri kümelerini inceleyerek neden daha iyi sonuç aldığımızı bulmaya çalıştık. Kullanıcıların ve k-NN algoritması ile ürettiğimiz komşularının oluşturduğu çizgeleri kullandığımız sosyal çizgeler ile karşılaştırdık. Bu kısımdaki çalışmalarımız bizi özelleşmiş alanlardan oluşturulan sosyal çizgelerin nispeten daha genel alanlardan oluşturulan sosyal çizgelere göre öneri başarısını daha arttırdığı sonucuna ulaştırdı.

Anahtar Kelimeler: Tek Sınıf Kolektif Filtreleme, Öneri Sistemleri, Sosyal Çizgeler

To My Parents

ACKNOWLEDGMENTS

I would like to thank to my supervisor Assoc. Prof. Dr. Ferda Nur Alpaslan for her guidance, advice and criticism throughout the research. I also want to thank to the other committee members for their comments and suggestions.

I would like to express my deepest gratitude to Ceren Özakıncı, who encouraged and supported me in this study.

I would also like to thank to Volkan Çetin for donating a 42″ HDTV to support my research.

Thanks are also to all of my friends for their direct or indirect help.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

1.1 RESEARCH MOTIVATION

As the World Wide Web evolves, the amount of information available to the users becomes nearly impossible to manage. The information overload problem was foreseen by Peter J. Denning in early 1980's. In the ACM President's Letter titled *Electronic Junk* [10], Denning argued that, at some time, increasing use of electronic mail will overwhelm users. This argument made in 1982 when Internet was a small child. During this time WWW turned to be a huge collection of web sites covering a wide range of areas including social networks, news achieves, movie databases as well as the preferred way of global communication. Obviously, information overload problem addressed by Denning is becoming more critical day by day.

Prior to *Web 2.0*, user collaboration and contribution was at minimum level. Major collection of web sites were used to contain static information related to a specific topic. Search engines, which are the first attempts to remedy information overload problem, were successful at some degree. With introduction of *Web 2.0* concept, people started to blog about their lives, upload images and videos, share their thoughts, create interest groups, follow their friends' activities on the Internet. In other words, *Web 2.0* brought a brand new conception to the web. WWW, with its new face, is growing much faster than anytime before and people become much more impatient about reaching the piece of information they want. With ease of communication, users tend to locate what they are looking for directly with the help of their friends rather than old-fashioned keyword-based search engines.

Other than the need of locating information, in which case user knows what he/she is looking for, users are looking for services that can filter and recommend pieces of information that

they are even not aware of it but probably be interested in. Early examples of these services includes news, movie, and music recommenders. The need for these kind of services also extended with *Web 2.0*. Bookmark filters, friend activity filters and even friend filters can be examples of new services. With introduction of *Web 3.0* or *Semantic Web* information filtering and need of such new services are going to be even more crucial.

Although *information filtering* is not a new research area and there are already several good approaches that attack information overload problem, existing approaches need to be revised and new methodologies have to be proposed as the WWW evolves. In this work we focus specifically on *one-class collaborative filtering* problem and look ways of using social network data which has a rising trend in current web. We believe that social networks will gain even more importance with their use in information filtering applications.

1.2 ORGANIZATION OF THE THESIS

This thesis is organized as follows: Chapter 2 is intended to give some background information about the recommender systems while pointing out previous works related to the recommender systems. Particularly, we give a formal definition of recommendation problem, discuss existing approaches with paying special attention to collaborative filtering and one-class classification. Finally, we close this chapter with a brief introduction to social graphs.

In Chapter 3, we give details of the first phase of our research. We present the aim of our research and the methodology that we followed. After giving details of the algorithms we had used, we discuss the empirical results we had obtained.

Chapter 4 covers the second phase of our research. We start our discussion by giving the problem definition and our methodology. After that, we present the datasets we had used and give the results of the experiments we had conducted.

Finally, Chapter 5 draw some conclusions forwarding the future developments of the work we presented.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 RECOMMENDATION SYSTEMS

2.1.1 OVERVIEW

Personal experience is the most valuable information when we have to make choices. However, most of the time, we need to look for other people's experiences that we *trust* due to lack of personal experience on alternatives [41]. Recommendation systems simply try to model this process. Recommendation systems can be viewed as specialized applications of information filtering which aims to provide a much smaller and manageable subset of a large collection of *items* that probably users will be interested in. Amazon's Item-to-item recommender [25], Netflix's Cinematch DVD recommender [32], Last.fm's music recommender [22] are well-known examples of recommendation systems.

Formally, recommendation systems try to maximize a utility function L that measures how related a given item is to a given user. Let U be the set of all users, and Y be set of all items. For a given user $u \in U$ and given item $y \in Y$, L can be defined as:

$$L : U \times Y \rightarrow S \tag{2.1}$$

where S is the subset of Y that contains relevant items for u. In most cases S desired to be a list which is ordered by relevance to the user's preferences. Recommendation problem turns to be a problem looking for valid subsets of Y for each user that maximizes users utility for each element in the set:

$$\forall u \in U,\ \forall y^{'} \in Y_{u}^{'},\ y^{'} = \underset{y \in Y}{\operatorname{argmax}}\, L(u, y) \tag{2.2}$$

Due to the information overload problem mentioned in previous chapter, it is mostly impossible for users to inspect all the items in set Y. Thus they rely on other people's experiences, or utilities, with at least a subset of Y. Recommendation systems automate this process and try to fulfill the need of searching for people that experienced some items.

Recommendation systems make the use of user inputs in order to define utility of an item to a user. As stated by [44], user inputs can be, but not limited to, user's previous purchase data, user's ratings to a given item or user's comments for that item. The most common user input used by current recommender systems is user ratings. Ratings can explicitly be expressed by the user as in the Netflix's recommendation system. However, in most cases it is much more easier to obtain implicit ratings as click or not-click, bookmark or not-bookmark [36]. In fact, as stated in [20], if a user is asked to provide ratings for the recommendation system, he/she would get a bad impression of the system and even may decline to use it. Either way, recommendation systems end up with a rating matrix containing known utilities of items to users and expected to fill missing entries of that matrix. A sample rating matrix is presented in Table 2.1.

Table 2.1: A sample rating matrix to be used by a movie recommender system. Each user may give ratings 1 to 5 to movies, where a 5 indicates that the user loves that movie. Missing entries are denoted with a dash (-).

	Usual Suspects	Once	Hero	Bridget Jones's Diary	Love Actually
User 1	5	-	-	1	2
User 2	-	-	2	5	5
User 3	-	-	4	1	-
User 4	3	3	-	-	-

Recommendation systems have been classified into three categories in literature [1]:

- *Content-based recommendation systems*: Recommendation system tries to find similar items to the ones that is known that given user was interested in. Item descriptions, user comments and other contents that describe items are used during similarity calculations.
- *Collaborative filtering*: Recommendation system tries to find users with similar taste.

- *Hybrid methods*: These systems are simply combine content-based methods with collaborative filtering methods.

Content-based recommendation systems and hybrid methods are beyond the scope of our work, thus we are going to give an introductory information only about systems that uses collaborative filtering. [2] and [6] discuss content-based recommendation systems and hybrid methods respectively. These two papers might be a good starting point for the interested readers. Also [29, 15, 37] focuses on content-based recommendation systems.

2.1.2 COLLABORATIVE FILTERING

In [17], author defines collaborative filtering as:

> "Collaborative filtering aims at learning predictive models of user preferences, interests or behavior from community data, that is, a database of available user preferences."

So collaborative filtering can be viewed as a recommendation technology that aims to learn user's tastes based on community's previous actions. Of course community covers the user, thus collaborative filtering makes use of user's previous actions as well. The main assumption of collaborative filtering is that those who agreed in the past tend to agree again in the future.

We can define collaborative filtering by using the same formal notation that we used to define the recommendation problem. Collaborative filtering uses *other* user's utility functions $L(u', y)$ to predict the value of utility of a user $L(u, y)$:

$$L(u_1, y), L(u_2, y), \ldots L(u_m, y) \rightarrow L(u, y), \text{ where } u_i \in M(u) \tag{2.3}$$

So $L(u, y)$ is a combination of several other user's utility functions. This is the *collaboration* part of the collaborative filtering. Note that a new function M is introduced in Equation (2.3). M is the *filtering* part of the collaborative filtering. It is simply responsible for selecting a *high quality* subset of all users. With the term *high quality* we refer that u relies on the tastes of users selected by M.

Several recommendation systems that are using collaborative filtering as a basis can be found in both the academia and the industry. Arguably, the most known and successful collaborative filtering application in industry is Amazon's Item-to-item recommender [25], which tries to find similar items in its inventory by comparing all items based on the users purchased them. Other than Amazon's recommender, several collaborative filtering methods had been proposed to Netflix Prize [32, 33]. Also Google News is known to be using collaborative filtering to recommend news articles to its users [7]. Although no details have been published officially, it is known that several *Web 2.0* companies like Del.icio.us [9], Yelp [51] and Last.fm [22] are also using collaborative filtering based systems at least as a part of their services. In academia, Tapestry [14] is one of the first systems developed that uses collaborative filtering. Tapestry was not an automated system. Particularly, it expected each user to identify like-minded users manually [1]. Also, GroupLens [40] and PHOAKS [48] are well known collaborative filtering applications. [44] gives a good overview of some well known collaborative filtering recommendation systems that are being used in e-commerce field.

In [5], authors grouped collaborative filtering algorithms into two distinct groups: *memory-based* and *model-based* approaches.

Memory-based algorithms use users' previous ratings to make new predictions. In a typical memory-based algorithm, an unknown rating of an item given by a user is simply aggregation of the ratings given by other users to that item. Most of the time each user is assigned a weight which affects the strength of that user in final rating calculation. Given a system with m users and n items, let $\mathbf{R}$ be the rating matrix and $R(i, j)$ denote the rating of item j given by user i. Then, rating calculation of memory-based algorithms can be generalized as:

$$R(i, j) = \sum_{i \neq a} R(a, j) W(i, a) \tag{2.4}$$

Where $\mathbf{W}$ is the $m \times m$ weight matrix that contains user similarities. $W(i, j) = 1$ means user i and user j has very similar (if not identical) tastes while $W(i, j) = 0$ denotes that user i and user j has nothing in common.

In order to understand the idea behind memory-based algorithms, *K-Nearest-Neighbour (k-NN)* is a good starting point. *k-NN* is a simple yet effective memory-based collaborative filtering algorithm. The algorithm first looks for K similar users for each user. After that, cal-

culates missing entries of matrix **R** depending on ratings of these K users. In *k-NN* algorithm, for a given user i, other than the corresponding entries of K neighbours selected, all entries of W_i are equals to zero. **W** can be defined as follows:

$$W(k,i) = \begin{cases} \text{sim}(k,i), & \text{if } k \in K \\ 0, & \text{if } k \notin K \end{cases} \tag{2.5}$$

Several methods have been proposed for computing the similarity function $sim(k,i)$. *Minkowski Distance*, *Pearson Correlation* and *Cosine Similarity* are the most popular ones.

Minkowski Distance is known as the standard metric in geometrical problems. In memory-based algorithms, a special case of *Minkowski Distance* is highly used which is also known as *Euclidean Distance*. *Euclidean Distance* is obtained when $p = 2$ for the following equation:

$$d_M = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \tag{2.6}$$

[16] defines *Pearson correlation* as "the measure of the extend to which there is a linear relationship between two variables".

Pearson correlation can be computed as shown in Equation (2.7):

$$\rho = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{X_i - \mu_x}{\sigma_x} \right) \left(\frac{Y_i - \mu_y}{\sigma_y} \right) \tag{2.7}$$

When computing *Cosine similarity* between two users, we treat user ratings as one-dimensional vectors. *Cosine similarity* of two rating vectors is accepted as the similarity between two users. *Cosine similarity* can be computed as in Equation (2.8):

$$\text{sim}(x,y) = \cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||\,||\vec{y}||} \tag{2.8}$$

Although metrics given above are three most popular metrics used in many previous works, another similarity metric is worth to be mentioned. As can be seen in later chapters, due to the problem definition of our work, *Jaccard similarity* is more suitable than *Pearson correlation* or *Cosine similarity*. The important part is that *Jaccard similarity* can be used with binary

rating vectors. *Jaccard coefficient* is known to measure the ratio of the number of shared attributes of two rating vectors to the number possessed by both vectors [18]. *Jaccard similarity* is given as in Equation (2.9).

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.9}$$

while *Jaccard distance* is defined as in Equation (2.10):

$$J_\delta(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{2.10}$$

A detailed discussion on similarity metrics can be found in [18].

Note that up to this point we have always talked about user-to-user similarities. However, without any modification, same memory-based algorithms can be used to find item-to-item similarities. In fact that is what Amazon did in its item-to-item recommender [25]. Item-to-item recommendation systems may overcome some of the issues with collaborative filtering approaches that we are going to talk in next section. [42] analyzes different item-based recommendation algorithms.

Unlike memory-based algorithms, *model-based* collaborative filtering algorithms uses the rating database to learn a model after which it uses this model to make predictions. Model-based algorithms uses the learned model to predict the ratings on unseen items [7]. In [5], authors provide one of the earliest examples of model-based approach. Authors propose two different approaches: *Cluster Models* and *Bayesian Network Model*. Most of the recent work captures multiple interests of users by classifying them into multiple clusters or classes [7]. Other than the methods proposed in [5], [7] uses *Probabilistic Latent Semantic Indexing* (PLSI) and *MinHash*, [43] uses *Latent Semantic Indexing* (LSI) and [4] uses *Latent Dirichlet Allocation*.

In this work we mainly used memory-based collaborative filtering algorithms, thus we are not going into details of model-based approaches.

Regardless of the recommendation technique choice, all recommendation systems face some issues. We are going to talk about these issues in next section.

2.1.3 COMMON PROBLEMS FACED IN RECOMMENDATION SYSTEMS

Other than some scientific applications, it is hard to find multipurpose recommendation systems, especially as a working product. Prediction methods and learning algorithms used in recommendation systems are highly customized to the service that is using the recommendation system [38]. This is due to the nature of datasets available. In fact, performance of the algorithms presented in this research area are highly dependent on the dataset it is dealing with. Although there is no generic system, all the systems developed faces a set of common problems: *Data sparsity*, *new items added to the system*, *new users added to the system*, and *computational resource requirements*.

Data sparsity is arguably the most problematic issue for all recommender systems. Usually, the number of ratings available to the system is very small compared to the number of ratings that the system is expected to predict [1]. In fact, data sparsity is the key point that raises the recommendation problem itself. If we knew all the ratings then there would be no need to compute missing ratings. Due to data sparsity, it becomes very hard for a recommendation system to mimic users' tastes. Some popular datasets that have been used by researchers are listed in Table 2.2. Unfortunately, most of the time, real world datasets are even more sparse.

Table 2.2: Popular datasets that have been used by researchers.

Dataset	**User Cnt.**	**Item Cnt.**	**Rating Cnt.**	**Density**
MovieLens	943	1,682	100,000	0.06
Book-Crossing	278,858	271,379	1,149,780	1.51×10^{-5}
Jester Jokes	73,496	100	4,100,000	0.56
EachMovie	72,916	1,628	2,811,983	0.02
Netflix	480,189	17,770	100,480,507	0.01

New items added to the system and *new users added to the system* problems are very similar. In order the recommendation systems to make good predictions about users' tastes, it has to know something about them. For instance, *k-NN* algorithm has no chance to find the neighbours of a new user unless he/she rates some items. Same problem exists for new items as well. Item similarity will simply fail, if no one has rated that item. Especially recommendation systems designed for industry need to handle these problems with care since there will always be new comers to the system and system has to make good predictions to achieve user

satisfaction. *New items* and *new users* problems can be faced on an ongoing system. A very similar problem, named *cold start*, exists for all systems that are running for the first time since initially system will have no ratings at all. [45], [39] and [52] presents ways to attack these problems.

Computational resource requirements of recommendation systems can be very critical. These requirements become crucial especially for systems that are expected to make recommendations in a specific time frame. A typical recommendation cycle can be divided into two distinct phases: *Training phase* and *prediction phase*. During the training phase, system completes intermediate computations like finding user neighbours, item clusters, etc. Later in prediction phase, system makes use of these intermediate data to complete actual predictions. Prediction phase is where end-users faces the outcomes of the recommender. The required run time of these two phases highly depends on the domain of the service. For example, a service that recommends movies to users can extend the training phase at will. In fact, most of such systems completes training phase *offline*. However, it is desirable for the system to make predictions in real time. End-user will not wait long for the system to recommend some movies to him/her. For an e-commerce service in which several new items added to the inventory and several expires at any time, it is important that the recommender system adjust accordingly since these modifications of the inventory will invalidate the training data set, thus training phase. Otherwise, inevitably, system may recommend expired items or never recommend newly added items. Recomputing the training phase is the trivial solution to this problem, if only it were a *cheap* operation. In a real-world service where the recommender is expected to make recommendations out of millions of items to millions of users, repeating the training phase is simply not affordable. For the basic nearest neighbour algorithm, system has to recompute user-to-user similarities each time a new user added to the database. Although the problems mentioned above are common among all recommendation approaches, memory-based techniques suffer more. Several techniques have been proposed to overcome this issue. Google News recommendation system tries to parallelize existing methods using their large-scale data processing paradigm MapReduce [8, 7]. In [53], authors propose instance selection techniques to reduce the training set size, thus make it more manageable. [42] and [25], focuses on item similarities instead of user similarities where number of users exceeds number of items.

2.2 ONE-CLASS CLASSIFICATION

2.2.1 OVERVIEW

Supervised learning aims to build a model of the distribution of class labels [21]. The main difference between supervised learning and unsupervised learning is that in supervised learning instances in a dataset are given with known labels. Thus, the learned function or model is expected to classify a newly seen instance into known labels (or classes). In other words, assigning a new object to one of the classes that are already known is the *classification* task [47].

In the existence of multiple classes, the classification task is named as *multi-class classification* in literature. A multi-class classification problem can be split into several *two-class classification* problems [11]. In a two-class classification task, a new object can be labeled as a member of either one class or the other. It is not possible for an object to be neither in the first class nor the second class. As the author of [47] pointed out, objects to be classified can be scattered all around the feature space, but one can expect that objects belong to same class will be located near to each other and form a cluster. Thus a normal two-class classifier can separate these clusters. However, in case of the existence of an outlier, the classifier will simply mis-classify it. This situation can be seen in Figure 2.1 where objects of the two classes are denoted with a (*) and (+) respectively. The outlier object (o) which is not a (+) is misclassified by the two-class classifier. This example brings a new problem: *outlier detection*. If we were to detect outliers prior to the classification than the trained classifier would perform much better.

Outlier detection is actually another classification problem. In this problem, classifier is expected to classify an object either as a genuine object or an outlier object [47]. The main concern of this classifier is that whether an object is genuine or not. The actual class of the outlier object does not matter. Outlier detection is also known as *one-class classification* in literature which is the main focus of our work. A one-class classifier that would find out the outliers for the example given in Figure 2.1 is shown in Figure 2.2.

Previously, we stated that one-class classifiers look whether an object is genuine or not. If we

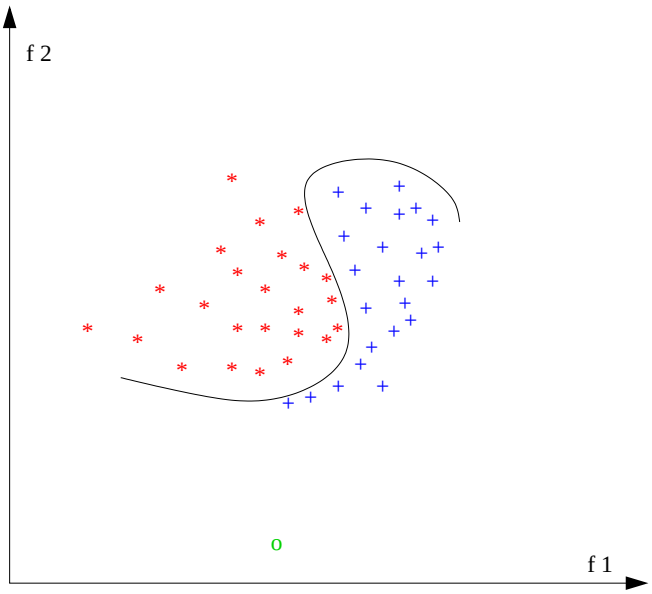

Figure 2.1: A two-class classifier applied to a dataset containing instances belong to two different classes. Each instance in dataset is represented by two features, named $f1$ and $f2$. The classifier is denoted by the solid line. The outlier (o) is misclassified as a (+) by the classifier.

think of *being genuine* and *not being genuine* as two classes, then one-class classification problem turns into a two-class classification problem, in which we try to label an object either as *genuine* or *not genuine*. However, there is an important difference between one-class and two-class classifiers which makes one-class classification problem harder and more interesting: Unlike two-class classification problems, we have examples of only one class in one-class classification problem. Thus we have a set of instances that we know they are genuine and based of these instances we try to learn a model which can recognize genuine objects.

Several methods have been proposed to solve one-class classification problem. One popular approach is to use *One-class Support Vector Machines* (SVM). In [46], authors propose to extend SVM methodology to handle one-class case. [26] applied this method to document classification problem. In [3] authors used density estimation techniques. Authors of [50] used an EM-like algorithm. A detailed list of previous works and analysis of approaches can be found in [47].

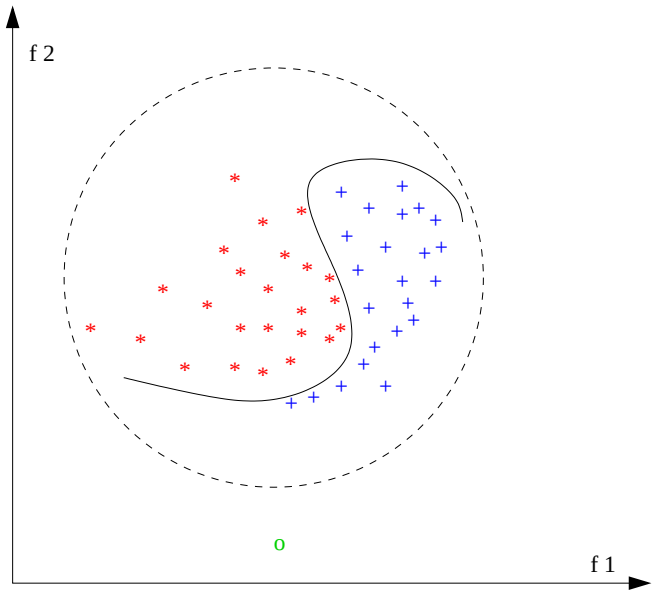

Figure 2.2: A one-class classifier applied to the same dataset used in Figure 2.1. The one-class classifier is denoted with dashes. One-class classifier effectively separates the outlier from the genuine objects.

2.2.2 ONE-CLASS COLLABORATIVE FILTERING

In previous section, we discussed recommendation systems that uses collaborative filtering. We divided collaborative filtering algorithms into two sets based on the approach they use. However, we did not make any distinction depending on the dataset available. In fact, most of previous research on collaborative filtering also did not make any distinction, depending on the assumption that both positive and negative examples exist in dataset.

For instance, in a movie recommendation service, if service lets users give explicit ratings from 1 to 5, then the recommendation problem can be seen as finding a 5-class classifier for each user. By applying the learned model to existing items, system can obtain the unknown ratings for a user. Most of the proposed methods assumes that training dataset contains at least one instance for each of the five classes.

Now, let us consider another recommendation service. This service lets its users to save their bookmarks on it. Also, depending on user's bookmarks, service makes recommendations to the user. In this case, the dataset available to the recommendation system is all bookmarks of all users. A possible portion of the actual dataset is given in Table 2.3. If a user bookmarked a

web site, we can conclude that user liked that web site, thus we put a 1 into the corresponding cell. However, if user did not bookmark a web site, either the user viewed that web site and did not find it bookmark-worthy or the user has not viewed the web site yet. User even may not be aware of the existence of such a web site. In other words, there is no way for the recommendation system to make distinction between the negative and missing positive entries in the dataset. After all, as can be seen in Table 2.3, dataset in such a system will consist of only positive (1's) and missing (dashes) entries.

Table 2.3: A sample dataset for a service that needs to use one-class collaborative filtering. A 1 in the dataset means that user has bookmarked the given web site. On the other hand, a dash (-) indicates that user has not bookmarked the web site, which means that either the user did not liked the website (negative example) or the user was not aware of that web site (actual missing data).

	Google.com	Twitter.com	Yahoo.com	Microsoft.com	Facebook.com
User 1	1	-	1	1	-
User 2	-	-	-	-	1
User 3	-	1	-	-	1
User 4	1	1	-	-	1
User 5	-	-	-	1	1

It is obvious that such a bookmarking service will face a one-class classification problem during the recommendation step. However, as authors of [36] stated, one-class collaborative filtering differs from one-class classification in that one-class collaborative filtering explores several concepts collaboratively while one-class classification aims to learn a single concept.

The key difference between traditional collaborative filtering and one-class collaborative filtering methods is that later one has only positive examples in training set [36]. However, traditional collaborative filtering methods can be used to attack one-class collaborative filtering problems. By interpreting missing data as negative examples one can obtain a dataset in which instances belongs to two classes. In fact, this approach already have been used by authors of [7]. Of course this approach will be biased as it will mark some positive examples as negative. In [27], authors focus on the issue of modeling rating distribution of missing data. Although authors worked on multi-class problems, their work can be used in one-class collaborative filtering problems as well. [36] discuss several strategies to distinguish negative examples out of missing data. In their work, authors experimentally compared several approaches including all missing data as negative, no missing data as negative and some weighting schemes to tag

an instance as negative. They had proposed a new method named *Weighted Alternating Least Squares* which is based on *Weighted Low-Rank Approximations* discussed in [31].

