Content and User Recommendation In Social Networks

Nathan Hale

Exeter College

University of Oxford

A dissertation submitted in partial fulfilment of the degree of

Master of Science

Summer 2012

Acknowledgements

Thanks Jamie!

Abstract

Social networking services have exploded in popularity over the past several years and the amount of content available on those services has grown correspondingly. With this much available content it is easy for users to be overwhelmed. Thus, it is in the interests of both the service operators and the users of such services to have access to effective methods for recommending novel content and users to connect with.

This dissertation proposes one such method, adapting and extending a method used in web search to the problem of recommending both users and content within social networks. It also demonstrates a procedure for incorporating links that would otherwise render a social graph non-bipartite into a proper bipartite graph, a technique which has never been used in research on social network content recommendation.

It is then demonstrated that this method is effective at recommending both content and users that a particular distinguished user may be interested in. This efficacy is demonstrated through both user evaluation and calculated metrics. A number of different variations to the methods are explored in order to refine the results further.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past decade, a number of internet services dedicated to 'social networking' have been created. These services allow people to share what they are doing with their friends and to see what their friends have done. They also frequently allow users to see the publicly shared content of other users who they find interesting, perhaps because they know them, perhaps because that person is a celebrity, or perhaps because that person consistently shares content that is interesting in some way such as breaking news, funny jokes, or interesting articles.

The popularity of these social networking services has grown tremendously over the past several years, and that growth has been accompanied by a corresponding growth in both the amount of content available on these networks and the number of services available. Facebook, the largest social network, had over 901 million monthly active users at the end of March 2012 and had over 125 billion friend connections between those users[1]. Twitter is another popular social network which has over 140 million active users as of May 2012, with more than 383 million users[2] having created accounts at some point since the service was founded in 2006. Dozens of other smaller social networks have been created as well, aimed both at general audiences and niche audiences.

With this explosion of content and users it has become much more difficult for users to find novel content of interest to them and to find new users to connect with who might consistently produce such content. Because this may negatively impact the user experience it is in the best interests of both users and the operators of social

[1]http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

[2]http://www.guardian.co.uk/technology/2012/may/15/twitter-uk-users-10m

networking services to provide methods to allow users to easily find interesting content and users.

Existing approaches to content and user recommendation (as described in Section 2.2.4) have focused on either a network-based approach, which examines the topology of the network and the connections between users, or a content-based approach, which seeks to take semantic and syntactic information from the content itself and use that to recommend other content. In many cases these approaches are still hamstrung by the same issues that vex individual users, namely the vast amount of data that exists in these social networks which can make analysing them a very difficult task.

Most social networks already provide some form of content recommendation to their users, usually in the form of a list of other users that they may know or be interested in which is created based on the connections that the user already has, i.e. a network-based approach. This is a step in the right direction, but it recommends only users, failing to recommend interesting individual pieces of content. On existing social networks where some content is recommended to users, content is usually only recommended in terms of what is popular amongst all users, without considering the individual user's own interests.

The purpose of this dissertation is to provide a novel and more effective method of recommending both users and content to users of social networking services despite the large amount of data involved. This dissertation will use a combination of a network-based approach and a content-based approach since both have been shown to have value. Taking both methods into account is also very helpful in making the process faster because it is possible to recommend both users and content while running only one algorithm.

In developing this method, it is hoped that such a technique could be put to use either directly by a social networking service or by a third party in order to improve the experience for their users by leading them to more interesting and relevant content. Furthermore, existing techniques for evaluating the effectiveness of such recommendations are very lacking, so it is hoped that new techniques can be used to improve the ability to evaluate both this scheme and future research in the field.

1.2 Structure

This dissertation is structured into six chapters, including this introduction describing the motivation behind the project and the problems it is intended to address.

Chapter 2 gives important background on what social networks are, with particular focus on features that most networks have in common. Two networks in particular, Facebook and Twitter, are discussed in depth to show how these generic features are implemented. This also provides insight on which of these features may be useful in recommending content and users. After giving this background information, existing research on social networks and content recommendation is examined as a means of seeing which network features might be useful for developing recommendations and what techniques have already been used.

Chapter 3 describes the general methodology used by this dissertation. The method is described without reference to any particular social network to emphasize that the technique being used can be applied to nearly any social network having the general features described in Chapter 2. The suitability of this method, known as the Co-HITS algorithm, to social network graphs is described, as are the methods used to transform the graph of the social network into a graph suitable for use by this algorithm.

Chapter 4 describes the specific implementation used by this dissertation. It describes how the data about the social network was selected and obtained and it describes how the challenges of the huge size of this data were dealt with. It then describes the implementation of creating the social graph and implementing the Co-HITS algorithm to run efficiently on this large graph.

Chapter 5 describes the results obtained using this method. Evaluating the results is not straightforward since there is not yet a canonical method of evaluating results in this research area. A number of techniques are proposed and the results of the algorithm for each evaluation technique are presented. Additionally, this chapter describes a number of modifications to the basic algorithm for recommendation and shows how each of these variations affects the results.

Chapter 6 concludes the dissertation. This chapter provides a discussion on how the performance of the specific method developed here could be improved and what future work could be done in this area to improve the overall method by modifying it more fundamentally. The dissertation ends with some concluding remarks on the method developed, its efficacy, and its suitability for real-world implementation.

Chapter 2

Background and Related Work

This chapter begins by describing what social networks are and what features one can expect to find in them. In particular, the features of two of the largest social networks, Facebook and Twitter, are discussed in detail, including how those features might be useful in recommending novel content to users. Using this description for background, the discussion then moves to research that has already been done in this field and which techniques for content recommendation have shown promise so far.

2.1 Social Networks

Facebook and Twitter are certainly the most popular social networking services, with more than 1 billion user accounts between them. There are dozens of other social networks, however, including many with millions of users such as Google+, Pinterest, and LinkedIn. It would be impractical and of limited value to discuss the nuances of all of these services and to what degree they implement particular features, but some background is necessary in order to understand the work undertaken in this project. Because Twitter and Facebook are the most popular social networking services and demonstrate all of the common features of social networks this discussion will be limited to only these two services.

Before describing these two services and their features in detail, it is valuable to begin with a brief discussion of the general features that are common to all social networks. Examining these general features provides a good basis for comparison of how the general features are implemented by the two test cases. It is also helpful as a reference to show that while the study undertaken for this project was specific to one particular social networking service, the features used were actually quite generic and could easily be extended to other services.

2.1.1 What are Social Networks?

At its core, a social network is a set of users, a set of edges connecting those users, and content produced by those users. Depending on the network and the relationships between users, the edges may be either directed or non-directed, and sometimes a given network may include edges of both types. A non-directed edge indicates that both users are subscribed to each other's content while a directed edge from user A to user B indicates that while user A is subscribed to content produced by user B and will see all public content from user B, user B will not see any content from user A.

The types of content produced also vary depending on the network. Twitter famously limits content to 140 characters of text, a limit designed to ensure that it can fit within the standard text message size limit of 160 characters. Still, those 140 characters can contain links to any other sort of content, and as the service has developed various conventions have been created in an ad hoc manner by its users in order to mark up the text to provide additional meaning. Facebook users, meanwhile, can share much longer text updates, photos, links, and videos.

Another common feature of social networks is the ability for users to create profiles to tell other users about themselves and to express their interests and desires. The importance of that profile depends on the network, and in some cases it is the primary form of content produced by users. LinkedIn, for example, is a social network designed for professional networking and job-seekers, and its user profile pages are essentially just extended CVs.

Social networks also frequently provide the ability for users to re-share or forward content from other users so that their connections can see that content. This allows links and pictures to make their way across the network and be seen by users who are not themselves connected to the original user who produced a piece of content. And, importantly for the task of recommending content, it provides a good insight into the types of content that the person re-sharing the content finds interesting.

The degree to which networks support these features varies widely, but most networks—and certainly the majority of the most popular networks—implement all of these features.

2.1.2 Facebook

Facebook was originally founded as a way for students at Harvard University to keep in contact with each other. Early versions of the website were rudimentary compared to the current behemoth—the only real features were the ability to become friends with other users and to fill out a profile indicating the user's birth date, home town, current location, interests, and similar biographical details. From those early beginnings the network expanded first to other universities and then to secondary schools before membership was eventually open to everyone.

As one of the oldest social networks which still enjoys a large user-base, Facebook has evolved quite a bit over the years since it was created. As other social networks have challenged it by introducing new features, Facebook has responded by adding versions of those same features with slight modifications to adapt them to the Facebook environment. Over the years of growth the modern version of Facebook has slowly taken shape as features such as photo sharing, the news feed, and the 'like button' have been introduced.

2.1.2.1 Features

The Facebook network is built around the concept of friends. The edges in the Facebook network are connections between users who (mostly) know each other in real life and are bi-directional. One user sends a friend request to a second user and if the second user accepts then the connection is created and each user can see content produced by the other.

Suppose that there is a user Alice who is friends with a user named Bob but not with a user named Carol. Bob, however is friends with Carol. Alice might share a piece of content C_A with her friends, in which case Bob will see the update but Carol will not. Similarly, if Carol shares a piece of content C_C, Bob will see it but Alice will not. Finally, if Bob shares a piece of content C_B, both Alice and Carol will see it since both users are friends with Bob. This visibility is shown in Figure 2.1.

The primary forms of content on Facebook are status updates, photographs, and links. Status updates are just text updates, sometimes telling other users what that person is doing, sometimes asking for advice, and sometimes just expressing emotions. Photographs and links are both self-explanatory, and are usually accompanied by captions or explanatory text from the user describing his or her thoughts on the content being shared. Each of these types of content is accompanied by a set of comments made by friends. Additionally, friends may 'like' the content by pressing the 'like' button to indicate some sort of agreement, support, or genuine like of the associated status, link, or photograph without needing to leave a comment.

More recently, perhaps in response to competition from Twitter, Facebook has introduced the ability to share (forward) content from another user and to subscribe

$$\begin{tikzcd}
Alice \arrow[r, shift left] \arrow[d, dashed, <->] & C_A \arrow[l, dotted, shift left] \arrow[dl, dotted] \\
Bob \arrow[r, shift left] \arrow[d, dashed, <->] & C_B \arrow[l, dotted, shift left] \arrow[ul, dotted] \arrow[dl, dotted] \\
Carol \arrow[r, shift left] & C_C \arrow[l, dotted, shift left] \arrow[ul, dotted]
\end{tikzcd}$$

Figure 2.1: An example of how visibility of content works on Facebook. The dashed edges indicate the friendship connections between users. Directed edges from users to content indicate authorship, and dotted directed edges from content to users indicate which users can see that content.

$$\begin{tikzcd}
Alice \arrow[d, dashed, <->] \arrow[dr] & \\
Bob \arrow[d, dashed, <->] & C_A \arrow[ul, dotted] \arrow[l, dotted] \\
Carol &
\end{tikzcd}$$

(a) Before Bob reshares C_A

$$\begin{tikzcd}
Alice \arrow[d, dashed, <->] \arrow[dr] & \\
Bob \arrow[d, dashed, <->] \arrow[r, shift left] & C_A \arrow[ul, dotted] \arrow[l, dotted, shift left] \arrow[dl, dotted] \\
Carol &
\end{tikzcd}$$

(b) After Bob reshares C_A

Figure 2.2: Visibility of content on Facebook before and after content is re-shared. The dashed edges indicate the friendship connections between users. Solid edges between users and content indicate content authorship and resharing, and dotted edges between content and users indicate visibility. This figure does not include the situation where Alice subscribes to the public updates of Carol.

to public posts of another user or group, essentially creating a directed edge between those users. Returning to the example of Alice, Bob, and Carol, if Alice shares content C_A, Bob will see it but Carol will not since Alice and Bob are friends while Alice and Carol are not. Bob can re-share this, however (provided that Alice's privacy settings allow this) at which point Carol will be able to see it. This visibility relationship is shown in Figure 2.2. Additionally suppose that Carol subscribes to the public updates of Alice. In this case, if Alice shares a photograph publicly, Carol will be able to see it without Bob having to re-share it since she subscribes to Alice's public updates.

As alluded to above, various privacy controls have been put in place over the years to give users control over who can see what content. Though the original idea of Facebook was that friends on Facebook would correspond to friends in real life, the actual mode of use didn't follow and people often found themselves in nominal 'friendships' with acquaintances or people from different social circles with whom they didn't necessarily want to share all of their content. Thus, users can choose to share specific content only with specific sets of people and to restrict the re-sharing of their content. Additionally, users must decide whether posts will be public, visible only to friends, or visible only to specific friends.

Other popular features of Facebook allow users to create events and send invitations to their Facebook friends and the ability to run applications and games developed by third-party developers which leverage the user's social data to provide additional experiences.

2.1.2.2 Content Recommendation

Many of the basic features of Facebook would be very applicable for use in a content recommendation scheme. The basic edges between users present an obvious starting point, but other features could be leveraged to put a more accurate weight on edges between users. Users who frequently comment on one another's posts are more closely connected than users who never interact, for example.

Similarly, if a user likes a post or comments on it then it indicates that the content itself was relevant to that user, information which could be used to recommend future content of interest to that user. Attending common events would be another method of determining just how close two nominal 'friends' actually are—users who go to the same events in the real world are obviously much more likely to be good friends than users who do not.

The large amount of information on a Facebook user's profile is another piece of information that could be leveraged to recommend content. If two users declare similar interests on their user profile then it would be more likely that those users would be interested in content generated by one another.

Facebook itself has developed a system based on many of these components to help it recommend other users that it believes you may know and want to become friends with. One such project is described in [4].

The major drawback to studying content recommendation on Facebook is that the data is difficult to access. Most of the content created by users and the edges between those users are private and thus inaccessible to researchers. Therefore most studies focusing on Facebook, including that in [4], are undertaken at least in part by employees of Facebook participating in research and development to help advance the interests of the company.

The private nature of this data made it unsuitable for study in this dissertation, though many of the other components of the network would have provided a rich set of features to use in content recommendation. Facebook is discussed here to show that it has analogues to most of the major features of Twitter and that all of the research undertaken in this dissertation could easily be applied to Facebook if the data were available.

2.1.3 Twitter

2.1.3.1 History

Created in 2006, Twitter is a social network noted for the brevity of its content. The basic concept is simple: users provide updates on what they are doing or thinking in up to 140 characters. The basic interface has changed very little since its inception and the feature set is still very similar to its beginnings, in marked contrast to the large number of features that are integrated into Facebook.

The original intent was that users could update their status and receive statuses from friends using SMS messages from their mobile phones, and thus the length of any given Twitter message was limited to be smaller than the maximum size of an SMS message, 160 characters.

The nature of the network began to change as smartphones—mobile phones capable of running applications and communicating with the internet—became ubiquitous following the release of Apple's first iPhone in 2007. Twitter provides an extensive Application Programming Interface (API), which has allowed application developers to create rich and visually pleasing interfaces to the service that have helped boost its popularity by making it even easier to use. As smartphones grew in popularity the need for and popularity of status updates via SMS faded, though the 140 character limit has remained.

Today, Twitter has over 140 million active users worldwide, and those users produce hundreds of millions of status updates each day. As a result, numerous attempts have been made to make sense of this vast source of public data about the feelings and activities of millions of people around the world. Research into this area is discussed in greater depth in Section 2.2.2.

$$\begin{tikzcd}
Alice \arrow[r, shift left] \arrow[d, dashed] & C_A \arrow[l, dotted, shift left] \\
Bob \arrow[r, shift left] & C_B \arrow[l, dotted, shift left] \arrow[ul, dotted]
\end{tikzcd}$$

Figure 2.3: An example of how visibility of content works on Twitter. The dashed edge indicates that Alice follows Bob, but Bob does not follow Alice. The dotted edges represent who will see each piece of content in their Twitter stream. Note that Alice or Bob could always see the other users's content by visiting their Twitter feed, regardless of any follower/followee relationship between them.

2.1.3.2 Features

Twitter updates are commonly known as *tweets*, and the act of producing one is known as *tweeting*. Within the research literature on Twitter, one person's *Twitter feed*, the collection of all of the tweets that they have produced, is commonly called a microblog in reference to the previously mentioned 140 character limit imposed on each tweet. Edges between users are directed. If a user Alice follows a user named Bob but Bob does not follow Alice, Alice will see all of Bob's tweets, but Bob won't see any of Alice's tweets. This is shown in Figure 2.3.

The terminology which will be used by this dissertation will be to refer to Alice as one of Bob's *followers* and to refer to Bob as one of Alice's *followees*. In contrast to Facebook, most tweets are visible to the public; the act of following Bob simply causes Bob's tweets to appear along with the tweets of Alice's other followees in Alice's *Twitter stream*, which is the set of tweets of everyone that Alice follows. If Bob cared to see all of Alice's tweets he could simply visit her Twitter feed, which collects all of Alice's tweets.

Initially, this was the complete feature set of the service; it was possible to post updates and to subscribe to the updates of others. As Twitter became more popular its users began to develop a shorthand for indicating things to other users. As that shorthand became more codified many of these features were given native support by Twitter. The most important of these features are hashtags, mentions, and retweets.

Since the service only permits the sharing of text directly, it does not internally support sharing pictures, videos, or links. However, web addresses contained within the text of a tweet are highlighted and automatically turned into links, allowing pictures, videos, or any other content to be shared by hosting it externally and providing a link. A number of image hosting websites were developed specifically to support user uploads of photos which can then be linked from Twitter. The biggest issue with linking is that many web addresses are by themselves more than 140 characters, and even those which are not often would not leave much room for comment if included in their entirety. As a result, URL shortening services (such as bit.ly, t.co, and ow.ly), which take a complete URL and hash it down to 5 or 6 characters, have become popular. An example of such a link is seen in Figure 2.5.

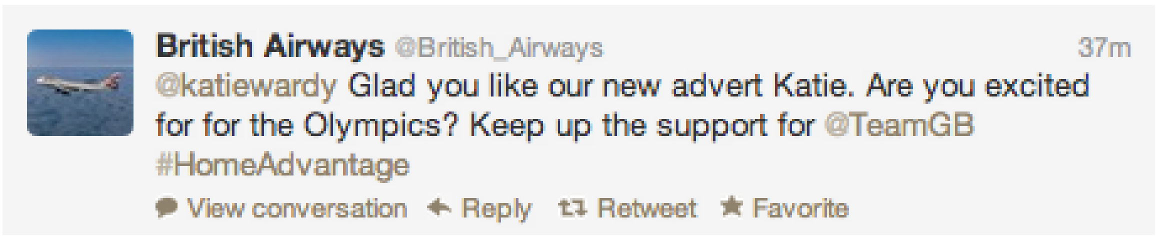
Hashtags Hashtags were created organically by users of Twitter[1] as a way of tagging tweets on a similar topic. Thus, someone might tag a tweet about the University of Oxford by appending '#oxford' to the message. Or, in keeping more with the brevity that is prized within tweets, they might integrate the hashtag into the message itself, something like 'Currently visiting the University of #Oxford'. Figure 2.5 contains a screenshot of a tweet which includes a hashtag.

Tagging tweets in this way clusters tweets which all discuss a particular topic and makes it easy to search. As this feature became widely adopted by users it was integrated to have native support by Twitter. Today, such hashtags appear as highlighted links within a tweet and clicking the link brings up other tweets that have used the same hashtag recently.

Twitter collects these hashtags and provides a list of topics that are currently popular amongst users. While these hashtags must not contain spaces or punctuation in order to be parsed properly, they are often stylized to contain entire phrases and questions to which users provide answers, such as #MyFavouriteSongIs for discussing music. While certainly commonly used for banalities and idle chatter, these hashtags can also take on great importance for providing information during breaking news events. For example, during the Iranian political unrest of 2009, hashtags such as #IranElections became popular for users in that country to disseminate information about protests [5].

Mentions and @replies Mentions are another important feature of Twitter, and like hashtags, they also started out as a convention amongst users before being adopted as a native feature by Twitter[12]. Mentions are simply a way of directing a tweet to the attention of a particular person. This is accomplished by pre-pending an @ symbol to their user name and including it in the tweet. Thus, to mention Bob, Alice would type '@bob' in her tweet. Mentions are collected by Twitter and can be seen

[1]https://support.twitter.com/articles/49309-what-are-hashtags-symbols


British Airways @British_Airways 37m

@katiewardy Glad you like our new advert Katie. Are you excited for for the Olympics? Keep up the support for @TeamGB #HomeAdvantage

Figure 2.4: This tweet is an @reply to a user named 'katiewardy' but also contains a mention of a user named 'TeamGB', the official Twitter page for the United Kingdom's Olympic Team

even if the user being mentioned, henceforth referred to as the *mentionee*, does not follow the mentioner. These mentions do not appear on a person's main Twitter feed, but are instead accessed from a special mentions page. If the username following the '@' character does not exist, the link is still highlighted but is not valid.

One distinction that is made is between an @reply and a mention. An @reply is the colloquial term given to mentions where the @username syntax appears at the beginning of the tweet as in the following sample tweet from Alice: "@bob Hope to meet you after the presentation today!". Because this is an @reply, this tweet would not show up in the main Twitter streams of Alice's followers (unless they also follow Bob). Since Bob doesn't follow Alice, this @reply also would not show up in his Twitter stream, only in his mentions. In contrast, a mention is a tweet that contains the @username syntax anywhere except at the beginning of the tweet, such as in this example tweet from Alice: "Hoping to meet @bob after the presentation today!". These tweets are visible on the Twitter streams of Alice's followers just like any other tweet, but are also collected for Bob along with all other tweets that mention him. Figure 2.4 is a screenshot of a tweet which includes both a mention and an @reply.

Retweets The final major feature of Twitter is yet another syntactic convention that was developed naturally by users and then given native support by Twitter: the retweet. A retweet is when a user repeats a tweet by another user to pass it along to their own followers, analogous to the re-sharing feature on Facebook. The common syntax is 'RT @username [original tweet]', with the RT standing for retweet. The retweet is often sent along with a comment from the person retweeting about how they feel about what was said. While no specific convention exists for where to place this comment, two conventions often seen are to place a comment before the RT and to place it after the original tweet with some indication of the divide between the

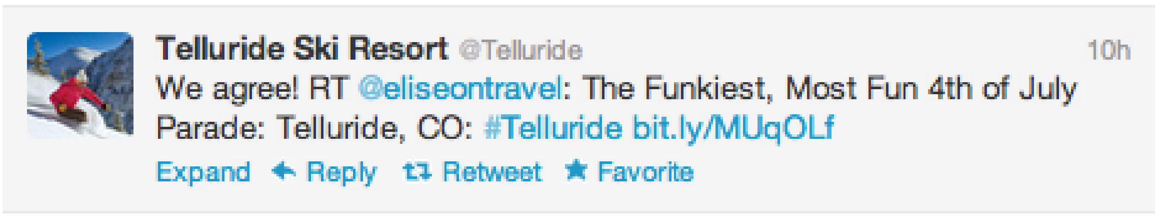

Telluride Ski Resort @Telluride 10h
We agree! RT @eliseontravel: The Funkiest, Most Fun 4th of July Parade: Telluride, CO: #Telluride bit.ly/MUqOLf

Figure 2.5: An example of a Twitter retweet. Here, the resort has forwarded the positive tweet by user 'eliseontravel' along with a comment of 'We agree!' to their own followers. This tweet is also an example of a tweet containing a hashtag which was used in the original tweet: '#Telluride'. The 'bit.ly/MUqOLf' at the end is a shortened link to an external website.

original tweet and the comment such as '//' or '...'. Figure 2.5 contains a screenshot of a tweet which includes a retweet which follows the convention of placing a comment before the RT.

2.1.3.3 Content Recommendation

All of these features of the Twitter network are useful in content recommendation, and many of them have been used in existing studies as described in section 2.2.4. The primary advantage of the Twitter network as an object of study, however, is that the data is freely available and accompanied by an extensive API to allow the data to be queried and searched. This API has been used by numerous researchers to download large sets of tweets and the social graph accompanying it indicating which users follow which other users.

In fact, while studying Twitter it is actually a case of having far too much data to deal with easily than of not having enough data. Even though each tweet is no more than 140 characters, if the assumption is made that 100 million tweets are produced on a particular day (a very low estimate with today's usage), then that single day will contain roughly 14GB of data. Fortunately, previous researchers such as [16] and [6] have already created some parsed versions of the data that can be used. More recently, a track of the Text Retrieval Conference (TREC)[2] has been opened with regards to microblog research along with an enormous collection of tweets from 2011. Though these datasets are all extremely large and thus useful for research, they still represent only a small fraction of the total data present.

[2]http://trec.nist.gov/tracks.html

This vast amount of readily available data meant that Twitter was by far the best option to study in this project, even though most of the concepts would be simple to apply to other social networks given the data since most of them have their major features in common with Twitter. As such, the remainder of this chapter will focus on Twitter since it is the object of study for this dissertation.

2.2 Existing Research on Social Networks

Twitter's simple and compact format and vast quantity of data has made it a popular target for research. The main topics of research are on the nature of the Twitter network itself, whether it is possible to gather useful information from Twitter, methods for determining what topics interest a particular Twitter user, and methods for ranking and recommending both tweets and users.

2.2.1 The Twitter Network

One of the earliest studies of the Twitter network was that of [12], from 2007, only a few months after the service launched. The study was undertaken when Twitter had fewer than 100,000 users and during the two month study period only 1.3 million tweets were made. Clearly things have changed since this study, but it does still provide a number of interesting results. For example, they found that Twitter showed a high degree correlation—users with a large number of followers also tended to have a large number of followees. In the same vein, the distribution of indegree and outdegree had a similar power exponent to that of the web and the blogosphere.

The most important findings of [12] were the different categories of users and of content. They found that the content produced mostly fell into four different categories: daily chatter, conversations, sharing information, and reporting news. Importantly, conversation comprised nearly one-eighth of the tweets in their dataset. Presumably if a similar study were repeated today it would find a fifth content category: spam. Users were found to fall into three categories: information seekers, users who may tweet infrequently but still commonly check the site for the tweets of others; information sources, users who post information that others find valuable; and friends. At the time of the study friendship links constituted most of the links between users, though with more than 100 million users today, this is unlikely to be the case.

A later study by [5] indirectly speaks to how much growth Twitter underwent between 2007 and 2010. At the time of that study in August 2009 Twitter had grown to nearly 54 million active users with 1.9 billion links between them and more than 1.7 billion tweets. The threshold for popularity investigated here, one million followers, is more than ten times greater than the total number of users in the 2007 study of [12]. [5] found that while Twitter users with one million followers might seem intuitively to be the most influential, the number of followers alone was in general a poor measure of the influence that a user has.

[22], a 2011 study, investigated which links in the twitter network (e.g. follower links, retweet links, etc.) were most likely to preserve topical relevance. They found that the most important link type for preserving this topical relevance is retweet links. Crucially, they also found that "traversing even a single follow link dramatically reduces the probability of topical relevance". In carrying out this study they repeated some of the experiments of [12] and found that most of the results from there still held true, despite the massive growth of the service.

A very important study for the purposes of this project is that of [16], an in-depth examination of characteristics of the Twitter network. In many ways it is similar to the early 2007 study of [12], but with a much more mature network. They studied things such as how likely a user was to reciprocate when someone followed them, the relationship between followee/follower numbers and number of tweets, the distribution of followee/follower numbers, the nature of trending topics, and the reasons behind and impacts of retweets, among other topics.

An important result from this study is that the number of followers/followees that users have behaves according to a strong power law with a long tail. There are hundreds of thousands of users with fewer than ten or twenty followers or followees, while only 40 users at the time of the study had more than one million followers. And of the users, 67% were not followed by any of their followees! The results also showed that how many tweets a user had was a strong predictor of both how many followers and how many followees that person had, at least up to about 100 follower or followees.

Those results are interesting, but the crucial outcome of [16] for this project is the large dataset that was produced in order to find those results, of which the complete social graph (more than 1.2 billion edges) which was included and used for this project is the most important component. The social graph from this study was the main one used in the project, as discussed in Section 4.1.

2.2.2 Gathering Information From Twitter

Given the vast size of the Twitter network and the vast amount of data produced by its users in talking about what they are doing and how they feel, it is not surprising that studies have been undertaken to try to mine this data to gain more insight into what people believe and what they're discussing.

The study undertaken by [19] attempted to predict the results of opinion polls based only on Twitter data. This process makes perfect sense when one considers that an opinion poll is really just an attempt to determine how a large population feels about something by sampling a small portion of that population and asking them how they feel. Analysing Twitter data, then, is akin to asking a much larger sample of that population.

Using a dataset of one billion tweets collected in 2008 and 2009, the research investigated three different possible opinion applications: election polls, job approval polls, and consumer confidence. For each application, tweets were analysed from a relevant period of time, such as in the run-up to the presidential election. For each tweet that contained certain trigger words that indicated relevance to a particular poll, the sentiment was analysed by simply seeing if a tweet had positive or negative words. All of the scores both positive and negative for a given period were summed and amalgamated and then compared to opinion polls from the same period of time.

The results were not perfect but did show a strong correlation for both job approval ratings and consumer confidence. Predicting election polling proved to be a more difficult task, and the correlation on those polls was not nearly as good. The strength of these results based on a very rudimentary method of sentiment analysis suggests that better techniques for analysing the tweets could improve the results.

A similar study was performed in [3], this time to try to predict the box office gross revenue of films. This study also reported some promising results from mining the sentiment of vast numbers of tweets in order to determine peoples' opinions.

2.2.3 Determining User Interest

A number of studies have focused on determining which topics are most often discussed by a user and which topics a user is most interested in reading about. The techniques from this area are very useful to the work undertaken in this project, particularly in determining the similarity of different pieces of content, e.g. as used in Section 3.3. Section 4.3.3 describes the use of named-entity recognition (NER) for purposes of determining which tweets are related, but any of the techniques used in this section could be used instead, probably with better results, albeit at the expense of greater run times.

One study, [18], used an ontology-based approach to determine a user's topics of interest. In this case, the ontology was the category structure of Wikipedia. For each tweet by a particular user, named entities were found using a named-entity recognizer. Those entities were then looked up in Wikipedia in order to disambiguate them and determine their categories. After repeating this process for all tweets by a user the process yields a pretty good set of the top ten topics of interest to that user.

A more mathematical approach was taken by [21]. They used a partially supervised machine learning algorithm and achieved good results when categorizing the tweets. Their particular machine learning approach was Labelled Latent Dirichlet Allocation, which finds distributions of words which tend to occur in similar documents. These sets of similar words are taken to be the topics, and the topics of interest to particular users can determined by analysing their tweets for these words. The authors were unsure at the outset if this technique, commonly used for long documents such as news articles, would work for documents as short as tweets, and their results indicate that it is quite successful.

A more recent method, [20], attempted to determine the topics of interest for a particular distinguished user by finding which of their followees were most important and setting the topics discussed by those users as the topics of interest for the distinguished user. The study used an approach based primarily on the Twitter network, including follower/followee relationships and retweet/mention relationships in order to determine the list of top users. As in other studies, retweets were found to be the most reliable and were thus weighted the most heavily. Upon identifying the top users, they took an approach similar to that of [18], using Wikipedia to look up terms and disambiguate them, though they used nouns returned from a part-of-speech tagger rather than named-entity recognition.

2.2.4 Content and User Recommendation

Two main approaches exist in current research on recommending content and users, the network-based approach and the content-based approach. In the network-based approach, the connections between users and their interactions with one another via retweets and mentions form the primary basis for recommendation. In the content-based approach, meanwhile, the analysis is primarily focused on the tweets themselves, such as by determining which tweets are most similar to one another. The two approaches are by no means mutually exclusive and are often used in complementary ways.

The research described here shows that when recommending users and content, each approach has had some degree of success.

2.2.4.1 Recommending Users

[16], the wide-ranging study that provided the dataset used for this project, included some analysis of potential network-based techniques for ranking users, though it was not the primary focus of the study. They used several techniques: ranking users in terms of number of followers, running the PageRank algorithm on the social graph, and ranking users in terms of how many times their content had been retweeted. As expected based on the results of [22], the number of retweets was the best of these ranking schemes. PageRank and the number of followers metric produced nearly identical results, suggesting based on the results of [5]—namely that a user's number of followers is a poor measure of their influence—that PageRank is not a good technique to use for recommending users.

Still, PageRank has been used by other studies, such as in [23]. They called their method TwitterRank because it differed slightly from original PageRank. Rather than visiting all users with uniform probability over links as in the random walk of PageRank, TwitterRank performs a topic-specific random walk using topics derived using Latent Dirichlet Allocation. Their results indicate that this modification allowed TwitterRank to outperform both the PageRank metric and a metric based on the indegree of nodes when ranking tweets. The performance improvement over page rank was small, however.

A hybrid approach utilizing both network and content information was taken by [2]. The goal of the research was to provide user recommendations in some sort of a ranked order. A network based approach was taken first in order to discover potentially interesting users. From that unordered list of potentially interesting users a ranked list is created by comparing the users based on the content that they produce to a reference document of what the user is interested in. Two strategies specifically used for building the user profile are the user's own tweets and the tweets of the user's followees.

Another piece of work focused on recommending users is presented in, [11], which recommends users based purely on content. As with a number of other approaches, the major choice explored is what to use as the reference content to which other users are compared. Several different strategies are compared, some of them overlapping with those examined by [2]. The tweets of the user, their followees, and their followers were all considered, as were the lists of users' followers and followees. Additionally, combinations of these methods were employed in a hybridized approach. For each means of building them, the user profile documents were compared as TF-IDF valued word vectors (see Section 3.4.2 for more details on TF-IDF weighting). The hybrid approaches performed the best, generally speaking.

2.2.4.2 Recommending Content

Attempts to rank individual tweets have also been made, such as in [8], which used a content-based approach to this recommendation. Here, various content features are gathered for each tweet and then a learning algorithm is employed to evaluate which of these features should be ranked most highly. The various features that were evaluated can be divided into three basic groups: those based on the content of the tweets themselves, those based on Twitter-specific features, and those based on the authority of the users who created the tweet.

The features for the content of the tweets were represented by the Okapi BM25 or cosine similarity score between a tweet and a query being evaluated. The twitter-specific group included features such as the length of the tweet, whether it contained hashtags, whether it contained a URL, and how many times it had been retweeted. Finally, the account authority group included metrics such as how many followers the author of the tweet has and their PageRank score within the twitter network. Surprisingly, the mere presence of a URL was found to be the most important feature, and the length of the tweet was also found to be very predictive.

2.2.4.3 Recommending Both Users And Content

The work of [13] used a machine learning algorithm to recommend both users and specific content in a system that they called TWITOBI. The approach is primarily based on the content of tweets, but some information about who follows whom in the network is utilized. A probabilistic model based on the expectation maximization algorithm is trained to properly rank both tweets and users via a similar technique.

The sample size in terms of number of users was fairly small, consisting of only 8,405 users, but the sample size in terms of number of tweets was very large, with more than 12 million tweets by those 8,405 users. No discussion of how those users were decided upon is included, which when combined with the small number of users makes some of the results a bit suspect.

2.2.4.4 Common Threads

Many of the methods for recommending users and content utilize similar ideas in making their recommendations. The literature indicates via multiple studies that retweets are a very important predictor of what a user is interested in and that retweets are also an excellent indicator of how much influence a user has over others. It is also clear from existing research that the PageRank algorithm, while useful to some degree, does not perform very well at predicting which users will be most influential on a given user.

Most recent research has not focused specifically on either the network-based approach or the content-based approach, preferring instead to take useful features from each. This dissertation takes the same tack, utilizing useful features from both the network and the content. In many cases, features used successfully in the research reviewed here were not used for various reasons, but would make excellent additions to the methodology and implementation described in chapters 3 and 4.

Chapter 3

Methodology

This chapter describes the methodology used in this dissertation to recommend both content and users to a particular user referred to here as the *distinguished user*. Items are recommended based on both the social network structure surrounding the distinguished user and on the actual content produced by the distinguished user and by others in the surrounding network.

The various aspects of the method are intentionally described as generally as possible and without reference to any particular social network. This generality is important because while the experiments described in Chapter 4 and Chapter 5 are specific to Twitter, the methodology described here could just as easily be applied to Facebook, Google+, or many other social networks.

3.1 Simple Approach

The nature of a social network—a group of users connected to one another by various relationships—suggests that creating a graph is the best way to analyse the network. There are many algorithms suitable for the task of analysing such a graph and much research has gone into applying these algorithms to Twitter and other social networks. Additionally, research has been done to analyse the nature of social graphs to determine things such as the average degree of nodes (i.e. users) in the network and the average distance between nodes. This research was discussed at length in the previous chapter.

The obvious approach to take is to build a graph where each user is a vertex and the relationships between those users are the edges, as in Figure 3.1. Thus, if user A follows user B then there will be a directed edge from vertex A to vertex B. When using the network-based approach for recommendation, this basic graph of the social network is often used, e.g. as in [10]. If it is desirable for the analysis to

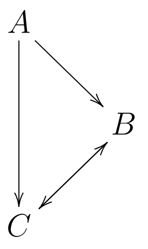


Figure 3.1: A simple graph of a social network containing three users. User A follows user C, user A and user C both follow user B, and user B follows only user C.

take the meaning of the content into account then this basic graph can be extended by creating edges between users who frequently discuss similar topics. This can be extended further by weighting the edges based on the strength of the relationship as determined by factors such as common activity as described in Section 2.1.2.2. [10], for example, augmented the basic social graph with additional links based on influence calculations.

Once a weighted graph such as this has been constructed a number of different algorithms can be employed including random walks, as in [4], the PageRank algorithm, as in [23], or simply searching for the strongest links to users who are not already connected to the distinguished user.

The obvious limitation to this simple approach is that it has no way of recommending particular pieces of content—the best it can hope to do is to select content from the top users using some similarity metric to compare that content to content that the user has already indicated interest in. Since the goal of this dissertation is to recommend both users and content this simple approach is clearly not sufficient.

3.2 Co-HITS

3.2.1 Background and Suitability

Deng et. al. [7] developed an algorithm that they dubbed Co-HITS to use for ranking web queries and documents. The name of this algorithm is in reference to the famous HITS algorithm [14] for ranking websites. Co-HITS is similar to HITS, and in fact with particular values of the personalized parameters (see Section 3.2.3) the algorithm reduces to the standard HITS algorithm. As with normal HITS, Co-HITS is an online

$$\begin{tikzcd}
A \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] & C_{A_1} \\
 & C_{A_2} \\
B \arrow[r, leftrightarrow] & C_B \\
C \arrow[r, leftrightarrow] & C_C
\end{tikzcd}$$

Figure 3.2: This simple bipartite version of the graph of a social network shows how a social network can be made to be bipartite by incorporating generated content into the graph and creating edges between users and the content generated by those users. This also demonstrates that this process alone is not enough to provide a useful graph since this graph is very disconnected.

algorithm, meaning that scores must be obtained for each unique query at query time rather than calculating the score once and then storing the results.

Two major differences between Co-HITS and HITS are the introduction of an initial score and that Co-HITS is designed to operate on a bipartite graph. The initial score factor is particularly important because it makes it possible to include content information in the algorithm by initializing scores based on content similarity. Co-HITS also does away with the concept of differing scores for hubs and authorities, instead keeping only one score.

The obvious objection to using this algorithm for recommending users and content is that the graph of a social network as in Figure 3.1 isn't bipartite at all. However, the two classes described by [7], search queries and web documents, can be adapted to the task of content and user recommendation by reformulating the social network graph not with users as vertices and connections between users as edges, but with both users and content as vertices and with edges between user vertices and the vertices representing content produced by that user rather than between users.

This new formulation of the social graph is clearly bipartite, as demonstrated in Figure 3.2, making the Co-HITS algorithm applicable to it. Unfortunately it is also clearly a very disconnected graph since users are only connected to the content that they produce, leaving no connections to other users or other content. This is addressed in section 3.3; for now it is enough that this graph is bipartite and thus that the Co-HITS algorithm can be applied to it.

3.2.2 Algorithm

This section contains a summary of the Co-HITS algorithm as described in [7] but applied to the case of ranking users and content in social networks rather than ranking web queries. Some of the variable names have been changed from that paper's notation to make it more clear what the variables refer to in the context of social networks.

Let the set of all content be called T and each member of that set be called t. Similarly, let the set of all users be called U and each member of that set be called u. Then let the probability of transitioning from a particular user u_i to a particular piece of content t_j be denoted as w_{ij}^{ut} and the probability of transitioning from content t_j to user u_i be denoted as w_{ji}^{tu}. The initial score (discussed in section 3.4) for a given user node and content node will be indicated as u_i^0 and t_k^0, respectively. In both cases the initial scores are normalised such that $\sum_{k \in T} t_k^0 = 1$ and $\sum_{i \in U} u_i^0 = 1$. Given these definitions, the generalized Co-HITS equations are given in [7] as follows:

$$u_i = (1 - \lambda_u)u_i^0 + \lambda_u \sum_{k \in T} w_{ki}^{tu} u_k$$

$$t_k = (1 - \lambda_t)t_k^0 + \lambda_t \sum_{j \in U} w_{jk}^{ut} x_j$$

The paper goes on to describe further refinements under the assumption that only one set of vertices is desired to be scored. However, the goal for this project is to score both sets of vertices, so that refinement, consisting of substituting one equation into the other, will not be discussed. Note that the personalised parameters λ_t and λ_u will be discussed in Section 3.2.3.

From these general Co-HITS equations, [7] describes two frameworks to arrive at final scores for each node in the graph, the iterative framework and the regularization framework. The approaches are similar to the multiple approaches to determining PageRank in that one involves iteration and propagation of scores while the other involves matrix operations and more intense computation.

As the name implies, the iterative framework involves iteratively propagating the scores between the nodes using the above equations until achieving convergence. The paper states that their empirical results assert that convergence usually occurs within approximately ten iterations. Experiments done for this project and discussed in

Chapter 4 indicate that this is accurate. The initial scores are normalised so that the scores for each set T and U sum to one. A convenient consequence of the above equations is that after each iteration of the algorithm the sums each of each set, $\sum\limits_{k \in T} t_k$ and $\sum\limits_{i \in U} u_i$, will still sum to one without need for an additional normalisation step.

The regularization framework, meanwhile, involves transforming the transition probabilities into a transition matrix and using that matrix along with some other derived equations to solve for the final results directly. The final step of this calculation involves a matrix inversion. As described in the paper, this operation can be done efficiently provided that the matrix is sparse and small. That works well for the test case described in [7], because their graph is indeed very sparse as well as being quite small (only 50,000 entries). This method is not well suited for a social network after application of the projection procedure described in section 3.3 because this graph is quite dense and extremely large since millions of pieces of content are produced each day, making the calculation computationally much more difficult and this framework less tractable than the iterative framework.

3.2.3 Parameters

The Co-HITS equations described above each include a so-called personalisation parameter $\lambda \in [0, 1]$, which determines how much weight the initial score has on the final outcome. The closer a given λ parameter is to 0, the more weight the initial score is given in calculating the score after each iteration.

Setting both λ parameters to 0 simply means that the final score will be equal to the initial score. Setting both λ parameters to 1, meanwhile, makes the algorithm into something much more akin to the standard HITS algorithm—the scores of each vertex are determined entirely by the scores of the nodes transitioning into that vertex. If only one of the λ values is set to 0 then the algorithm will converge after only one iteration once the scores of the corresponding set of the bipartite graph have been propagated across to the other set. Finally, [7] indicates that if only one of the λ values is set to 1 then the algorithm becomes something akin to the Personal Page Rank algorithm.

Conceptually it makes sense to set the λ parameters according to the confidence in the initial scores. If they are believed to be quite accurate for one or both sets of vertices in the bipartite graph then more weight should be placed on the initial score by moving the λ value closer to 0. If, on the other hand, very little is known about the accuracy or values of the initial scores for a particular set of vertices then the λ value should be closer to 1.

For the particular data set used in [7], λ values of 0.7 and 0.4 were found to have particularly good results. Chapter 4 of this dissertation has more information on the values that were found to have good results for the social networking dataset used here.

3.3 Projecting to a Bipartite Graph

One of the key insights of this project was how to transform the social graph of a social network, which is very clearly not a bipartite graph, into a bipartite graph suitable for use with the Co-HITS algorithm or other algorithms specific to bipartite graphs such as [15]. Section 3.2.1 has already described how a social network can be made into a simple bipartite graph by taking users and content as vertices and connecting users to the content which they produced.

Unfortunately, this basic process leaves a very disconnected graph and discards all of the network information about which users are connected to which other users, which is a very important piece of information. It also ignores the connections between pieces of content, such as whether they are forwards of a particular piece of content and whether they discuss similar topics to other pieces of content.

This information can be recovered by projecting the network connections between users so that they are represented by connections between user vertices and content vertices. If a user Alice is connected to a user named Bob and a user named Carol in the social graph (as in Figure 3.3a), then this projection would connect content C_A produced by Alice to both Bob and Carol in the output bipartite graph, Figure 3.3b. This process makes sense intuitively because it is clear that if Bob and Carol see this content—which they probably will, given their connection to Alice—then it is exerting some influence on them.