2.3 SOCIAL GRAPHS

In graph theory, a *graph* is a combination of a set of *points* (or *nodes*) and a set of *lines* (or *edges*) that connects pairs of points. A *social graph* or *social network* is a graph where each node represents a person and each edge represents the connection between two people. Similar to usual graphs, social graphs can be either directed or undirected. While *friendship* between two people can be represented by an undirected social graph, *fan* relationships have to be represented by a directed social graph since one does not necessarily be a fan of his/her fans. Depending on the relationship used as edges, several different types of social networks can be built like friendship, mutual support, cooperation, and similarity [35].

From now on, unless explicitly stated, a social graph will be considered to be undirected for the sake of simplicity. However, all the ideas can easily be applied to directed social graphs.

With no doubt, social graphs store a highly valuable implicit information: the *trust* among people. If a node (a person) is connected with another node than we can conclude that these people trust each other on the relationship emphasized by the edges of that graph. Especially in domain specific social graphs trust may replace the neighbours of a user that a recommender system thinks that have similar tastes with that user. For example, a list of trusted friends of a math professor drawn from his friendship social graph will not be so useful for a recommendation system that aims to recommend research papers to the professor. On the contrary, a social graph of colleagues would be extraordinarily helpful in making such recommendations. Actually, this simple example points out one common property of social graphs and recommendation systems: *collaboration*. Collaborative filtering recommendation systems is known to be a strong type of recommendation systems that tries to make recommendations based the collaboration among its users. By nature, collaboration is the main flavor of a social graph. Every day we ask for our friends' ideas and give advices on movies, songs, restaurants, brands, etc. and let the collaboration flow over the edges of our social graph.

Although it is rather a new topic, social graphs have already been analyzed vastly. However, with the advances in web itself, there is still much work to do to fully understand social graphs.

ReferralWeb is one of the earliest systems that combines social networks and collaborative filtering [19]. Unlike most of recommender systems, *ReferralWeb* does not expect its users to create their profiles. It builds those profiles using public documents found on the web. Similarly, it does not expect its users to create a social network. Instead, *ReferralWeb* attempts to uncover existing social graphs [19]. Authors of [30] used trust in recommender systems. Their method looks for similar agents to make recommendations. Also [34] and [13] discusses using trust in recommender systems. Other than these papers, [28], [24] and [35] are some research papers worth to mention.

In conclusion, we believe that people have a default trust to their friends' tastes, which collaborative filtering algorithms try to reveal. Thus, having a list of trusted people in hand, collaborative filtering methods can be improved. In fact, this is what we are going to do in the first part of our research by specially focusing on one-class collaborative filtering problem. To the best of our knowledge [36] is the most recent paper that deals with our problem. In literature, several researchers had applied traditional collaborative filtering methods to attack one-class collaborative filtering problems by assuming all missing instances as negative. However, it is obvious that such problems needs special treatment. As authors of [36] argued, observing all missing entries as negative would be highly biased. In [36], authors named these kind of problems as "One-Class Collaborative Filtering" problems and, unlike previous researchers, proposed specialized methods for this problem set. In the first part of our research, we followed the same path as the authors of [36] and proposed specialized methods to one-class collaborative filtering problem with the help of social graphs. We believe that in a perfect recommendation world, a social graph would be the same of a neighbour graph created by a collaborative filtering recommender system. In the second part of our research we focused on this issue. To explore the resemblance of social graphs and neighbour graphs, we applied one-class collaborative filtering methods to generate neighbour graphs. In literature, [23] is the only work we know that explores the properties of a neighbour graph created by a recommender system. However, [23] only inspects the neighbour graphs. They did not focused on the resemblance of that graph with an actual relation graph. By comparing the social graphs and neighbour graphs, we found that social graphs created from a specialized domain better resembles to their neighbour graphs than the social graphs created from a more generic domain.

CHAPTER 3

A NEW DATASET FOR OCCF PROBLEM: SOCIAL GRAPHS

In this chapter, we are going to give details of the first part of our research. For this part, we modified some popular memory-based collaborative filtering algorithms, that have been proposed to solve one-class collaborative filtering problem, in order to take advantage of social graph data. We empirically compared recommendation performances of new algorithms with their original forms.

In the first section we are going to state the aim of this research and the methodologies we followed. In second section, we are going to give details of the datasets that we had used. Third section will cover the details of algorithmic modifications. In this section we are going to inspect each algorithm we used separately. Finally, we will close this chapter by going over the evaluation methods we used and results we obtained.

3.1 AIM AND METHODOLOGY

For almost all of the collaborative filtering methods, a very sparse rating matrix is provided and it is expected to make accurate predictions based on this matrix. Some of them look for implicit features while others look for collaboration patterns. However, all meet at the same point: *if training set had a higher quality, we could provide more accurate predictions*. In one-class collaborative filtering applications, dataset becomes even more problematic since it only contains positive examples. In this part of our research, we introduced social graphs as a new data source for classical approaches and we looked for ways to make this data useful. Using social graph data with accordance to training dataset, we believe that one-class collaborative filtering methods can perform better.

Basically, we had used social graphs in two different ways: *pre-processing the training dataset* and *generating weighting schemes for algorithms*. Also for the *k-NN* algorithm we used social graphs in a trivial way by selecting neighbours of users directly from the social map. Pre-processing the training dataset is explained in next section while weighting schemes are going to be explained in Section 3.3.

3.2 DATASET SELECTION

Like other data mining fields, recommendation systems also have standard datasets which are being used by researchers to compare performance of their methods. There are several datasets, some of which are listed in Table 2.2. Unfortunately, in our research we were unable to use any of these standard datasets since none of them contains social relationships between its users. To test the effect of using social data we needed a dataset that contains ratings as well as user relations. Similar to the authors of [36], we chose *Del.icio.us* [9] as our base data source.

Del.icio.us is a famous bookmarking service, where users can save their bookmarks, write a short description about them and tag them. Also users can build networks by making friends or be fan of other users. Although, *Del.icio.us* officially provides an API, due to its restrictions we manually crawled it. Starting from a base tag, our crawler initially fetches all URLs labeled with that tag. After all URLs are retrieved, for each URL, crawler starts to fetch a list of users that bookmarked it and for each user, crawler fetches friends of that user. Finally, we pre-process the datasets obtained to remove users that had not bookmarked any URL and URLs that had not been bookmarked by any user. This procedure results in a $m \times n$ rating matrix $\mathbf{R}$ and a $m \times m$ adjacency matrix $\mathbf{G}$ where m is the user count and n is the URL count. Entries of $\mathbf{R}$ is defined as in Equation 3.1.

$$R(i, j) = \begin{cases} 1, & \text{if URL } j \text{ has been bookmarked by user } i \\ 0, & \text{if URL } j \text{ has not been bookmarked by user } i \end{cases} \tag{3.1}$$

Similarly, entries of $\mathbf{G}$ is defined as in Equation 3.2.

$$G(i,j) = \begin{cases} 1, & \text{if user } i \text{ is a friend of user } j \\ 0, & \text{if user } i \text{ is not a friend of user } j \end{cases} \tag{3.2}$$

For the first part of our research, we have created four different datasets. For two of them, we had used bookmarks labeled with more than one tag. Dataset properties are listed in Table 3.1. The idea behind selecting bookmarks labeled with more than one tag is that those URLs are from a more specific domain. For example, a dataset consists of URLs tagged with *blog*, *programming*, and *JavaScript* is definitely more focused than a dataset consists of URLs tagged only with *blog*. In the former case we know that the user is related with *JavaScript programming* and we can simply assume that same user will also be interested in *Python programming* rather than *cooking*. However, in the later case, we cannot make such an assumption. Users are equally likely to be interested in *Python programming*, *cooking* or *politics*.

Table 3.1: Properties of training datasets crawled from *Del.icio.us*.

Tags Used	**User Cnt.**	**URL Cnt.**	**Bookmark Cnt.**	**Density**
blog	27,142	1,296	59,888	0.0017
photography	31,085	1,273	71,910	0.0018
blog, *programming*, *java*	23,955	1,084	60,869	0.0023
blog, *programming*, *python*	14,140	929	39,579	0.0030

As can be seen in Table 3.1, datasets generated are highly sparse, which is an expected result. Due to the nature of one-class collaborative filtering problems, datasets contain only positive examples. One other expected property of our datasets is that most of the URLs had been bookmarked only a couple of times, while a small set of URLs had been bookmarked by hundreds of users. This issue is perfectly visualized by dataset histograms. Figure 3.1 shows a histogram of the dataset created using *blog* tag. Histograms of other training datasets can be found in Appendix A.1.

Del.icio.us is one of the best options to create a dataset that suits one-class collaborative filtering problem with social graphs. However, *Del.icio.us* mainly focuses on bookmarking URLs, it does not encourage its users to socialize. That is why the social graphs we had created for each dataset is even more sparse than the training datasets. The properties of

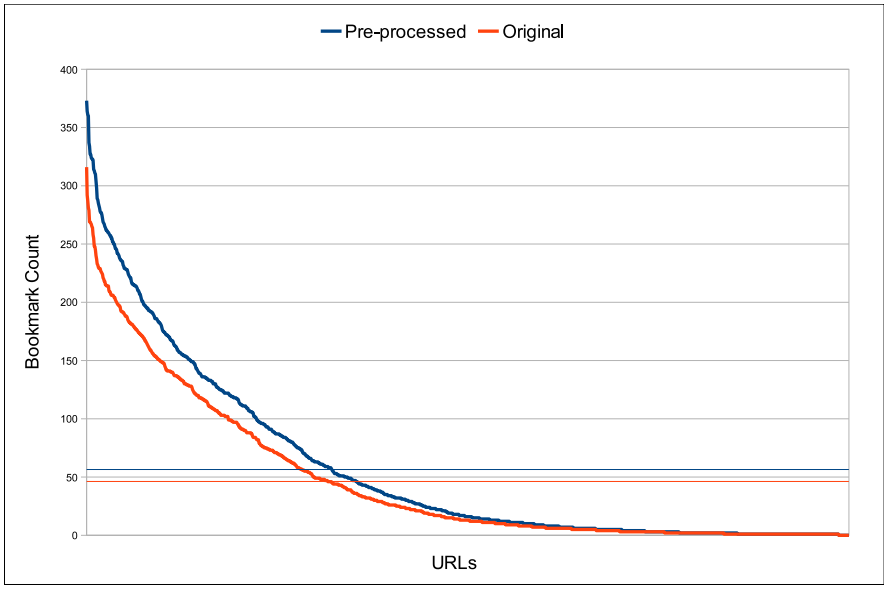

Figure 3.1: Histogram of pre-processed and original training dataset created with *blog* tag. Columns represent the number of users that bookmarked a URL. Horizontal lines represent the average bookmark count.

social graphs are listed in Table 3.2, while histogram of the social graph created using the *blog* tag is shown in Figure 3.2. To increase the visual quality of histogram, we truncated top-50 users. If we were to add those users as well, the graph would be much more left-skewed. For histograms of other social graphs refer to Appendix A.1.

Table 3.2: Properties of social graphs crawled from *Del.icio.us*.

Tags Used	**User Cnt.**	**Edge Cnt.**	**Density**
blog	27,142	71,110	0.000096
photography	31,085	87,046	0.000090
blog, *programming*, *java*	23,955	77,016	0.000134
blog, *programming*, *python*	14,140	47,090	0.000235

In order to find out how algorithms perform, we created a test set for each of training datasets. Under normal circumstances, standard test set selection methodologies could have been used. However, due to high sparsity rates, we had forced to use a custom selection technique. Nor-

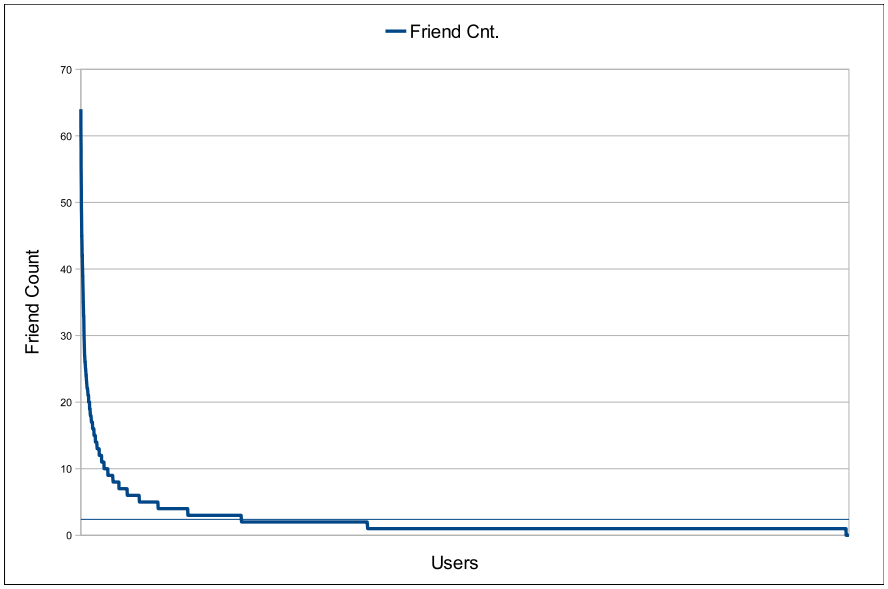

Figure 3.2: Histogram of social graph dataset created with *blog* tag. Columns represent the number of friends of a user.

mally, in *K-fold cross validation* method, one is expected to partition the training dataset into K subsamples which are generally drawn in random order. During our first attempts, we had noticed that random picks from training dataset would lead us to a training set which contains several *ghost* users that have not bookmarked any web sites. Thus we implemented a simple logic that selects less test cases from users that have less bookmarks which drastically decreased number of *ghost* users. We had selected five test sets for each dataset. During our tests, we noticed that all test cases more or less perform same. Thus we discarded the test results of other four sets and focused on a single test set for each dataset for the sake of simplicity. Similar to the previous datasets, properties of test sets are given in Table 3.3 and histogram of *blog* dataset is given in Figure 3.3. Appendix A.1 contains rest of histograms.

Table 3.1 summarizes the level of sparsity problem that we faced. Having a denser dataset could help us to make better recommendations, however we have no complaints about this issue since sparsity problem is one of the main reasons that we need a recommendation system. To remedy the sparsity problem, at least to some degree, we pre-processed training datasets

Table 3.3: Properties of test datasets.

Tags Used	User Cnt.	URL Cnt.	Test Case Cnt.
blog	27,142	1,296	5,695
photography	31,085	1,273	7,155
blog, *programming*, *java*	23,955	1,084	6,826
blog, *programming*, *python*	14,140	929	4,997

to populate them with the help of social graphs. Simply using a user's friends as neighbours, we followed a *k-NN-like* approach to populate the rating matrices. This approach differs from *k-NN* in two main aspects: neighbour selection method and prediction confidence level. Unlike the *k-NN* algorithm, we accepted positive ratings that we are *highly confident* about. As we are going to explain in the next section, weighted average of neighbours' ratings is used as predicted rating in *k-NN*. If we were to use the same prediction schema, we would result in a highly degenerated dataset. We used the confidence level to maintain the trade-off between a degenerated dataset and a denser one. We had chosen the confidence levels empirically. The resulted datasets are shown in Table 3.4. Figure 3.1 shows the histogram of the newly generated training dataset for *blog* tag as well as original *blog* training dataset.

Table 3.4: Properties of pre-processed training datasets.

Tags Used	User Cnt.	URL Cnt.	Bookmark Cnt.	Density
blog	27,142	1,296	73,191	0.0020
photography	31,085	1,273	88,953	0.0022
blog, *programming*, *java*	23,955	1,084	90,513	0.0035
blog, *programming*, *python*	14,140	929	69,082	0.0053

3.3 ALGORITHMS

There are several algorithms proposed in academia to solve the collaborative filtering problem. We had chosen three popular algorithms to focus our research on the value of social graphs for one-class collaborative filtering problem rather than on algorithmic details. Choosing well studied algorithms also makes us feel confident with the robustness of the algorithms and the results they produce. In fact, several algorithms that have been proposed perform well on a specific dataset while hardly tolerable to any dataset changes.

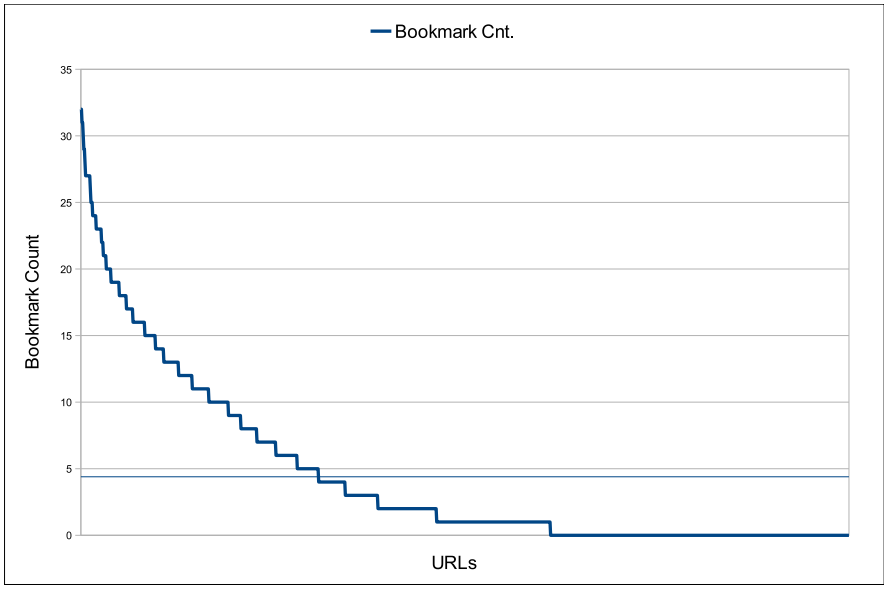

Figure 3.3: Histogram of test dataset created with *blog* tag. Columns represent the number of users that bookmarked a URL. On average we selected 5 test cases for each URL.

Although we chose popular algorithms, our results are mostly not comparable to previous research since we were unable to use a standard dataset. However, the behavior of algorithms (convergence rates, running times, memory consumption, etc.) were as expected in our test runs.

This section covers the details of the standard algorithms we had used and the modifications we made to them. We start our discussion with a de facto memory-based algorithm: *k-Nearest Neighbour*. In the second part, we will be talking about *Singular Value Decomposition*, which is a method that is highly popular nowadays. Finally, we are going to talk about *Weighted Alternating Least Squares* algorithm, a variation of *Low Rank Approximation* algorithm, proposed by the authors of [36].

3.3.1 K-NEAREST NEIGHBOUR

With no doubt, *k-Nearest Neighbour* algorithm is the most known algorithm in machine learning field. This algorithm is simple to implement and produce acceptable results for different

datasets. Basically, *k-NN* searches for k most similar users to each user, which are called *neighbours* of users. After finding neighbours for all users, algorithm makes predictions for a user based on that user's neighbours' ratings. The process of selecting neighbours is shown in Algorithm 1.

Algorithm 1 Neigbour selection process of *k-NN* algorithm.

Require: Neigbour number k, user list **U**, user rating vector set **V**
Ensure: User neighbour vector set **N**
for user u_i in **U** **do**
  Initialize priority queue Q_i
  $\vec{v}_i \leftarrow \mathbf{V}[u_i]$
  for user u_j in **U** **do**
    if $u_i \neq u_j$ **then**
      $\vec{v}_j \leftarrow \mathbf{V}[u_j]$
      $s \leftarrow \text{similarity}(\vec{v}_i, \vec{v}_j)$
      enqueue(Q_i, u_j, s)
    end if
  end for
  $\mathbf{N}[u_i] \leftarrow dequeue(Q_i, k)$
end for

Similarity metric selection is the critical part of Algorithm 1. We already discussed several popular similarity metrics in Section 2.1.2. As we stated, *Pearson Correlation* and *Cosine Similarity* have been preferred by most of the previous researchers. However, we believe that *Jaccard Similarity* is more suitable to one-class collaborative filtering applications since we are dealing with binary rating vectors. Thus, we had used *Jaccard Similarity* as defined in Equation (2.9) in the *similarity* procedure called in Algorithm 1.

Neighbour selection can be seen as the training phase of *k-NN* algorithm. Training phase can be extended to cover prediction calculations if the algorithm is expected to find predictions for all possible user-item pairs. However, generally prediction step is done in testing phase in which the algorithm is expected to calculate predictions only for test cases. The basic approach to make predictions is to equally value the ideas of neighbours. In other words, the final rating is the average rating of all neighbours. For the one class collaborative filtering problem, predicted rating of user u to item i can be formulated as in Equation (3.3).

$$\rho(u, N_u, i) = \left\| \frac{\sum_{n \in N_u} \delta(n, i)}{k} \right\| \tag{3.3}$$

where N_u is precomputed list of neighbours of user u. $\delta(n, i)$ is equal to 1 if user n rated item i and is equal to 0 otherwise.

Another common approach for making predictions using *k-NN* algorithm is that each neighbour is assigned a weight which is most of the time the similarity between the user and that neighbour. In that way, final rating can be computed as in Equation (3.4).

$$\rho(u, N_u, i) = \frac{\sum_{n \in N_u} R(n, i) W(u, n)}{\sum_{w_1} \sum_{w_2} W(w_1, w_2)} \tag{3.4}$$

where $\mathbf{R}$ is the training dataset matrix and $\mathbf{W}$ is a square matrix that stores the weights (i.e. similarities) of users.

Note that Equation (3.4) will always produce a positive real number as the predicted rating. If we were expected to find an integer rating between 1 and 5, then we would simply round the final result. However, if we had only two possible ratings, 1 and 0, then rounding would be highly biased. In fact, in our tests, this prediction schema produced very poor results. To overcome this problem, we used a normalized form of weighting which is shown in Equation (3.5). This equation simply uses the average weights of positive ratings and average weights of negative ratings to compute the predicted rating.

$$\rho(u, N_u, i) = \left| \left(\frac{\sum_{n \in N_u} \delta(n, i) W(u, n)}{\sum_{n \in N_u} \delta(n, i)} \right) - \left(\frac{\sum_{n \in N_u} (1 - \delta(n, i)) W(u, n)}{\sum_{n \in N_u} (1 - \delta(n, i))} \right) \right| \tag{3.5}$$

Until now we only talked about standard *k-NN* algorithm which is expected to run on a dataset that contains positive examples as well as negative examples. Obviously, this is not our case. In order to make classical *k-NN* algorithm function as expected, we marked all missing entries as negative which is an assumption made by many researchers [36]. Although this assumption is biased and will lead us in some false negatives, the overall effect of it will be minor since most of the people are expected to be interested in only a small set of items. In fact, this is the main cause of data sparsity.

We had run *k-NN* algorithm with some small modifications as explained above to constitute a base for other modifications we made using social graphs. The results of our experiments are given in Section 3.4.2.

Before modifying the *k-NN* algorithm further, we ran an algorithm which is a trivial combination of *k-NN* and social graphs. This algorithm basically operates as *k-NN*, other than its neighbour selection process. We directly used users' friends as neighbours and used all other *k-NN* steps as they are. Due to sparsity of social graph we did not limit neighbour count, in other words, we did not use a k value. However, for denser social graphs, usage of an appropriate k value should be considered. The results of these experiments also can be found in Section 3.4.2.

As we already stated, in this part of our research, we only focused on using social graphs in creating different weighting schemes. For *k-NN* algorithm, we had used social graphs in two different ways. First modification is done in the neighbour selection phase while second modification is in the prediction phase.

Referring to Algorithm 1, in neighbour selection phase actual decision depends on the value returned by similarity procedure call. By modifying this procedure we have the opportunity to alter the neighbour selection process. We added a new term to similarity equation to favor a user's friends during selection phase. The final similarity equation is given by Equation (3.6). In this equation, r_{u_i} is the rating vector of user u_i and $J(\vec{r}_{u_i}, \vec{r}_{u_j})$ is the Jaccard similarity between users u_i and u_j. $\eta(u_i, u_j)$ is an indicator function where its value is 1 if u_i and u_j are friends.

$$\text{similarity}(u_i, u_j) \;=\; \lambda\, J(\vec{r}_{u_i}, \vec{r}_{u_j}) + (1-\lambda)\, \eta(u_i, u_j), \text{where } 0 \leq \lambda \leq 1 \tag{3.6}$$

By deploying a new variable λ into the similarity calculation, we linearly combined the Jaccard similarity with the data we took from social graphs. In this equation, η will always result in either 0 or 1. Since Jaccard similarity between two vectors is always in the range $[0, 1]$, by forcing the λ to be in the range $[0, 1]$ also, we ensure that the new similarity value is normalized. This is an important property since, depending on the λ value, the similarity results can be highly skewed in favor of one side.

Note that λ value controls the balance between favoring social and collaborative aspects. At

first glance, equally favoring both sides seems to be reasonable. However, due to the high sparsity rates of social graphs we had used, we noticed that favoring the social graphs led us to poor results.