Besides the obvious and explicit connections between users in a normal social network there are also implicit connections between the individual pieces of content that those users create. For example, in the case of Twitter, two tweets which include the same hashtag (e.g. #Oxford) have an implicit connection, as in Figure 3.4a. The projection procedure can easily be extended to apply to content by connecting users who have produced content to all of the other pieces of content which are linked to their own. Thus if Bob mentioned #Oxford in a piece of content, he would

$$\begin{tikzcd}
Alice \arrow[r, leftrightarrow] & C_A \\
Bob \arrow[r, leftrightarrow] \arrow[u, dashed] & C_B \\
Carol \arrow[r, leftrightarrow] \arrow[uu, dashed, bend left] & C_C
\end{tikzcd}$$

(a) Before projection

$$\begin{tikzcd}
Alice \arrow[r, leftrightarrow] & C_A \\
Bob \arrow[r, leftrightarrow] \arrow[ur, leftrightarrow] & C_B \\
Carol \arrow[r, leftrightarrow] \arrow[uur, leftrightarrow] & C_C
\end{tikzcd}$$

(b) After projection

Figure 3.3: Demonstration of projecting network connections, indicated in (a) by the dashed lines, to make the graph bipartite and more connected.

$$\begin{tikzcd}
Alice \arrow[r, leftrightarrow] & C_A \\
Bob \arrow[r, leftrightarrow] & C_B \arrow[dd, leftrightarrow, dashed, "\#Oxford" description] \\
 & \\
Carol \arrow[r, leftrightarrow] & C_C
\end{tikzcd}$$

(a) Before projection

$$\begin{tikzcd}
Alice \arrow[r, leftrightarrow] & C_A \\
Bob \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] & C_B \\
Carol \arrow[r, leftrightarrow] \arrow[ur, leftrightarrow] & C_C
\end{tikzcd}$$

(b) After projection

Figure 3.4: Demonstration of projecting connections between related pieces of content, indicated in (a) by the dashed lines, to increase the connectivity of the bipartite graph.

be connected to all other content in the graph which also mentioned #Oxford, as indicated in Figure 3.4b.

This projection also makes sense because if Alice creates a piece of content then she is likely to be interested in other content which is related to it. This process of projecting the content links isn't limited to just hashtags. Any two pieces of content determined to be similar could have their implicit links projected in the same way. That could mean that they both mention similar named entities, that it can be determined that they discuss the same topic, that they link to the same external web address, or any number of other means of determining the similarity of two pieces of content.

A number of other projections can be made to add other features of the social network to the bipartite graph. For example, [22] reveals that in Twitter retweets are one of the most important indicators of what is interesting to a user. Retweets—or more generally, re-shares of the content of others—can be included in the bipartite graph in a number of ways such as by strengthening the weight of edges connecting the sharer to the content of the original author, by connecting the content produced by the original content author's connections to the sharer, or by connecting the content produced by the original content author to the connections of the sharer.

Projecting out the links between users and the links between content makes the bipartite graph much more connected and also models the real-world influence that particular network characteristics exert. Depending on the test being run, some or all of these edges may be directed rather than undirected and they may be weighted or not. Some analysis of those design decisions is presented in Chapter 4.

3.4 Method for Initializing Scores

A key factor in the accuracy of the final scores when using the Co-HITS algorithm is the initial scores provided for each set of vertices. How much of an effect the initial scores have is dependent on the λ parameters, but unless both are set to 1 the initial scores will still have a major impact on the final results. It is therefore quite important to choose a good method for determining those initial scores. In the absence of any known metric for determining the scores it is necessary to set the initial scores for a particular set to a uniform value, which indicates that all of them are equally likely to be relevant at the outset.

Fortunately, in social networks, it is possible to make some very accurate initial determinations about content and users which may be relevant.

3.4.1 User Scores

Much work has been done on the so-called link prediction problem which seeks to predict which links will be created between users in a social network, i.e. which other users are most relevant to a given user. [17], [15], and [4] are all examples of studies that have been done on this subject.

Because this process of determining original scores is only the initial step in the recommendations, for this project it makes sense to choose a method which is as fast as possible while still maintaining a high degree of accuracy. [17] studies a number of different methods of link-prediction, ranging from very fast and easy to calculate to very complex calculations. Fortunately, one of the most accurate methods is also one of the easiest to calculate, which is the method of Adamic and Adar from [1].

If $\Gamma(x)$ is defined as the set of all connections (neighbours) of vertex x, then the similarity score for two users x and y is given as:

$$\textstyle\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$$

The initial score for all of the user vertices in the bipartite graph can be computed by iterating through all of the users and comparing them to the distinguished user with this metric. This function handily penalizes the effect that more popular users have on the scores by virtue of appearing more frequently while weighting less popular users more highly.

Depending on which particular social network is being examined, it may make sense to use slightly different versions of the Γ function. For example, in Facebook it might make sense to consider only the neighbourhood of close connections, while in Twitter it might make sense to find the intersection of the followees of the distinguished user and the followers of the user being scored rather than the intersection of the followers of both.

Both the simple common neighbours function, given by:

$$|\Gamma(x) \cap \Gamma(y)|$$

and the Jacard Coefficient, given by:

$$\frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}$$

were found by [17] to present good results as well and could be used in cases where a more lightweight function is desired.

3.4.2 Content Scores

Much research has also been done on comparing the similarity of content, usually in the form of document similarity. Such algorithms can be a straightforward comparison of the number of words which overlap in the two pieces of content or they can be much more complex and use ontologies or other methods of deriving semantic meaning to compare the two pieces of content.

As with the initial user scores, it is desirable that the initial content scores be fast and easy to compute. A natural choice is the cosine similarity metric, which is both very common and very easy to compute. This metric requires that the pieces of content (referred to as documents in the remainder of this section) be represented as vectors in the standard term frequency–inverse document frequency (tf-idf) form.

In this form, the documents are represented as vectors where each dimension corresponds to a different term that appears somewhere in one of the documents in the collection. The value of each dimension is the tf-idf value. In its most common form, the tf-idf value for a given term is computed as:

$$tf(t) \times idf(t) = |t| \times \log(\frac{|D|}{df(t)})$$

where $tf(t)$ is the frequency of term t within the document in question (represented above by $|t|$), and $idf(t)$ is the inverse document frequency for that term, which is found by dividing the number of documents in the collection (represented above as $|D|$) by the number of documents in the collection which containing term t (represented as $df(t)$). The denominator of the idf function can lead to a division by zero if a term does not appear in the collection, so either that needs to be detected and the corresponding tf-idf value set to zero or each df(t) values should be incremented by one.

Once tf-idf vectors have been created for both of the documents $\hat{a}$ and $\hat{b}$ being scored, the cosine similarity metric itself can be computed as:

$$\frac{\hat{a} \cdot \hat{b}}{|\hat{a}||\hat{b}|} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i (a_i)^2}\sqrt{\sum_i (b_i)^2}}$$

This equation for the similarity between two vectors lies in the range $[-1, 1]$ in the general case. However for document vectors the individual terms can never be negative and so the range of the similarity lies in the range $[0, 1]$.

For actually computing the initial scores of the bipartite graph it is necessary only to iterate through all of the pieces of content and compare each one to a reference document using the cosine similarity metric. The only remaining question, then, is the choice of the reference document which best indicates which documents will be relevant. Based on [22], the best option would likely be the set of content which the distinguished user has re-shared, since that is the best indicator of the type of content that user would like to see. If that set is empty, then some secondary choices might be the set of all documents produced by connections of the distinguished user or the set of all documents produced by the distinguished user.

While the scores produced by this method are not as accurate as the methods for initialising user scores discussed in Section 3.4.1, the resulting scores are still quite reasonable and the scoring process is very fast. Plus, tf-idf scoring and cosine similarity are very widely used methods and are supported by a wide range of tools and libraries. The fact that the initial scores produced for the content are not quite as accurate as those for users can be dealt with by simply moving the value of the λ_t parameter closer to 1, dampening the impact of the initial score.

Chapter 4

Implementation

This chapter describes how the method of Chapter 3 was actually implemented for this project, a non-trivial step considering the major difficulties associated with the vast amount of data used. It begins by describing how the dataset was selected and the challenges that the large dataset presented in terms of storing and accessing the data. From there it moves on to describe how the bipartite graph was constructed, including selection of tweets and users, construction of edges, and initialization of scores. From there, the actual implementation of the algorithm is described both in words and in pseudocode before finally describing the method of retrieving the results from the database.

Java was chosen as the programming language for this project due to the wide adoption of libraries available to interface with it and the ease of development with those libraries. The Lucene information retrieval library[1], for example, is originally written in Java and was used to index the tweets in a way that made tf-idf cosine-similarity calculations easy. It also facilitated easy search of the tweet database which was useful during the development process to learn more about the nature of the dataset. In addition to Lucene, the Java Database Connectivity library (JDBC) was used to interact with the PostgreSQL database and the Stanford named-entity recognizer was used for the named entity recognition.

4.1 Selecting a Dataset

When beginning the project, one of the first questions was which dataset to use. There were four possibilities: downloading data via the Twitter API specifically for this project, the Choudhury data ([6]), the Stanford data ([24]) using the social graph

[1]http://lucene.apache.org/core/

of Kwak et. al. ([16]), and the database created for the Text Retrieval Conference (TREC). Each had advantages and disadvantages.

Given the availability of existing datasets that would meet the needs of the project, the possibility of downloading a social graph and set of tweets specifically for this project was dismissed early on, leaving the task as one of selecting which of the three datasets was best.

The smallest dataset was the Choudhury data. It consisted of tweets collected between late 2008 and late 2009 and contained more than 10 million tweets and more than 800,000 edges. This database is much smaller than the network was at the time it was created; it contains a tiny fraction of the tweets which were produced during that time and a much smaller fraction of the number of users. It was created by starting a seed set of users and then building the database out from there, but does not include the complete set of followers or followees for any user, though it is a connected graph. For example, a number of users have a list of some followers and then are shown to follow no one. This lack of completeness made this dataset unacceptable for this project.

The second dataset was the TREC dataset[2], consisting of 16 million tweets collected over a two week period in late January and early February of 2011. It is not intended to be a complete list of the tweets from that time period, merely a representative sample which includes spam tweets amongst the important tweets. The major problem with this dataset, however, is that it did not include any social graph information, just the tweets. Since a major portion of this project was based on the impact of the social graph on the recommendations, this clearly was not possible to use.

The Stanford Twitter dataset contains 476 million tweets from more than 17 million users. These tweets were collected over a period of 7 months between 1 June, 2009 and 31 December, 2009. The authors of [24] estimate that these tweets covered approximately 20-30% of the tweets published in that time period[3]. By itself, this dataset would be unusable for the same reason as the TREC database since it does not include any social graph information. Fortunately the authors link to the published social graph of [16], which was crawled via the Twitter API contemporaneously with the collection of the Stanford tweet data. Though the vastness of this dataset presented many challenges during the project, it also was the best choice for putting together a recommendation system to be as accurate as possible.

[2]http://trec.nist.gov/data/tweets/

[3]http://snap.stanford.edu/data/twitter7.html

Of these three datasets, only the TREC tweet data and the social graph of [16] are currently available to the public. Twitter changed their terms of service, requiring care to be taken with the data and researchers to remove tweets from their databases if they were deleted by their original authors. The TREC data is accompanied by a tool which queries the Twitter service to accomplish this, which is why it remains, though even it requires that the United States National Institute of Standards and Technology issue a password to allow the data to be downloaded.

The tweets from the Stanford dataset were able to be used here because one of the researchers within the Oxford University Department of Computer Science had downloaded it before this change came into effect.

4.2 Data Storage

Considering the vast amount of data available in the dataset, an efficient data storage mechanism was vital. The major consideration was the ease and efficiency of retrieving the data. PostgreSQL was selected to store the information because it has excellent support and is a mature database management system capable of being used with large amounts of data. It has a good interface with the Java programming language and is also open source, making it possible to tweak the implementations of certain features if necessary, though that was not done here.

The biggest source of data to be put into the PostgreSQL database was the social graph. The tweets took up far more disk space, but not all of them needed to be stored in the database because most of them were not used, as described in Section 4.3. The complete social graph, meanwhile, was approximately 24 GB, with each line containing a pair of user ids separated by a tab delimiter. Unfortunately the tweets data from Stanford had each tweet indicated only by the user's name, meaning that the data from the social graph which had only user id pairs had to be translated before being stored.

The final `namenetwork` table, whose schema is shown in Table 4.1, used the actual usernames as looked up from another database that accompanied the social graph data. Indices were created around both the username and followername columns, with the username column being clustered since it is used the most frequently and has the most data returned from it since the most popular users have far more followers than followees.

Still, it was helpful for certain queries to have fast access to the followees of a given user, so a second copy of the namenetwork table was created to make these queries

Field Name	Field Type	Description
username	VARCHAR	The name of a user
followername	VARCHAR	The name of a user who follows the user in the username column for this row

Table 4.1: Schema of the 'namenetwork' table

Field Name	Field Type	Description
username	VARCHAR	The name of a user
count	INT	The number of followers of the user in the username column for this row

Table 4.2: Schema of the 'followercount' table

faster—this version was called `namenetwork_followers` and had the same schema as the namenetwork table but was clustered around the followername field rather than the username field.

The usernames were all converted to lower case, and indeed all usernames which could potentially be parsed in mixed case found while indexing the tweets were converted to lower case. Additionally, the social graph data contained a number of duplicate user names. This happened because the program which aggregated the data apparently ran over an extended period of time over which some user accounts were deleted and re-created. Additionally, a large number of user ids corresponded to the name 'n/a'. All of these duplicate names were removed from the database.

For the initialization of the user scores (cf. Section 4.4) using the method of Adamic and Adair (cf. Section 3.4.1) it was necessary to know how many followers each user had. This could be calculated on the fly by running a simple SQL `COUNT()` query on the namenetwork table, but this would sacrifice performance, so it made more sense to do this query once and create a table for it. The `followercount` table, whose schema is shown in Table 4.2, was created using the following SQL command:

```
CREATE TABLE followercount AS
    (SELECT username, COUNT(followername) AS count
        FROM namenetwork GROUP BY username);
```

A hash index was created on the username column of the followercount table to allow for quickly looking up the number of followers that a user has.

The tweets themselves were not all used in any given run of the algorithm, and in fact only a tiny portion of them were used. One guideline for the project was that the recommendations themselves be only for a particular time period on the order of one to two weeks in order to reflect fleeting interests and recommend things that are of interest to the user at that time. For example, during the Olympics someone may be very interested in tweets about the Olympics, but they would probably not be so interested a few weeks after the Olympics have ended. For this project 500,000 tweets were indexed when the algorithm was initialized, corresponding to about 15 hours of the data stretching between 21:03 on 30 November, 2009 to 11:15 on 1 December, 2009. Considering that the full dataset contains nearly half a billion tweets, this is a very small portion.

It was useful to store this small portion of tweets in two different ways for two different purposes. For purposes of initializing the tweet scores (again, cf. Section 4.4) it was very useful to index the tweets in a Lucene index because such indices automatically count the frequency of each term in each document and in the document collection as a whole, making tf-idf cosine similarity scoring easy.

At the same time, it was also valuable to be able to write SQL statements to quickly select particular tweets and to update their Co-HITS scores quickly, a task much better accomplished via a relational database such as PostgreSQL. This storage of the tweets represented the tweet vertices in the graph, and thus information was included about the Co-HITS score and the original Co-HITS score for each of these nodes. The schema for the `tweet_vertices` table is shown in Table 4.3. This table was indexed on the tweetid field using a btree index. After the initial index was built, an operation which only needed to happen one time for the series of tweets under investigation, the index was clustered. During the creation of the edges this table was clustered on the names, since that was how most of the edges where created—by joining user vertices with their tweets. Once this was completed, the tweets were clustered by their tweetids for actually running the algorithm.

The tweets were stored in the database, so it obviously makes sense that the user vertices would be stored in the database as well. The schema for the `user_vertices` table is shown in Table 4.4. Similar information needed to be tracked with these vertices as with the tweet vertices, most notably the current Co-HITS score and the original Co-HITS score. The index on this table was a btree index on the name field, and as with the tweet vertices this index was clustered after it was populated since this only needed to happen one time.

Storage in a PostgreSQL database facilitated some easy implementations of certain operations. For example, by storing the original score in the vertex it was possible to

Field Name	**Field Type**	**Description**
tweetid	INT	The unique id of the tweet that this vertex represents
name	VARCHAR	The author of the tweet that this vertex represents
date	TIMESTAMP	The timestamp of when the tweet that this vertex represents was published
tweet	VARCHAR	The text of the tweet that this vertex represents
score	FLOAT	The Co-HITS score for this tweet
original_score	FLOAT	The original Co-HITS score for this tweet before the algorithm's first iteration.

Table 4.3: Schema of the 'tweet_vertices' table

Field Name	**Field Type**	**Description**
name	VARCHAR	The name of the user that this vertex represents
score	FLOAT	The Co-HITS score for this user
original_score	FLOAT	The original Co-HITS score for this user before the algorithm's first iteration.

Table 4.4: Schema of the 'user_vertices' table

quickly determine whether the original scores had been initialized and normalized by simply checking whether the sum of the original_score column was equal to one:

```
SELECT SUM(original_score) FROM user_vertices;
```

Similarly, the scores could be normalized with only one SQL command:

```
UPDATE tweet_vertices
    SET original_score=(original_score / S.sum)
    FROM (SELECT SUM(original_score) FROM tweet_vertices) S;
```

And if the algorithm needed to be reset so that it could be run again, it was a simple matter of copying over the original_score field to the score field. In the actual implementation, three additional columns were present in the two vertex tables, one for each of the distinguished users. These columns stored the original scores for each of the users and made it possible to calculate this value only once for each user, facilitating faster testing.

The final component that was stored in the PostgreSQL database was the list of edges between the two different classes of vertices. The schema of the `edges` table is

Field Name	**Field Type**	**Description**
name	VARCHAR	The name of the user that this edge connects to.
tweetid	INTEGER	The unique identifier of the tweet that this edge connects to.
type	INTEGER	The type of edge that this represents (see Table 4.6).

Table 4.5: Schema of the 'edges' table

shown in Table 4.5 and is quite basic. The name of the user and the id of the tweet that are connected is shown, along with the type of the edge. The directionality (if any) of that edge is indicated purely by its type. Experimentation with making edges of particular types directed in particular ways is addressed in Section 5.2.5.

As with the `namenetwork` table, different clusterings were useful for different purposes. For purposes of running the actual algorithm, it was necessary to sometimes order the edges by username and sometimes by tweetid. The edges table was clustered around the username column, so when retrieving the edges in order of tweetid, the query took far more time: two orders of magnitude more, in fact. Thus a copy of the edges table called `edges_tweetids` was created for this purpose, this one clustered around the tweetids, allowing both queries to be much faster.

4.3 Building the Graph

Building the graph is a very important part of the process. This was taken care of in the GraphManager class (cf. Appendix A), and in particular in the `createGraph()` method and the other methods called from there. First the users and tweets are selected and indexed and then all the appropriate edges between the two sides of the bipartite graph are created.

4.3.1 Selecting Users and Tweets

How to build the graph itself is not as simple a question as it would seem to be at first. Which users should be present in the graph? Which tweets should be present? How the graph is created and the decisions about which nodes are present dictates how connected the graph will be and thus how the scores from each set of vertices will interact with one another. Having users who are not connected to the rest of graph serves no purpose because no information about their suitability can be gained.

Tweets were added by opening one of the files containing tweet data and parsing until the appropriate number of tweets had been indexed. Certain tweets in the data were invalid and were thrown out. The tweet ids were created by beginning at 1 and then incrementing for each tweet that was added to the database (i.e. each valid tweet). Proceeding in this manner ensured that the tweets were all from the same time period, all were valid, and all had a unique identifier.

In order to ensure that the graph was maximally connected, only users who would have connections were added. Practically speaking, this meant that users were added as the tweets were placed into the tweet_vertices database. All users who had authored a tweet were added, but so were all users who had been retweeted or mentioned. Because of the types of edges created, as described in Section 4.3.2, these users are guaranteed to at least be connected to one tweet and will likely be connected to several others.

4.3.2 Edge Creation

Once the users and tweets have been added to the database, the next step is to add the edges between the users and the tweets. There were twelve different types of edges used in this project, though the list is by no means exhaustive of the possibilities. The different edge types are listed in Table 4.6.

The authorship edges are simple and are actually created when the users and tweets are initially indexed. For each tweet added to the database its author is added if that user is not already present and then an edge is added between the tweet and the author. These edges are simple.

The remainder of the edge types are more complicated and fall broadly into two categories. The first category is the follower, retweet, mention, and @reply edges, which correspond to the network edges connecting users to other users and the edges connecting users to particular tweets. These edges all have an obvious directionality implied by the type of edge. The second category of edge is the content edges which correspond to the edges between tweets, and these edges don't have any obvious directionality.

4.3.2.1 Network Edges

The follower edges are the most straightforward to describe and are simply the result of following the procedure for projection of user connections described in Section 3.3.

The retweet and mention edges are more complicated but are very analogous to one another. The simplest of these edges is the simple retweet or mention edge which

Edge Type	**Description**	**Direction**
Authorship	Connects a user to a tweet authored by that user	User → Tweet
Follower	Connects a user to a tweet authored by one of their followees, as described in Section 3.3	Tweet → User
Retweet	Connects the tweet to the person being retweeted	User → Tweet
Retweet Followees	Connects the tweets of the followees of the person being retweeted to the retweeter	Tweet → User
Retweet Followers	Connects the tweets of the person being retweeted to the followers of the retweeter	Tweet → User
Mention	Connects the tweet to the person being mentioned	User → Tweet
Mention Followees	Connects the tweets of the followees of the person being mentioned to the mentioner	Tweet → User
Mention Followers	Connects the tweets of the person being mentioned to the followers of the mentioner	Tweet → User
At Reply	Connects the tweet to the person being @replied to	User → Tweet
At Reply Content	Connects the tweets of the person being @replied to to the tweeter	Tweet → User
Hashtag	Connects all tweets which use a given hashtag to the authors of those tweets	Bi-Directional
Content	Connects all tweets discussing similar content (cf. Section 4.3.3) to the authors of those tweets.	Bi-Directional

Table 4.6: The types of edges in the graph

connects the tweet to the person being mentioned or retweeted; as with the authorship edges, these are created at the time that the tweets are initially indexed. These edges capture the intuition that a high scoring tweet which mentions or retweets someone suggests that the person in question should also be scored highly and that a person is likely to be interested in a tweet that mentions them or comments on something they say. The followees and followers versions of the retweet and mention edges, meanwhile, are more complicated.

The followees edge creates a connection between the person who is retweeting or mentioning someone and the followees of the person being mentioned or retweeted. This is because that user has proven by their retweet or mention that they are very interested in content from the user who authored the original tweet, so it stands to reason that they are correspondingly more likely to be interested in the same things as that person.

The followers edge connects the tweets of the person who was retweeted or mentioned to all of the followers of the person who retweeted or mentioned them. If a user is interested enough in someone to follow them and see their tweets then it stands to reason that they would also be interested in someone that their followee is very interested in.

The @reply edges are quite straightforward, though they probably provide less impact on the final rankings because they generally only reinforce the existing follower and followee connections. These edges connect the tweet to the person the @reply was directed at and connect the tweets of the person the @reply was directed towards to the person who sent it.

4.3.2.2 Content Edges

There are far fewer content edges, but they convey a great deal of information. These edges are based on the system of projecting edges between tweets into the bipartite graph by connecting the edges to users instead using the procedure detailed in Section 3.3.

Hashtag edges represent the connections between tweets using the same hashtag. These edges connect all the tweets using a particular hashtag to all of the authors of tweets containing those hashtags. Figure 3.4 demonstrates this process. The hashtags are converted to lowercase for normalization purposes and because Twitter itself treats hashtags as being case insensitive.

The content edges connect tweets in the same way, but using the implicit links between tweets with similar content as the basis for the projection. How the similarity in content is determined can vary between implementations and range from something complicated such as Latent Dirichlet Allocation topic models to something much simpler.

4.3.3 Determining Content Similarity

Determining which tweets contain similar content to one another is an important part of the algorithm used here because it provides many connections between users and tweets that are not connected via straightforward network connections, allowing novel tweets to be recommended by the algorithm.

The method for determining this similarity need not be very complicated, though it certainly can be. Using topic models and other techniques from statistics and machine learning would certainly generate good results for determining content similarity, but at the cost of being quite computationally expensive.

For this project a much simpler and faster method was used: named-entity recognition (NER). Using the Stanford Named-Entity Recognizer [9], each tweet was run through the tool in order to find any named entities such as people, places, and organizations. This was done as the tweets were being processed for other content information such as hashtags.

Each named entity found was converted to lower case and stored in memory along with the id of the tweet in which it appeared. If another tweet was found containing the same named entity then that tweet's id was stored along with the ids of the other tweets already found with that id. Once all of the tweets had been processed, each list of tweetids was connected to the authors of the respective tweets.

This process is far from perfect, but provides a good approximation of the people, places, and things that users are talking about. In experiments done prior to incorporating this tool into the algorithm, the NER of the Stanford library was found to be generally accurate, despite the lack of context provided by the space constraints of tweets.

The biggest problem with the use of the recognizer in this algorithm is that it returns compound entities as separate entities, meaning that entities such as 'United Kingdom' would be returned as two separate words. Similarly, names when given as first and last names (e.g. 'David Beckham') would be returned in two chunks. This presents a question as to whether these words should be treated individually or combined into one entity.

Leaving them as separate words gives some poor results such as 'David' or 'United' being counted as entities even though they are not very useful, while combining them separates cases where people are referred to by last name, meaning that 'Beckham' and 'David Beckham' would be two different entities even though they refer to the same thing. Similarly, phrases such as 'David Beckham, United Kingdom' would be returned as one big entity, which is clearly nonsensical.

For the purposes of this project, the former approach of leaving all entities as separate was taken. This leads to a small number of garbage edges but with the benefit of more correct connections and a simpler implementation. Put another way, this choice increases the recall of the connections between content while decreasing the precision.

4.3.4 Graph Statistics

The average degree of a user node and a tweet node can be taken by dividing the number of edges by the number of users and the number of tweets, respectively. For the set of tweet data used in this project, the first 500,000 tweets of the December data from the Stanford tweet dataset. When building the edges as described here, there were 121,788,048 edges, 500,000 tweet vertices, and 400,293 user vertices, meaning that the average user was connected to 304.2 tweets and the average tweet was connected to 243.5 users.

There are quite a few duplicate edges created using the above process because checking for duplicates at the time of graph creation slows that down dramatically; it is far easier to either ignore them when running the algorithm or to simply remove them from the graph after it has been created. Of the more than 120 million edges created, only 110,585,315 distinct edges were created. Experiments were performed to determine whether to ignore duplicate edges or whether to leave them in place to act as a weighting factor on the output—those results are discussed in Chapter 5. Additionally, in this graph a very large portion of the edges are content edges: 80,535,848. These were split more or less equally between edges based in hashtags and edges based on named entities: 42,608,235 of the edges were hashtag edges. The speed of the algorithm can be increased dramatically by eliminating these edges, and the results of that on the quality of the recommendations are discussed in Chapter 5.

4.4 Initializing Scores

Broadly speaking, the initialization of scores was implemented as described in Section 3.4. The user scores were initialized according to the method of Adamic and Adar ([1]) as described in [17] and the tweet scores were initialized via cosine similarity.

4.4.1 Tweet Scores

Calculating the initial tweet scores was mostly trivial. For purposes of finding a reference document which indicated the interests of the distinguished user against which all tweets could be compared, several possibilities were used. The first thing checked was whether a particular user had retweeted anything during the time period under study since retweets are the best predictor of what a user is interested in. If they had retweeted any content then those retweets were combined into one document with the 'RT' removed and used as the reference document. Unfortunately, given that the data presented was a small subset of the overall data, most users had no retweets and those who did have any usually only had one, which decreases the accuracy of this technique by focusing on one particular interesting tweet.

For those users with no recorded retweets in the timeframe under study, the composite document can be formed using the tweets of all of their followees. These can conveniently be recovered based on the edges by selecting all follower edges which connect to the distinguished user.

The initialization of the tweet scores uses the previously discussed Lucene library to aid with content similarity. Lucene stores term frequency information within its index, simplifying the calculation of tf-idf information for use with cosine similarity. All of the tweet vertices are selected from the database and then iterated through with a tf-idf document vector created for each tweet and then compared to the reference document using cosine similarity. Once all of the scores have been added to the database, they can be normalized to add to one by using the SQL statement shown in Section 4.2.

The canonical method of tf-idf calculation had an interesting side effect when used with the very short tweet documents. The brevity of the format means that many short words are used, and the social nature of the network also means that many personal pronouns are used. The canonical method of calculating cosine similarity using tf-idf did not adequately discount these highly common words, resulting in very highly scoring tweets which contained things such as "I love you so much", or which included multiple pronouns, like "I want you I need you I love you".

These tweets clearly had no value on their own, so the decision was made to provide an additional factor to increase the amount by which these words were discounted. To do this, the idf score provided by the canonical tf-idf measurement was modified by multiplying it by an additional factor:

$$tf(t) \times idf(t) \times discount(t) = |t| \times \log(\frac{|D|}{df(t)}) \times \frac{1}{\log(df(t)+1)}$$

The value of this discount factor would range from 0.175 for a term that occurred in all 500,000 tweets to 3.32 for a term that occurred in only one tweet. This serves to increase the value of rare words while doing a better job of discounting extremely common words and dramatically improved the quality of the initial tweet scores. They still were not overly interesting by themselves, but certainly provided a better starting point than with the canonical tf-idf measurement.

4.4.2 User Scores

Calculating the initial user scores was one of the more time-intensive parts of the process and led to the creation of extra columns in the user_vertices table to allow all of the users being investigated to have their initial scores stored so that they did not need to be calculated each time a different user was used for experimentation. Clustering the namenetwork table and creating the followercount table were both motivated largely by speed gains while calculating the initial user score.

Recall the scoring method of Adamic and Adar from Section 3.4. If $\Gamma(x)$ is defined as the set of all connections (neighbours) of vertex x, then the similarity score for two users x and y is given as:

$$\textstyle\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$$

For initializing the user scores, user x is always the distinguished user and user y is the user currently being scored. In the experiments of both [1] and [17] utilizing this formula the networks under study were non-directed, so the simple metric for the Γ function of neighbouring vertices was adequate. In the Twitter network, however, connections are directed.

Clearly it does not make sense to look for the intersection of the followers of two users to determine their similarity, since users have no control over this. A better method would be to look for the intersection of the followees of two users since it would at least indicate that they were interested in seeing the same things. Still, it would not indicate whether one user actually created content of interest to the other user.

For this project, the function used was the intersection of the followees of the distinguished user with the followers of the user being scored. If users that the distinguished user has interest in express interest in another user, then it stands to reason that that user will be more likely to be of interest to the distinguished user.

For each user z in this intersection, the score component from that user is calculated according to $\frac{1}{\log|\Gamma(z)|}$, and this is summed for all such users z. In this formula, $|\Gamma(z)|$ represents the size of the neighbourhood $\Gamma(z)$, i.e. the number of followers that the user z has, as found in the followercount table. This formula discounts the scoring effect of very popular users because they will naturally have more overlap in their follower lists with the followees of the distinguished user.

This process is accomplished in the project by running a SQL query to retrieve the follower counts of all of these users and then summing them in the code. That SQL query is:

```
SELECT C.count FROM
  ( (SELECT username AS name FROM namenetwork_followers
            WHERE followername=?)
      INTERSECT
    (SELECT followername AS name FROM namenetwork
            WHERE username=?)
  ) S, followercount C
  WHERE S.name=C.username;
```

where followername is parametrized with the username of the distinguished user and username is parametrized with the username of the user being scored.

The one user for whom this method does not work is the distinguished user—that user's score would not be properly returned using this method. Instead, the distinguished user has their initial score set to some factor of the largest score found before normalization. For this implementation this factor was 1.5, meaning that the distinguished user's initial score before normalization would be set to 1.5 times the highest score from any other user. This is designed to provide a strong score impact from the things that the distinguished user is connected to since that provides a great deal of information about content and users they are likely to be interested in.

4.5 Implementation of the Co-HITS Algorithm

The iterative version of the Co-HITS algorithm itself can be implemented to run in linear time and space once the graph and its associated indexes have been created. The process of indexing these various fields during the creation of the graph is $\mathcal{O}(n \cdot \log(n))$, but this only needs to be performed one time for the graph and once it is complete any number of users can have their recommendations built off of that graph with only linear time complexity. Furthermore, it is easy to update the graph incrementally as new users or tweets are added.

More specifically, the complexity of the algorithm is $\mathcal{O}(E \cdot I)$, where E is the number of edge vertices and I is the number of iterations. For each iteration the edges are sorted first by the user name, when calculating the tweet scores, and then by the tweet id, when calculating the user scores. This sort is performed when the tweets are indexed, however, not when running the algorithm. When updating the tweet scores, the following SQL statement is used to select the edges:

```
SELECT U.score as user_score, U.name, E.type, T.tweetid
  FROM user_vertices U, edges E, tweet_vertices T
  WHERE T.tweetid=E.tweetid AND E.name=U.name
  GROUP BY U.name, E.type, T.tweetid;
```

For updating the user scores, the following SQL statement is used to select the edges:

```
SELECT U.name, E.type, T.score as tweet_score, T.tweetid
  FROM user_vertices U, edges E, tweet_vertices T
  WHERE U.name=E.name AND E.tweetid=T.tweetid
  GROUP BY T.tweetid, E.type, U.name;
```

Note that in both cases the scores of the opposite vertex type from that being updated must be selected so that the update can be performed appropriately. For example, in the case of updating the tweet scores, the user score for each vertex must be known so that the score contribution from that vertex over the edges emanating from it can be calculated.

Each vertex is considered in order and its score contribution to the various nodes on the other side is considered. The sorting makes it easy to determine the total number and type of edges emanating from a particular vertex because all of the edges from that vertex are considered one after another. Knowing the total number of edges leaving a particular vertex is necessary when calculating the transition probability.

Upon iterating through all edges sorted by one component (either users or tweets), the scores for the other half of the graph can be updated with the newly calculated values.

Pseudocode of the implementation for one direction, updating the tweet scores, is shown for one iteration in Algorithm 1, while the actual implementation is shown in the code listing in Appendix A in the `updateTweetScores()` and `updateUserScores()` methods of the GraphManager class.

Algorithm 1: The Co-HITS Implementation for one iteration of updating the tweet scores. Updating user scores is identical, but with changes to the appropriate variable names.

```
tweetScores = $\emptyset$;
curUser ← first_edge.username;
curUserEdges = $\emptyset$;
while database cursor for edges not at end do
    edge ← edge at current database cursor position;
    if curUser $\neq$ edge.username then
        for edges e $\in$ curUserEdges do
            tweetProbability ← 1 / (size of curUserEdges);
            scoreEffectFromThisEdge ← tweetProbability * curUser.score * $\lambda_t$;
            if tweetScores contains key edge.tweetid then
                curScore ← tweetScores.get value for key edge.tweetid;
                curScore ← curScore + scoreEffectFromThisEdge;
                tweetScores.update(edge.tweetid , curScore);
            else
                tweetScores.put(edge.tweetid , scoreEffectFromThisEdge);
            end
        end
        clear curUserEdges;
    end
    add edge to curUserEdges
    curUser = edge.username
    move database cursor to next edge;
end
update the tweet scores in the database according to tweetScores
```

It is clear from examining Algorithm 1 that each edge is actually traversed twice, and since this algorithm is repeated twice for each iteration, the edges are actually traversed 4 times for each iteration. This is an unfortunate consequence of the need to know the number of edges leaving a particular vertex before calculating that vertex's effect on the vertices to which it connects.

When updating the scores, two steps are used. First, the original score is set according to the original_score column, the λ_t parameter, and the total score from vertices which did not have an edge in the current direction because of the directionality of the edges. These dead-end vertices still need to be accounted for, so their scores are distributed evenly between all vertices on the other side of the graph. So, before the scores are set for specific vertices, the original score for all vertices can be updated with the following query:

```
UPDATE tweet_vertices SET score=(? + (original_score * ?));
```

Here, the first question mark indicates the sum of the scores from all dead-end vertices, and the second represents the λ_t parameter. After the baseline score is added uniformly (and very quickly), the base score can be augmented with the calculated score factor for only those vertices which have been changed. This is performed as a batch operation using the following SQL query:

```
UPDATE tweet_vertices SET score=? WHERE tweetid=?;
```

Only the version for updating the tweet scores is shown, but the user score update proceeds in a nearly identical manner.

The Co-HITS algorithm is intended to be run until convergence, i.e. when the score was not updated for either the tweets or the users during a particular iteration. This would require knowing when the update statement did not result in any content updates, which could be accomplished by introducing an additional constraint on the WHERE clause in the above update statement.

In their paper introducing the Co-HITS algorithm, [7], Deng, et. al. say that the iterative algorithm generally converges in under 10 iterations. This was found to be generally true in this project, depending on how it was defined.

Using a WHERE statement to detect convergence never resulted in absolute convergence in the sense that no additional vertices were updated, and relaxing the requirement of how close the value had to be in order to determine convergence simply resulted in the normalization of the scores being knocked out of balance and no longer summing to 1 properly. Still, after ten iterations, and usually much fewer, it was clear by examining the values as they changed that the changes that were still occurring were very small, not actually affecting the final outcome or ordering of the tweets or users. For this project, the algorithm was allowed to run for up to ten iterations before terminating.

4.6 Retrieving Results

Retrieving the results of the algorithm is made easy by the structure of the tables within PostgreSQL; it is simply a matter of running a SQL command. For this project it was thought that users could find and become aware of users with a lot of followers very easily. By extension it was also assumed that users could easily find content by these users.

Thus, when retrieving the recommendations, only those whose follower count fell below a certain threshold were considered. Similarly it made no sense to recommend content that the user had already seen, users who the distinguished user already followed were ignored, as were their tweets. It may be reasonable to include tweets from existing followees in the recommendations for some applications (and Twitter's own new recommendation system does this) but for evaluation purposes within this project they were ignored.

The SQL statements for selecting the highest scoring tweets is:

```
SELECT T.name, T.score, T.tweet FROM
  tweet_vertices U,
  (
     (SELECT T.name FROM tweet_vertices T, followercount F
         WHERE T.name=F.username AND
                     F.count < 100000 ORDER BY T.score DESC)
   EXCEPT
     (SELECT username AS name FROM namenetwork_followers
         WHERE followername=?)
  ) S
  WHERE T.name = S.name ORDER BY T.score DESC;
```

The SQL statement for selecting the highest scoring users is:

```
SELECT U.name, U.score FROM
  user_vertices U,
  (
     (SELECT U.name FROM user_vertices U, followercount F
         WHERE U.name=F.username AND
                     F.count < 100000 ORDER BY U.score DESC)
   EXCEPT
```

```
    (SELECT username AS name FROM namenetwork_followers
        WHERE followername=?)
) S
WHERE U.name = S.name ORDER BY U.score DESC;
```

These two SQL statements both take 100,000 followers as the threshold of followers below which the content is deemed useful for recommendation purposes, but this can easily be adjusted.

By default this does not filter out two classes of tweets that a user is unlikely to be interested in: his own and those containing @replies to other users. Similarly, for the sake of brevity, the SQL statement for retrieving users does not filter out the distinguished user, though it would be simple to do so. In both cases, these tweets are filtered out before the recommendations are delivered to the user.

4.7 Parallelization and Performance Improvement

Though the speed of the algorithm has not yet been described except in broad theoretical terms, it is quite slow in practice. This is not due to the inherent speed of the algorithm, but rather it owes to the fact that all of the data being used is stored in a series of databases on an external storage device and accessing that data is very slow and must be performed in serial.

The creation of the graph, for example, takes more than 24 hours due to the large number of lookups and insertions that must be done in order to connect more than 120 million edges. To illustrate this point, consider the follower edge type. For each tweet the followers of the author need to be looked up in order to determine the follower edge tweets, and then each of these edges must be inserted into the edges table in the database. The result is a slow process. Running the algorithm itself is much faster, but still on the order of hours rather than minutes or seconds. As with the speed of creating the graph, the slowness in running the algorithm is caused by the slowness of retrieving so much data from the hard disk.

With many of the steps of the algorithm it would be very reasonable to the step in parallel using an algorithm such as MapReduce and thus to dramatically increase the speed of the entire process. If many different external storage devices are used then the speed of initializing the user scores can be increased by running the initialization completely in parallel. The maximum limit of this process would be to have one external storage device for each user in the graph, allowing the initialization to be done in seconds.

The creation of the edges in the graph can also be run in parallel. In the existing algorithm, the creation of edges is done by iterating through each tweet and creating appropriate edges based on the type of tweet (mention, retweet, @reply), which entities and hashtags it contains, and who follows the author. It would be simple to duplicate the database information on multiple external storage devices and to determine the edges for multiple tweets in parallel.

And of course, the algorithm itself could also be run in parallel. Recall that the edges are ordered when they are retrieved for updating the scores for each side of the bipartite graph. The score contributions onto the other side are considered for each vertex, a process made easier by the fact that the edges are are ordered by the vertex being considered. The edges could be split so that these vertices were considered in parallel by being selected from separate external storage devices.

Similarly, for a long-running system it would be possible to build much of the graph in real time as tweets came in, thus amortizing the costs of building it. Similarly, it would be possible to keep a matrix that had the similarity scores between all users which could be used when initializing the user scores, one of the longer running steps. Building such a matrix initially would be very time consuming, obviously, but once it was built then it could be updated incrementally at a much smaller time cost.

Unfortunately, this parallelization was not possible for this project due to the large amount of hardware required and the time required to implement it. As such, each run of the algorithm was quite slow, limiting the number of experiments that it was possible to perform. Still, as will be seen in the next chapter, good results were obtained and quite a bit of experimental variations on those results were obtained to gain good insight into which parts of this implementation were good and which need to be tweaked in future work in this field.

Chapter 5

Results

Having detailed both the method used for recommendation and the actual implementation of that method, the discussion now moves to the method for evaluating the results and the presentation of the results themselves in order to demonstrate the efficacy of the method.

First the method of evaluating the results is described; for such a large dataset this is a challenging question, and it is not yet well-defined for this research area. The selection of the distinguished users is then described and their effect on the evaluation methods is considered before arriving at several metrics for evaluating the results.

Next the baseline results are shown using the established metrics. Finally, variations on the algorithm are experimented with and their results presented in order to establish which parameters and edge types are the most valuable for this task and whether the algorithm can be improved over its baseline result.

5.1 Evaluation Methodology

5.1.1 Challenges

A review of the existing research on content recommendation discussed in Section 2.2.4, reveals no consensus on how best to evaluate the results of a recommendation method. This is due primarily to two separate but related issues: the lack of a common dataset and the lack of ground truth judgements on the relevance of either content or users.

None of the research projects on Twitter content recommendation reviewed in Section 2.2.4 used the same set of Twitter data for their experiments. All of the researchers used the Twitter API to download their own datasets of varying sizes and using various procedures. Because of this the datasets turn out to be very different in terms of size, user composition, and tweet content, which makes results difficult to compare between papers. The creation of the microblog track of the Text Retrieval Conference (TREC) was accompanied by a very large collection of tweet data which may be commonly used in the future, but as discussed in Section 4.1, it was not sufficient for this project.

The bigger issue, however, is the lack of a set of ground truth judgements of the relevance of tweets and users in any dataset. One obvious cause of this is the lack of a canonical dataset upon which recommendations are made. Perhaps if ground truth judgements were available such a dataset would come into common usage, but the fact is that the job is too complex to realistically be performed reliably for any decent sample size. The content and users that are relevant for one user would be vastly different from the content and users relevant for another. And any database of a usable size would have millions and millions of tweets and users, far too many for all of them to be judged for even one user.

5.1.2 Distinguished User Selection

With these challenges in mind, it was necessary to develop methods of evaluating the performance of the algorithm that did not rely on a pre-judged set of relevant content and users. The first step was to choose distinguished users to whom the content should be recommended. For privacy reasons, those users are not named here. Overall, three users were chosen.

Two users were chosen because they are known personally by the author and are long-time users who are present in the dataset. These facts made them ideal candidates for the user studies discussed in Section 5.1.3. Both of these users are computer scientists, so technology is a major interest, but there are some distinguishing interests as well. These two users will be known as users K_i and K_j, with the choice of K meant to indicate that they are the users **K**nown to the author. Both users joined Twitter in early 2009.

The other user was chosen because his interests are clear and easy to distinguish and because he was well connected to the small sample of tweets that were used during initial implementation. This user has more than 7,000 followers as of this writing, up from approximately 3,700 three years ago when the dataset was collected. As of this writing he has published nearly 25,000 tweets. His interests are also in technology, but with a business focus. A large number of his current followees are technology entrepreneurs, but he also posts tweets and has followees related to popular culture such as films, television, and music. This user will be known as user U, indicating that he is **U**nknown to the author.

5.1.3 User Study

The goal of the user study is primarily to determine whether the content recommended to the users was of interest to the distinguished user.

A user study is perhaps the closest method of evaluation to a set of ground truth relevance judgements, though on a smaller scale. For the tweets ranked by the users it is possible to say with certainty how relevant they are and thus to evaluate the precision and recall of the algorithm. The user study could obviously only be performed on the users known to the author, which was part of what motivated the choice of known users as distinguished users.

One major drawback of the user study is that it recommends content to these users based on what was happening three years ago. Thus, the recommendations are based on the interests of three years ago which may no longer be relevant. Before filling out the user survey these users were asked to put themselves in the frame of reference of what might have been interesting to them three years ago, but this is obviously inexact, so the efficacy of user study depends largely on the idea that interests change slowly.

The user studies consisted of two components: recommending users and recommending individual tweets.

For recommending individual tweets, the top thirty tweets from each experiment that was run, retrieved as described in Section 4.6, were retrieved after the algorithm was run. Crucially, tweets which were an @reply to another user were filtered out since a user would not see those if they were to follow the user in question. After running this procedure for all of the experiments, there were approximately 150 tweets for each user to rank.

These filtered tweets were then presented to the user in a random order and each user was asked to rank each of the one hundred tweets from 1 to 5 using the values from Table 5.1. From these scores, the precision was evaluated by considering tweets scored as either 4 or 5 to be relevant tweets and seeing how many of the top 20 tweets were relevant. Recall is not really possible to evaluate with a currently existing Twitter dataset, unfortunately, because of the lack of ground truth relevance judgements on the Twitter data.

Recommending users proceeded in a largely similar manner. A surprising number of the recommended users either no longer appear as users or have set their Twitter feeds to be private, preventing effective evaluation, so the option for the users to rate the user as not available was added. These users were then filtered out of the results when calculating precision.