$\eta(u_i, u_j)$ only checks whether there is a direct connection between u_i and u_j in social graph. That is, we are looking for only depth one relationships. The idea given by Equation (3.6) can be extended to use also depth two relationships (friend of a friend). One possible formulation can be as given by Equation (3.7). We put experiments on this equation on our feature work list.

$$\text{similarity}(u_i, u_j) = (1 - \lambda - \lambda^2)\, J(\vec{r}_{u_i}, \vec{r}_{u_j}) + \lambda\, \eta(u_i, u_j) + \lambda^2\, \eta'(u_i, u_j), \text{where } 0 \leq \lambda \leq 1 \quad (3.7)$$

We made a similar use of social graphs in prediction phase. In Equation (3.5), we checked whether the average weight of positive ratings is greater than the average weight of negative ratings. In our original *k-NN* implementation, we had used user similarities as weights. By replacing this weighting schema with the one we used in Equation (3.6), we obtained the Equation (3.10).

$$\overline{w}_{pos} = \left(\frac{\sum_{n \in N_u} \delta(n, i)(\lambda\, J(\vec{r}_u, \vec{r}_n) + (1 - \lambda)\, \eta(u, n))}{\sum_{n \in N_u} \delta(n, i)} \right) \quad (3.8)$$

$$\overline{w}_{neg} = \left(\frac{\sum_{n \in N_u} (1 - \delta(n, i))(\lambda\, J(\vec{r}_u, \vec{r}_n) + (1 - \lambda)\, \eta(u, n))}{\sum_{n \in N_u} (1 - \delta(n, i))} \right) \quad (3.9)$$

$$\rho(u, N_u, i) = \left\lceil \overline{w}_{pos} - \overline{w}_{neg} \right\rceil \quad (3.10)$$

Of course, Equation (3.10) can be extended to use depth two relationships as well. However, we left this as future work, as well.

During our experiments we strictly separated two modifications in order to observe the effects of each modification clearly. For this part, we ran four versions of *k-NN*: *original k-NN*, *k-NN with neighbours taken from social graph*, *k-NN with weighted neigbour selection* and *k-NN with weighted predictions*. Results of our experiments are listed in Section 3.4.2.

3.3.2 SINGULAR VALUE DECOMPOSITION

Singular Value Decomposition is a powerful factorization method used in linear algebra. Although many application areas of *SVD* can be found in academia, our main concern is its uses in information retrieval and collaborative filtering fields. In information retrieval it is mostly mentioned with *Latent Semantic Indexing*. Also its uses in *Principal Component Analysis* are worth to mention. [49] has a good discussion on *SVD* and *PCA*. Especially in collaborative filtering area, it gains attention due to *Netflix Prize* [32, 33]. Several *SVD-based* methods have been proposed to *Netflix*, including [54]. What makes *SVD* so popular is that it can find concepts while reducing dimensionality. In other words, you have to deal with a much smaller rating matrix.

Before diving into the formal definition of *SVD* and its application in collaborative filtering, let us stop for a moment and analyze a hypothetical movie recommendation system which defines all the movies in its database with a set of predefined features. The feature set F for this system is given as follows:

$$
\begin{aligned}
F \;=\; \{ & \\
& f_1 \;\rightarrow\; \textit{genre} : \text{“crime”}, \\
& f_2 \;\rightarrow\; \textit{genre} : \text{“comedy”}, \\
& f_3 \;\rightarrow\; \textit{director} : \text{“Quentin Tarantino”}, \\
& f_4 \;\rightarrow\; \textit{director} : \text{“Stanley Kubrick”} \\
\} &
\end{aligned}
$$

This system also gets feedbacks from its users in order to model their tastes. Depending on these features and their relations to users and movies, our hypothetical system aims to recommend some worth-to-see movies to its users. Table 3.5 represents a snapshot of our system's database.

By combining feature set F with the data given in Table 3.5, our system can find out the relation between users and movies. Simply calculating the dot products of feature vectors of every user and every movie in its database, it can figure out the set of movies of interest for each user. A simple calculation is given as follows:

Table 3.5: Hypothetical taste matrix of users and movies for an example movie recommender setup. Each feature can take a value in the range [0, 5].

Users / Movies	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$
User 1	4	2	4	1
User 2	1	5	2	0
User 3	4	1	3	5
Reservoir Dogs	5	1	5	0
Monty Python and the Holy Grail	0	5	0	0
The Shining	0	0	0	5

$$
\begin{aligned}
\vec{F}_{user_1} &= \{f_1 = 4, f_2 = 2, f_3 = 4, f_4 = 1\} \\
\vec{F}_{RD} &= \{f_1 = 5, f_2 = 1, f_3 = 5, f_4 = 0\} \\
\vec{F}_{MPHG} &= \{f_1 = 0, f_2 = 5, f_3 = 0, f_4 = 0\} \\
\vec{F}_{user_1} \cdot \vec{F}_{RD} &= (4 \times 5) + (2 \times 1) + (4 \times 5) + (1 \times 0) \\
&= \mathbf{42} \\
\vec{F}_{user_1} \cdot \vec{F}_{MPHG} &= (4 \times 0) + (2 \times 5) + (4 \times 0) + (1 \times 0) \\
&= \mathbf{10}
\end{aligned}
$$

As above calculation states, $user_1$ will most probably prefer *Reservoir Dogs* rather than *Monty Python and the Holy Grail*.

Unlike the *k-NN* algorithm explained in Section 3.3.1, our system makes the use of features to predict ratings. Probably, this system will remain hypothetical for the rest of the time since it depends on a predefined feature set and explicitly ask its users to rate the importance of each feature to them. Fortunately, *SVD* can be used to draw out these features that are hidden in the rating matrix $\mathbf{R}$. Better yet, it can reveal the relation of users / movies to features automatically.

Although, *SVD* is an exact decomposition method, it can be used to approximate a matrix by two rank r matrices [54]. Formally *SVD* is defined as in Equation (3.11).

$$
\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{T}}, \text{where } \mathbf{R} \in \Re_{+}^{m \times n}, \mathbf{U} \in \Re^{m \times m}, \mathbf{\Sigma} \in \Re_{+}^{m \times n}, \mathbf{V} \in \Re^{n \times n} \tag{3.11}
$$

where $\mathbf{\Sigma}$ is a diagonal matrix which holds the singular values of this decomposition and its values are assumed to be in decreasing order.

Using Equation (3.11), $\mathbf{R}$ can be constructed exactly. However, in most cases it is enough to find an approximation of $\mathbf{R}$. This procedure is known as *Compact SVD* in literature. *Compact SVD* is formally defined as in Equation (3.12).

$$\tilde{\mathbf{R}} = \mathbf{U_r}\mathbf{\Sigma_r}\mathbf{V_r^T}, \text{where } \tilde{\mathbf{R}} \in \Re_+^{m\times n}, \mathbf{U_r} \in \Re^{m\times r}, \mathbf{\Sigma_r} \in \Re_+^{r\times r}, \mathbf{V_r} \in \Re^{r\times n} \tag{3.12}$$

By choosing $r \ll n$ in Equation (3.12), we can get a good approximation $\tilde{\mathbf{R}}$ of actual matrix $\mathbf{R}$ much efficiently. To get this approximation programatically, we only need to store two much smaller matrices $\mathbf{U_r}$ and $\mathbf{V_r^T}$ as well as a vector $\mathbf{\Sigma_r}$ that contains top r singular values of actual decomposition.

In collaborative filtering perspective, after decomposing a large rating matrix $\mathbf{R}$, we can get an acceptable approximation of ratings by truncating $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{\Sigma}$. Diagonal entries of $\mathbf{\Sigma}$ can be seen as the implicit features that are hidden in training set $\mathbf{R}$. Each positive singular value denotes the strength of that feature. $\mathbf{U}$ stores the relations of users to the features and similarly $\mathbf{V^T}$ stores the relations of items to the features. By using only the top r features in compact *SVD*, we make use of only strong features and eliminate useless features in computations.

SVD seems a good solution to recommendation problem since it can highly reduce the computational requirements of such systems. In fact, computational requirements are as important as prediction performance in real-world recommendation systems since a perfect prediction will not make any sense if it cannot be found in a reasonable time.

There is one big obstacle in front of *SVD* approach that we have not yet discussed. Up to now, we focused on finding a good approximation of $\mathbf{R}$ by decomposing it and truncating resulting matrices. What we will obtain afterwards is that a good approximation of training set $\mathbf{R}$ which we know that most of its entries is missing. If even $\mathbf{R}$ were perfect, it would not be that easy to decompose such a huge matrix. The solution of this issue is a well known method called *Expectation Maximization*. Using this method, we try to fit matrices $\mathbf{U_r}$ and $\mathbf{V_r^T}$ in order to minimize the error in Frobenius form which is given in Equation (3.13). In fact this is the solution of author of [12] to *Netflix Prize* problem.

$$\left\|\mathbf{R} - \mathbf{U_r^T}\mathbf{\Sigma_r}\mathbf{V_r}\right\|_F^2 = \sum_{ij}\left(R_{ij} - \sum_r u_{ri}\sigma_r v_{rj}\right)^2 \tag{3.13}$$

In [12], author proposed to find the ratings as in the Equation (3.14), where $\vec{u}_i^T$ and $\vec{v}_j$ are user and item feature vectors respectively.

$$\tilde{R}_{ij} = \vec{u}_i^T \cdot \vec{v}_j \tag{3.14}$$

Starting from randomly filled two feature matrices $\mathbf{U_r}$ and $\mathbf{V_r}$, training phase of *SVD* is given by the author of [12] with the following equations:

$$\epsilon_{ij} = R_{ij} - \tilde{R}_{ij} \tag{3.15}$$

$$u_{ir} = u'_{ir} + \tau(\epsilon_{ij} v_{jr} - \lambda u'_{ir}) \tag{3.16}$$

$$v_{jr} = v'_{jr} + \tau(\epsilon_{ij} u_{ir} - \lambda v'_{jr}) \tag{3.17}$$

where ϵ is the error rate, τ is the learning rate, λ is the regularization rate and u' and v' are the user and item features of previous iteration respectively. In Equation (3.16) and Equation (3.17), we cross train feature vectors. To overcome the overfitting issue, author of [12] proposed a regularization term λ. Note that, in each iteration we add previously computed feature values as well, which makes the training phase an iterative process.

In our research, we had not modified *SVD*, instead we used it as a reference point for the methods explained in Section 3.3.1 and Section 3.3.3. *SVD* has been proven to be a successful method in collaborative filtering area, however, we were unable to find any track record of a related work which applied *SVD* to a one-class collaborative filtering problem other than [36]. Thus, we wanted to observe how *SVD* will perform on a one-class collaborative filtering problem like ours.

Similar to Section 3.3.1, we made the assumption that all the missing data is negative, which results in a rating matrix $\mathbf{R}$ such that $R_{ij} \in \{0, 1\}$. Fortunately, the matrix $\mathbf{R}$ we obtained for each dataset we used were small enough to be decomposed by a linear algebra toolbox. After calculating exact $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{\Sigma}$ matrices, we truncated them to use their top r features. Using these truncated matrices we recalculated the rating matrix $\tilde{\mathbf{R}}$ and made predictions based on this matrix. Empirical results of this section can be found in Section 3.4.3.

3.3.3 WEIGHTED ALTERNATING LEAST SQUARES

Weighted Alternating Least Squares (wALS) is the method proposed in [36]. Authors proposed *wALS* directly to the one-class collaborative filtering problem. In fact, this paper is the only one we could find that propose a specialized solution to the one-class collaborative filtering problem. *wALS* is based on the *Weighted Low-Rank Approximation (wLRA)* method proposed in [31]. Both *wALS* and *wLRA* methods introduce a new matrix $\mathbf{W}$ which stores weights. Weight of a rating can be seen as the confidence level of us about that rating, which means that if a rating has a high weight, then we are highly confident that this rating is correct. Although, accommodating a weight matrix is the common idea of *wALS* and *wLRA*, the weight matrices used is the main distinguishing point between two algorithms. In [31], authors proposed a naïve weighting scheme where observed ratings had a weight of "1" while missing entries had a weight of "0". As authors of [36] discussed, this is an extreme weighting scheme in which negative missing entries and positive missing entries are not separable. Instead, they proposed some dynamic weighting schemes which constitutes the *"alternating"* part of their algorithm.

Like *SVD*, *wALS* is also based on matrix decomposition. The ultimate goal of *wALS* is to find a matrix $\tilde{\mathbf{R}} \in \mathfrak{R}_{+}^{m \times n}$ such that $\tilde{\mathbf{R}} = \mathbf{U}\mathbf{V}^{\mathbf{T}}$ where $\mathbf{U} \in \mathfrak{R}^{m \times r}$ and $\mathbf{V} \in \mathfrak{R}^{n \times r}$. $\tilde{\mathbf{R}}$ is expected to minimize the error in Frobenius form as given by Equation (3.18).

$$\epsilon = \left\| \mathbf{R} - \mathbf{U}\mathbf{V}^{\mathbf{T}} \right\|_F^2 \tag{3.18}$$

wALS is an improved version of *wLRA*. Both methods try to solve the optimization problem $\underset{\tilde{R}}{\operatorname{argmin}}\, L(\tilde{R})$, where $L(\tilde{R})$ is defined with Equation (3.19).

$$\begin{aligned} L(\tilde{R}) &= \sum_{ij} W_{ij} \left(R_{ij} - \tilde{R}_{ij} \right)^2 \\ L(U, V) &= \sum_{ij} W_{ij} \left(R_{ij} - U_i V_j^T \right)^2 \end{aligned} \tag{3.19}$$

To overcome the overfitting issue of *wLRA*, authors of [36] proposed a regularization term to Equation (3.19) which is shown in Equation (3.20).

$$L(U, V) = \left(\sum_{ij} W_{ij} \left(R_{ij} - U_i V_j^T \right)^2 \right) + \lambda \left(\|U_i\|_F^2 + \left\|V_j\right\|_F^2 \right) \tag{3.20}$$

Mathematical details of *wALS* method can be found in [36]. Also [31] includes a detailed discussion on *wLRA*. In our research, we had used the exact implementation of *wALS* as given in [36]. The main difference between our work and [36] is the weighting schemes we had used. In fact, this point is the main difference between *wALS* and *wLRA*.

As we stated in the beginning of this section, *wLRA* uses a naïve weighting scheme. *wALS* had been proposed to reduce the side effects of this naïve approach. In [36], authors assigned a "1" to weight matrix cell W_{ij} if only they had high confidence on the observed example. In other words W_{ij} would be "1" if and only if $R_{ij} = 1$. This approach is the same in *wLRA*. For non-positive entries of $\mathbf{R}$, authors proposed to lower the weights. They had set $W_{ij} \in [0, 1]$ where $R_{ij} = 0$. They had listed three different weighting schemes in order to fill actual W_{ij} values when $R_{ij} = 0$. These schemes are given in Table 3.6. The first scheme assigns a constant weight to all missing ratings. The second scheme is user-oriented. Authors of [36] explained this scheme as follows:

> "... if a user has more positive examples, it is more likely that she does not like the other items, that is, the missing data for this user is negative with higher probability."

Last scheme is item-oriented which points that if an item has less positive ratings than probably missing entries will be negative.

Table 3.6: Weighting schemes used for wALS in [36].

	$\mathbf{R_{ij} = 1}$	$\mathbf{R_{ij} = 0}$
Uniform	$W_{ij} = 1$	$W_{ij} = \delta$
User-Oriented	$W_{ij} = 1$	$W_{ij} \propto \sum_j R_{ij}$
Item-Oriented	$W_{ij} = 1$	$W_{ij} \propto m - \sum_i R_{ij}$

In our research, we used social graphs to fill the weight matrix $\mathbf{W}$ in order to exploit the *trust* between users and their friends. Similar to previous works, we assigned a "1" to the ratings that we are confident with. For all other ratings, we checked the ratings given to the same item by users' friends. This scheme posits that if a URL is bookmarked by user's all friends than that user will also be interested in that URL. Similarly, if a URL is not bookmarked by user's any friends than algorithm should block that URL to appear in recommendations. These are

two extreme cases where all friends of a user are same-minded. For other *middle* cases, we assigned a weight proportional to the *majority's thought*. The weighting scheme we had used is given as in Equation (3.21).

$$W_{i,j,d_1} = \begin{cases} 1, & \text{if } R_{ij} = 1 \\ \frac{\sum_{n \in N_i} \delta(n,j)}{n}, & \text{if } R_{ij} = 0 \end{cases} \tag{3.21}$$

where N_i is the friend list of user i, and $\delta(n, j)$ is an indicator function where its value is 1 if user n has bookmarked URL j. Note that W_{ij} will always be in the range $[0, 1]$ so we do not need any normalizations.

Equation (3.21) uses only *depth 1* relationships. We can easily extend it to make use of *depth 2* connections. *Depth 2* weights can be computed as given by Equation (3.22).

$$W_{i,j,d_2} = \frac{\sum_{n \in N_i} \sum_{k \in N_n \wedge k \neq i} \delta(k, j)}{\sum_{n \in N_i} \sum_{k \in N_n \wedge k \neq i} 1} \tag{3.22}$$

A linear combination of Equation (3.21) and Equation (3.22) would result in the desired weighting scheme. Final weighting scheme is given in Equation (3.23).

$$W_{ij} = \begin{cases} 1, & \text{if } R_{ij} = 1 \\ \lambda W_{i,j,d_1} + (1 - \lambda) W_{i,j,d_2}, & \text{if } R_{ij} = 0 \end{cases} \tag{3.23}$$

Using the *wALS* implementation given in [36] with weighting schemes defined in Equation (3.21) and Equation (3.23), we compared results with other implementations given in previous sections. Details of experimental results are given in Section 3.4.4.

3.4 EVALUATION

In previous sections of this chapter, we gave details of our research in which we seek ways to use social graph data to improve performance of some well-known collaborative filtering algorithms when applied to one-class collaborative filtering problem. As stated in Section 2.2.2, one-class collaborative filtering problems are harder to solve than multi-class recommenders

since datasets that require one-class collaborative filtering solutions contain only positive examples. With the absence of counter-examples it is hard to train algorithms. For our case, the dataset problem was even drastic. Even though we spent special effort on building high quality datasets, results were not as appealing as we expected. Our datasets listed in Table 3.1 are much more sparse than the ones listed in Table 2.2.

Unlike other recommendation methods, collaborative filtering relies on the implicit social relations. If people were not tend to agree in the future if they had agreed in the past, there would be no such notion like *collaborative filtering*. That is why we thought that social graphs may contain some valuable information that we can use to improve the recommendation performances. Unluckily enough, the first thing we had noticed at the beginning of our research is that if the system's main focus is not to lead people to socialize, then the social graph obtained from such a system will be even more sparse than the rating dataset. Table 3.2 summarizes this issue.

With a set of very sparse training datasets (Table 3.1) and a set of even more sparse social graph datasets (Table 3.2) we implemented two different groups of collaborative filtering algorithms. The first group was *k-NN* based, while the second group was based on *Weighted Alternating Least Squares* method. Table 3.7 lists all the algorithms we had used along with the abbreviations that we will use through this section.

Table 3.7: Algorithms and their abbreviations used in the first part of our research.

Abbreviation	**Algorithm**
knn-base	Basic *k-NN*
knn-sg	*k-NN* with neighbours selected from social graph
knn-sg-neigh	*k-NN* - Social graph used in neighbour selection phase
knn-sg-pred	*k-NN* - Social graph used in prediction phase
svd	*SVD* with exact decomposition using a third party software
wals-base	*wALS* implementation as in [36]
wals-sg-d1	Weight matrix filled using level one connections of social graphs
wals-sg-d2	Weight matrix filled using level one and level two connections

We had conducted two experiments for each algorithm: one with original training dataset and one with pre-processed training dataset. In the next section we are going to talk on measurement methods we had used and later on we are going to give the results of our experiments.

3.4.1 EXPERIMENT MEASUREMENT

In order to measure the performance of our algorithms, we used a method similar to standard *K-Fold Cross Validation*. The main difference between our method and cross validation is that we implemented a basic logic into the test case selection phase. Normally, in cross validation, validation datasets are randomly split into training and test datasets with a 80/20 ratio. After our initial attempts, we noticed that random selection had led us to a training dataset in which many users had no bookmarks. By selecting more test cases from users that have more bookmarks, we were able to create a more unified test dataset. Although, we had created and used 5 training/test dataset pairs for each validation dataset, we are going to list results of one training/test dataset pair for each validation dataset since the results of different dataset pairs were consistent with each other. Some properties of created test datasets are listed in Table 3.3.

In collaborative filtering applications, *Root Mean Squared Error (RMSE)* is a commonly used error measure. *RMSE* is the aggregation of residuals, which are simply the difference between the predicted value and actual values. *RMSE* can formally be defined as in Equation (3.24).

$$RMSE = \sqrt{\frac{\sum_{ij} \left(r_{ij} - \tilde{r}_{ij}\right)^2}{n}} \tag{3.24}$$

In one-class collaborative filtering problems, final values obtained would be either a “1” or a “0”. During prediction phase, any value between the range $(0, 1)$ are rounded. The effect of a predicted rating of 0.6 would be same as the effect of a predicted rating of 0.99. Thus we believe that *RMSE* does not perfectly fit in one-class collaborative filtering case. That is why we seek for another performance measure.

Note that, similar to training datasets that we had used, our test datasets were also contained only positive test cases. So we could not find out the number of *false negative* and *true negative* examples after any experiment. That is why we chose *Positive Predictive Value (PPV)* as our primary performance metric. *Predictive Value* is a measure commonly used by clinicians to interpret diagnostic test results. *PPV* indicates the probability of a patient to actually have a condition if he/she has a positive test result. For our case, *PPV* indicated that whether the user will really be interested in a URL if we recommend that URL to her/him.

PPV can be calculated as in Equation (3.25).

$$PPV = \frac{\text{number of true positives}}{\text{number of true positives + number of false positives}} \tag{3.25}$$

3.4.2 RESULTS OF KNN-BASED ALGORITHMS

In this part, we are going to discuss the results of *k-NN* based algorithms. First, we will discuss the effect of pre-processing. Later, we will go over the effect of different weighting schemes.

3.4.2.1 IMPACT OF PRE-PROCESSING

As stated in Section 3.2, we used social graphs to pre-process training datasets. Specifically, we had used *knn-sg* algorithm to populate training dataset. To measure the effect of pre-processing correctly we only applied *knn-base* algorithm on both original datasets and populated datasets. We had not used *knn-sg*, *knn-sg-pred* or *knn-sg-neigh* since we already used social graphs in pre-processing phase. We believe that re-using social graphs would mislead us. The results for *blog-programming-java* dataset is given in Figure 3.4.

Although *PPV* has a rapid growth rate for small k values in pre-processed dataset, final results was very similar. *knn-base* converged to the same *PPV* value at approximately the same k value for both the pre-processed and original dataset. Table 3.8 summarizes the performance of *knn-base* for *blog-programming-java* dataset. For our case, we can conclude that pre-processing the dataset did not help us to improve *knn-base*. Results of other datasets can be found in Appendix B.1.1.

Table 3.8: Results of *knn-base* algorithm for *blog-programming-java* dataset.

	Convergence k value	**Best PPV**
blog-programming-java	305	~ 0.62
pre-prop. *blog-programming-java*	340	~ 0.62

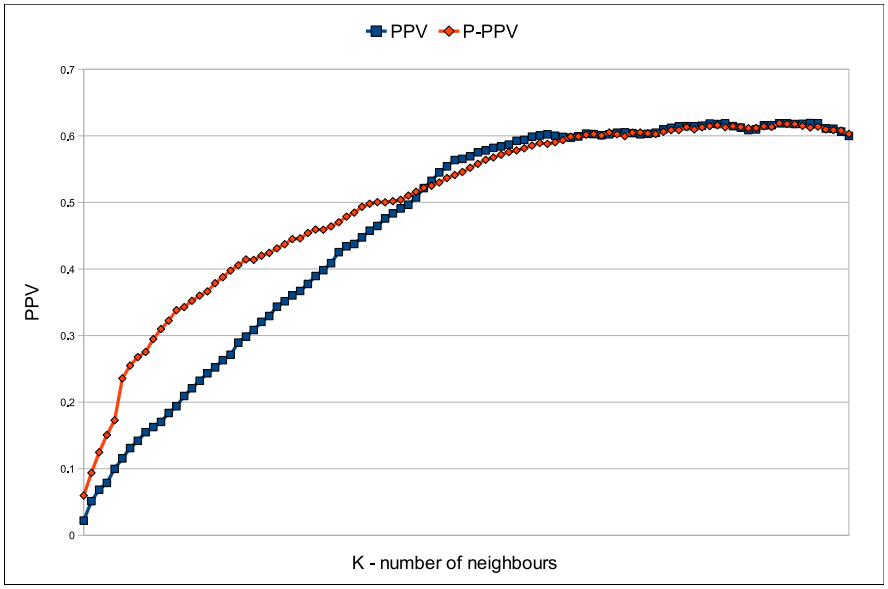

Figure 3.4: *knn-base* results for pre-processed and original datasets created using *blog*, *programming* and *java* tags. *P-PPV* stands for the *PPV* value of pre-processed dataset.

3.4.2.2 IMPACT OF WEIGHTING SCHEMES

In order to measure the impact of different weighting schemes we had tested *knn-sg-pred* and *knn-sg-neigh* algorithms as well as *knn-base* algorithm on our datasets.

knn-sg-pred algorithm blends the similarity metric used for *knn-base* with a neighborhood weight obtained from social graph. This process is formally given by Equation (3.10). Note that the λ variable in Equation (3.10) is the blending factor. In other words, λ indicates at which rate we blended Jaccard similarity with neighborhood similarity. Choosing $\lambda = 1$ will result in totally discarding neighborhood weights, while choosing $\lambda = 0$ will totally discard the effect of similarity metric we used. We tested *knn-sg-pred* algorithm for different k values and λ choices. Results of *blog-programming-java* dataset is given in Figure 3.5 while Figure 3.6 gives a more detailed view of results for *PPV* values in the range of $[0.45, 0.65]$.