Score	Description of tweets with this score
1	Not relevant at all, e.g. non-English tweets
2	Useless tweets
3	Average tweet; not particularly useful, but not completely without value
4	Relevant or interesting tweet
5	Very relevant tweets

Table 5.1: User ranking scores for tweets

Score	Description of users with this score
0	Users whose Twitter feeds are no longer available
1	Not relevant at all, e.g. users tweeting in another language
2	Uninteresting users
3	Average user; not particularly interesting, but not without value
4	Interesting users; includes users who once were followed but no longer are
5	Users that the distinguished user now follows

Table 5.2: User ranking scores for users

For each recommended user, the distinguished user under study was presented with a link to their Twitter page so that their profile and recent tweets could be viewed. Each user to be ranked was presented in random order. The users under study were then asked to rank the recommended users according the scale of Table 5.2, which is largely similar to the scale for ranking tweets. The results of this ranking were then used in the same way to determine precision scores by considering users scored as either 4 or 5 to be relevant users.

As with the tweets, it was not really possible to evaluate the recall of current users because it is not possible to say how many total relevant users there are. Additionally, since the time requirements for ranking users are higher, only 30 users were presented to the distinguished users to rank.

5.1.4 Comparison to Current Tweets

The user study is effective for evaluating the results for the known users but is subject to biases and changes in interests and does nothing for evaluating the results for unknown or unavailable users. As such, it was also necessary to develop more automated techniques.

Perhaps the best possibility would be to use the closest thing to a ground truth dataset that exists: the list of all the tweets that a particular distinguished user retweeted. If the user's own connections to each of these tweets were severed then it would be possible to test just how many of these tweets were actually recommended. This presents two major problems, however. First, because the dataset is limited to only approximately 20%-30% of the tweets that were published in the time period, many of the initial tweets were missed and only their retweets were captured. Second, in the small time period being examined, most distinguished users would be unlikely to retweet even one or two tweets, leaving not enough data to evaluate.

A more realistic technique is to assume that a user's interests remain static between the time period represented by the dataset and the present day. This makes it possible to compare the tweets recommended by the system to the tweets that the user is actually interested in while using a separate data source to avoid overfitting.

As described by [22] and mentioned elsewhere in this dissertation, retweets are the most effective means of determining which content a person is interested in. Thus, each tweet recommended by the system can be compared to each of the most recent retweets from the present day twitter stream of the user for whom the recommendations are being created, and the score can be averaged together. These numbers can be compared to a baseline created by comparing each tweet in the system to the reference tweets and taking an average of their similarity scores. This process can be repeated for tweets that the distinguished user has authored.

For this project, the method of determining similarity was the modified cosine similarity metric as described in Section 4.4.1. Because of the large number of tweets in the database the baseline against which these numbers were compared was determined by taking the average of the similarity scores for 2,500 tweets rather than for the entire collection. Looking at the similarity scores for each 500 tweets showed that the average scores did not change very much from the score after evaluating the first 500, validating the choice of 2,500 as the limit.

As with all of the means of evaluation, this is an imperfect measurement. Given their technical interests and the fast-paced nature of that field, many of the technologies that these users are interested in today may not have existed at the time the data was collected. Still, it does allow a comparison to show that the recommendations provided have value above random recommendation and because it is automated it allows far more documents to be ranked than the user study.

5.1.5 Comparison to Current Network State

Much of the research on link prediction in social networks focuses on predicting whether links will be created between users in the future, and the evaluation involves comparing the predicted links to those actually formed later. For this project, the only data available on the state of the social graph is that of the dataset and that of the present day.

Thus, one evaluation method used was to compare the users recommended for each distinguished user to the set of actual followees of that person in the present social network. For the two known users it was also possible to include users who were included in the recommendations whom the distinguished user may once have followed but no longer does. The number of recommended users likely to be in the set of present-day followees is extremely small given that most users follow a small set of people, but by comparing the number of users in the top 25 recommended users who overlap with the present-day followee list to the probability of a random user appearing in that list it is possible to demonstrate that the recommendations have value.

Calculating the probability of a random user from the data being amongst the present-day followees requires a major assumption: that the network today is the same size and has the same users as the network of the dataset. This is necessary because there simply is not data available for the relevant network information at any point except the dataset used here and the present day. This assumption is obviously not true, but it means that the actual probability is smaller than the calculated probability, so the calculated probability can be taken as an upper boundary. Using this assumption, it is possible to calculate the approximate probability that a random user would be a present-day followee of the distinguished user with the following formula:

$$\frac{\Delta_{followees}}{count(user\ vertices) - count(dataset\ followees)}$$

User K_i has 43 followees in the dataset used here compared to 443 today and 72 followers in the dataset used here compared to 791 today. User K_j has 105 followees in the dataset compared to 505 today and 87 followers in the dataset used here compared to 495 today. User U has 279 followees in the dataset compared to 442 today to go along with 3,720 followers in the dataset compared to 7,191 today. Given these numbers, the probability that a random user would be a present day followee of each user is listed in Table 5.3. These probabilities can then be compared to the number of users recommended by the algorithm who the user currently follows in order to establish the value of the algorithm.

User	$\Delta_{followees}$	Probability
U	163	0.04%
K_i	400	0.10%
K_j	400	0.09%

Table 5.3: Probability of a random user being added as a followee

5.2 Results

This section describes the results which were obtained. Most of the scores are presented in the form of precision at rank and are based on the results of the user study. These scores are labelled in the various tables as 'P@5', 'P@10', 'P@15', and 'P@20', representing the precision at 5, 10, 15, and 20. Recall from above that for purposes of calculating precision, a tweet or a user is considered to be relevant if it ranked at either 4 or 5 in the user study. Since user U was not part of the user study, he has no numbers for this measurement. All of the rankings for the known users K_i and K_j also include a result for the average score at rank 5, 10, 15, and 20, which imparts some additional information about the quality of the rankings.

For the user recommendation results, the tables also include a column for '% 5s @ 5', '% 5s @ 10', etc. This value is the percentage of users rated as a 5 at various ranks in the recommendations. This corresponds to the users that they actually have started following since this data was collected in 2009, though in the case of the users included in the user study, at least one or two of the users ranked as 5 were users who they did not follow before seeing the recommendation. This measurement is the metric described in Section 5.1.5 of comparing the list of recommended users to the users that the distinguished user now follows. Because it is possible to determine this for all users, this number is included for user U. These numbers compare very favourably with the chance of a random user appearing in this list as discussed in Section 5.1.5, demonstrating that the user recommendations are very useful, though as will become apparent most of the value in user recommendations comes from the initial score.

For most of the variations on the baseline results only one user was studied for each variation, providing a very small sample size. So while the results are very promising for many of the experiments, caution is warranted because there simply is not very much information. The need for improving the sample size is discussed further in Chapter 6. Another interesting outcome of the experimental variations was that the differences between many of the configurations were very small, usually with only a few users or tweets being different or sometimes simply with them being in a slightly different position.

5.2.1 Baseline Results

The results presented here as the baseline results are based on a λ_{users} parameter value of 0.7 and a λ_{tweets} parameter value of 0.9. These values represent the fact that the initial scores of the users are more useful as a basis for recommendation than the initial scores of the tweets.

It is clear just from looking at the initial scores for both users and tweets that the tweet scores are significantly less valuable than the user scores. While the initial user scores would provide an excellent ranking before the algorithm is even run, the initial tweet scores are not nearly so useful. By keeping the value of λ_{tweets} closer to 1 the impact of these less useful initial scores on the final outcome is mitigated.

These results are also based on the presence of all of the edge types listed in Table 4.6 in Section 4.3.2, with the directionality indicated there and an equal weighting for all edges.

5.2.1.1 User Recommendation Results

For all users, the run under these default parameters provided very good results on the user recommendations. In particular, user K_i remarked that the user rankings were very good and that some of the recommended users for him were people that he found interesting enough to start following after seeing the recommendation. Table 5.4 shows the results for user recommendation using the base configuration and compares these to the results obtained from just the initial scores. Note that in the case of the user U, the scores were the same for both, since no current followees were found by the default algorithm. One of his current followees did appear at a rank between 20 and 25 and the same user also showed up in some of the experimental variations which included user U, though those results are not described due to the lack of user ratings.

These results are quite good, and in particular the number of users in the recommendation list who are current followees of the distinguished users (% 5s@Rank) compare very favourably with the probability of a random user from the dataset appearing in the followees list as presented in Table 5.3, where all of the probabilities were less than 1%. That being said, at least for the case of user recommendations the algorithm provides little to no improvement over the results that would be achieved

Metric	**User U**	**User K_i Init. Score**	**User K_i**	**User K_j Init. Score**	**User K_j**
P@5	-	0.6	0.6	0.8	0.6
P@10	-	0.8	0.8	0.7	0.6
P@15	-	0.533	0.733	0.666	0.6
P@20	-	0.6	0.7	0.6	0.55
% 5s@5	0%	40%	20%	40%	40%
% 5s@10	0%	30%	30%	40%	40%
% 5s@15	0%	20%	26.6%	33%	33%
% 5s@20	0%	20%	20%	25%	25%
Avg. Score @5	-	3.2	3.0	3.6	3.4
Avg. Score @10	-	3.7	3.7	3.8	3.5
Avg. Score @15	-	3.4	3.733	3.666	3.6
Avg. Score @20	-	3.55	3.65	3.6	3.35

Table 5.4: The user recommendation results achieved using only the initial user scores and using the algorithm in the baseline configuration

if using only the initial score. As will become clear, however, the major value of these accurate user scores are that they make it possible to drastically improve the quality of the tweet recommendations over the initial scores.

5.2.1.2 Tweet Recommendation Results

The method of using cosine similarity for comparison as described in Section 5.1.4 turned out not to be particularly useful, though it did demonstrate at a rudimentary level that the recommended tweets were at least somewhat valuable over noise. Because the information provided by this metric was of limited utility, it was only done for one set of recommendations. The results below come from the recommendations provided by one of the combined configurations described in Section 5.2.6 rather than for the base configuration, but since the tweets themselves recommended by these two configurations were so similar the results for either set of recommendations would be similar.

Table 5.5 shows the average cosine similarity between the recommended tweets and the reference tweets for user U. The table shows, for example, that the average of the similarity of the first 20 recommended tweets was 0.00330 for the reference set consisting of the tweets authored in the present day by the distinguished user and 0.00268 for the reference set consisting of the tweets retweeted in the present day by the distinguished user.

Recommendation List Position	Authored Tweets Similarity	Retweeted Tweets Similarity
5	0.00286	0.00100
10	0.00285	0.00190
15	0.00308	0.00227
20	0.00330	0.00268

Table 5.5: Average cosine similarity between recommended tweets and current tweets of user U

# of Random Tweets	Authored Tweets Similarity	Retweeted Tweets Similarity
500	0.00217	0.00198
1000	0.00215	0.00200
1500	0.00214	0.00200
2000	0.00211	0.00197
2500	0.00210	0.00199

Table 5.6: Average cosine similarity between random tweets and current tweets of user U

Table 5.6 shows the average cosine similarity between random tweets taken from the database and the various reference sets. The table shows, for example, that after all 2,500 tweets had been considered their average similarity to the reference set was 0.00210 and 0.00199 for the authored tweets and the retweeted tweets, respectively. The results from the recommended list of tweets are noticeably better than those for the random tweets. Still, given the inaccuracy of the cosine similarity measurement to begin with and the lack of a major difference in the similarity to retweeted tweets, this measurement was not particularly useful and will not be revisited. It does, however, suggest that the recommendations have at least some value.

The comparison between the current network and the results of the user study provided a much better source of results. These results are shown in Table 5.7, including a comparison with the results obtained using the initial tweet scores only. Since no user study information was available for user U, he is not included in this table. After rating the recommended tweets as part of the user study, user K_i mentioned that the recommendations were at least as good as those provided by Twitter. User K_j was generally a harsher critic of the tweets and he remarked that in general he was quite discerning about what he liked on Twitter and would automatically dislike a tweet which contained spelling errors, which on Twitter provides a major limitation.

As with the user recommendations, these results are very good. In the original

Metric	**User K_i Init. Score**	**User K_i**	**User K_j Init. Score**	**User K_j**
P@5	0	0.6	0.4	0.0
P@10	0.1	0.6	0.3	0.2
P@15	0.066	0.4	0.266	0.4
P@20	0.05	0.3	0.2	0.4
Avg. Score @5	1.8	3.4	2.6	2.8
Avg. Score @10	2.4	3.3	2.9	3.1
Avg. Score @15	2.133	2.933	2.666	3.266
Avg. Score @20	2.05	2.9	2.55	3.3

Table 5.7: The tweet recommendation results achieved using only the initial tweet scores and using the algorithm in the baseline configuration

Co-HITS paper ([7]), Deng et. al. had results for precision at rank 5 of between 0.35 and 0.39 and precision at 10 of between 0.31 and 0.35 for a web search application, and the results here are in the same neighbourhood when averaged together. It is not surprising to see that the results from the base configuration are quite a bit better than those based off of the original tweet scores given how poor the tweets with the highest initial scores were. Though the precision scores show that user K_j did not like the tweets in the top 10 of the recommendation list very much, the results from the rest of the top 20 and from the average score still suggest that the rankings were quite good and an improvement on the initial scores.

5.2.2 Varying λ Parameters

The choice of the λ parameters from the previous section was not arbitrary, but rather was based on experimentation to see which values for these parameters produced the best results. Two experiments were run. The first, on user K_i, held the value of λ_{tweets} constant at 0.9 while varying the value of the λ_{users} parameter. The second experiment was run on user K_j and held the value of λ_{users} constant at 0.7 while varying the value of the λ_{tweets}.

Recall that the closer the λ parameters are to 0, the more of an impact the initial score has on the final outcome—the fact which drove the selection of λ values to hold constant for each experiment. Since the tweet scores were so bad, it made sense to keep that parameter much closer to 1. The results of these experiments were obtained and graphed for P@5, P@10, and P@20.

Figure 5.1 shows the results of the user recommendations for user K_i when varying the λ_{tweets} parameter, while Figure 5.2 shows the results of the tweet recommendations

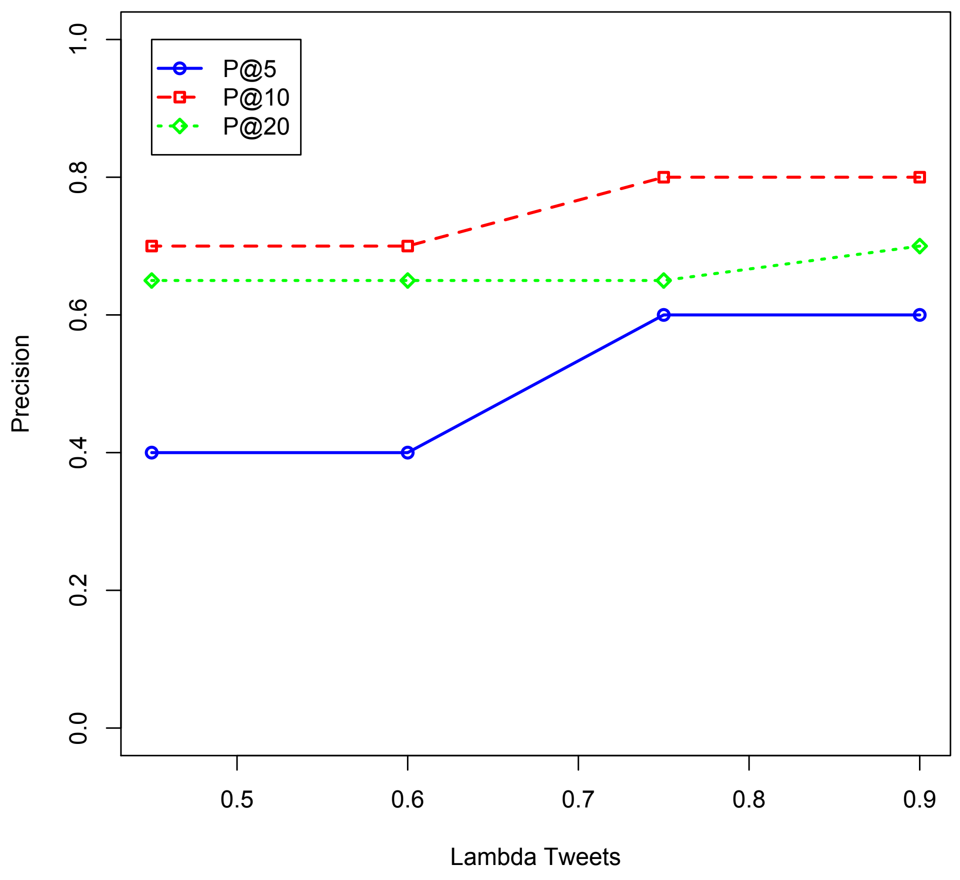


Figure 5.1: User recommendation results for user K_i when varying λ_{tweets}. The value of λ_{users} is held constant at 0.7.

for user K_i when varying the λ_{tweets} parameter. Both of these graphs confirm that a value of lambda tweets closer to 1, as was used for the baseline configuration, provides better results.

Figure 5.3 shows the results of the user recommendations for user K_i when varying the λ_{users} parameter, while Figure 5.4 shows the results of the tweet recommendations for user K_i when varying the λ_{users} parameter. Both of these graphs confirm that a value of lambda tweets too close to 1 will provide terrible results. Because the initial scores for the users are so good, it does not particularly matter how close to 0 the value of lambda users gets.

Another test was undertaken for user K_j, which was to hold λ_{users} constant at 0.9 while varying the tweets parameter. Though not many data points were collected, it does still suggest that keeping the value of the λ_{tweets} parameter closer to 1 is

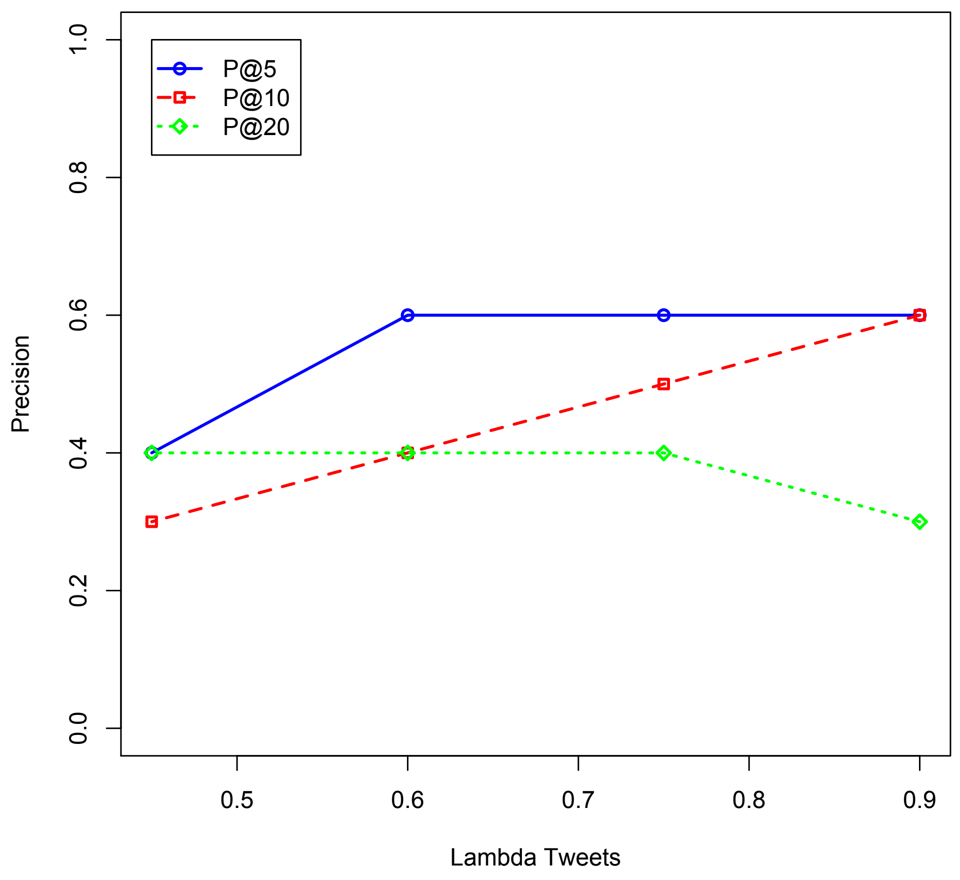


Figure 5.2: Tweet recommendation results for user K_i when varying λ_{tweets}. The value of λ_{users} is held constant at 0.7.

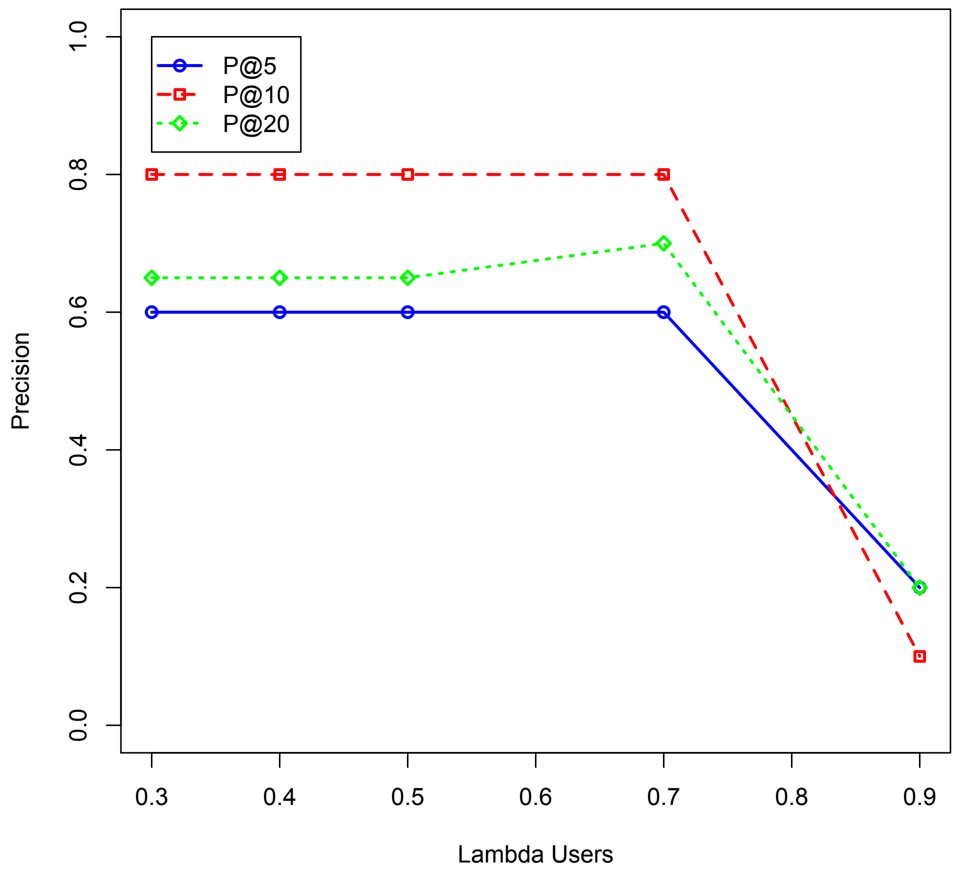


Figure 5.3: User recommendation results for user K_i when varying λ_{users}. The value of λ_{tweets} is held constant at 0.9.

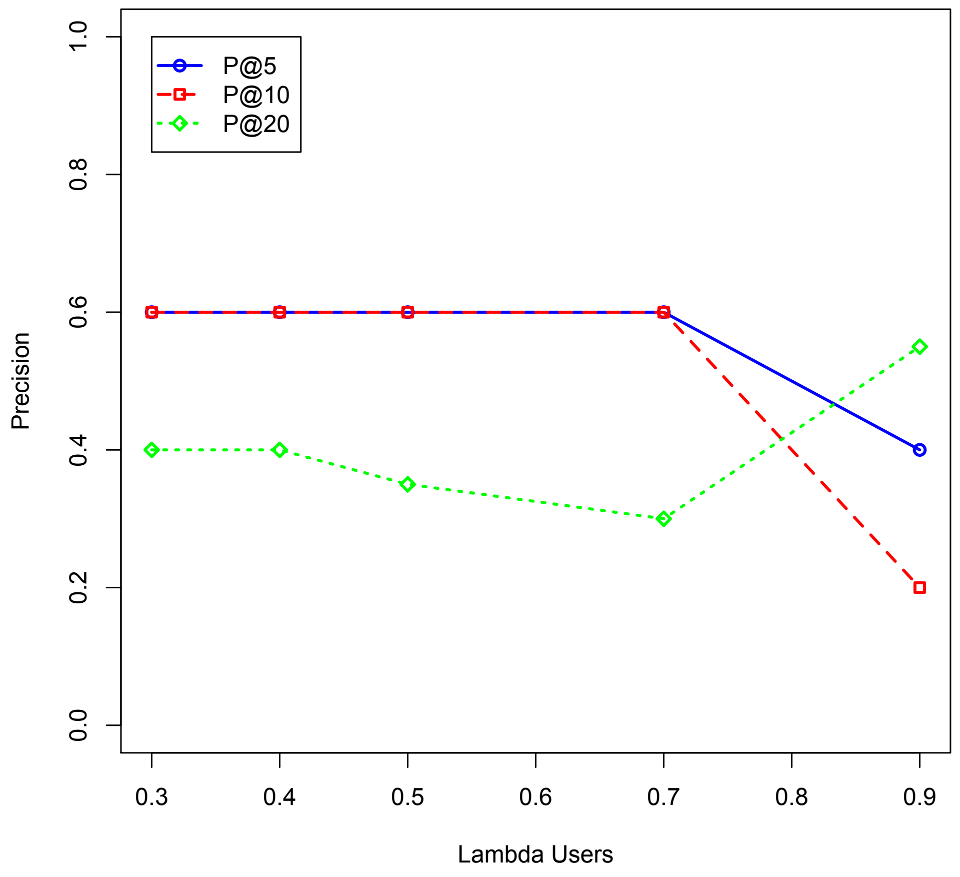


Figure 5.4: Tweet recommendation results for user K_i when varying λ_{users}. The value of λ_{tweets} is held constant at 0.9.

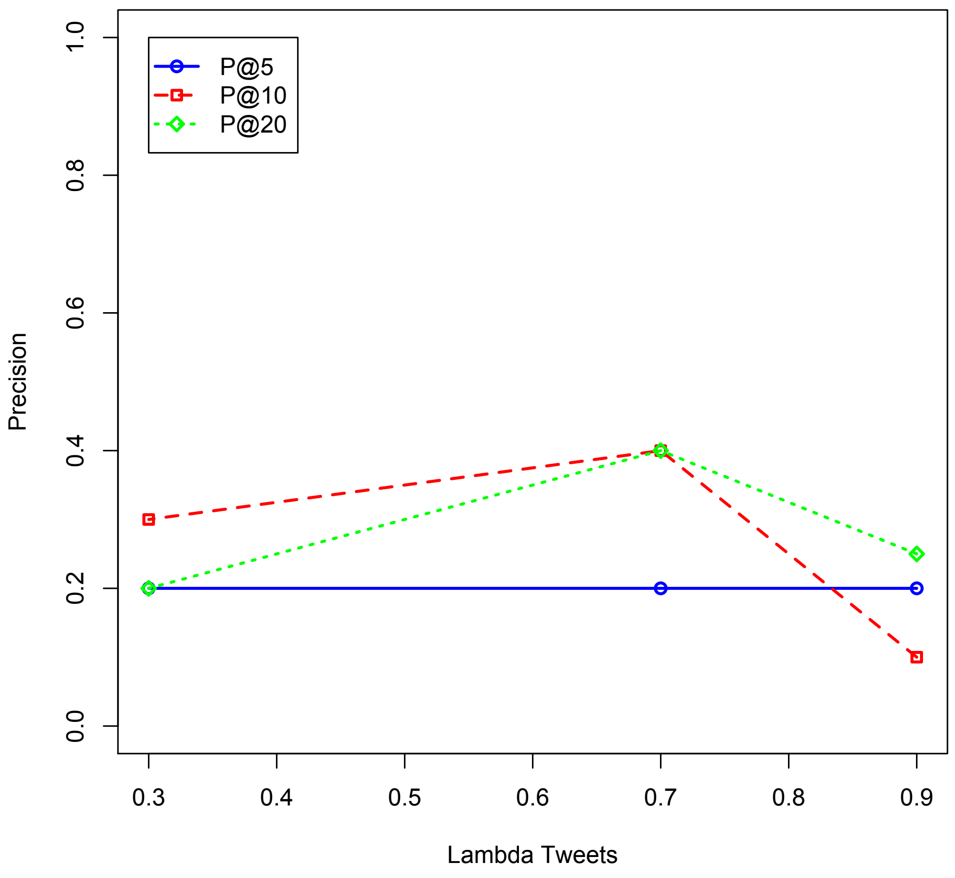


Figure 5.5: User recommendation results for user K_j when varying λ_{tweets}. The value of λ_{users} is held constant at 0.9.

preferable. The graph for the user recommendation results on this data is shown in Figure 5.5, while the graph for the user recommendation results on this data is shown in Figure 5.6. The results are not as good as when the value of λ_{users} is closer to the optimal values revealed in the other experiments in this section.

A final experiment on the lambda parameters was undertaken which moved both λ parameters to 1, thereby removing much of the impact of the initial scores. The results for this configuration were terrible, with the precision for both tweets and users being 0.2, the average scores being between 2.5 and 2.75, and no scores of 5 for any of the user recommendations. The results for this experiment are included as part of the experiments on varying the types of edge included in the algorithm described in Section 5.2.3.

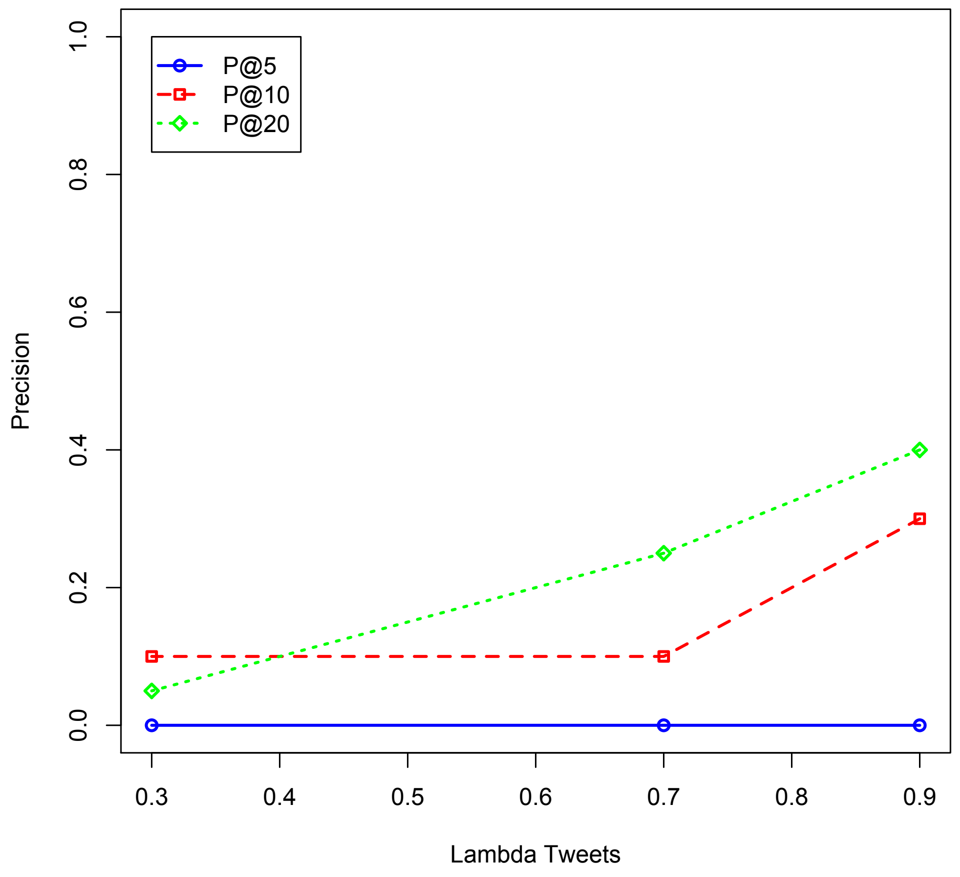


Figure 5.6: Tweet recommendation results for user K_j when varying λ_{tweets}. The value of λ_{users} is held constant at 0.9.

5.2.3 Varying Edge Types Included

As was mentioned in Chapter 4, the algorithm was slow to run due to database access times. This made it difficult to experiment with removing edge types because with twelve different edge types there were a large number of possible combinations that could have been removed. Instead, the experiments focused on removing the edge types that seemed like they might be less valuable.

Recall the edge types from Table 4.6. The retweet, mention, and at-reply edges connect the user who was retweeted, mentioned, or at-replied to with the tweet that retweeted, mentioned, or at-replied to them. The logic behind this was previously discussed, but the link is a tenuous one and thus these are prime candidates for removal to see the effect on the results. Note that the other five retweet, mention, and at-reply edges remained in place.

There were 292,475 edges of these types, which is a very small amount in the context of 120 million total edges, and as can be seen from the results from removing these edges shown in Table 5.8, removing these edges had very little impact on the overall quality of the results when compared with the baseline results already presented, and what little impact there was tended to be negative. This experiment was done for user K_i only and it is difficult to draw too many conclusions from such a small smaple size, but since the edges do not seem to be particularly valuable they could perhaps be removed from the graph entirely in order to speed up the algorithm.

The results from removing one or both of the types of content edges were only slightly clearer, though the impact improved the results rather than hurting them. Two basic experiments were run: one that removed all content edges (cf Table 5.9) and one that removed only the hashtag edges (cf Table 5.10). There were a lot of the content edges: about 42 million hashtag edges and about 38 million entity-based edges. It was surprising, then, when removing them had such a small effect. Comparing the results in Table 5.9 with the baseline results already presented shows very little difference. As with the other experiment with removing edges, this experiment was only run for user K_i.

In order to attempt to isolate the effect of removing the edges from the effect of the initial scores, the removal of the content edges was also performed with both λ parameters set to 1. The results for this are particularly interesting in that the results for the quality of the user recommendations were much lower than the default configuration and even lower than with the content edges in place, while the tweet scores improved greatly. This result is shown in Table 5.11, which compares the results of the user and tweet recommendations for user K_i with the content edges in

Metric	K_i **Base Config. User Scores**	K_j **User Scores**	K_i **Base Config. Tweet Scores**	K_j **Tweet Scores**
P@5	0.6	0.6	0.6	0.4
P@10	0.8	0.8	0.6	0.5
P@15	0.733	0.666	0.4	0.333
P@20	0.7	0.65	0.3	0.25
Avg. Score @5	3.0	3.0	3.4	3.4
Avg. Score @10	3.7	3.7	3.3	3.4
Avg. Score @15	3.733	3.6	2.933	2.866
Avg. Score @20	3.65	3.6	2.9	2.7
% 5s@5	20%	20%	-	-
% 5s@10	30%	30%	-	-
% 5s@15	26.6%	26.6%	-	-
% 5s@20	20%	20%	-	-

Table 5.8: The tweet score and user score results for user K_i when removing the simple retweet, mention, and @reply edges.

Metric	K_i **Base User Scores**	K_i **User Scores**	K_i **Base Tweet Scores**	K_i **Tweet Scores**
P@5	0.6	0.6	0.6	0.6
P@10	0.8	0.8	0.6	0.6
P@15	0.733	0.866	0.4	0.466
P@20	0.7	0.8	0.3	0.45
Avg. Score @5	3.0	2.8	3.4	3.6
Avg. Score @10	3.7	3.5	3.3	3.7
Avg. Score @15	3.733	3.8	2.933	3.333
Avg. Score @20	3.65	3.7	2.9	3.25
% 5s@5	20%	0%	-	-
% 5s@10	30%	10%	-	-
% 5s@15	26.6%	20%	-	-
% 5s@20	20%	20%	-	-

Table 5.9: Results for user K_i from removing all content edges, as compared with the base configuration

Metric	K_i **Base User Scores**	K_i **User Scores**	K_i **Base Tweet Scores**	K_i **Tweet Scores**
P@5	0.6	0.6	0.6	0.6
P@10	0.8	0.8	0.6	0.7
P@15	0.733	0.733	0.4	0.533
P@20	0.7	0.7	0.3	0.45
Avg. Score @5	3.0	3.0	3.4	3.4
Avg. Score @10	3.7	3.7	3.3	3.7
Avg. Score @15	3.733	3.666	2.933	3.4
Avg. Score @20	3.65	3.65	2.9	3.25
% 5s@5	20%	20%	-	-
% 5s@10	30%	30%	-	-
% 5s@15	26.6%	26.6%	-	-
% 5s@20	20%	20%	-	-

Table 5.10: Results for user K_i from removing the hashtags edges only, as compared with the base configuration

place but the λ parameters set to 1 and the recommendations with the λ parameters set to 1 but the content edges removed. As with all of the experimental variations described here, the sample size is too small to draw any firm conclusions, but it still suggests an interesting avenue for further exploration in the future.

One final test on removing edges combined the two main edge types that were experimented on by removing the retweet, mention, and at-reply edges in addition to the two types of content edges. Again, the λ parameters were set to 1 in an attempt to remove the impact of the initial scores on the outcome. For the user recommendations these results were slightly better than those from removing only the content edges, while for the tweet recommendations they were more mixed, with the top 10 showing worse results and the rest of the top 20 showing improved results. The usual caveats about small sample size apply here as well, of course.

5.2.4 Varying Edge Weights

The slowness of running the algorithm made experimentation on edge weights particularly difficult because with so many different edges and possible weightings for them a rigorous experiment would require far more time than was available. As such, the experiments which were done were based largely on intuition and were small in number, with only four different weightings being explored.

Metric	$\lambda_{t,u} = 1$ **Base User Scores**	**User Scores**	$\lambda_{t,u} = 1$ **Base Tweet Scores**	**Tweet Scores**
P@5	0.0	0.2	0.2	0.6
P@10	0.2	0.1	0.2	0.7
P@15	0.2	0.133	0.2	0.533
P@20	0.2	0.1	0.2	0.45
Avg. Score @5	2.4	2.4	2.8	3.8
Avg. Score @10	2.7	2.3	2.8	3.9
Avg. Score @15	2.8	2.2	2.733	3.4
Avg. Score @20	2.6	2.05	2.7	3.15
% 5s@5	0%	0%	-	-
% 5s@10	0%	0%	-	-
% 5s@15	0%	0%	-	-
% 5s@20	0%	0%	-	-

Table 5.11: The tweet score and user score results for user K_i when removing the content edges with the λ parameters set to 1

Metric	$\lambda_{t,u} = 1$ **Base User Scores**	**User Scores**	$\lambda_{t,u} = 1$ **Base Tweet Scores**	**Tweet Scores**
P@5	0.0	0.4	0.2	0.6
P@10	0.2	0.2	0.2	0.5
P@15	0.2	0.2	0.2	0.666
P@20	0.2	0.15	0.2	0.75
Avg. Score @5	2.4	3.2	2.8	3.8
Avg. Score @10	2.7	2.7	2.8	3.5
Avg. Score @15	2.8	2.666	2.733	3.666
Avg. Score @20	2.6	2.15	2.7	3.75
% 5s@5	0%	0%	-	-
% 5s@10	0%	0%	-	-
% 5s@15	0%	0%	-	-
% 5s@20	0%	0%	-	-

Table 5.12: The tweet score and user score results for user K_i when removing the content edges and the simple retweet, mention, and @reply edges with the λ parameters set to 1.

Edge Type	**Test 1 Weight**	**Test 2 Weight**	**Test 3 Weight**	**Test 4 Weight**
Authorship	2	2	2	4
Follower	1.25	1.25	1.25	1.5
Retweet Followees	2	3	3	3
Retweet Followers	0.75	1.25	1.25	1.25
Mention Followees	2	2	2	1.5
Mention Followers	0.75	1	1	1
@reply Content	0.5	0.5	0.25	0.5
Hashtag	1	1	3.5	1
Content	1	1	1.5	1

Table 5.13: Edge weights used for each experiment

The experiments done here were performed on user K_j, with directionality per the default, though authorship edges were bi-directional. Table 5.13 shows the weights that were used for each edge type in each experiment. Table 5.14 shows the results for each experiment on the user recommendations and Table 5.15 shows the results for each experiment on the tweet recommendations. Note that the edges corresponding to basic retweets, mentions, and at replies were not included for any of these experiments, and thus no weights are listed for these edge types.

The methodology used was to sum the total weight of all edges emanating from a vertex, and the chance of taking a particular edge was just the weight of that edge's type divided by the total weight of all edges. So if a particular vertex had two edges with a weight of 2 each and one edge with a weight of 0.5 then the total weight would be 4.5, so the chance of going to each of the edges with weight 2 would be $2 \div 4.5 = 0.44$ and the chance of going to the edge with weight 0.5 would be $0.5 \div 4.5 = 0.11$.

One very clear result from the experiments is that none of them gave results which were particularly distinct, though the weights in the fourth experiment were perhaps slightly better. The most likely reason for this is that the values for the weights of the various edges were not very distinct between the different experiments. Instead, they were based on a few assumptions, such as that retweets, followers, and authorship are very important. Existing research and some of the other experiments support these intuitions, but the experiments run here neither proved nor disproved them.

5.2.5 Varying Edge Directionality

The directionality of the edges when producing the baseline results described in Section 5.2.1 followed the description in Table 4.6. As shown in that section, this direc-

Metric	K_j Base User Scores	Test 1 User Scores	Test 2 User Scores	Test 3 User Scores	Test 4 User Scores
P@5	0.6	0.6	0.6	0.6	0.6
P@10	0.6	0.6	0.6	0.6	0.6
P@15	0.6	0.533	0.6	0.6	0.6
P@20	0.55	0.6	0.65	0.6	0.6
Avg. Score @5	3.4	3.4	3.4	3.4	3.4
Avg. Score @10	3.5	3.7	3.7	3.7	3.7
Avg. Score @15	3.6	3.4	3.666	3.666	3.666
Avg. Score @20	3.35	3.5	3.65	3.65	3.65
% 5s@5	20%	40%	40%	40%	40%
% 5s@10	40%	40%	40%	40%	40%
% 5s@15	33.3%	33.3%	40%	40%	40%
% 5s@20	25%	30%	30%	30%	30%

Table 5.14: Results of the user recommendations for user K_j when varying the weights of the edges

Metric	K_j Base Tweet Scores	Test 1 Tweet Scores	Test 2 Tweet Scores	Test 3 Tweet Scores	Test 4 Tweet Scores
P@5	0.0	0.4	0.4	0.4	0.4
P@10	0.2	0.3	0.4	0.4	0.4
P@15	0.4	0.4	0.4	0.4	0.4
P@20	0.4	0.35	0.35	0.35	0.35
Avg. Score @5	2.8	3.4	3.4	3.4	3.4
Avg. Score @10	3.1	3.3	3.4	3.4	3.4
Avg. Score @15	3.266	3.4	3.4	3.4	3.4
Avg. Score @20	3.3	3.3	3.3	3.3	3.4

Table 5.15: Results of the tweet recommendations for user K_j when varying the weights of the edges

tionality produced generally good recommendations for both users and tweets.

One obvious variation on these baseline results is to remove edge directionality altogether and assume that each edge type has an influence in both directions. For at least some of the edge types, this makes perfect sense. Consider the authorship edge, for example. It certainly makes sense that if a particular tweet received a high score because of something it was connected to then that score should be passed along to the author of the tweet. Similarly, a highly scoring user should obviously transfer some of that high score along to the tweets that they author.

Running the algorithm with all edge types being bi-directional was one of the first experiments that was run for user U. The results from doing this were very clearly terrible. The tweets were dominated by one or two users and the tweets of those users were political in nature and diametrically opposed to the political views of the distinguished user. On the scale used for the user study, all of these would have ranked as a 2 had user U been part of the user study.

The reason for this was that the user who produced the tweets followed over 40,000 people, and today follows more than 80,000 people. Clearly this user does not interact with or even read all of the tweets from these 40,000 people, but the algorithm cannot take that into account. So because this user tweeted frequently and followed so many people, their tweets were highly scored by the system with bi-directional edges because each of their 19 tweets was connected to thousands of users whose scores all contributed to high scores for these tweets.

This result demonstrates that for the edges based on the network state it does not makes sense to vary the directionality of the edges, either by reversing them or by making them bi-directional. Take follower edges for example: clearly there is not an influence by a user on the tweets of someone they follow. This was the biggest issue in the experiment done with no directionality at all—it assumed influence where there was none and came up with correspondingly terrible results.

Eliminating the network edges from consideration then leaves only the two content-based edges and the authorship edge as candidates for changing directionality. With these edges it is not clear that any one direction should be the one that has the influence; for instance, it could make sense that all tweets on a given subject should provide their scores to a user or that a highly scoring user should provide an impact on all tweets mentioning subjects of a similar subject. Similarly, leaving these as bi-directional as in the default configuration also makes sense.