Figure 3.5 clearly shows that for small λ values *knn-sg-pred* performs worse than *knn-base*.

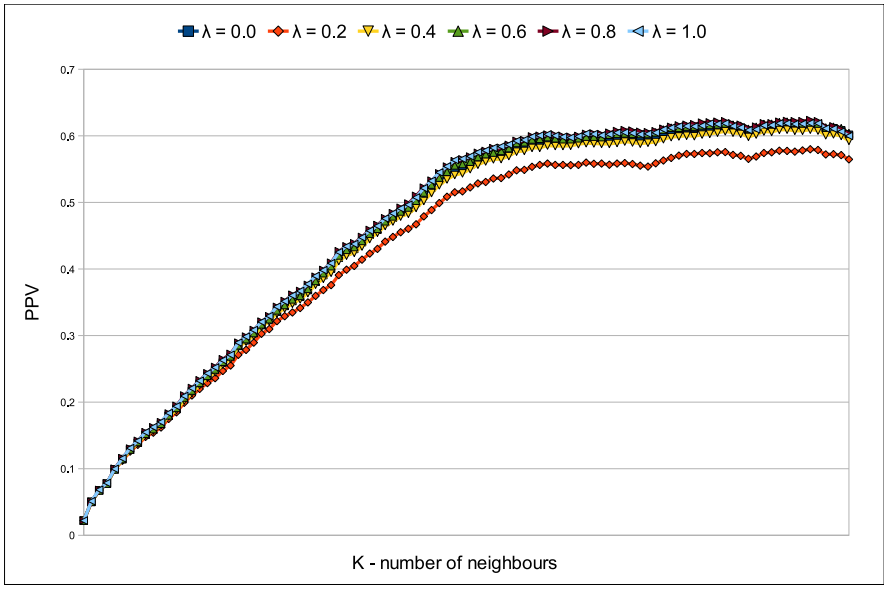

Figure 3.5: *knn-sg-pred* results for *blog-programming-java* dataset for different λ values.

Mixing similarity metric with neighborhood weights in favor of social maps produced worst results, while two extreme cases performed better (where $\lambda = 1$ and $\lambda = 0$). For all datasets we tested, $\lambda = 0.8$ outperformed rest of the λ choices. However, the overall performance gain was too small which made us think that it is not worth to modify the *knn-base*. Modification requires handling the social graph data as well, which is usually a matrix larger than rating matrix. Results of *knn-sg-pred* over other datasets is given in Appendix B.1.2.

We also tested *knn-sg-pred* on pre-processed datasets. Generally speaking, pre-processing did not affect the performance of *knn-sg-pred*. As can be seen from Figure 3.7, results are pretty similar to results of original dataset. The only difference is that *PPV* growth rate is higher for smaller k values for pre-processed dataset. In fact, this is the same result we obtained from the tests we discussed in Section 3.4.2.1. Figure 3.8 shows a detailed view of *knn-sg-pred* algorithm's behavior on pre-processed dataset. Results of other datasets are listed in Appendix B.1.2.

Second weighting scheme we had used was *knn-sg-neigh*, which follows a similar approach

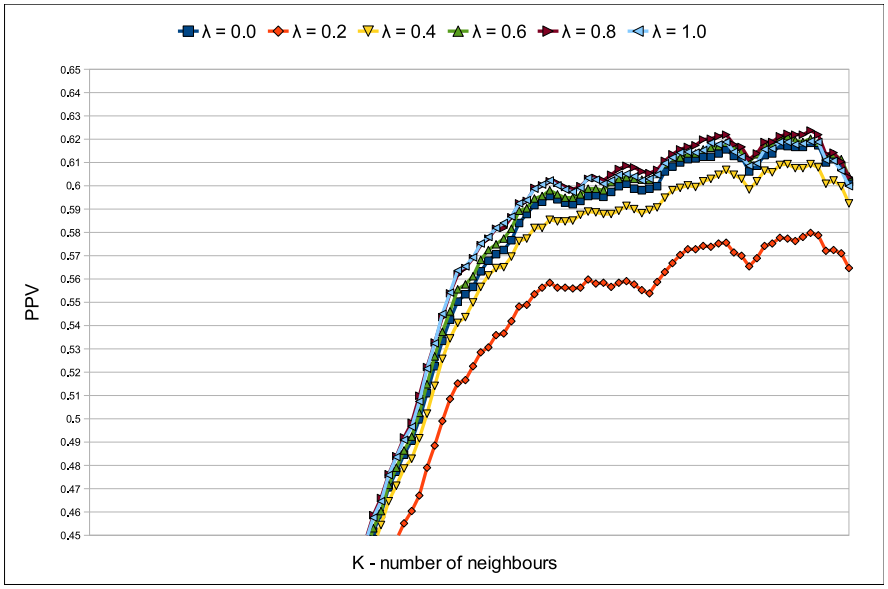

Figure 3.6: A detailed view of Figure 3.5 for $PPV \in [0.45, 0.65]$

as *knn-sg-pred*. Unlike *knn-sg-pred*, *knn-sg-neigh* uses the blending during the neighbour selection phase. This algorithm blends neighborhood weight with Jaccard similarity at a rate of λ (Equation (3.6)). We had run *knn-sg-neigh* on all of our datasets as well as their pre-processed versions. The results of this algorithm for *blog-programming-java* dataset is given in Figure 3.9. And a closer view is given in Figure 3.10.

In our experiments, *knn-sg-neigh* produced interesting results. Unlike previous experiments, for $\lambda = 1$, algorithm performed worst. Although, $\lambda = 0$ outperformed $\lambda = 1$, it was not the winning λ choice either. For this dataset, $\lambda = 0.6$ seems to outperform rest. This situation can be seen in Figure 3.10. Also the best *PPV* value obtained from $\lambda = 0.6$ is 0.68 while this value decreases to 0.62 for $\lambda = 1$ which is the performance of *knn-base*. For *blog-programming-java dataset*, *knn-sg-neigh* algorithm improved *PPV* value of *knn-base* by almost 10%.

Another important note on *knn-sg-neigh* experiments is that, the algorithm's behavior changed based on dataset choice. With *blog-programming-python* dataset, best *PPV* value improved by 6.5% (Figure B.10), while with *blog* (Figure B.12) and *photography* (Figure B.14) datasets

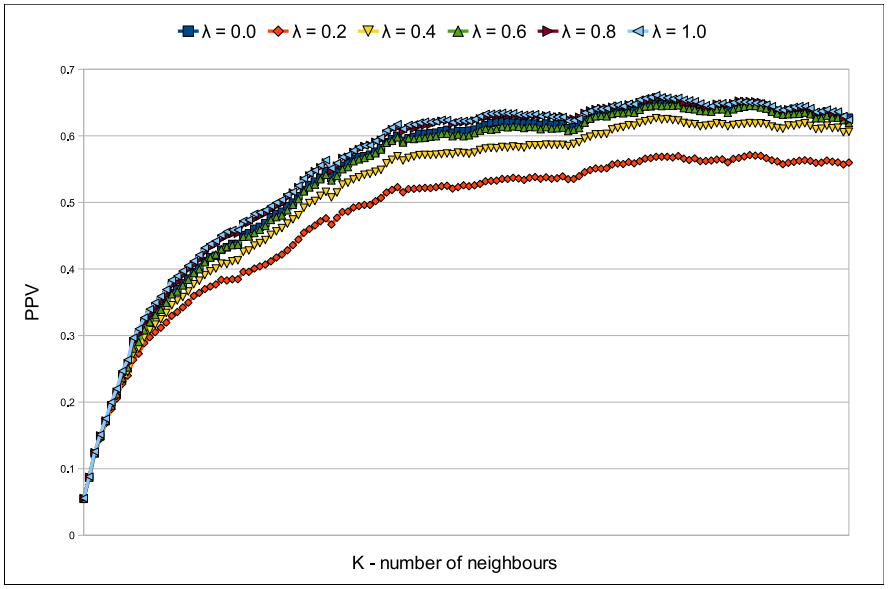

Figure 3.7: *knn-sg-pred* results for pre-processed *blog-programming-java* dataset for different λ values.

almost no improvement observed. Also for these two datasets, the best λ choice seems to be $\lambda = 0.8$. We believe that, this is due to the internal dynamics of a social environment. In a system like *Del.icio.us*, a social graph formed around a single tag represents loose connections between its users. However, a graph created from several tags (i.e. *blog*, *programming*, *java*) represents tight connections. In other words, the more tags users provided for their bookmarks, the deeper knowledge we obtain from that social graph. In fact, this issue is the main focus of second part of our research that we are going to discuss in next chapter.

Results of *knn-sg-neigh* algorithm for other datasets can be found in Appendix B.1.2.

knn-sg-neigh performed better than *knn-base* and *knn-sg-pred* for *blog-programming-python* and *blog-programming-java* datasets. For these two datasets $\lambda = 0.6$ seems to be the best choice for *knn-sg-neigh*. However, for other datasets $\lambda = 0.8$ and $\lambda = 1$ produced best results. In fact, these λ choices are the best ones for other algorithms as well.

Accommodating social graphs into the neighbour selection phase seems reasonable only if the

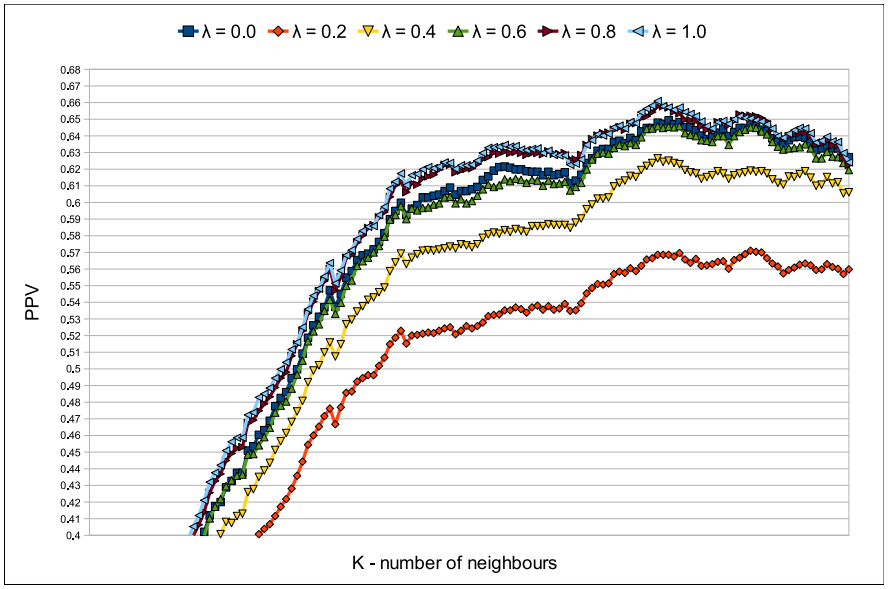

Figure 3.8: A detailed view of Figure 3.7

social graph is *strong* enough. Otherwise, the social graph itself may not be as powerful as to change the results. This idea is supported by the results of *knn-sg-neigh* algorithm applied to pre-processed dataset. We had pre-processed datasets using the *knn-sg* algorithm by which datasets are populated according to their social graphs. After this phase, we again used social graphs to select neighbours. So running *knn-sg-neigh* on a pre-processed dataset means that we had used social graphs twice. Doubling the use of social graphs can be thought as doubling the effect of social graphs. With double effect of social graphs, we were able to change the results. *knn-sg-neigh* algorithm's results on pre-processed *blog-programming-java* dataset is given in Figure 3.11. Unlike the results given in Figure 3.10, the *PPV* values for experiments $\lambda \neq 1$ are very close to each other. Results are similar for other datasets as well. However, we cannot conclude that using pre-processed datasets always produce better results. As can be seen from the results, the overall effect of pre-processing is minimal. It did not help improving the *PPV* value, but made the algorithm provide closer results for $\lambda \neq 1$.

Table 3.9 summarizes the results of all *k-NN* based algorithms for *blog-programming-java* dataset. *knn-sg-neigh* seems to outperform rest of the algorithms. And best results obtained by

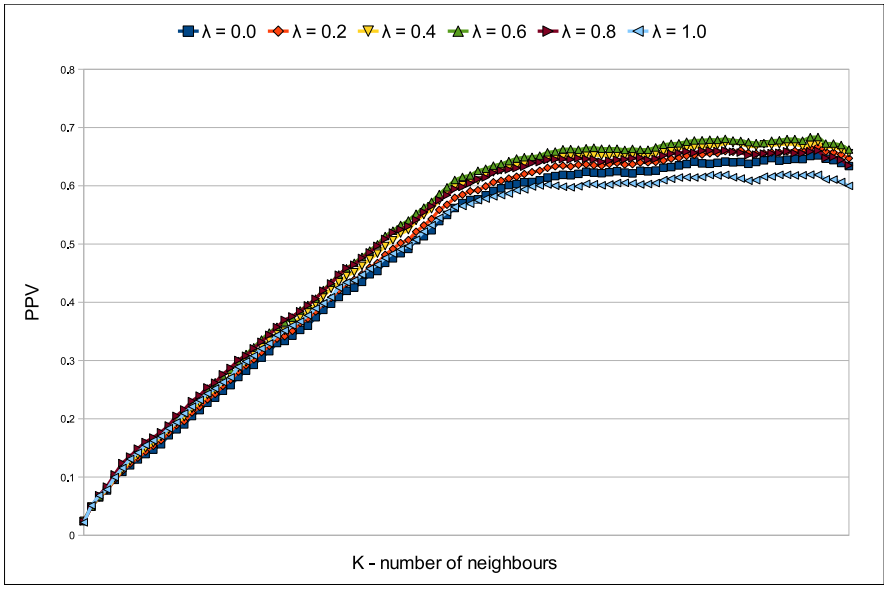

Figure 3.9: *knn-sg-neigh* results for *blog-programming-java* dataset for different λ values.

merging the usual similarities used in *knn-base* ($\lambda = 1$) and similarities used in *knn-sg* ($\lambda = 0$). Also *knn-sg-neigh* produced best results on *blog-programming-java* and *blog-programming-python* datasets when compared to other datasets. This is due to the fact that these datasets are more domain specific and less sparse. Although pre-processing the dataset slightly improved results for *blog-programming-java* dataset, it was not that helpful for other datasets. We believe that pre-processing would help in increasing *PPV* values if social graphs were less sparse. Generally, in one-class collaborative filtering problems training datasets are much more sparse than other collaborative filtering problems. Also missing counter examples makes it even harder to predict ratings. A dense social graph would be of great use to overcome these issues. Unfortunately, for our case, social graphs were even more sparse than training datasets, so they were not so helpful.

In conclusion, social graphs may help us improve the performance of *k-NN* algorithm when applied to one-class collaborative filtering problems. However, the performance gain highly depend on the quality of social graph itself. *knn-sg-neigh* algorithm, which makes use of social graphs during the neighbour selection phase, may be preferred over the classical *knn-*

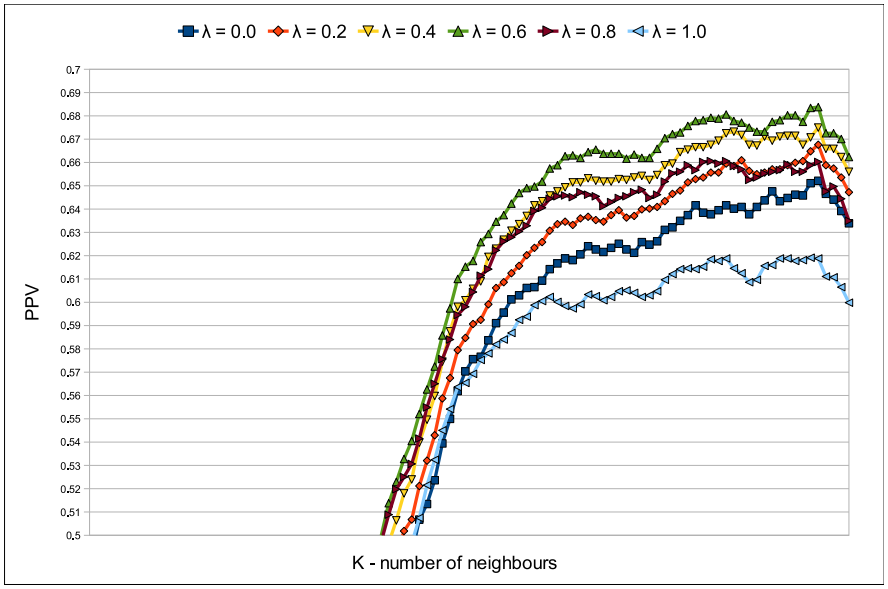

Figure 3.10: A detailed view of Figure 3.9

base algorithm in one-class collaborative filtering problems.

3.4.3 RESULTS OF SINGULAR VALUE DECOMPOSITION

In this section, we are going to discuss the results of *SVD* algorithm on our datasets. These results will form a baseline for the next section in which we will discuss the results of *wALS* algorithm.

We had tested the *SVD* algorithm on four datasets as well as their pre-processed versions. The important variable that may change the results of *SVD* is the number of features used in calculations. Figure 3.12 shows the outcomes of *SVD* algorithm for different feature counts for the *blog-programming-java* dataset.

PPV values of our test runs first increased and then decreased as we increased the number of features. This behavior is consistent with the results obtained by authors of [12] and [36]. In [12], optimal feature count is said to be around 40, while in [36], 10 seems to be the

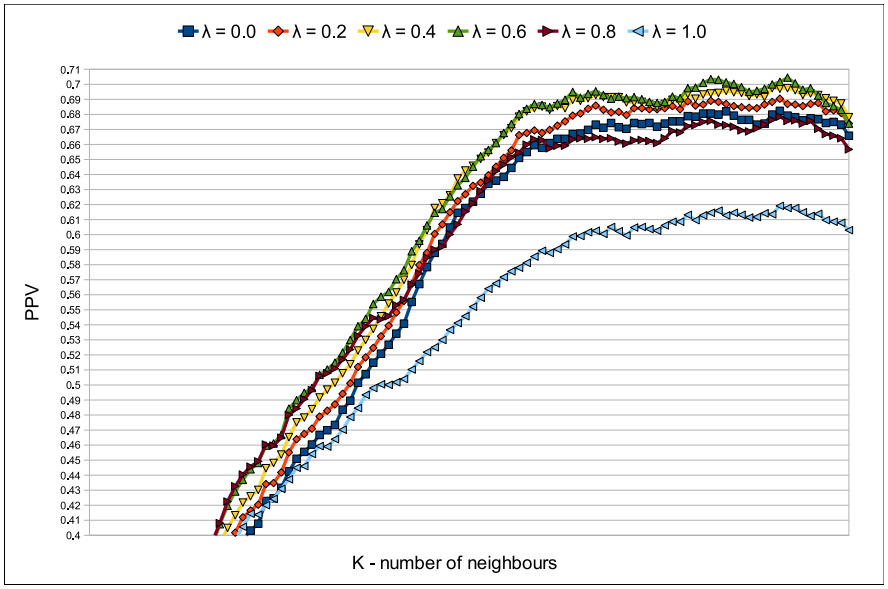

Figure 3.11: *knn-sg-neigh* results for pre-processed *blog-programming-java* dataset for different λ values.

optimal feature count. For our case, as can be seen from Figure 3.12, 12 is the optimal feature count. Even with the optimal feature count, *svd* seems to perform worse than any *k-NN* based algorithm. Best *PPV* value obtained from *svd* for *blog-programming-java* dataset was 0.57 which is 16% worse than the *PPV* value obtained from *knn-sg-neigh*. *svd* algorithm produced similar results for other datasets as well. Results of all other datasets are listed in Appendix B.2.

As can be seen in Figure 3.13, *svd* produced slightly better results for pre-processed dataset. Note that, the best *PPV* value obtained for pre-processed dataset is at $r = 24$, while it was at $r = 12$ for original dataset. Also decay rate is smaller in pre-processed dataset. Putting it all together, we can say that pre-processing the datasets did not have a great impact on performance of *svd* algorithm.

svd, without any modifications, seems not to perform as good as *knn-sg-neigh*. However, results were legitimate when compared to the results of [36] and [12].

Table 3.9: Results of *k-NN* based algorithms for pre-processed and original datasets generated using *blog*, *programming* and *java* tags.

	blog-programming-java		Pre-processed blog-programming-java	
	Best PPV	Best λ	Best PPV	Best λ
knn-base	~ 0.62	1.0	~ 0.62	1.0
knn-sg	~ 0.65	0.0	~ 0.67	0.0
knn-sg-pred	~ 0.63	0.8	~ 0.66	1.0 and 0.8
knn-sg-neigh	~ **0.68**	0.6	~ **0.70**	0.6

3.4.4 RESULTS OF wALS-BASED ALGORITHMS

This section covers the results of the final group of experiments we had conducted.

We had tested the *wALS* algorithm with the modifications explained in Section 3.3.3 over four datasets. In [36], authors proposed three different weighting schemes for *wALS*: *uniform*, *user oriented* and *item oriented*. Their test results had shown that user oriented scheme outperformed other schemes. Thus, we had chosen user oriented scheme as a baseline for our weighting schemes. We proposed two different weighting schemes for *wALS*. First one fills out the weight matrix of *wALS* using the first level connections of the social graph while the second one uses second level connections as well as the first level connections. The details of these weighting schemes can be found in Section 3.3.3.

To measure the impact of pre-processing, we used pre-processed datasets as well. Thus, we conducted a total of 24 experiments using three weighting schemes over eight datasets.

We divided this section into three parts and for each part we are going to discuss the results of a different weighting scheme. For each weighting scheme, we will be looking for the impact of pre-processing and impact of the number of features used during training. *wals-base* is the first scheme we are going to talk about. Later on we are going to give results of *wals-sg-d1* and complete this section with results of *wals-sg-d2*.

3.4.4.1 wALS-BASE

Figure 3.14 shows the behavior of *wals-base* algorithm when applied to *blog-programming-java* dataset. As can be seen from the figure, *PPV* values obtained increased as we increased

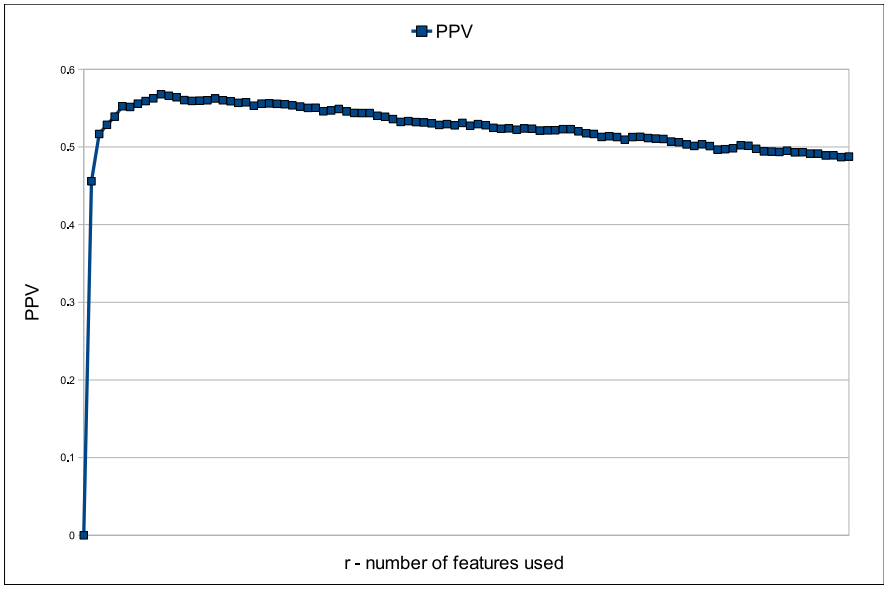

Figure 3.12: *svd* results for *blog-programming-java* dataset for different r values. r is the number of the features used in calculations.

the number of features we had used. However, after $r = 12$, *PPV* values tend to converge. Unlike the *svd* algorithm discussed in Section 3.4.3, *PPV* values did not decreased as we increased number of features. So *wals-base* produced quite stable results.

The best *PPV* value obtained was 0.58 which is slightly better than best *PPV* value of *svd*. Best *PPV* value obtained at $r = 14$ which is around the optimal feature count of *svd*.

Pre-processing the dataset did not help much for *blog-programming-java* dataset. Figure 3.14 clearly shows that *P-PPV* values are very close to *PPV* values.

Other datasets produced similar results. Although, they converged at different r values, best *PPV* values were always slightly better than *svd*. However, for none of them, *PPV* values reached the ones obtained from *k-NN* based algorithms.

Our *wals-base* results were parallel to the ones provided by authors of [36]. The main difference between our results and results of [36] is that their results outperformed *svd*. This was most probably due to the different datasets we had used. Generally speaking, our *wals-base*

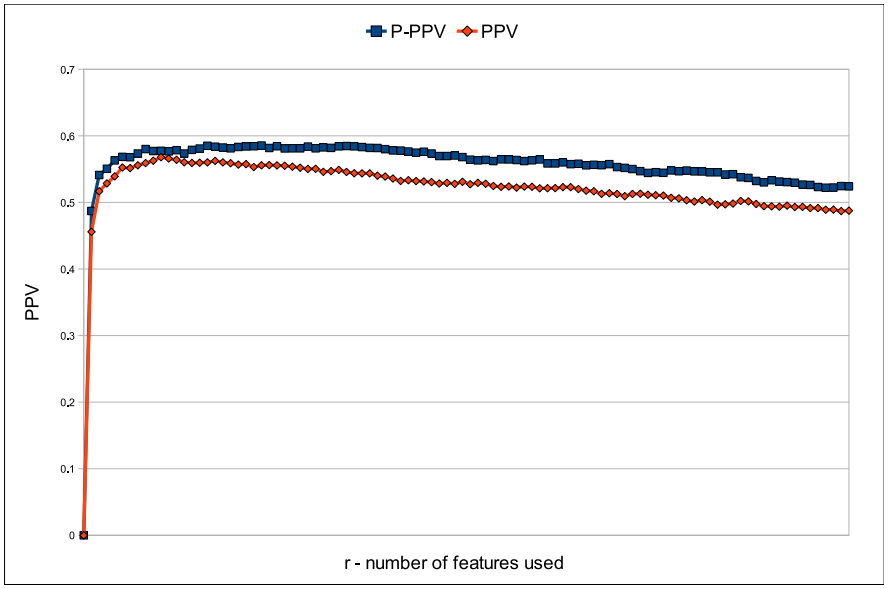

Figure 3.13: *svd* results for pre-processed *blog-programming-java* dataset for different r values. *P-PPV* represents the results of pre-processed dataset.

weighting scheme produced consistent results but the overall performance was not as good as *knn-base* or *knn-sg-neigh*. Appendix B.3 lists results of *wals-base* for other datasets.

3.4.4.2 wALS-SG-D1

wals-sg-d1 algorithm uses a weight matrix filled according to Equation (3.21). This is the first weighting scheme that uses social graphs. Results of this weighting scheme on *blog-programming-java* dataset is shown in Figure 3.15. *wals-sg-d1* produced comparable results to *knn-base*. Best *PPV* value produced is 0.61, which is obtained at $r = 28$. Although, best value reached is at $r = 28$, *PPV* values for $r > 18$ are pretty close.

The red line in Figure 3.15 represents the results obtained from the pre-processed *blog-programming-java* dataset. Similar to *wals-base*, pre-processing slightly increased the performance. Also the best *PPV* value is reached at $r = 18$, thus we can conclude that pre-processing the dataset quickens the convergence. In fact, this is supported with the test results

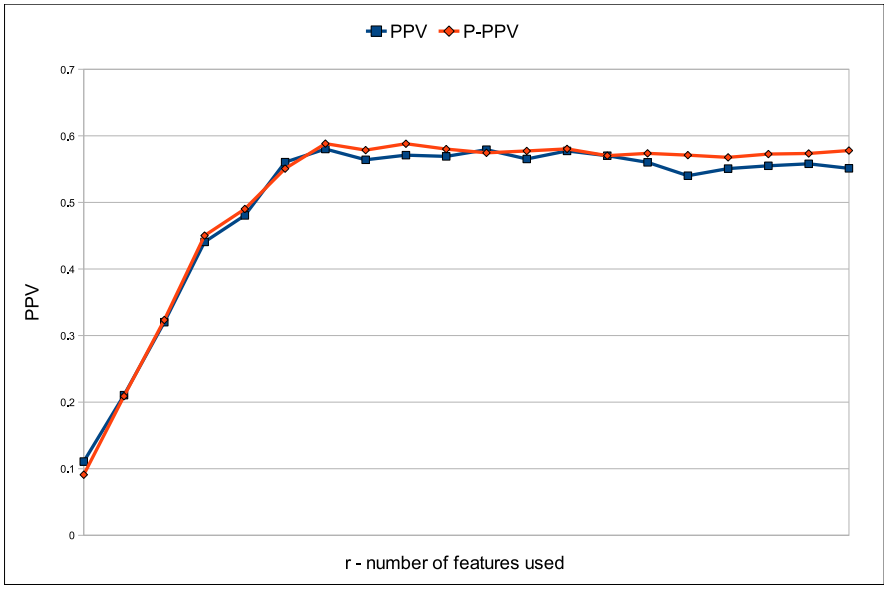

Figure 3.14: *wals-base* results for pre-processed and original *blog-programming-java* datasets for different r values. r is the number of features used and *P-PPV* is the results of pre-processed dataset.

of other datasets as well.

Although, *wals-sg-d1* performed much better than *svd* and *wals-base*, it was not able to outperform *knn-sg-neigh* algorithm. *wals-sg-d1* results for other datasets can be found in Appendix B.3.

3.4.4.3 wALS-SG-D2

wals-sg-d2 is the second weighting scheme that we consider using social graphs. Unlike our previous uses of social graphs, in this weighting scheme we also consider second level connections. By employing second level connections we were able to create a denser social graph (of course with different edge weights). Using the linear combination offered by Equation (3.23), we obtained the results for *blog-programming-java* dataset as shown in Figure 3.16. We chose λ to be 0.6 for our experiments. Note that this λ value has nothing to do with the λ value introduced in Section 3.4.2.

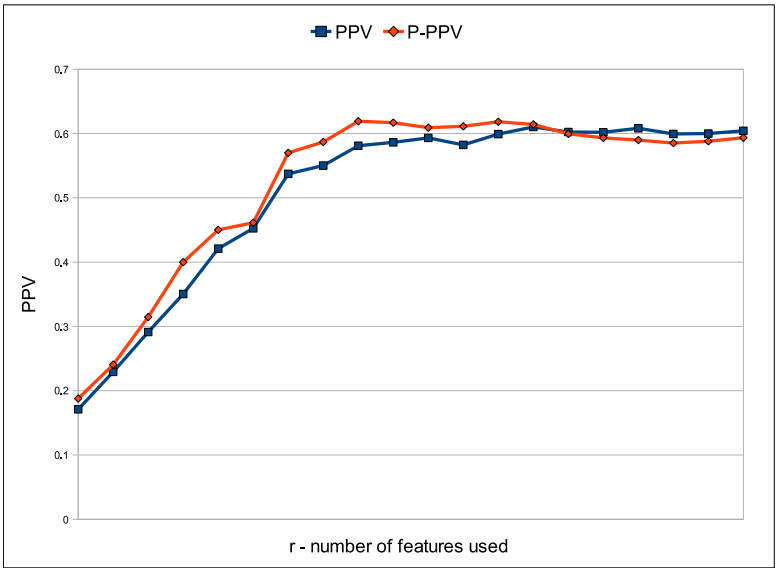

Figure 3.15: *wals-sg-d1* results for pre-processed and original *blog-programming-java* datasets for different r values. r is the number of features used and *P-PPV* is the results of pre-processed dataset.

wals-sg-d2 seems to be more successful than all *wALS* based weighting schemes as well as the *svd* algorithm. At $r = 20$, its *PPV* value reached to 0.63 which outperforms *knn-base*, *knn-sg-pred* and *svd* algorithms. Also for pre-processed dataset, *wals-sg-d2* produced a *PPV* value of 0.64 which is again comparable to results of other algorithms except *knn-sg-neigh*. *wals-sg-d2* brought the most significant improvement out to the *blog-programming-python* dataset. This could be due to the sparsity rate of social graph of this dataset. According to Table 3.2, *blog-programming-python* has the highest density value for its social graph. Results of *wals-sg-d2* for other datasets are listed in Appendix B.3.

Obviously, considering *depth-2* connections had a positive impact on test results. However, this impact was not intense when compared to results of *knn-sg-neigh* over the *blog-programming-java* and *blog-programming-python* datasets. Although, we do not have enough evidence to assure, in a real world system, one should consider using or at least testing *depth-2* connections. We believe that for denser social graphs *wals-sg-d2* would perform better.

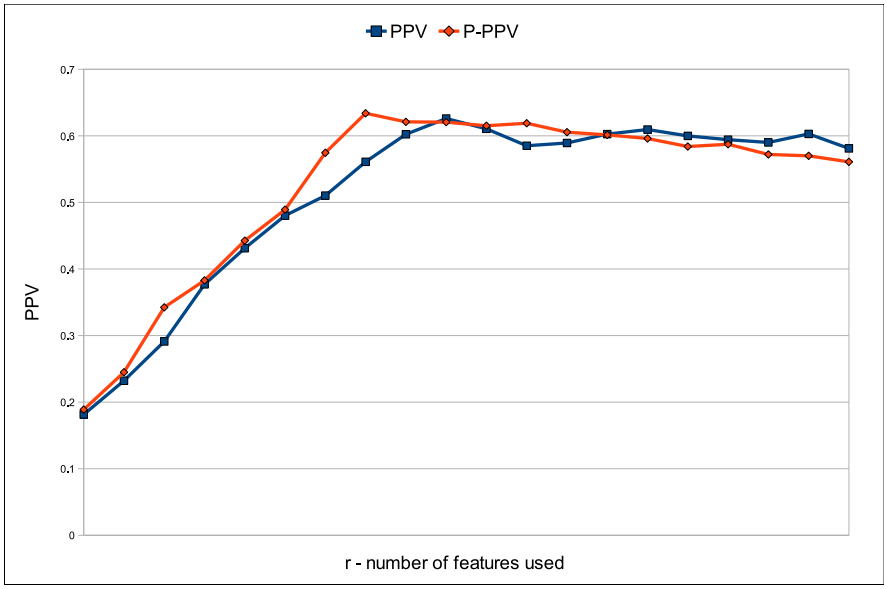

Figure 3.16: *wals-sg-d2* results for pre-processed and original *blog-programming-java* datasets for different r values. r is the number of features used and *P-PPV* is the results of pre-processed dataset.

3.5 CONCLUSION

In this chapter, we gave the details of the first part of our research. Most of the previous work focused on multi-class collaborative filtering problems and proposed methods accordingly. In fact, in some of them, researchers had used datasets that perfectly fit to the one-class collaborative filtering problems. With paying special attention to the properties of the datasets, we believe that traditional algorithms can perform better in one-class collaborative filtering problems.

One-class collaborative filtering problems came with two main obstacles: *missing counter examples* and *excessive sparsity of the datasets*. Unfortunately, these obstacles are extra and added to other obstacles of the multi-class collaborative filtering problems. To remedy these issues at least to some degree, we proposed the use of social graphs. The notion of collaboration involves in social graphs inherently. Thus they are perfect data sources that we might exploit in order to help standard methods to solve one-class collaborative filtering problems.

This constituted the heart of our research.

To begin with, we used social graphs to pre-process the datasets to see whether we could overcome the sparsity problem. By applying a *k-NN* like algorithm that uses social graphs to our datasets, we managed to populate training datasets. For *blog-programming-java* dataset we increased the number of positive examples by 22.2% and increased density rate from 0.0017 to 0.0020.

Without the counter examples, standard collaborative filtering methods would result in bad performance if not totally failed. One way of using standard methods in one-class collaborative filtering problems is to assume that all missing entries are counter examples which involves a high bias. To overcome this problem weighting mechanisms had been proposed. By assigning a weight value in the range $[0, 1]$ to each instance in a dataset, we define confidence levels for that instances. For observed examples, we assigned a 1 which implies that we are pretty confident about that rating. For missing counter examples we defined weighting schemes based on social graphs that weights higher if a URL is highly rated by connections of the social map. These weighting schemes are based on the idea that "if your friends liked a URL probably you will like that as well".

Table 3.10: Results of all algorithms for pre-processed and original datasets generated using *blog*, *programming* and *java* tags.

	blog-programming-java	**Pre-processed blog-programming-java**
knn-base	~ 0.62	~ 0.62
knn-sg	~ 0.65	~ 0.67
knn-sg-pred	~ 0.63	~ 0.66
knn-sg-neigh	~ **0.68**	~ **0.70**
svd	0.57	0.59
wals-base	0.58	0.59
wals-sg-d1	0.61	0.62
wals-sg-d2	0.63	0.64

Table 3.10 lists results of all methods we used for *blog-programming-java* dataset. Also second column lists the results for pre-processed version of *blog-programming-java* dataset. *knn-base*, *svd* and *wals-base* methods were our baselines. Among all methods we had used, *knn-sg-neigh* and *wals-sg-d2* were the most successful ones. Although results of *knn-sg-neigh* were not consistent among all datasets, the results generated were quite good. In fact,

for *blog-programming-java* dataset, it offered about 10% improvement over *knn-base* method. *knn-sg-neigh* produced good results also for *blog-programming-python* dataset, which made us think that this algorithm works well on domain specific datasets and social graphs. To find out whether actually this is the case, we conducted a series of experiments which constitutes second part of our research given in next chapter. *wals-sg-d2* did not performed as good as *knn-sg-neigh* for *blog-programming-java* dataset. However, its results were consistent among all datasets and all results outperformed our baselines.

As can be seen from Table 3.10, pre-processing the datasets had a small impact on performance of algorithms. Pre-processing helps us to reduce the sparsity rates but have no effect on the missing counter-example issue which is the primary problem with one-class collaborative filtering algorithms. Using social graphs in pre-processing phase increased *PPV* values slightly. Improvement was promising, thus it should be considered to use social graphs in pre-processing. However, there is no guarantee on good results.

Despite being out of the scope of our research, we would like to spent some words on the computational resource requirements of the methods we had used. One, if not the only, good point with one-class collaborative filtering methods is that the dataset of concern is binary. In other words, the huge rating matrix can be stored in bit matrices which drastically lowers the memory requirements. *k-NN* is known to be a high memory intensive algorithm. For *knn-base* algorithm we did not face any memory problems. Also the similarity metric we had used was consist of Boolean operations over bit vectors. Thus they required much less CPU clocks than usual similarity metrics used in other collaborative filtering problems. However, employing a weight matrix that consists of double precision numbers changed this situation. Even for small datasets that we had used, we needed to rethink the structure of our algorithms in order to made them run on a modest PC with 2 gigabytes of RAM. For *svd* and *wALS* based methods, memory problems were at critical levels. These methods required to store weight matrices as well as two large double precision matrices that stores the feature vectors. For *svd* method, we needed to decompose training matrices of approximately $30K \times 1K$, which was quite a challenge. For larger matrices this method is definitely not an option. *wALS* based methods are highly CPU intensive since they require several matrix operations. For all methods, parallelized versions have to be considered in real world systems. Especially, for a system that is required to make online predictions computational requirements become more important. At this point social graphs can be very handy. A system that uses only friends of

users as their neighbours can make predictions on the fly. And the only data structure that has to be stored in memory is an adjacency matrix representing the social graph. Although we did not do any optimization, for real world applications, algorithm's that use social graphs can be highly optimized in terms of memory requirements.

Another point that we want to go over is the depth of connections to be used during calculations. For all but the *wals-sg-d2* algorithm, we used only *depth-1* connections, which are the direct links between a user and his/her friends. However, *depth-2* or even *depth-3* connections can be helpful as well, since there will be much more connections in *depth-2* and even more in *depth-3*. In fact, in *wals-sg-d2* experiments, we were able to get better *PPV* values than *wals-sg-d1* which uses only *depth-1* connections. Note that, in *wals-sg-d2*, we used a linear combination of *depth-1* and *depth-2* connections in which we put more emphasis on *depth-1* connections. Similarly, one could employ *depth-3* connections as well with an even lower emphasis. However, we should be aware not to overuse connections. This may lead us to a point where the emphasis of *depth-1* connections diminished. *Depth-1* connections are the most valuable connections to us and we should never let them loose their importance.

To sum up, we can conclude that social graphs will be helpful for one-class collaborative filtering problems at least for some types of datasets. Also due to the structure of the social graphs, some optimizations may be done in order to reduce memory requirements of algorithms.

CHAPTER 4

DOMAIN SPECIFIC VS. GENERIC SOCIAL GRAPHS

In the first part of our research, we tried to benefit from social maps in order to increase the performances of standard collaborative filtering algorithms when applied to one-class collaborative filtering problems. Specifically, we had modified *k-NN* and *wALS* algorithms such that we introduced new weighting schemes based on social graphs. We also used social graphs to pre-process training datasets to remedy the sparsity problems we faced. In spite of the fact that social graphs were not as fruitful as we expected to one-class collaborative filtering problem, our research revealed some interesting points about them.

As we discussed in Section 3.4.2.2, *knn-sg-neigh* algorithm outperformed the rest of the algorithms when we provided a more *domain specific* dataset to it. For our case, the more the tags used to create a dataset are, the more the domain specific that dataset is. For instance, a dataset created using the URLs bookmarked with *blog*, *programming*, and *java* tags is more domain specific than a dataset created using the URLs bookmarked with only *blog* tag. We believe that *knn-sg-neigh* performed better for *blog-programming-java* and *blog-programming-python* datasets since these two datasets are more domain specific, thus the connections between their social graphs more resembles to the connections between their *collaborative graphs*.

In the second part of our research, we focused on this issue and tried to figure out which types of social graphs could be used by *knn-sg-neigh* algorithm for one-class collaborative filtering problems.

First of all, we are going to give a problem definition for this part of our research and discuss our methodology. Later, we are going to cover the datasets we had used in our experiments. Finally, we are going to talk on empirical results and close this chapter.

4.1 PROBLEM DEFINITION AND METHODOLOGY

The main motivation behind this part of our research is the results of *knn-sg-neigh* algorithm we had studied in Section 3.3.1. In our experiments, we noticed that *knn-sg-neigh* algorithm performed interestingly well for *blog-programming-java* and *blog-programming-python* datasets. In fact, these results were the most successful ones among all other methods and datasets. In the second part of our research we sought the answer of the following question:

> For what types of one-class collaborative filtering datasets is it appropriate to use social graphs in order to improve prediction performances?

As we noted in previous chapter, social graphs includes the notion of collaboration naturally. In most of the times, we value ideas of our friends rather than ideas of a stranger. If we are looking for advice in a technical field, then ideas of our colleagues are much more important to us than ideas of one of our family members. Actually whom to value his/her idea totally depends on the domain we are seeking an advice for. So for a recommendation system that is expected to recommend research papers, social graphs of users and their colleagues are arguable the most important dataset, while a social graph of users and their friends would best suit a movie recommender.

In the first part of our research, we modified algorithms to use social graphs and measured their performances over different datasets. To find evidence that supports this idea, we followed a reverse path. After *k-NN* algorithm finishes training, we end up with a set of neighbours for each user. By taking those like-minded neighbours as friends of each user, we obtain a graph of users which is, indeed, another social graph. From now on we will call this graph as *collaborative graph* in order to prevent possible confusions.

We had argued that *knn-sg-neigh* algorithm was successful on *blog-programming-java* and *blog-programming-python* datasets since the users of those social graphs were belong to more specific domains. Thus we can expect that social graphs of those datasets resemble the collaborative graphs of same datasets. In other words, for any user, list of his/her neighbours should overlap the list of his/her friends.

So for this part of our research, we are aiming to find out how similar the collaborative graphs and social graphs are for several datasets. A complete analysis of both graphs is out of the scope of our research. Instead, we focus on the similarities of the nodes and edges connecting them.

As in the previous chapter, we assume that both the collaborative graphs and the social graphs are undirected and unweighted. Since we do not consider edge weights, graphs can be represented with binary adjacency matrices. A valid adjacency matrix should obey the Equation (4.1).

$$G(i,j) = \begin{cases} 1, & \text{if } u_i \text{ and } u_j \text{ connected} \\ 0, & \text{if } i = j \text{ or } u_i \text{ and } u_j \text{ not connected} \end{cases} \tag{4.1}$$

For each dataset, we generated a collaborative graph using *knn-base* algorithm as explained in Section 3.3.1. To find how similar two adjacency matrices, we used an intuitive way which is given by Equation (4.2).

$$\text{similarity}(\mathbf{S}, \mathbf{C}, k) = \frac{\text{cardinality}(\mathbf{S} \cap \mathbf{C})}{\text{cardinality}(\mathbf{S})} \tag{4.2}$$

where $\mathbf{S}$ is the adjacency matrix of social graph, $\mathbf{C}$ is the adjacency matrix of collaborative graph obtained using top k neighbours of each user.

Note that Equation (4.2) uses only *depth-1* connections. However, *depth-2* connections are important as well. Thus, for each dataset we looked for similarity of $\mathbf{C}$ to the adjacency matrix that stores second level connections as well.

Equation (4.2) simply calculates number of common terms in user's neighbour list and user's friend list. It does not pay attention to the order of the lists. However, user's neighbour list is ordered by the similarity of the user to the neighbour. So a match that occurs at the top of the list should gain more importance than a match that occurs at the bottom of the list. To catch up this notion we used a metric called *Half Life Utility (HLU)*. *HLU* was presented in [5] to be used as a new evaluation for recommendation systems. *HLU* depends on the idea that when a user is presented a ranked list of items, he/she is very unlikely to browse to deeper items. So the probability of the items to be clicked decays as we go through the list. *HLU* is given as in

Equation (4.3).

$$\text{HLU} = 100 \times \frac{\sum_u R_u}{\sum_u R_u^{max}} \tag{4.3}$$

and R_u is defined as in Equation (4.4).

$$R_u = \sum_{u_i \in N(u)} \frac{\eta(u, u_i)}{2^{(i-1)(\beta-1)}} \tag{4.4}$$

where $N(u)$ is the neighbour list of user u, $\eta(u, u_i)$ is an indicator function that is 1 only if user u and user u_i are friends. R_u is the utility of user u and R_u^{max} is the maximal utility that can be achieved for user u. In other words, R_u^{max} is the utility that will be obtained if all friends of user u were at the top of the neighbour list obtained from *knn-base* algorithm. β is the decay rate proposed in [5]. We used the original value of 5 for β.

Using these two metrics, we conducted experiments on our datasets for both *depth-1* and *depth-2* connections. After going over the details of the datasets we used in the next section, we will be giving results of our experiments in Section 4.3.

4.2 DATASETS

For this part we had created four new datasets and used them with the datasets we created for the previous part. In the first part of our research, we had used two groups of datasets. Datasets of the first group was created using the *blog* and *photography* tags. Second group of datasets were created using more tags for each dataset. In the second part of our research, our main focus was to investigate the properties of domain specific datasets. Thus, we had created more domain specific datasets as well as some middle state datasets. Specifically, datasets created were *blog-programming*, *photography-camera*, *photography-camera-canon* and *photography-camera-nikon*. Properties of these datasets are given in Table 4.1, while properties of social maps are given in Table 4.2.

Histograms of these newly added datasets are very similar to the ones used in previous chapter. These histograms are listed in Appendix A.2.

Table 4.1: Properties of training datasets used in second part.

Tags Used	**User Cnt.**	**URL Cnt.**	**Bookmark Cnt.**	**Density**
blog	27,142	1,296	59,888	0.0017
photography	31,085	1,273	71,910	0.0018
blog, *programming*	31,025	1,332	88,014	0.0021
photography, *camera*	13,338	995	28,328	0.0025
blog, *programming*, *java*	23,955	1,084	60,869	0.0023
blog, *programming*, *python*	14,140	929	39,579	0.0030
photography, *camera*, *canon*	8,478	735	17,855	0.0029
photography, *camera*, *nikon*	6,951	849	14,126	0.0024

Table 4.2: Properties of social graphs used in second part.

Tags Used	**User Cnt.**	**Edge Cnt.**	**Density**
blog	27,142	71,110	0.000096
photography	31,085	87,046	0.000090
blog, *programming*	31,025	91,404	0.000095
photography, *camera*	11,338	27,294	0.000212
blog, *programming*, *java*	23,955	77,016	0.000134
blog, *programming*, *python*	14,140	47,090	0.000235
photography, *camera*, *canon*	8,478	21,520	0.000299
photography, *camera*, *nikon*	6,951	17,692	0.000366

4.3 EVALUATION

To find out similarities between collaborative graphs and social graphs, we measured adjacency matrix similarities and *HLU* values for all of the datasets given in Table 4.1. Collaborative graphs used in our tests were obtained for the k values that produced highest *PPV* values in *knn-base* algorithm.

Table 4.3 lists similarities between adjacency matrices of collaborative graphs and social graphs.

Table 4.3 can be interpreted as follows: For *blog* dataset, 10.48% of friends of users were found to be in the computed collaborative graph. In other words, 10.48% of the nodes of the social graph of *blog* dataset were also nodes of corresponding collaborative graph. If we consider *depth-2* connections as well, this rate increases to 11.44% which means that the

Table 4.3: Similarities between collaborative graphs and social graphs computed according to Equation (4.2).

Dataset	**Depth-1 Similarity**	**Depth-2 Similarity**	**Growth**
blog	0.1048	0.1144	9.18%
photography	0.0929	0.0981	5.65%
blog, *programming*	0.0931	0.0998	7.19%
photography, *camera*	0.1376	0.1525	10.79%
blog, *programming*, *java*	0.1193	0.1464	22.71 %
blog, *programming*, *python*	0.1359	0.1594	17.31%
photography, *camera*, *canon*	0.1914	0.2195	14.68%
photography, *camera*, *nikon*	0.1880	0.2103	11.83%

depth-1 match count improved by 9.18%.

At first glance, the numerical values given in Table 4.3 may not be so informative about the datasets. However, after a little inspection, we can easily end up with two deductions.

The first deduction is directly about our research problem. We started our discussion in order to find out whether datasets from specialized domains resemble more to corresponding collaborative graphs than datasets from more general domains or not. As can be seen in Table 4.3, *photography* dataset has a similarity of 9.29%, while dataset *photography-camera*, which covers a more specialized domain, has a similarity rate of 13.76%. Moreover, *photography-camera-canon* and *photography-camera-nikon* datasets have similarity rates of 19.14% and 18.80% respectively. Similarity rates of the datasets belong to *blog* domain comply with this situation as well. These results directly supports that social graphs of datasets of specialized domains better resembles to their collaborative graphs.