Both directions for the content edges yield improved results for the tweet recommendations, which is quite a surprising result. Directing the content edges from tweet

Metric	K_i Base User Scores	K_i User Scores	K_i Base Tweet Scores	K_i Tweet Scores
P@5	0.6	0.6	0.6	0.6
P@10	0.8	0.8	0.6	0.7
P@15	0.733	0.866	0.4	0.533
P@20	0.7	0.75	0.3	0.45
Avg. Score @5	3.0	3.0	3.4	3.6
Avg. Score @10	3.7	3.6	3.3	3.8
Avg. Score @15	3.733	3.866	2.933	3.4
Avg. Score @20	3.65	3.7	2.9	3.25
% 5s@5	20%	20%	-	-
% 5s@10	30%	20%	-	-
% 5s@15	26.6%	26.6%	-	-
% 5s@20	20%	20%	-	-

Table 5.16: Results for user K_i when content edges are directed from tweet to user

Metric	K_i Base User Scores	K_i User Scores	K_i Base Tweet Scores	K_i Tweet Scores
P@5	0.6	0.6	0.6	0.8
P@10	0.8	0.6	0.6	0.6
P@15	0.733	0.733	0.4	0.466
P@20	0.7	0.7	0.3	0.35
Avg. Score @5	3.0	3.0	3.4	4.4
Avg. Score @10	3.7	3.2	3.3	3.6
Avg. Score @15	3.733	3.6	2.933	3.266
Avg. Score @20	3.65	3.6	2.9	2.9
% 5s@5	20%	20%	-	-
% 5s@10	30%	10%	-	-
% 5s@15	26.6%	20%	-	-
% 5s@20	20%	20%	-	-

Table 5.17: Results for user K_i when content edges are directed from user to tweet

Metric	K_i **Base User Scores**	K_i **User Scores**	K_i **Base Tweet Scores**	K_i **Tweet Scores**
P@5	0.6	0.6	0.6	0.6
P@10	0.8	0.8	0.6	0.5
P@15	0.733	0.733	0.4	0.4
P@20	0.7	0.65	0.3	0.4
Avg. Score @5	3.0	3.0	3.4	3.6
Avg. Score @10	3.7	3.7	3.3	3.3
Avg. Score @15	3.733	3.733	2.933	3.0
Avg. Score @20	3.65	3.6	2.9	3.1
% 5s@5	20%	20%	-	-
% 5s@10	30%	30%	-	-
% 5s@15	26.6%	26.6%	-	-
% 5s@20	20%	20%	-	-

Table 5.18: Results for user K_i when the authorship edge is bi-directional.

to user yields slightly better user recommendations, though the recommendations are quite good in both cases. Table 5.16 shows the results when content edges are directed from tweet vertices to user vertices and Table 5.17 shows the results when the content edges go in the other direction.

Experiments were also done with the authorship edge. By default this edge was directed from the user to the tweet, but as was previously discussed this could certainly have been bi-directional. The results shown in Table 5.18 are from running the algorithm for user K_i with all the standard edges and directions but with the authorship edge being bi-directional. Table 5.19, meanwhile, shows the same experiment for user K_j. The results for this configuration show that changing the direction of these 500,000 edges has almost no effect.

5.2.6 Combined Configuration

Most of the experiments showed very mixed results for individual variations, with most of the experiments making very little difference on the final outcome. The values for the λ parameters were fairly clear, but changing the included edges and the edge directions had little effect. Still, one thing that has not been explored thus far is a combination of some these different features. The results in this section combine some of the past experiments into a combined configuration in hopes that the combination will produce better results than the generally middling results of the previous experiments. Unfortunately, the results are generally comparable to those of

Metric	K_j **Base User Scores**	K_j **User Scores**	K_j **Base Tweet Scores**	K_j **Tweet Scores**
P@5	0.6	0.6	0.0	0.0
P@10	0.6	0.6	0.2	0.2
P@15	0.6	0.6	0.4	0.266
P@20	0.55	0.6	0.4	0.3
Avg. Score @5	3.4	3.4	2.8	2.8
Avg. Score @10	3.5	3.7	3.1	3.0
Avg. Score @15	3.6	3.6	3.266	3.133
Avg. Score @20	3.35	3.5	3.3	3.2
% 5s@5	20%	40%	-	-
% 5s@10	40%	40%	-	-
% 5s@15	33.3%	33%	-	-
% 5s@20	25%	30%	-	-

Table 5.19: Results for user K_j when the authorship edge is bi-directional.

the baseline configuration, though perhaps a more robust experiment which studied more users would reveal more clear-cut results.

Table 5.20 shows the results for user K_i with the authorship edge being bi-directional and the simple retweet, mention, and at-reply edges being removed. As with many of the other experimental variations the results differ only slightly from the baseline results, and only one or two tweets and users are different. Table 5.21 shows the results of the same experiment for K_j, which shows slightly better results. The tweet recommendations for user K_j do show quite a bit of improvement early in the recommendations list, though this is due in no small part to the fact that the baseline tweet results for this user were poor to begin with.

By default, the algorithm ignores the duplicate edges of the same type linking a user and a tweet that exist in the graph. As an additional experiment on user K_i, the experiment shown was repeated with the duplicate edges in the graph being left in. The results for this were identical to the results when leaving these edges out. This surprising result suggests a possible way to speed up the algorithm, since more than 10 million of the edges are duplicates.

One final experiment was run on K_j, similar to those on user K_i, but with the two content edge types not included. The results for this experiment are shown in Table 5.22 and show that the early tweet recommendations were much improved over the baseline, while the user recommendations were about the same.

Metric	K_i **Base User Scores**	K_i **User Scores**	K_i **Base Tweet Scores**	K_i **Tweet Scores**
P@5	0.6	0.6	0.6	0.4
P@10	0.8	0.8	0.6	0.4
P@15	0.733	0.666	0.4	0.4
P@20	0.7	0.6	0.3	0.3
Avg. Score @5	3.0	3.0	3.4	3.4
Avg. Score @10	3.7	3.7	3.3	3.2
Avg. Score @15	3.733	3.6	2.933	3.066
Avg. Score @20	3.65	3.55	2.9	2.8
% 5s@5	20%	20%	-	-
% 5s@10	30%	30%	-	-
% 5s@15	26.6%	26.6%	-	-
% 5s@20	20%	20%	-	-

Table 5.20: Results for user K_i with the authorship edge being bi-directional and the retweet, mention, and @reply edges not included.

Metric	K_j **Base User Scores**	K_j **User Scores**	K_j **Base Tweet Scores**	K_j **Tweet Scores**
P@5	0.6	0.6	0.0	0.4
P@10	0.6	0.6	0.2	0.4
P@15	0.6	0.6	0.4	0.333
P@20	0.55	0.65	0.4	0.4
Avg. Score @5	3.4	3.4	2.8	3.4
Avg. Score @10	3.5	3.7	3.1	3.5
Avg. Score @15	3.6	3.6	3.266	3.266
Avg. Score @20	3.35	3.65	3.3	3.35
% 5s@5	20%	40%	-	-
% 5s@10	40%	40%	-	-
% 5s@15	33.3%	33.3%	-	-
% 5s@20	25%	30%	-	-

Table 5.21: Results for user K_j with the authorship edge being bi-directional and the retweet, mention, and @reply edges not included.

Metric	K_j **Base User Scores**	K_j **User Scores**	K_j **Base Tweet Scores**	K_j **Tweet Scores**
P@5	0.6	0.6	0.0	0.6
P@10	0.6	0.5	0.2	0.5
P@15	0.6	0.533	0.4	0.466
P@20	0.55	0.5	0.4	0.4
Avg. Score @5	3.4	3.4	2.8	3.6
Avg. Score @10	3.5	3.3	3.1	3.5
Avg. Score @15	3.6	3.4	3.266	3.466
Avg. Score @20	3.35	3.3	3.3	3.4
% 5s@5	20%	40%	-	-
% 5s@10	40%	30%	-	-
% 5s@15	33.3%	26.6%	-	-
% 5s@20	25%	25%	-	-

Table 5.22: Results for user K_j with the authorship edge being bi-directional and the retweet, mention, @reply, hashtag, and entity-based edges not included.

5.3 Dataset Biases

The results described in this chapter revealed some issues that come with selecting tweets to use for such a study that can end up having a major impact on the final results.

One drawback of the Twitter network is that particular communities are often over-represented, which can make it more difficult to find relevant content for people not in those communities. The tweet recommendations here show that effect quite clearly. User U is a part of the Silicon Valley entrepreneurship community, which is very strongly represented on Twitter. Within the 500,000 tweets studied here that user's results were very good since there were a large number of tweets that would be interesting to someone from that community. But the trade-off is that there were far fewer tweets of interest available for the other users who were not part of that community.

Another bias in the Twitter dataset is that the subjects of the tweets can be overwhelmed by one particular event. The day that the tweets used here were collected happened to be World AIDS Day in 2009, resulting in a much higher level of discussion of things related to that than would normally be expected. This also had the effect of limiting the number of interesting tweets for someone who wasn't interested in content related to World AIDS Day.

Finally, because of limitations within the Twitter API and limitations of storing and collecting a large amount of data, the Twitter data used here only represents 20-30% of tweets during the time period. This results in a lot of cases where a retweet is among the recommendations while the original tweet was never seen. The results could be improved if the repeated retweets of the same tweet were removed from the recommendations in favour of the original tweet.

All of these biases were noticeable in the results, though the impact was generally not a major one and the algorithm still produced excellent recommendations. It would be interesting and valuable to study the effect that these biases have on the results, such as by studying a larger number of tweets, a more complete dataset, or multiple different time periods.

That is far from the only work that remains to be done in this still very new research area, of course. While the results here are very promising, the experiments and methodologies suggest that plenty of improvements could be made and further experiments performed to improve the results described in this chapter. The final chapter will discuss some of the other future work that could be done to expand on and improve this project as well as providing concluding remarks on the project as a whole.

Chapter 6

Conclusion

6.1 Concluding Remarks

The goal of this project was to create an effective content recommendation system for social networks, one capable of recommending both interesting users and interesting content. The results certainly indicate that this goal was achieved.

Between 5% and 40% of the top 20 users recommended under the various experiments were users that the distinguished users had decided to follow independently 3 years after the data used was collected. And of the remaining users in the list, the user study revealed that most of them were interesting, with as many as 80% of the recommended users being of interest. While results close to these could have been achieved with existing methods alone, these good results worked well within the Co-HITS algorithm to provide excellent rankings of tweets.

The more difficult task of recommending individual pieces of content was also very successful. Using the coarse method of cosine similarity to compare the tweets recommended from the time when the data was collected to tweets from today which the user has shown interest in suggests that the tweet recommendations have at least some value over random selections. As a human evaluator it is also clear that the recommendations are good simply by examining what the distinguished users are interested in and the style and content of the tweets recommended to them.

The user study revealed that of the top 20 tweets recommended, between 20% and 60% of them were interesting, at least for the two users studied. Both users indicated that the recommendations were excellent, with the obvious caveat that the recommendations are based on their interests of three years ago.

Because of the ad hoc nature of many of the datasets used by various studies in this area it is very difficult to compare the results obtained here to the results of other studies. Still, the results compare favourably with the results that one would expect from using such a recommendation system. Retrieving 40% relevant content within the top 20 results is quite comparable to the results that are expected when using a search engine, for example. The original paper describing the Co-HITS algorithm ([7]) had a precision at rank 5 of between 0.35 and 0.39 and precision at 10 of between 0.31 and 0.35 for a web search application, for example.

Additionally, most existing research fails to recommend particular tweets, or if they do it is done simply by selecting tweets from amongst the recommended users. This project not only provided very good tweet recommendations, but it also included many tweets from users other than those in the list of recommended users.

It is also possible to subjectively compare the recommendations generated here to those produced by Twitter and sent to users, an effort which began around the time that this project was begun. The recommended users that Twitter provides have always been generated using a technique similar to that used for the initial user scores here, so while the results here were slightly better than those of Twitter, the advantage was not tremendous.

But Twitter recently began sending users personalized recommendations of particular tweets. Simple examination of these recommendations reveals them to be inferior to the results generated here because they frequently are authored by users who are already followed by the user receiving the recommendation and also seem to be based largely on the number of users who have retweeted the tweet, which means that tweets by users with fewer followers will necessarily not be included.

By all measures available, the algorithm and method described here are very good at recommending content on Twitter. Both the background information on social networks from Chapter 2 and the very general nature of the Co-HITS algorithm described in Chapter 3 suggest that the implementation described in Chapter 4 could be expanded beyond just Twitter and instead be applied to many other social networking services. This is a very important piece of future work that should be done in this area, but it is far from the only one.

6.2 Future Work

6.2.1 Experimental Variations

There are a large number of possible experiments on how best to implement the Co-HITS algorithm for recommending content on Twitter which could not be included in this project due to the time constraints of running them.

The most important of these would be to run the algorithm for a much greater number of distinguished users with a much greater number of repetitions of each of the experiments that have already been done. Many of the experiments described in Chapter 5 were run for only a single user, and that small sample size impacts the reliability of the results. Another important experiment that should be expanded is the experimentation on the weighting of the various edge types. Some work was done on this, as described in Chapter 5, but a much more robust set of experiments would be ideal. The best possible way to do this would be if a (large) set of ground-truth relevance judgements were to emerge for some particular dataset. This would then allow the proper weights to be learned much more accurately using a machine learning algorithm.

Other content edges may also be useful to add. In particular, a URL edge which connects tweets containing the same URL—in a shortened URL form using a service such as bit.ly, most likely—to the authors of those tweets would probably useful. It wouldn't be feasible to expand all of the shortened URLs and to compare them that way, but since a given URL will always map to a particular shortened URL (at least for a period of several years) it is not necessary to expand them. Since URLs are one of the more interesting pieces of content to share by virtue of the greater amount of information they convey, this might be a very valuable edge type.

For calculating the initial user scores, the score of the distinguished user is determined by taking the maximum score from amongst the other users and multiplying that maximum score by 1.5. But the value of 1.5 which was used here is based purely on intuition. It would be good going forward to determine whether this is the correct value or whether it should be lower or higher to produce better results.

6.2.2 Improvements

Another important piece of future work would be to improve the initial tweet scores and more generally to improve the integration of information based on tweet content and style into the algorithm. The results presented in Chapter 5 demonstrated that completely removing the content edges had very little impact on which tweets and users had the highest scores, suggesting that there is plenty of room for improvement on the way in which content is included. This might take the form of Latent Dirichlet Allocation to determine the similarity of the tweets for the initial scores, as was done in several existing research projects, for example.

Content could also be integrated more effectively into the content-based edges. For example, it might be possible to incorporate sentiment analysis into the content edges by only creating links between tweets which discussed the content with a similar sentiment. Alternatively, it might be possible to determine the topics of some of the tweets and to connect tweets based on more concrete measures of their topics to avoid creating the large number of content edges which diluted their effectiveness in the experiments done here.

Yet another option might be to perform clustering on users, tweets, or both, and to use this to make the problem more tractable. Content could be recommended only within a particular cluster to other users within that cluster or links could be made between clusters in order to find groups of users or tweets to recommend.

6.2.3 Other Social Networks

Finally, and perhaps most importantly, it would be valuable to expand the experimentation done here with the Twitter network to some other social network. Given that the data of most other social networks is generally not available to the public this will be a difficult task, but the algorithm used here is general enough that if the data were available then it would be possible to use on any network.

Some things would need to be adapted, of course. For example with Facebook the retweet edges would change to reshare edges and be less prevalent while hashtags would have to be dropped in favour of some other method of linking content since hashtags are not widely used on Facebook. The promising results shown here suggest that adapting to other networks would provide good results.

Research into content recommendation on social networks is still very much in its early stages, so many of the possibilities have not yet been explored. Recommending individual pieces of interesting content, in particular, has seen very little study so far and the results here do an excellent job of recommending interesting tweets and demonstrating that an approach based primarily on the structure of the social graph can be effective at recommending individual pieces of content. Continuing with this further research would do much to build on the very promising results shown in this dissertation.

Appendix A

Source Code

Listing A.1: Recommender.java

```java
package com.wyeknot.serendiptwitty;

import java.awt.EventQueue;
import java.util.HashSet;

import edu.stanford.nlp.ie.AbstractSequenceClassifier;
import edu.stanford.nlp.ling.CoreLabel;


public class Recommender {

  private LuceneIndexManager indexMgr;
  private DatabaseInterface database;
  private GraphManager graph;

  public static final String DEFAULT_TWEET_DATA_PATH = "/Users/nathan/Documents/MSc
      Project/Twitter Data/Stanford/";
  public static final String DEFAULT_LUCENE_INDEX_PATH = DEFAULT_TWEET_DATA_PATH + "
      lucene_index/";
  public static final String DEFAULT_NER_CLASSIFIERS_PATH = "/Users/nathan/Documents/MSc
      Project/classifiers/";

  public static String tweetDataPath = DEFAULT_TWEET_DATA_PATH;
  public static String luceneIndexPath = DEFAULT_LUCENE_INDEX_PATH;
  public static String nerClassifiersPath = DEFAULT_NER_CLASSIFIERS_PATH;

  AbstractSequenceClassifier<CoreLabel> classifier;

  public static final int NUM_TWEETS_TO_INDEX = 500000;

  public static void main(String[] args) {

    if (args.length > 0) {
      Recommender.tweetDataPath = args[0];

      if (args.length > 1) {
        Recommender.luceneIndexPath = args[1];

        if (args.length > 2) {
          Recommender.nerClassifiersPath = args[2];
        }
      }
    }

    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          Recommender recommender = new Recommender();
          recommender.recommend();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }

  public void recommend() {
    HashSet<String> otherDistinguishedUsers = new HashSet<String>(2);
    graph = new GraphManager(database, indexMgr, "____", otherDistinguishedUsers);
    graph.createGraph();
    graph.runAlgorithm();
  }

  public Recommender() {
    try {
      database = new DatabaseInterface();

      indexMgr = new LuceneIndexManager(Recommender.luceneIndexPath);
      if (!indexMgr.indexExists()) {
        indexMgr.indexTweets(Recommender.tweetDataPath);
        indexMgr.closeIndexForWriting();
      }
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(-1);
    }
  }
}
```