Second deduction is about the depth of the connections. As can be seen from Table 4.3, all of the *depth-2* graphs have positive growth rates. Although, the magnitude of the growth rates does not imply any sensible results, the sign of them indicates that when we used *depth-2* connections the number of common nodes among collaborative graphs and social graphs of datasets had increased.

As we already stated, checking only the similarities of adjacency matrices does not put the importance of match position into account. In all of the *k-NN* based algorithms we mentioned in previous chapter, we used the similarity between two users as an indicator of the confidence

Table 4.4: Similarities between collaborative graphs and social graphs computed according to Equation (4.3).

Dataset	**Depth-1 HLU**	**Depth-2 HLU**	**Growth**
blog	18.49	19.03	2.93%
photography	16.46	17.00	3.26%
blog, *programming*	16.58	17.08	2.99%
photography, *camera*	19.94	20.94	5.02%
blog, *programming*, *java*	19.28	20.34	5.48%
blog, *programming*, *python*	23.59	24.68	4.64%
photography, *camera*, *canon*	26.24	27.22	3.73%
photography, *camera*, *nikon*	25.84	26.66	3.19%

of that user about the taste of the other user. Ratings of a neighbour with high similarity has more effect on the final rating prediction than ratings of a neighbour with less similarity. Thus, it is important for us to find not only if a friend of a user exists in his/her neighbour list but also the actual position of that friend in neighbour list. That is why we proposed the *HLU* metric. Table 4.4 lists the computed *HLU* values of all datasets for both *depth-1* and *depth-2* connections. A quick overview of Table 4.4 would reveal the resemblance between these results and the ones given in Table 4.3. *HLU* results also directly support our argument about domain specific social graphs. Although, the growth rates are much smaller when compared to results of Table 4.3, we can conclude that considering *depth-2* connections had a positive impact on our test results.

4.4 CONCLUSION

As we discussed in previous chapter, in the first part of our research we noticed that *knn-sg-neigh* algorithm performed interestingly good on some of our datasets. We decided to go over this issue with the hope of defining the dataset types on which *knn-sg-neigh* is expected to produce acceptable results. By defining such properties of datasets, we would at least save some time of researchers, which they could spend on solving other recommender system problems. Or they could pretend to be working on those problems and play Warcraft instead.

We argued that URLs labeled with *blog* and *programming* tags address a more specific domain than URLs labeled with only *blog* tag. Thus, the datasets created with these URLs are more *domain specific*. Also we argued that domain specific datasets should contain more

collaboration than a dataset from a more general domain. In other words, for a domain specific dataset, a graph that represents the collaboration flows among users should resemble the actual social graph obtained from the same dataset. To find evidence for our argument, we had created collaborative graphs and social graphs for eight datasets and compared them according to two different similarity metrics. The results we obtained were highly promising. Resemblance rates were prominently higher for datasets created from a specific domain which directly supports our argument.

Also we researched the impact of using deeper levels of connections in social graphs. We had compared similarities of collaborative graphs and social graphs with only *depth-1* connections and with *depth-1* and *depth-2* connections. Our experiments had shown that social graphs with *depth-1* and *depth-2* connections resembles more than the social graphs with only *depth-1* connections. We did not extend our research to cover deeper levels since this was not the main focus of our research. However, we saved it as a future work.

In conclusion, this part of our research revealed that domain specific datasets better embodies the notion of collaboration. Thus, they should be expected to outperform more general datasets in a collaborative filtering environment.

CHAPTER 5

CONCLUSION

We started our discussion with the *information overload* problem. Every passing second, people publish new websites, upload their photographs to their favorite services, write reviews about products or search for something they are interested in. And every passing second, Internet services become more inadequate to respond the needs of their users. To address this information overload problem, at least to some degree, major service providers started to build personalized services that offer customized solutions. These services either explicitly or implicitly inspects their users' behaviors and act accordingly. With the increased awareness of such systems, users started to seek for such services to locate their needs. Recommendation systems are at the heart of such services. Although, several examples of successful recommendation systems exists, their number is still inadequate to solve the whole information overload problem. It is obvious that such personalized services and particularly recommendation systems will gain even more importance as the web evolves.

In order the recommendation systems to function correctly, they need to learn users' tastes. Obviously, the most direct way to achieve this is to explicitly ask the user for what he/she likes/dislikes. However, users hardly cooperate in such a situation. Thus, the only option remains is to look for implicit data hidden in user behaviors. This implicit data can be anything like clicking on a link, time spent on a web page, bookmarking a URL or even highlighting part of a text. The problem with such a data is that this data is not that informative. In most of the cases, this data will only reveal that a user *is* interested in something. It is hardly possible to find cases where a recommender system able to conclude that a user *is not* interested in something. Lack of counter-examples makes such cases a natural candidate for applications that should use one-class collaborative filtering methods. Being able to deal with cases that counter-examples does not exists, makes one-class collaborative filtering applications

remarkable. One-class collaborative filtering is rather a new field of research. In previous works, researchers tend to solve one-class collaborative filtering problems with classical collaborative filtering approaches. However, it is obvious that one-class collaborative filtering problems need special attention.

Social networks is one of the *trendy* topics in current web. And it seems that they will remain trendy for quite a long time. Apart from being trendy, social networks are good fellows of collaborative filtering applications since they involve the notion of *collaboration* naturally. In our research, we tried to benefit from this implicit collaboration in one-class collaborative filtering applications.

We had divided our research into two phases. In the first phase, we searched the ways of making the use of social networks in classical one-class collaborative filtering algorithms. Specifically, we used social networks to pre-process datasets in order to reduce the sparsity rates and to create weighting schemes for some well-known algorithms. Our experiments showed that using social graphs in pre-processing the datasets may not be spectacular. For some of our datasets, pre-processing slightly increased the prediction performances. In the second part of this phase, we modified *k-NN* and *wALS* algorithms to accommodate social data. *knn-sg-pred* and *wals-sg-d1* algorithms produced comparable results to our baselines. On the other hand, *knn-sg-neigh* algorithm produced remarkable results for two of our datasets. Also *wals-sg-d2* produced better results than our baselines.

In the second phase of our research, we tried to understand why *knn-sg-neigh* was successful for two of our datasets. We also tried to find out why *wals-sg-d2* performed better than *wals-sg-d1*. After analyzing our test results, we concluded that *knn-sg-neigh* was successful on those datasets since they were more *domain specific*. Also we concluded that using *depth-2* connections of a social graph as well as *depth-1* connections may help in improving prediction performances.

Although, there are still too many open issues related to social graphs and one-class collaborative filtering problem, we had to stop our research at some point. We leave further analysis of social graphs in one-class collaborative filtering applications as future work. Our future work list includes detailed analysis of collaborative graphs and social graphs, effects of using deeper levels of connections in social graphs, new weighting schemes based on social graphs and integrating social graphs to other one-class collaborative filtering algorithms.

REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749, 2005.

[2] M. Balabanović and Y. Shoham. Fab: content-based, collaborative recommendation. *Commun. ACM*, 40(3):66–72, 1997.

[3] S. Ben-David and M. Lindenbaum. Learning distributions by their density-levels - a paradigm for learning without a teacher. In *EuroCOLT '95: Proceedings of the Second European Conference on Computational Learning Theory*, pages 53–68, London, UK, 1995. Springer-Verlag.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[5] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. pages 43–52. Morgan Kaufmann, 1998.

[6] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12:331–370(40), November 2002.

[7] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 271–280, New York, NY, USA, 2007. ACM.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] Del.icio.us. Del.icio.us. http://delicious.com. Last accessed on August 2009.

[10] P. J. Denning. Acm President's letter: Electronic Junk. *Commun. ACM*, 25(3):163–165, 1982.

[11] K. Fukunaga. Statistical pattern recognition. pages 33–60, 1993.

[12] S. Funk. Netflix update: Try this at home. http://sifter.org/simon/journal/20061211.html. Last accessed on August 2009.

[13] J. Golbeck and J. Hendler. FilmTrust: Movie recommendations using trust in web-based social networks. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 1, pages 282–286, 2006.

[14] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.

[15] N. Good, J. B. Schafer, J. A. Konstan, A. Borchers, B. Sarwar, J. Herlocker, and J. Riedl. Combining collaborative filtering with personal agents for better recommendations. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 439–446, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[16] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.

[17] T. Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.*, 22(1):89–115, 2004.

[18] A. S. Joydeep, E. Strehl, J. Ghosh, and R. Mooney. Impact of similarity measures on web-page clustering. In *In Workshop on Artificial Intelligence for Web Search (AAAI 2000*, pages 58–64. AAAI, 2000.

[19] H. Kautz, B. Selman, and M. Shah. ReferralWeb: Combining social networks and collaborative filtering. *Commun. ACM*, 40(3):63–65, March 1997.

[20] D. Kelly and J. Teevan. Implicit feedback for inferring user preference: a bibliography. *SIGIR Forum*, 37(2):18–28, 2003.

[21] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas. Machine Learning: A review of classification and combining techniques. *Artif. Intell. Rev.*, 26(3):159–190, 2006.

[22] Last.fm. Last.fm Audioscrobbler. http://www.audioscrobbler.net. Last accessed on August 2009.

[23] N. Lathia, S. Hailes, and L. Capra. kNN CF: A temporal social network. In *RecSys '08: Proceedings of the 2008 ACM conference on Recommender systems*, pages 227–234, New York, NY, USA, 2008. ACM.

[24] K. Lerman. Social networks and social information filtering on digg, Dec 2006.

[25] G. Linden, N. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. Technical report, Amazon.com, 2003.

[26] L. M. Manevitz and M. Yousef. One-class SVMs for document classification. *J. Mach. Learn. Res.*, 2:139–154, 2002.

[27] B. M. Marlin, R. S. Zemel, S. Roweis, and M. Slaney. Collaborative filtering and the missing at random assumption. In *proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI2007)*, 2007.

[28] D. W. McDonald. Recommending collaboration with social networks: a comparative evaluation. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 593–600, New York, NY, USA, 2003. ACM.

[29] P. Melville, R. J. Mooney, and R. Nagarajan. Content-boosted collaborative filtering for improved recommendations. In *Eighteenth national conference on Artificial intelligence*, pages 187–192, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[30] M. Montaner, B. López, and J. L. de la Rosa. Developing trust in recommender agents. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 304–305, New York, NY, USA, 2002. ACM.

[31] N. S. Nati and T. Jaakkola. Weighted low-rank approximations. In *In 20th International Conference on Machine Learning*, pages 720–727. AAAI Press, 2003.

[32] Netflix. Netflix CineMatch. http://www.netflix.com. Last accessed on August 2009.

[33] Netflix. Netflix Prize. http://www.netflixprize.com. Last accessed on August 2009.

[34] J. O'Donovan and B. Smyth. Trust in recommender systems. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 167–174, New York, NY, USA, 2005. ACM.

[35] J. Palau, M. Montaner, B. López, and J. L. De La Rosa. Collaboration analysis in recommender systems using social networks. *Lecture Notes in Computer Science*, pages 137–151, 2004.

[36] R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. One-class collaborative filtering. *Data Mining, IEEE International Conference on*, 0:502–511, 2008.

[37] M. J. Pazzani and D. Billsus. Content-based recommendation systems. *Lecture Notes in Computer Science*, pages 325–341, 2007.

[38] C. Rack, S. Arbanowski, and S. Steglich. A generic multipurpose recommender system for contextual recommendations. In *ISADS '07: Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems*, pages 445–450, Washington, DC, USA, 2007. IEEE Computer Society.

[39] A. M. Rashid, I. Albert, D. Cosley, S. K. Lam, S. M. McNee, J. A. Konstan, and J. Riedl. Getting to know you: learning new user preferences in recommender systems. In *IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces*, pages 127–134, New York, NY, USA, 2002. ACM.

[40] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM.

[41] P. Resnick and H. R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.

[42] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295, New York, NY, USA, 2001. ACM.

[43] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl. Application of dimensionality reduction in recommender system - a case study. Technical report, Department of Computer Science and Engineering, University of Minnesota, 2000.

[44] J. B. Schafer, J. Konstan, and J. Riedi. Recommender systems in e-commerce. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, New York, NY, USA, 1999. ACM.

[45] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 253–260, New York, NY, USA, 2002. ACM.

[46] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Comput.*, 13(7):1443–1471, 2001.

[47] D. M. J. Tax. *One-class classification; Concept-learning in the absence of counter-examples*. PhD thesis, Delft University of Technology, 2001. ISBN 90-75691-05-x.

[48] L. Terveen, W. Hill, B. Amento, D. McDonald, and J. Creter. PHOAKS: a system for sharing recommendations. *Commun. ACM*, 40(3):59–62, 1997.

[49] M. E. Wall, A. Rechtsteiner, and L. M. Rocha. Singular value decomposition and principal component analysis, 2003.

[50] G. Ward, T. Hastie, S. Barry, J. Elith, and J. R. Leathwick. Presence-Only Data and the EM algorithm. *Biometrics*, 65:554–563(10), June 2009.

[51] Yelp. Yelp. http://www.yelp.com. Last accessed on August 2009.

[52] K. Yu, A. Schwaighofer, V. Tresp, X. Xu, and H. Kriegel. Probabilistic memory-based collaborative filtering. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):56–69, 2004.

[53] K. Yu, X. Xu, M. Ester, and H. Kriegel. Selecting relevant instances for efficient and accurate collaborative filtering. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 239–246, New York, NY, USA, 2001. ACM.

[54] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM '08: Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, pages 337–348, Berlin, Heidelberg, 2008. Springer-Verlag.

APPENDIX A

DATASET HISTOGRAMS

A.1 PART-I DATASET HISTOGRAMS

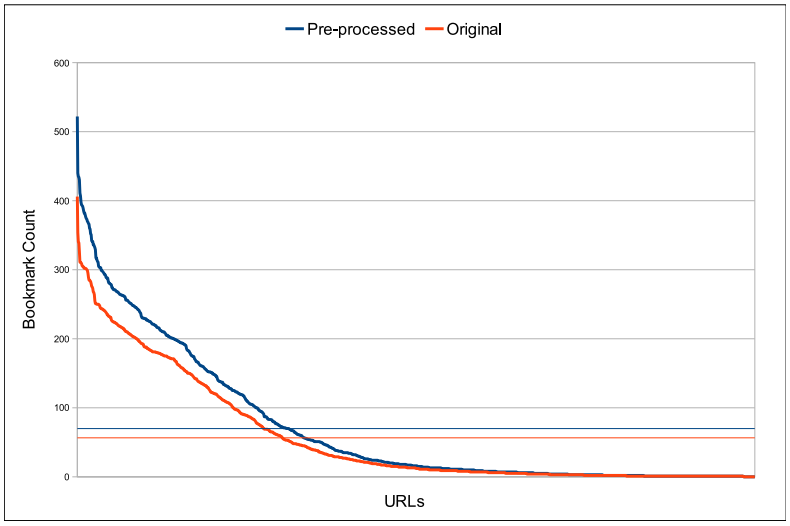

Figure A.1: Pre-processed and original *photography* training dataset histograms.

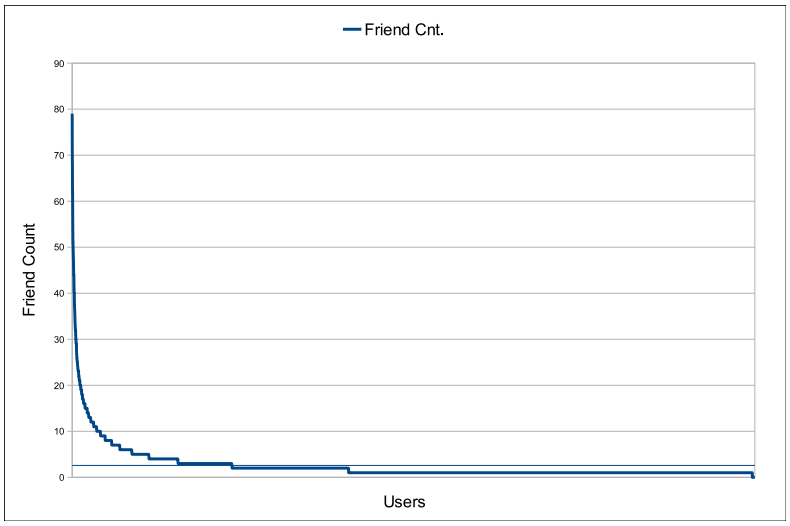

Figure A.2: *photography* social graph histogram.

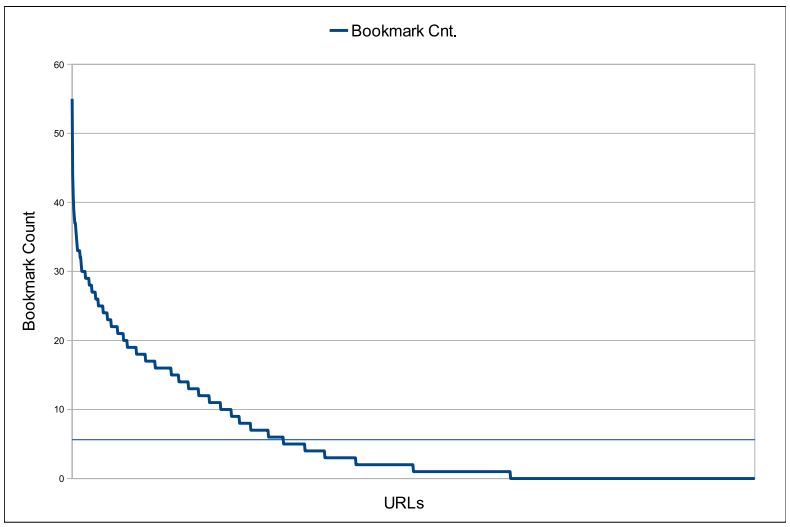

Figure A.3: *photography* test dataset histogram.

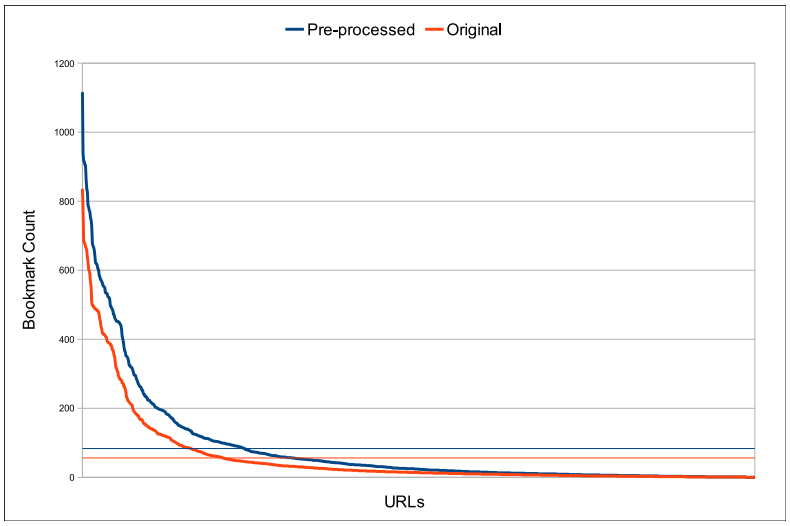

Figure A.4: Pre-processed and original *blog-programming-java* training dataset histograms.

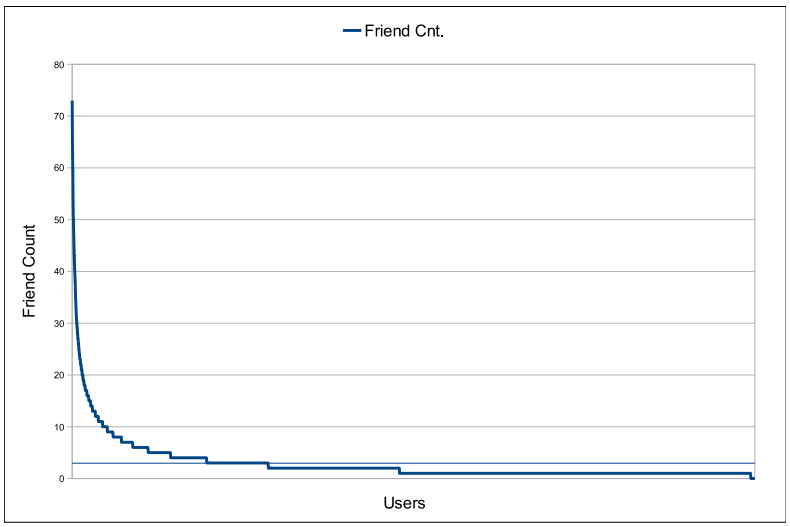

Figure A.5: *blog-programming-java* social graph histogram.

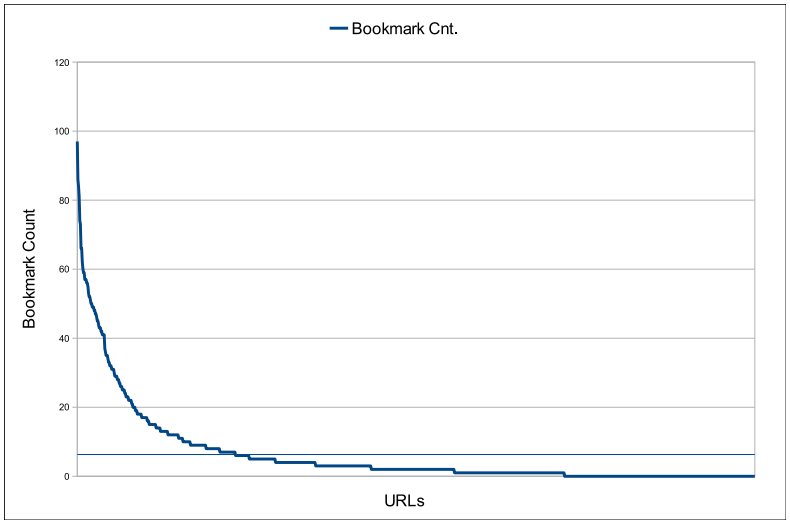

Figure A.6: *blog-programming-java* test dataset histogram.

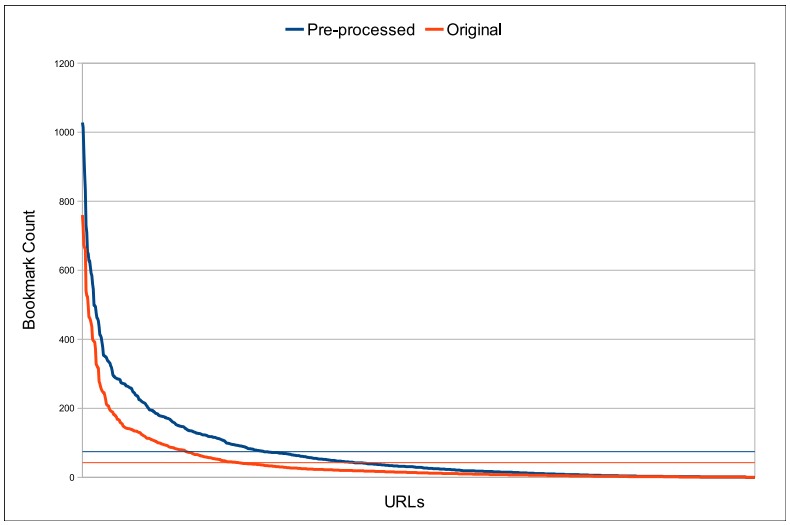

Figure A.7: Pre-processed and original *blog-programming-python* training dataset histograms.

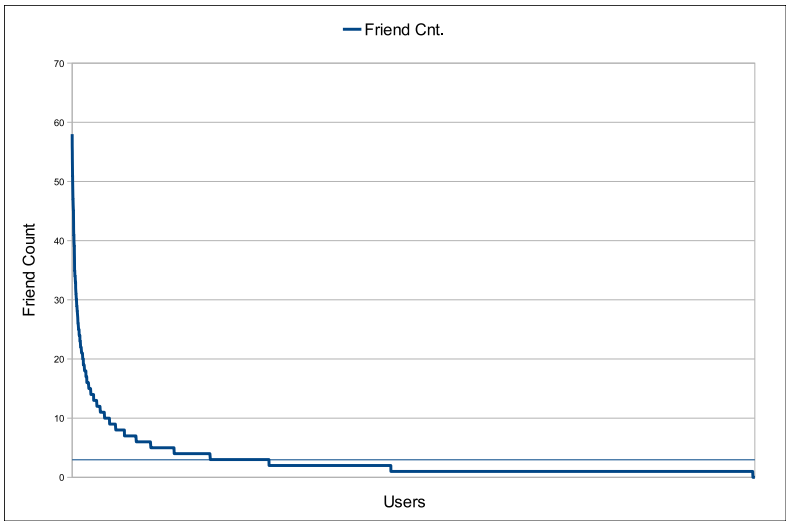

Figure A.8: *blog-programming-python* social graph histogram.

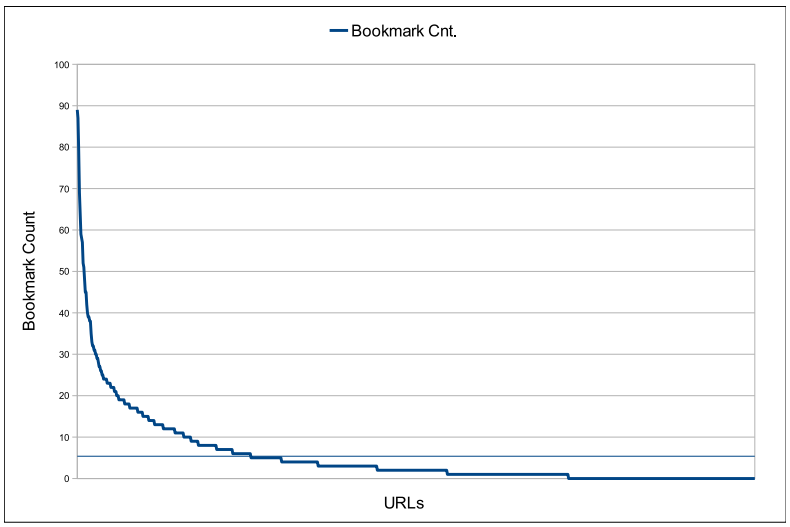

Figure A.9: *blog-programming-python* test dataset histogram.

A.2 PART-II DATASET HISTOGRAMS

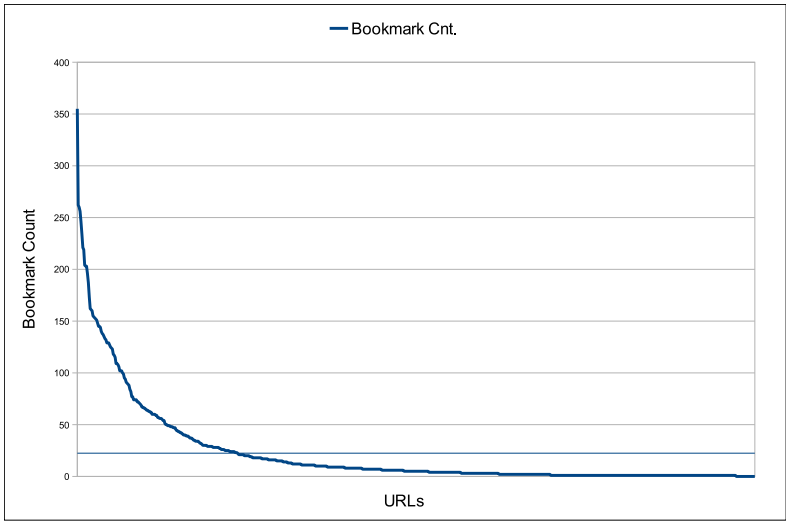

Figure A.10: Pre-processed and original *photography-camera-canon* training dataset histograms.

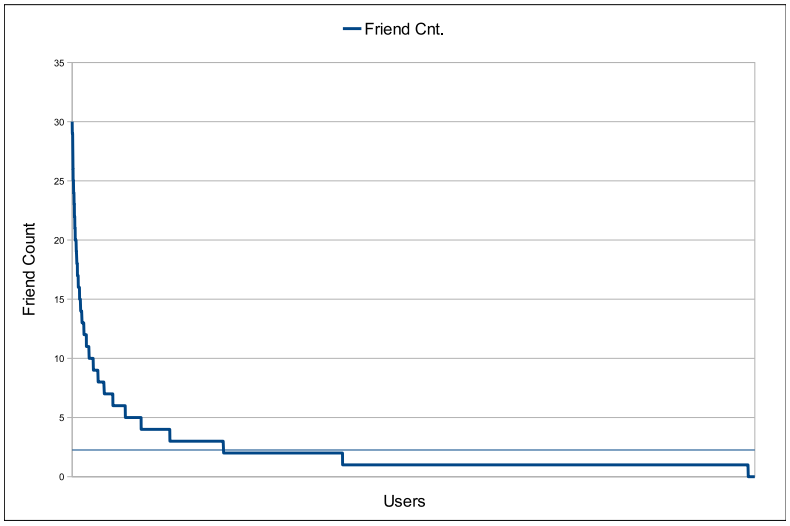

Figure A.11: *photography-camera-canon* social graph histogram.

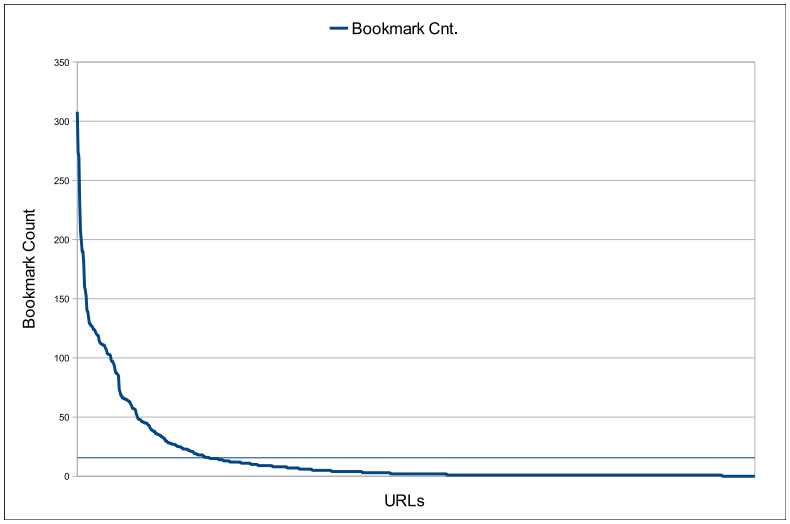

Figure A.12: Pre-processed and original *photography-camera-nikon* training dataset histograms.

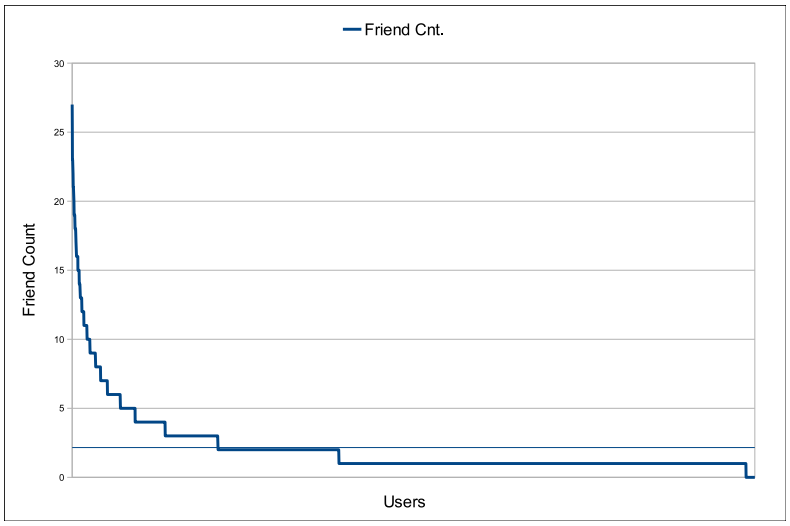

Figure A.13: *photography-camera-nikon* social graph histogram.

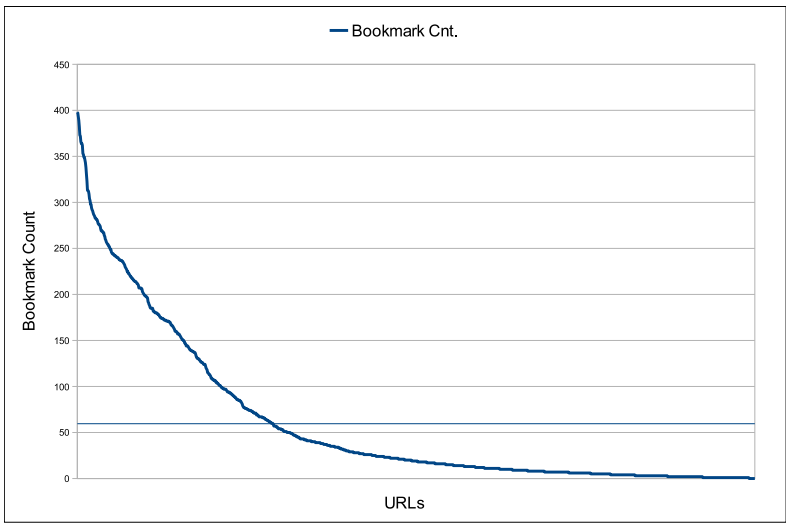

Figure A.14: Pre-processed and original *blog-programming* training dataset histograms.

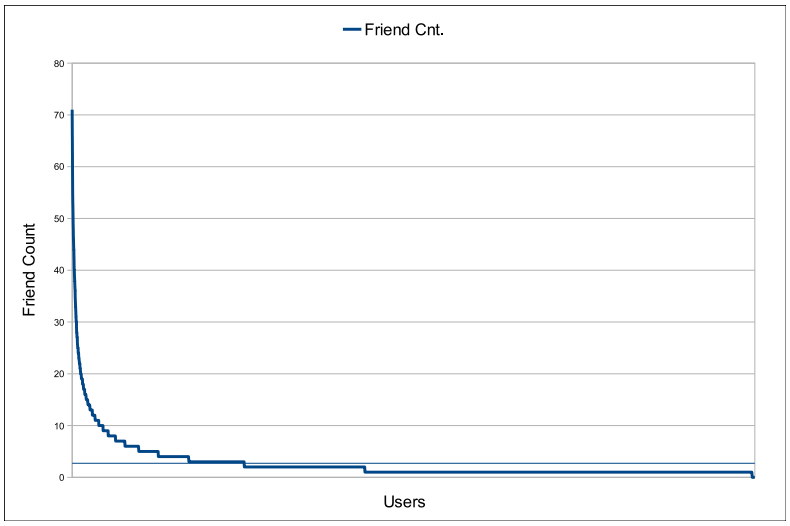

Figure A.15: *blog-programming* social graph histogram.

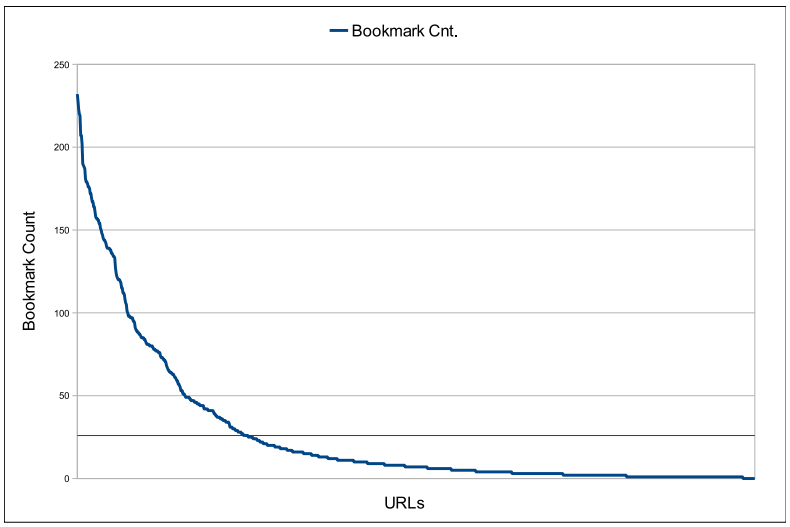

Figure A.16: Pre-processed and original *photography-camera* training dataset histograms.

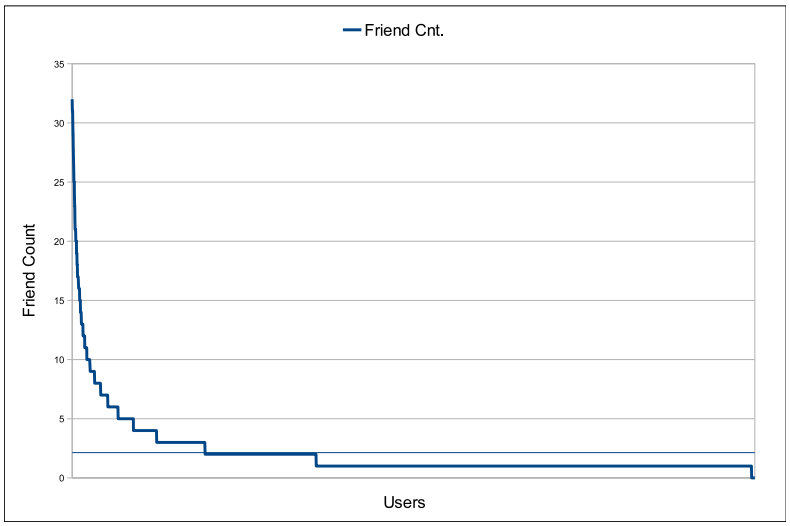

Figure A.17: *photography-camera* social graph histogram.

APPENDIX B

EVALUATION RESULTS

B.1 RESULTS OF KNN-BASED ALGORITHMS

B.1.1 IMPACT OF PRE-PROCESSING

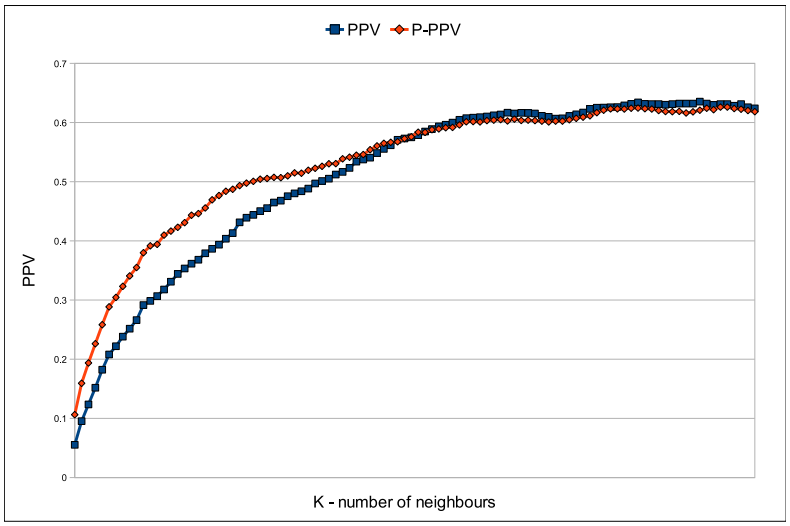

Figure B.1: *knn-base* results for *blog-programming-python* dataset and pre-processed *blog-programming-python* dataset.

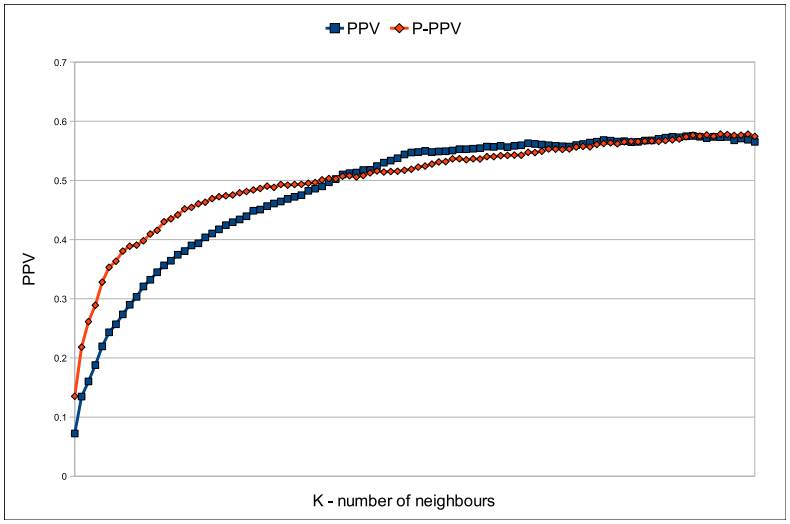

Figure B.2: *knn-base* results for *blog* dataset and pre-processed *blog* dataset.

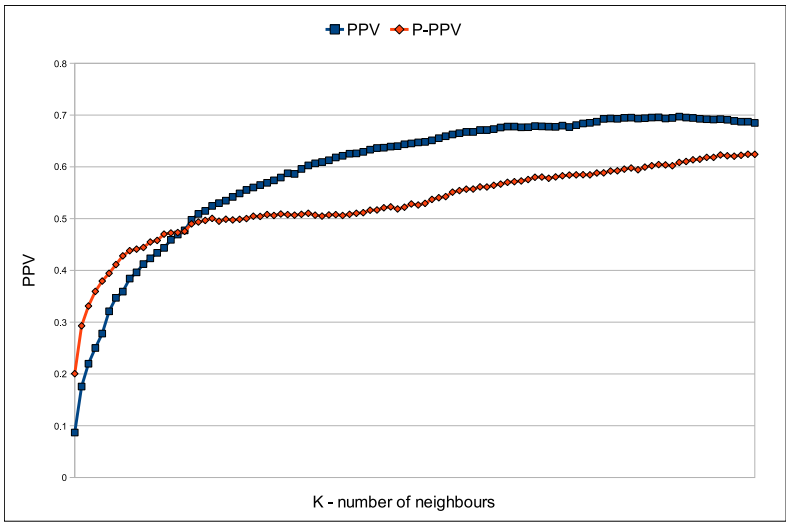

Figure B.3: *knn-base* results for *photography* dataset and pre-processed *photography* dataset.

B.1.2 IMPACT OF WEIGHTING SCHEMES

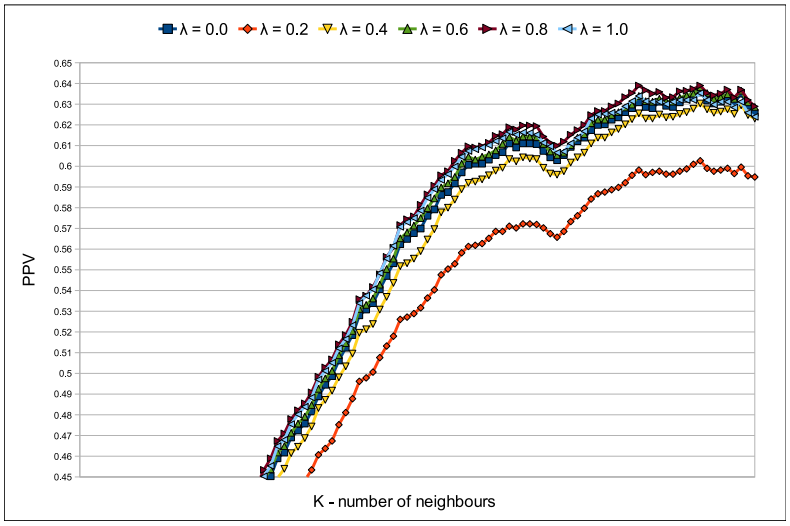

Figure B.4: *knn-sg-pred* results for *blog-programming-python* dataset.

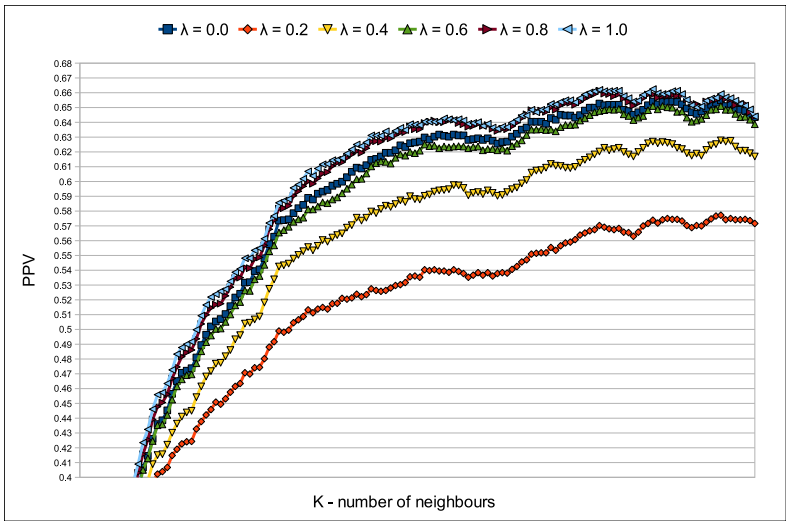

Figure B.5: *knn-sg-pred* results for pre-processed *blog-programming-python* dataset.

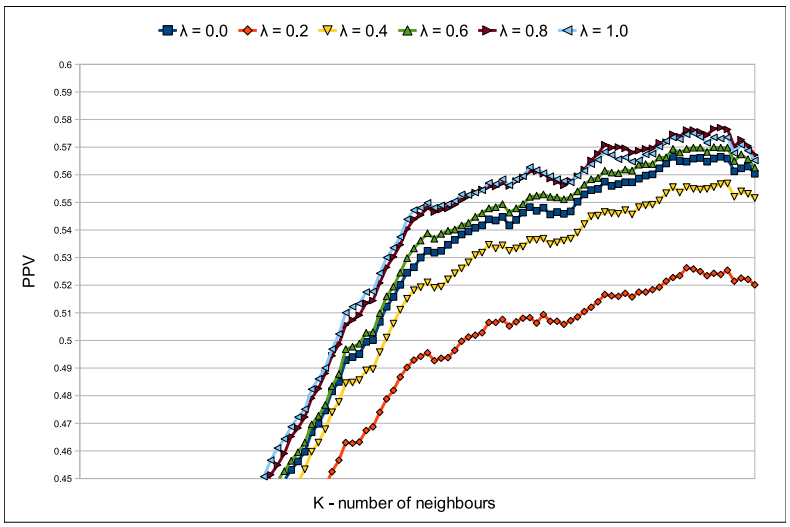

Figure B.6: *knn-sg-pred* results for *blog* dataset.

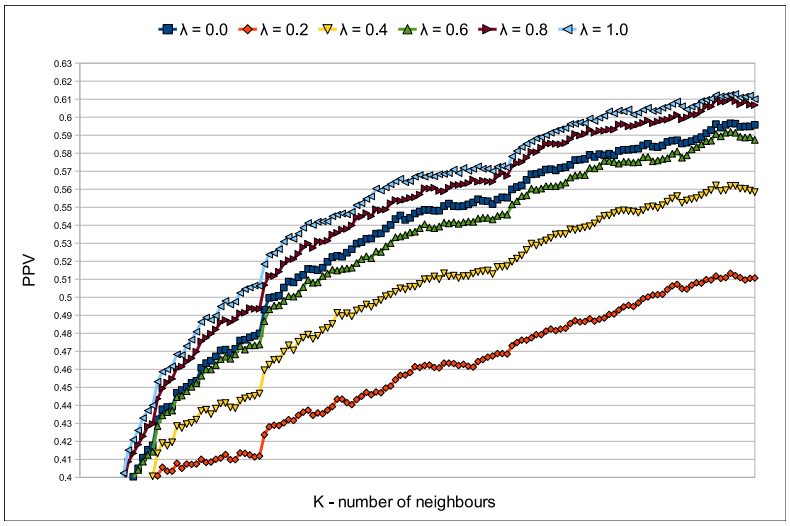

Figure B.7: *knn-sg-pred* results for pre-processed *blog* dataset.

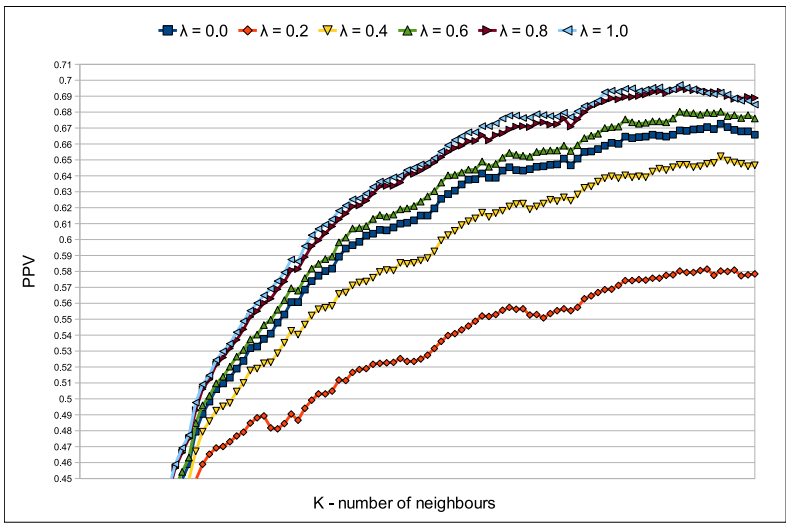

Figure B.8: *knn-sg-pred* results for *photography* dataset.

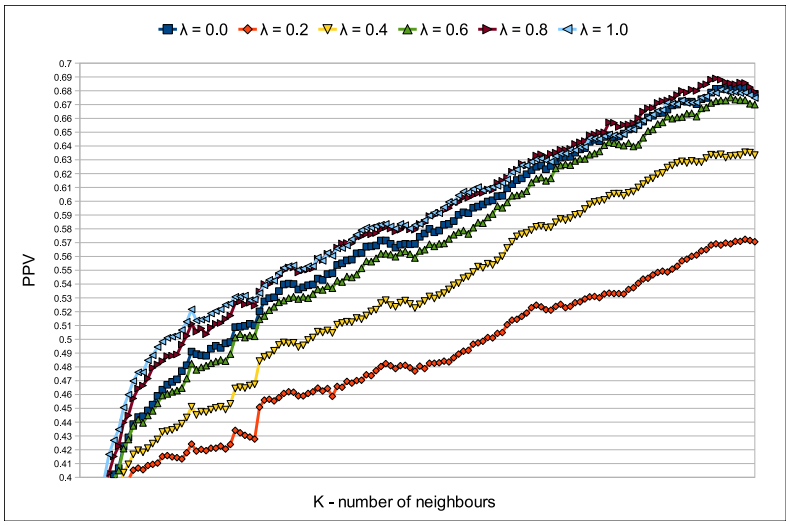

Figure B.9: *knn-sg-pred* results for pre-processed *photography* dataset.

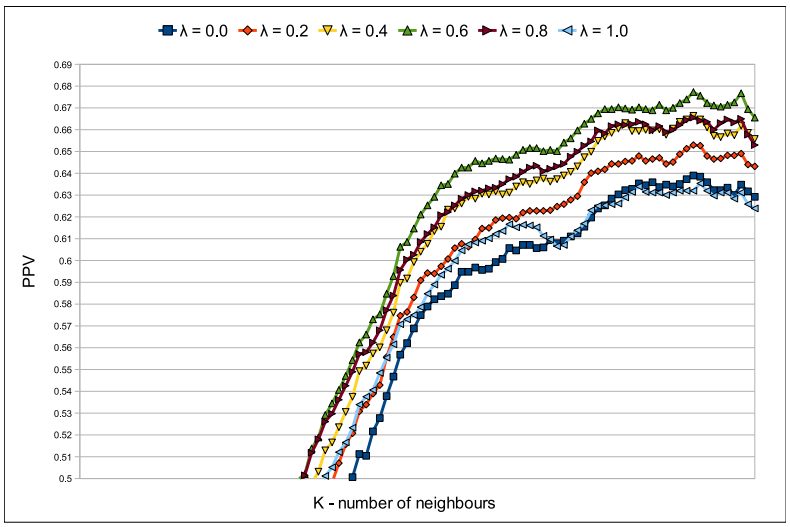

Figure B.10: *knn-sg-neigh* results for *blog-programming-python* dataset.

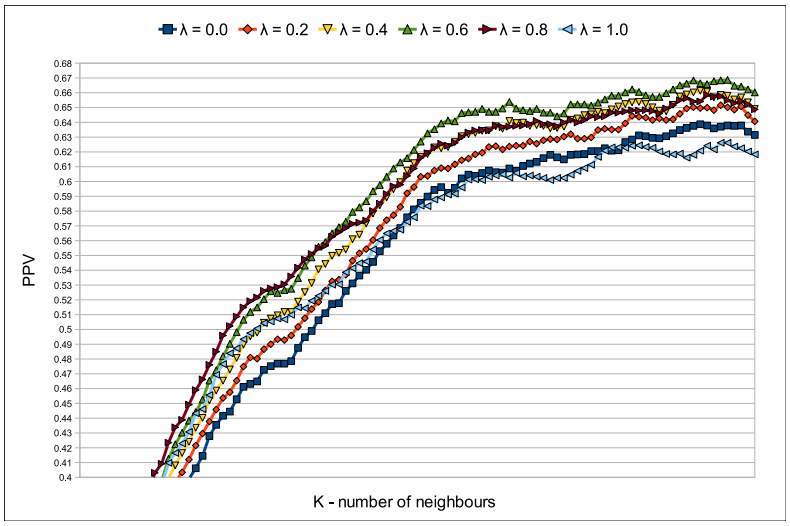

Figure B.11: *knn-sg-neigh* results for pre-processed *blog-programming-python* dataset.

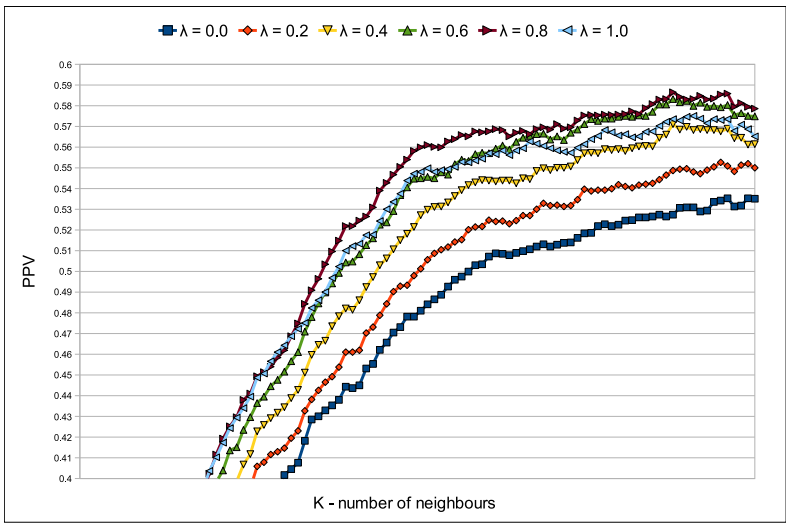

Figure B.12: *knn-sg-neigh* results for *blog* dataset.

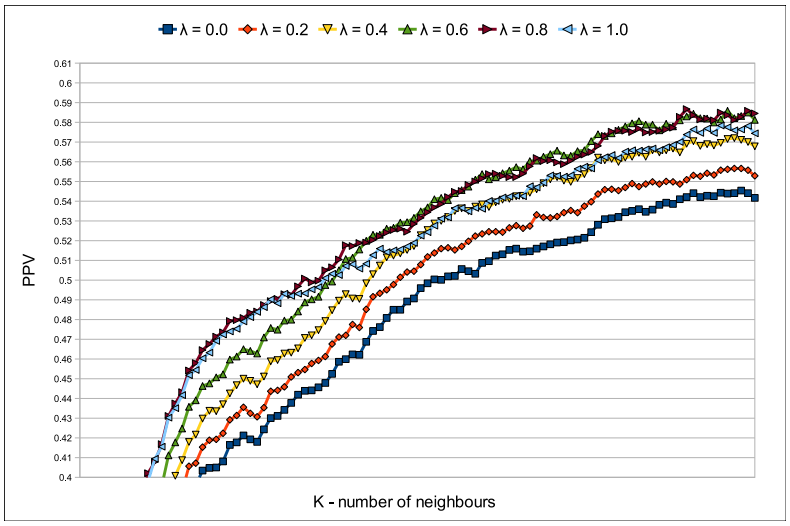

Figure B.13: *knn-sg-neigh* results for pre-processed *blog* dataset.

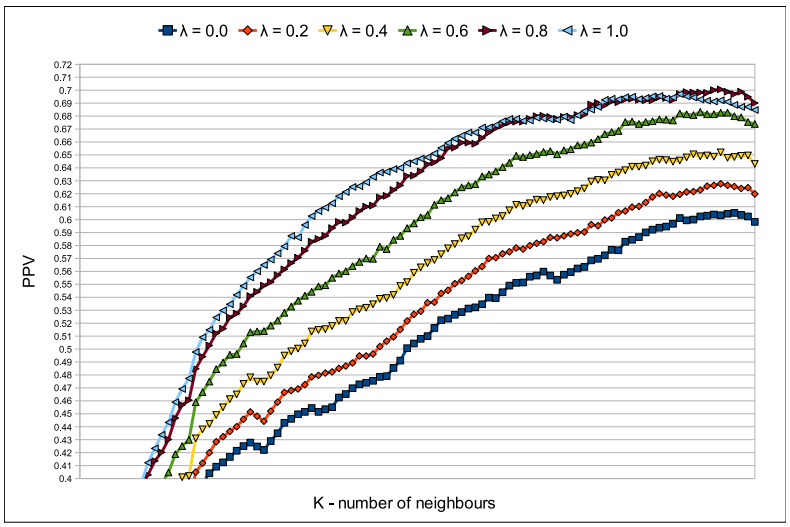

Figure B.14: *knn-sg-neigh* results for *photography* dataset.

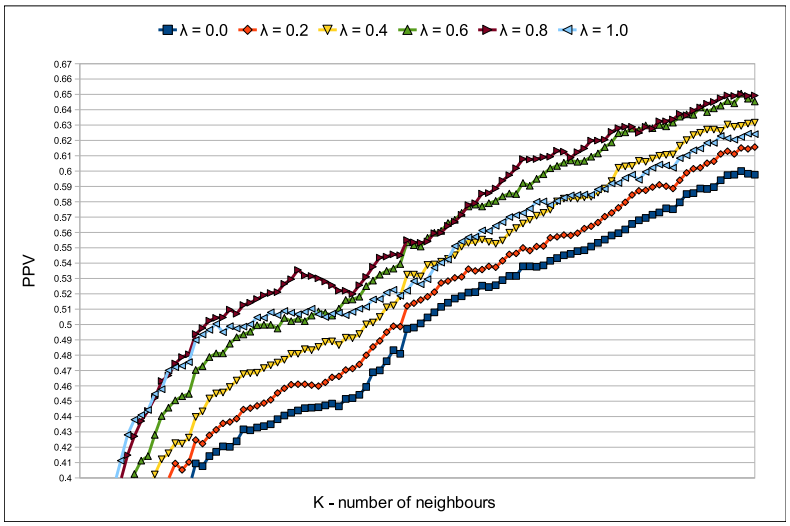

Figure B.15: *knn-sg-neigh* results for pre-processed *photography* dataset.

B.2 RESULTS OF SINGULAR VALUE DECOMPOSITION

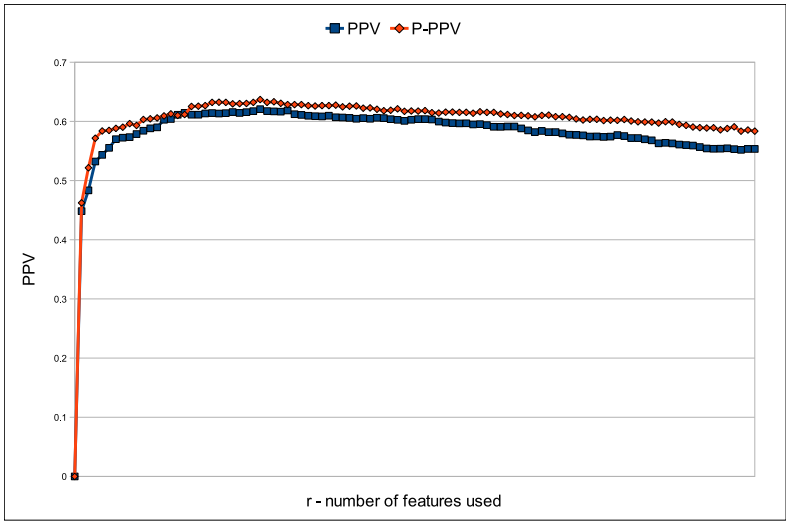

Figure B.16: *svd* results for original and pre-processed *blog-programming-python* dataset.

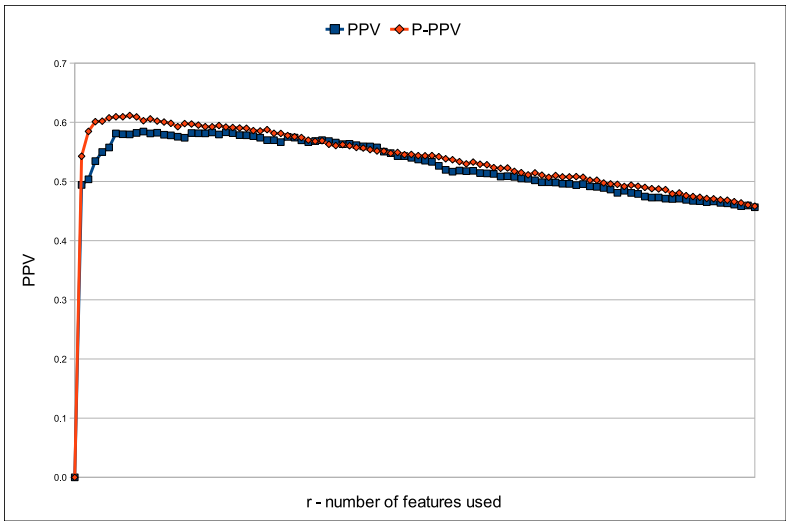

Figure B.17: *svd* results for original and pre-processed *blog* dataset.

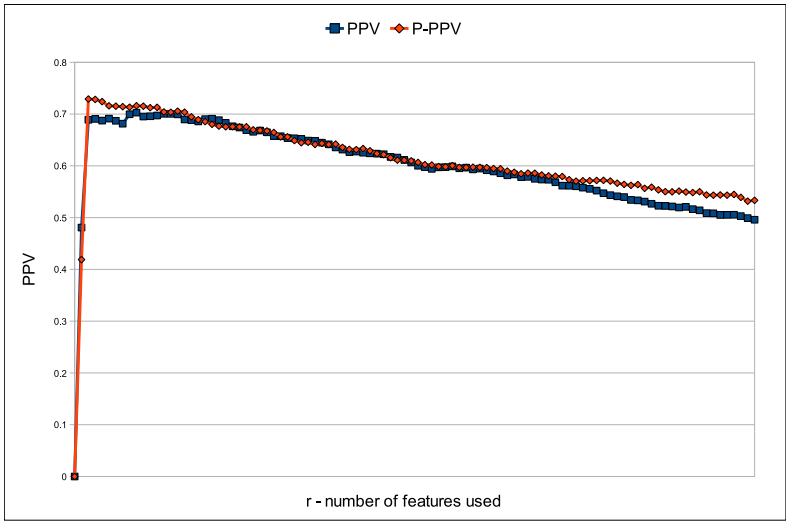

Figure B.18: *svd* results for original and pre-processed *photography* dataset.

B.3 RESULTS OF wALS BASED ALGORITHMS

B.3.1 wALS-BASE

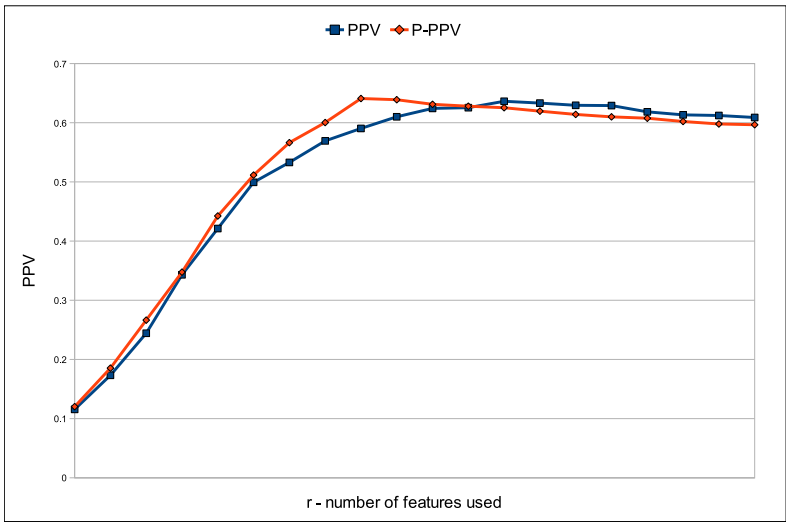

Figure B.19: *wals-base* results for original and pre-processed *blog-programming-python* dataset.

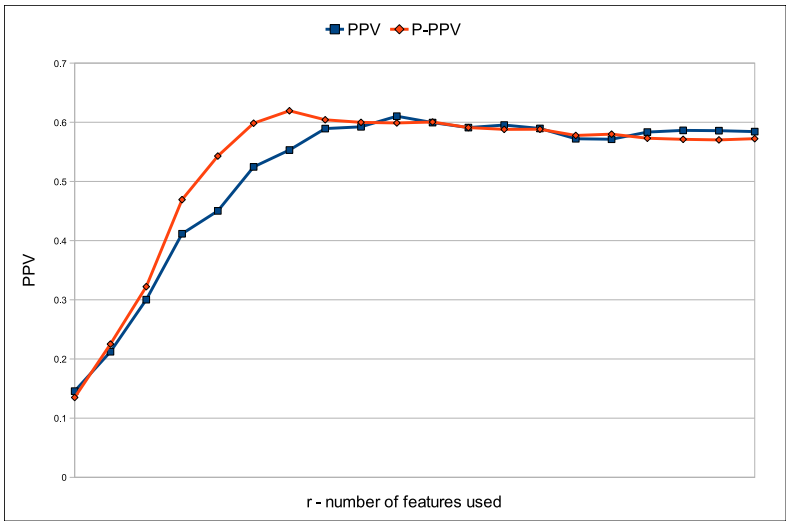

Figure B.20: *wals-base* results for original and pre-processed *blog* dataset.

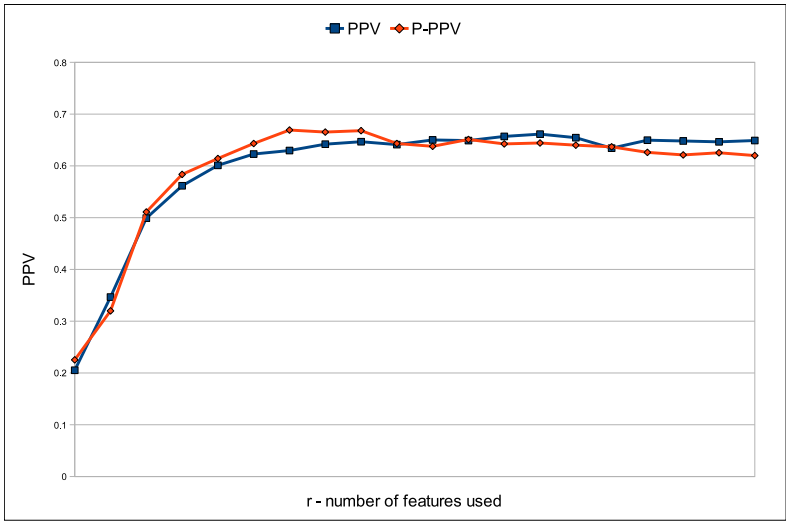

Figure B.21: *wals-base* results for original and pre-processed *photography* dataset.

B.3.2 wALS-SG-D1

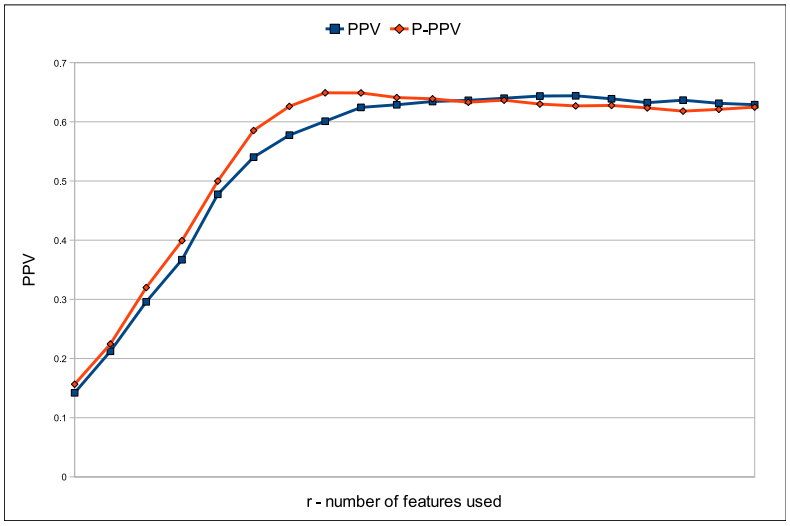

Figure B.22: *wals-sg-d1* results for original and pre-processed *blog-programming-python* dataset.

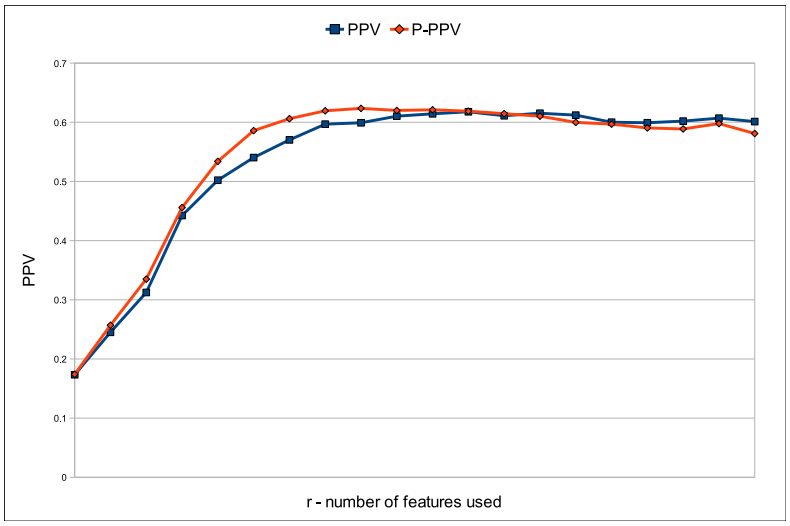

Figure B.23: *wals-sg-d1* results for original and pre-processed *blog* dataset.

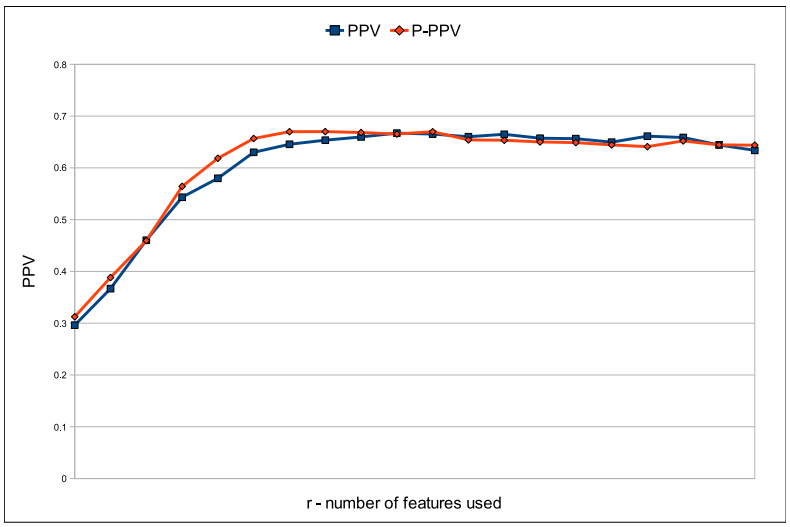

Figure B.24: *wals-sg-d1* results for original and pre-processed *photography* dataset.

B.3.3 wALS-SG-D2

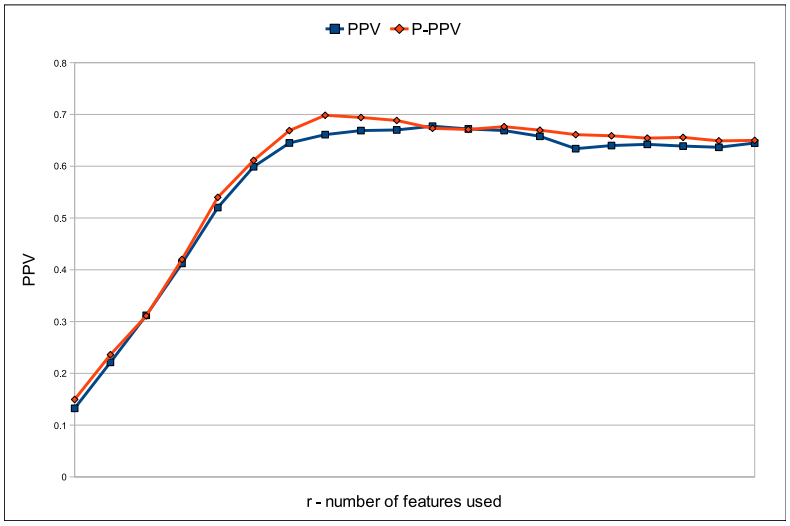

Figure B.25: *wals-sg-d2* results for original and pre-processed *blog-programming-python* dataset.

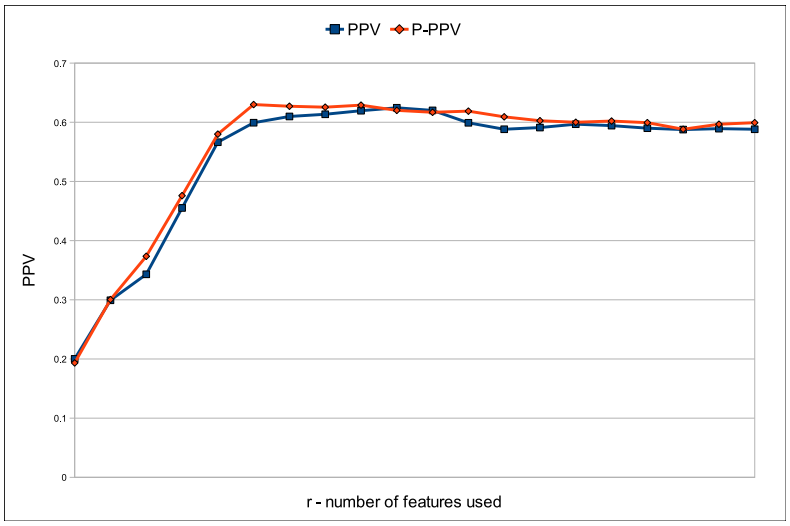

Figure B.26: *wals-sg-d2* results for original and pre-processed *blog* dataset.

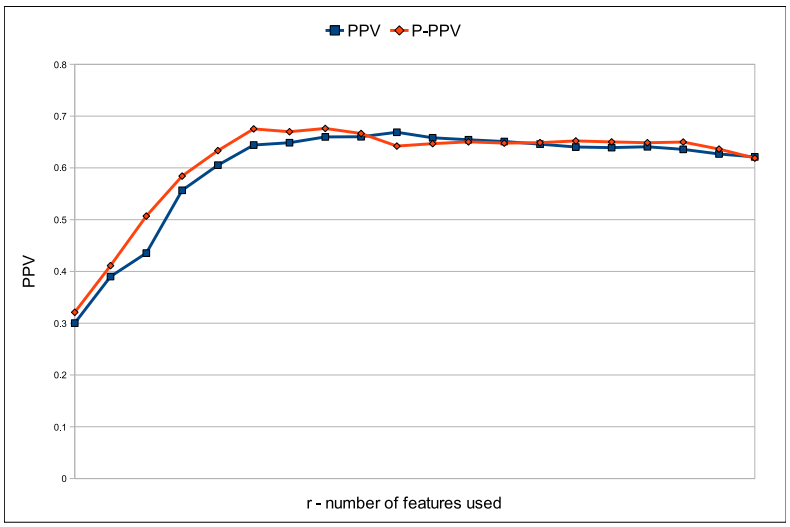

Figure B.27: *wals-sg-d2* results for original and pre-processed *photography* dataset.