Listing A.2: GraphManager.java

```java
package com.wyeknot.serendiptwitty;


import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.Field.TermVector;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.TermFreqVector;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.RAMDirectory;

import com.wyeknot.serendiptwitty.LuceneIndexManager.DocVector;

import edu.stanford.nlp.ie.AbstractSequenceClassifier;
import edu.stanford.nlp.ie.crf.CRFClassifier;
import edu.stanford.nlp.ling.CoreAnnotations.AnswerAnnotation;
import edu.stanford.nlp.ling.CoreLabel;


public class GraphManager {

  public static final double DEFAULT_ORIGINAL_SCORE = 0;

  private LuceneIndexManager indexMgr;
  private DatabaseInterface database;

  private String distinguishedUser;
  private Set<String> otherDistinguishedUsers;

  //Protects us from adding a user into the tweets twice
  private HashSet<String> usersInTweets;

  private HashSet<String> userBatch;
  private HashSet<Tweet> tweetBatch;
  private HashSet<Edge> edgeBatch;

  private static final int MAX_ITERATIONS = 10;

  //Values closer to 0 put more weight on the original score
  private static final double lambdaUsers = 0.7;
  private static final double lambdaTweets = 0.9;

  private static final double MAX_USER_SCORE_MULTIPLIER = 1.5;


  GraphManager(DatabaseInterface database, LuceneIndexManager index, String
      distinguishedUser, Set<String> otherDistinguishedUsers) {
    this.database = database;
    this.indexMgr = index;
    this.distinguishedUser = distinguishedUser.toLowerCase();
    this.otherDistinguishedUsers = otherDistinguishedUsers;

    usersInTweets = new HashSet<String>();

    userBatch = new HashSet<String>();
    tweetBatch = new HashSet<Tweet>();
    edgeBatch = new HashSet<Edge>();
  }


  /*
   *
   * Code for creating the graph, including edges and vertices
   *
   */

  public void createGraph() {
    if (!database.tableHasRows("edges")) {

      usersInTweets.add(distinguishedUser);
      userBatch.add(distinguishedUser);

      for (String user : otherDistinguishedUsers) {
        user = user.toLowerCase();
        userBatch.add(user);
        usersInTweets.add(user);
      }

      tweetsParser.indexTweets(Recommender.tweetDataPath, Recommender.NUM_TWEETS_TO_INDEX)
          ;

      if (userBatch.size() > 0) {
        database.addUserBatch(userBatch);
        userBatch.clear();
      }

      if (tweetBatch.size() > 0) {
        database.addTweetBatch(tweetBatch);
        tweetBatch.clear();
      }

      database.clusterUsers();
      database.clusterTweetsByAuthor();

      //Now create the remainder of the edges
      createFollowerRetweetAndMentionEdges();
      createContentEdges();

      if (edgeBatch.size() > 0) {
        database.addEdgeBatch(edgeBatch);
        edgeBatch.clear();
      }


      database.clusterEdges();
      database.clusterTweetsById();

      database.createAndClusterEdgesTweetIds();
      database.analyzeTables();
    }
  }

  private void createFollowerRetweetAndMentionEdges() {

    database.acquireCursorForTweetVertices();

    String tweet = null;
    List<Long> curUserTweetIds = new ArrayList<Long>();
    String lastUser = null;

    while (null != (tweet = database.getNextTweetFromCursor())) {

      String tweeter = database.getNameFromCurrentCursorPos();
      long tweetId = database.getTweetIdFromCurrentCursorPos();

      if (!tweeter.equals(lastUser) && (lastUser != null)) {
        createFollowerEdgesForUser(lastUser,curUserTweetIds);
        curUserTweetIds.clear();
      }

      lastUser = tweeter;
      curUserTweetIds.add(Long.valueOf(tweetId));

      Set<String> retweets = Tweet.findRetweetedUsers(tweet);
      String atReply = Tweet.findAtReply(tweet);
      Set<String> mentions = Tweet.findMentionedUsers(tweet, retweets);

      createRetweetEdgesForTweet(tweeter, tweetId, retweets);

      createMentionEdgesForTweet(tweeter, tweetId, mentions);

      if (null != atReply) {
        database.acquireInternalCursorForTweets(atReply);
        long internalTweetId = -1;
        while (-1 != (internalTweetId = database.getNextTweetIdFromInternalCursor())) {
          addEdge(tweeter, internalTweetId, Edge.Types.EDGE_TYPE_AT_REPLY_CONTENT, false);
        }
      }
    }
  }

  private void createFollowerEdgesForUser(String user, List<Long> curUserTweetIds) {
    database.acquireInternalCursorForFollowersInUserVertices(user);

    String internalUser = null;
    while (null != (internalUser = database.getNextNameFromInternalCursor())) {
      for (Long id : curUserTweetIds) {
        addEdge(internalUser, id.longValue(), Edge.Types.EDGE_TYPE_FOLLOWER, false);
      }
```

```java
    }
  }

  private void createRetweetEdgesForTweet(String tweeter, long tweetId, Set<String> retweets) {
    for (String retweetee : retweets) {
      //Connects the tweets of the followees of the person being retweeted to the retweeter
      database.acquireInternalCursorForFolloweesEdges(retweetee, tweeter,
          Edge.Types.EDGE_TYPE_AUTHORSHIP.id(), Edge.Types.EDGE_TYPE_RETWEET_FOLLOWEES.id());

      long internalTweetId = -1;
      while (-1 != (internalTweetId = database.getNextTweetIdFromInternalCursor())) {
        addEdge(tweeter, internalTweetId, Edge.Types.EDGE_TYPE_RETWEET_FOLLOWEES, false);
      }

      //Connects the tweets of the person being retweeted to the followers of the retweeter
      database.acquireInternalCursorForFollowersEdges(retweetee, tweeter,
          Edge.Types.EDGE_TYPE_AUTHORSHIP.id(), Edge.Types.EDGE_TYPE_RETWEET_FOLLOWERS.id());
      internalTweetId = -1;

      while (-1 != (internalTweetId = database.getNextTweetIdFromInternalCursor())) {
        String internalUserName = database.getNameFromCurrentInternalCursorPos();
        addEdge(internalUserName, internalTweetId, Edge.Types.EDGE_TYPE_RETWEET_FOLLOWERS, false);
      }
    }
  }

  private void createMentionEdgesForTweet(String tweeter, long tweetId, Set<String> mentions) {
    for (String mentionee : mentions) {
      //Connects the tweets of the followees of the person being mentioned to the mentioner
      database.acquireInternalCursorForFolloweesEdges(mentionee, tweeter,
          Edge.Types.EDGE_TYPE_AUTHORSHIP.id(), Edge.Types.EDGE_TYPE_MENTION_FOLLOWEES.id());


      long internalTweetId = -1;
      while (-1 != (internalTweetId = database.getNextTweetIdFromInternalCursor())) {
        addEdge(tweeter, internalTweetId, Edge.Types.EDGE_TYPE_MENTION_FOLLOWEES, false);
      }

      //Connects the tweets of the person being mentioned to the followers of the mentioner
      database.acquireInternalCursorForFollowersEdges(mentionee, tweeter,
          Edge.Types.EDGE_TYPE_AUTHORSHIP.id(), Edge.Types.EDGE_TYPE_MENTION_FOLLOWERS.id());
      internalTweetId = -1;

      while (-1 != (internalTweetId = database.getNextTweetIdFromInternalCursor())) {
        String internalUserName = database.getNameFromCurrentInternalCursorPos();
        addEdge(internalUserName, internalTweetId, Edge.Types.EDGE_TYPE_MENTION_FOLLOWERS, false);
      }
    }
  }


  private void createContentEdges() {
    String serializedClassifier = Recommender.nerClassifiersPath + "english.all.3class.distsim.crf.ser.gz";

    @SuppressWarnings("unchecked")
    AbstractSequenceClassifier<CoreLabel> classifier = CRFClassifier.
        getClassifierNoExceptions(serializedClassifier);

    /*
     * These HashMaps have the entity/hashtag as the key, and then for
     * each hashtag there is a List of tweetId/author pairs and a set
     * of authors who have used this entity/hashtag.
     *
     * We need to keep track of the tweet id AND the author of each
     * tweet because we don't want to create a link between the author's
     * own tweet and their vertex.
     *
     * Note: If we have a set of tweetids instead of a list then we can
     * prevent duplicate edges, but I think that the duplicate edges
     * add extra value, so I don't plan to implement it that way.
     */
    HashMap<String , Pair< List<Pair<Long,String>> , Set<String> > > entities =
        new HashMap<String,Pair<List<Pair<Long,String>>,Set<String>>>();
    HashMap<String , Pair< List<Pair<Long,String>> , Set<String> > > hashtags =
        new HashMap<String,Pair<List<Pair<Long,String>>,Set<String>>>();

    database.acquireCursorForTweetVertices();

    String tweet = null;

    while (null != (tweet = database.getNextTweetFromCursor())) {
      String tweeter = database.getNameFromCurrentCursorPos();
      Long tweetId = Long.valueOf(database.getTweetIdFromCurrentCursorPos());

      List<List<CoreLabel>> out = classifier.classify(tweet);
      for (List<CoreLabel> sentence : out) {
        for (CoreLabel word : sentence) {
          String type = word.get(AnswerAnnotation.class);

          if (!type.equals("O")) {
            String key = word.word().toLowerCase();

            if (!Character.isLetter(key.charAt(0)) || key.equals("rt")) {
              continue;
            }

            if (!entities.containsKey(key)) {
              ArrayList<Pair<Long,String>> tweets = new ArrayList<Pair<Long,String>>();
              HashSet<String> users = new HashSet<String>();

              tweets.add(new Pair<Long,String>(tweetId,tweeter));
              users.add(tweeter);

              Pair<List<Pair<Long,String>>,Set<String>> value =
                  new Pair<List<Pair<Long,String>>,Set<String>>(tweets, users);

              entities.put(key, value);
            }
            else {
              Pair<List<Pair<Long,String>>,Set<String>> value = entities.get(key);
              value.getFirst().add(new Pair<Long,String>(tweetId,tweeter));
              value.getSecond().add(tweeter);
            }
          }

          /* Ignore something if it is not a proper word AND
           * it isn't BOTH preceded and followed by another entity
           */

          /* Can see if this words .beginPosition is one above the
           * last word's .endPosition to ignore spaces.
           */
        }
      }

      Set<String> hashtagsInTweet = Tweet.findHashTags(tweet);
      for (String tag : hashtagsInTweet) {
        String key = tag.toLowerCase();

        if (!hashtags.containsKey(tag)) {
          ArrayList<Pair<Long,String>> tweets = new ArrayList<Pair<Long,String>>();
          HashSet<String> users = new HashSet<String>();

          tweets.add(new Pair<Long,String>(tweetId,tweeter));
          users.add(tweeter);

          Pair<List<Pair<Long,String>>,Set<String>> value = new Pair<List<Pair<Long,String
              >>,Set<String>>(tweets, users);

          hashtags.put(key, value);
        }
        else {
          Pair<List<Pair<Long,String>>,Set<String>> value = hashtags.get(key);
          value.getFirst().add(new Pair<Long,String>(tweetId,tweeter));
          value.getSecond().add(tweeter);
        }
      }
    }

    //Evaluate the entities

    for (Pair<List<Pair<Long,String>>,Set<String>> edges : entities.values()) {
      List<Pair<Long,String>> tweetIds = edges.getFirst();
      Set<String> authors = edges.getSecond();


      for (Pair<Long,String> tweetIdAndAuthor : tweetIds) {
        for (String author : authors) {
          if (!author.equals(tweetIdAndAuthor.getSecond())) {
            addEdge(author, tweetIdAndAuthor.getFirst(), Edge.Types.EDGE_TYPE_CONTENT,
                false);
          }
        }
      }
    }


    //Evaluate the hashtags

    for (Pair<List<Pair<Long,String>>,Set<String>> edges : hashtags.values()) {
      List<Pair<Long,String>> tweetIds = edges.getFirst();
      Set<String> authors = edges.getSecond();

      for (Pair<Long,String> tweetIdAndAuthor : tweetIds) {
        for (String author : authors) {
          if (!author.equals(tweetIdAndAuthor.getSecond())) {
            addEdge(author, tweetIdAndAuthor.getFirst(), Edge.Types.EDGE_TYPE_HASHTAG,
                false);
          }
        }
      }
    }
  }


  Set<String> createBasicRetweetEdges(String tweet, long curTweetId) {
    Set<String> retweets = Tweet.findRetweetedUsers(tweet);
    for (String s : retweets) {
      addEdge(s, curTweetId, Edge.Types.EDGE_TYPE_RETWEET, true);
    }

    return retweets;
  }

  void createBasicAtReplyEdge(String tweet, long curTweetId) {
    String atReply = Tweet.findAtReply(tweet);
    if (null != atReply) {
      addEdge(atReply, curTweetId, Edge.Types.EDGE_TYPE_AT_REPLY, true);
    }
  }

  void createBasicMentionEdges(String tweet, long curTweetId, Set<String> retweetNames) {
    Set<String> mentions = Tweet.findMentionedUsers(tweet, retweetNames);
    for (String s : mentions) {
      addEdge(s, curTweetId, Edge.Types.EDGE_TYPE_MENTION, true);
    }
  }


  /*
   *
   * Code for updating the scores
   *
   */

  private void updateTweetsFromUser(String userName, double userScore, Set<Pair<Integer,
      Long>> edgeTypeAndDest,
      Map<Long,Double> updatedTweetScores, double totalEdgeWeight) {

    for (Pair<Integer,Long> t : edgeTypeAndDest) {
      double chanceOfGoingToTweet = (Edge.idToEdgeType[t.getFirst()].probability() /
          totalEdgeWeight);
      double scoreEffectFromThisEdge = chanceOfGoingToTweet * userScore * lambdaTweets;

      Long tweetId = t.getSecond();

      if (!updatedTweetScores.containsKey(tweetId)) {
        updatedTweetScores.put(tweetId, Double.valueOf(scoreEffectFromThisEdge));
      }
      else {
        Double currentScore = updatedTweetScores.get(tweetId);
        double newScore = scoreEffectFromThisEdge + currentScore.doubleValue();
        updatedTweetScores.put(tweetId, Double.valueOf(newScore));
      }
    }
  }

  private void updateUsersFromTweet(long tweetId, double tweetScore, Set<Pair<Integer,
      String>> edgeTypeAndDest,
      Map<String,Double> updatedUserScores, double totalEdgeWeight) {

    for (Pair<Integer,String> t : edgeTypeAndDest) {
      double chanceOfGoingToUser = Edge.idToEdgeType[t.getFirst()].probability() /
          totalEdgeWeight;
      double scoreEffectFromThisEdge = chanceOfGoingToUser * tweetScore * lambdaUsers;

      String userName = t.getSecond();

      if (!updatedUserScores.containsKey(userName)) {
        updatedUserScores.put(userName, Double.valueOf(scoreEffectFromThisEdge));
      }
      else {
        Double currentScore = updatedUserScores.get(userName);
        double newScore = scoreEffectFromThisEdge + currentScore.doubleValue();
        updatedUserScores.put(userName, Double.valueOf(newScore));
      }
    }
  }

  private void updateTweetScores() {
    database.acquireCursorForUpdatingTweetScores();

    long tweetId = -1;

    String userName = null;
    String lastUserName = null;
    double lastUserScore = -1;
    double lastUserTotalEdgeWeight = 0;

    double scoreTotalFromVerticesWithNoExit = 0;

    //Holds a record of the scores of all the tweets that we've updated the score for
    Map<Long,Double> updatedTweetScores = new HashMap<Long,Double>();

    Set<Pair<Integer,Long>> currentUserTypesAndDestinations = new HashSet<Pair<Integer,
        Long>>();

    while (null != (userName = database.getNextNameFromCursor())) {
      if (!userName.equals(lastUserName) && (lastUserName != null)) {
        if (currentUserTypesAndDestinations.isEmpty()) {
          scoreTotalFromVerticesWithNoExit += lastUserScore;
        }
        else {
          updateTweetsFromUser(lastUserName, lastUserScore,
              currentUserTypesAndDestinations,
              updatedTweetScores,lastUserTotalEdgeWeight);
          currentUserTypesAndDestinations.clear();
          lastUserTotalEdgeWeight = 0;
        }
      }

      tweetId = database.getTweetIdFromCurrentCursorPos();
      if (-1 == tweetId) {
        throw new RuntimeException("Error retrieving tweetId while calculating tweet
            scores!");
      }

      Edge.Types type = Edge.idToEdgeType[database.getEdgeTypeFromCurrentCursorPos()];
      if (type.userToTweetDir()) {
        if (currentUserTypesAndDestinations.add(new Pair<Integer,Long>(
            Integer.valueOf(type.id()),
            Long.valueOf(tweetId)))) {
          lastUserTotalEdgeWeight += type.probability();
        }
      }

      lastUserName = userName;
      lastUserScore = database.getUserScoreFromCurrentCursorPos();
    }

    //And update the last tweet
    if (currentUserTypesAndDestinations.isEmpty()) {
      scoreTotalFromVerticesWithNoExit += lastUserScore;
    }
    else {
      updateTweetsFromUser(lastUserName, lastUserScore, currentUserTypesAndDestinations,
          updatedTweetScores,
          lastUserTotalEdgeWeight);
    }

    database.updateBaseTweetScores(lambdaTweets, scoreTotalFromVerticesWithNoExit);
    database.updateTweetScores(updatedTweetScores);
  }

  private void updateUserScores() {
    database.acquireCursorForUpdatingUserScores();

    String userName = null;

    long tweetId = -1;
    long lastTweetId = -1;
    double lastTweetScore = -1;
    double lastUserTotalEdgeWeight = 0;

    double scoreTotalFromVerticesWithNoExit = 0;

    //Holds a record of the scores of all the users that we've updated the score for
    Map<String,Double> updatedUserScores = new HashMap<String,Double>();

    Set<Pair<Integer,String>> currentTweetTypesAndDestinations = new HashSet<Pair<Integer,
        String>>();

    while (-1 != (tweetId = database.getNextTweetIdFromCursor())) {

      if ((lastTweetId != tweetId) && (lastTweetId != -1)) {
        if (currentTweetTypesAndDestinations.isEmpty()) {
          scoreTotalFromVerticesWithNoExit += lastTweetScore;
        }
        else {
          updateUsersFromTweet(lastTweetId, lastTweetScore,
              currentTweetTypesAndDestinations,
              updatedUserScores, lastUserTotalEdgeWeight);
          currentTweetTypesAndDestinations.clear();
          lastUserTotalEdgeWeight = 0;
        }
      }

      userName = database.getNameFromCurrentCursorPos();
      if (null == userName) {
        throw new RuntimeException("Error retrieving userId while calculating user scores!
            ");
      }

      Edge.Types type = Edge.idToEdgeType[database.getEdgeTypeFromCurrentCursorPos()];

      if (type.tweetToUserDir()) {
        if (currentTweetTypesAndDestinations.add(new Pair<Integer,String>(
            Integer.valueOf(type.id()),
            userName))) {
          lastUserTotalEdgeWeight += type.probability();
        }
      }

      lastTweetId = tweetId;
      lastTweetScore = database.getTweetScoreFromCurrentCursorPos();
    }

    //And update the last user
    if (currentTweetTypesAndDestinations.isEmpty()) {
      scoreTotalFromVerticesWithNoExit += lastTweetScore;
    }
    else {
      updateUsersFromTweet(lastTweetId, lastTweetScore, currentTweetTypesAndDestinations,
          updatedUserScores, lastUserTotalEdgeWeight);
    }

    database.updateBaseUserScores(lambdaUsers, scoreTotalFromVerticesWithNoExit);
    database.updateUserScores(updatedUserScores);
  }


  private void initializeUserScores() {

    database.acquireCursorForInitializingUserScores();

```

```java
    String userName = null;

    /* This is a variation on the Adamic/Adair method of similarity
     * as described in Liben-Nowell, 2007. Followees of
     * distinguished_user (gamma(x)) intersection with followers of
     * current_user (gamma(y))
     *
     * gamma(z) = followers user z
     */

    double maxScore = 0;

    HashMap<String,Double> scoreBatch = new HashMap<String,Double>(DatabaseInterface.
        MAX_DATABASE_BATCH_SIZE);

    while (null != (userName = database.getNextNameFromCursor())) {
      if (userName.equals(distinguishedUser)) {
        continue;
      }

      //Gets the follower counts of the overlap between the distinguished user's followees
           and this user's followers
      List<Integer> overlap = database.getFollowerCountsOfOverlappingUserSet(
          distinguishedUser, userName);
      if (overlap == null) {
        scoreBatch.put(userName, Double.valueOf(0));
        if (scoreBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
          database.setOriginalUserScoreBatch(scoreBatch);
          scoreBatch.clear();
        }
        continue;
      }

      double score = 0;

      for (Integer i : overlap) {
        score += 1 / Math.log10(i.intValue());
      }

      scoreBatch.put(userName, Double.valueOf(score));
      if (scoreBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
        database.setOriginalUserScoreBatch(scoreBatch);
        scoreBatch.clear();
      }

      if (score > maxScore) {
        maxScore = score;
      }
    }

    scoreBatch.put(distinguishedUser, Double.valueOf(maxScore * MAX_USER_SCORE_MULTIPLIER)
        );
    database.setOriginalUserScoreBatch(scoreBatch);

    //The values will be normalized elsewhere
  }

  /*
   *
   * Code for initializing tweet and user scores
   *
   */


  private void initializeTweetScores() {

    /* If the distinguished user has retweeted anything, then those people are most
     * indicative of content that he likes, so we combine their tweets as a reference
     * document against which all tweets have their similarity compared. If not, then
     * we'll just use the tweets of everyone that the distinguished user follows.
     */

    database.acquireCursorForTweets(distinguishedUser);

    String combinedTweet = "";
    String tweet = null;

    while (null != (tweet = database.getNextTweetFromCursor())) {
      combinedTweet += " " + tweet;
    }

    Set<String> retweetedUsers = Tweet.findRetweetedUsers(combinedTweet);

    if (retweetedUsers.size() > 5) {
      //Our combined document for comparison will be all of these people's tweets, if
          there are enough of them
      database.acquireCursorForTweetsOfUsers(retweetedUsers);
    }
    else {
      database.acquireCursorForAllFollowerEdgeTweets(distinguishedUser, Edge.Types.
          EDGE_TYPE_FOLLOWER.id());
    }

    combinedTweet = "";
    tweet = null;
    while (null != (tweet = database.getNextTweetFromCursor())) {
      combinedTweet += " " + tweet;
    }

    if (combinedTweet.equals("")) {
      database.acquireCursorForTweets(distinguishedUser);
      while (null != (tweet = database.getNextTweetFromCursor())) {
        combinedTweet += " " + tweet;
      }
    }

    combinedTweet = combinedTweet.replaceAll("RT", "");
    combinedTweet = combinedTweet.replaceAll("rt", "");

    Document referenceDoc = new Document();
    referenceDoc.add(new Field("tweet", combinedTweet, Field.Store.YES, Field.Index.
        ANALYZED, TermVector.YES));

    //This is used only to get the term frequencies of the reference document
    Directory referenceIndex = new RAMDirectory();

    HashMap<Long,Double> scoreBatch = new HashMap<Long,Double>(DatabaseInterface.
        MAX_DATABASE_BATCH_SIZE);

    database.setTweetScoresAndOriginalScoresToZero();

    try {
      /* To ensure that stemming and stop words are identical to
       * the main lucene index, we index the reference document in
       * the same way, but in RAM and with only one document.
       *
       * This is used purely to get the term frequencies using Lucene.
       */
      IndexWriter writer = LuceneIndexManager.getIndexWriter(referenceIndex);
      writer.addDocument(referenceDoc);
      writer.close();

      indexMgr.createDocumentFrequency();

      String[] terms;
      int[] termFreqs;

      IndexReader referenceIndexReader = IndexReader.open(referenceIndex);
      TermFreqVector tf = referenceIndexReader.getTermFreqVector(referenceIndexReader.
          maxDoc() - 1, "tweet");

      terms = tf.getTerms();
      termFreqs = tf.getTermFrequencies();

      /* Note that we use the main index's document frequencies, which are
       * valid for all terms of the reference document since the tweets
       * which make up the reference document are contained within the
       * main index.
       */
      DocVector referenceDocVector = new DocVector(terms,termFreqs,indexMgr.
          getDocumentFrequency(), database.getNumTweetVertices());

      for (int ii = 0 ; ii < indexMgr.reader.maxDoc(); ii++) {
        if (indexMgr.reader.isDeleted(ii)) {
          continue;
        }

        if (null == (tf = indexMgr.reader.getTermFreqVector(ii,"tweet"))) {
          continue;
        }

        terms = tf.getTerms();
        termFreqs = tf.getTermFrequencies();

        DocVector docVector = new DocVector(terms, termFreqs, indexMgr.
            getDocumentFrequency(), database.getNumTweetVertices());

        double similarity = docVector.cosineSimilarity(referenceDocVector);

        Document doc = indexMgr.reader.document(ii);
        Long tweetId = Long.valueOf(doc.get("tweetid"));

        scoreBatch.put(tweetId, Double.valueOf(similarity));
        if (scoreBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
          database.setOriginalTweetScoreBatch(scoreBatch);
          scoreBatch.clear();
        }
      }

      if (!scoreBatch.isEmpty()) {
        database.setOriginalTweetScoreBatch(scoreBatch);
        scoreBatch.clear();
      }

      referenceIndexReader.close();
    }
    catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Couldn't initialize tweet scores!");
    }
  }

  /*
   *
   * Code for running the actual algorithm
   *
   */


  public void runAlgorithm() {
    System.out.println("Running the Co-HITS algorithm!");

    if (!database.originalUserScoreCalculated()) {
      initializeUserScores();
      database.normalizeUserScores();
    }

    if (!database.originalTweetScoreCalculated()) {
      initializeTweetScores();
      database.normalizeTweetScores();
    }

    //Sets score to original_score for doing multiple runs in a row
    database.resetScores();

    int iterations = 0;

    Date d1 = new Date();
    Date d2 = new Date();

    do {
      System.out.println("Iteration #" + (iterations + 1));

      updateTweetScores();
      updateUserScores();
    } while (++iterations < MAX_ITERATIONS);//*/

    d2 = new Date();

    System.out.println("Finished running Co-HITS after " + (iterations + 1) + " iterations
        ");
    System.out.println("Started at " + d1 + " and finished at " + d2);
    System.out.println("Roughly " + ((float)(d2.getTime() - d1.getTime()) / 1000.0) + "
        seconds");
    System.out.println("LambdaTweets was " + lambdaTweets + " and lambdaUsers was " +
        lambdaUsers);
  }

  /*
   *
   * Code for adding edges
   *
   */

  private void addEdge(String userName, long tweetId, Edge.Types type, boolean
      createUserIfNeeded) {
    if (userName == null) {
      return;
    }

    if (createUserIfNeeded && !usersInTweets.contains(userName)) {
      usersInTweets.add(userName);

      userBatch.add(userName);
      if (userBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
        database.addUserBatch(userBatch);
        userBatch.clear();
      }
    }

    edgeBatch.add(new Edge(userName, tweetId, type.id()));
    if (edgeBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
      database.addEdgeBatch(edgeBatch);
      edgeBatch.clear();
    }
  }


  /*
   *
   * Implementation for the tweets parser which is used to create
   * the edges.
   *
   */

  private StanfordTweetIndexer.StanfordParser tweetsParser = new StanfordTweetIndexer.
      StanfordParser() {

    @Override
    boolean tweetHandler(Tweet tweet) {

      if ((tweet.user.length() >= 32) || (tweet.tweet.length() >= 255)) {
        return false;
      }

      //addEdge creates the user if needed
      addEdge(tweet.getUser(),tweet.id, Edge.Types.EDGE_TYPE_AUTHORSHIP, true);

      Set<String> retweets = createBasicRetweetEdges(tweet.getTweet(), tweet.id);
      createBasicAtReplyEdge(tweet.getTweet(), tweet.id);
      createBasicMentionEdges(tweet.getTweet(), tweet.id, retweets);

      tweetBatch.add(tweet);
      if (tweetBatch.size() > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
        database.addTweetBatch(tweetBatch);
        tweetBatch.clear();
      }

      return true;
    }
  };
}
```

Listing A.3: DatabaseInterface.java

```
package com.wyeknot.serendiptwitty;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;


//Not thread safe!!
public class DatabaseInterface {

  private Connection dbConnection;
  private ResultSet curResults = null;
  private PreparedStatement curStatement = null;

  private ResultSet curInternalResults = null;
  private PreparedStatement curInternalStatement = null;

  public static final int MAX_DATABASE_BATCH_SIZE = 50000;

  private int numTweetVertices = 0;
  private int numUserVertices = 0;

  public DatabaseInterface() throws SQLException {
    String connectionURL = "jdbc:postgresql://localhost:5432/tweet";
    dbConnection = DriverManager.getConnection(connectionURL,"twitter","tweets357");
  }

  public void clusterEdges() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("CLUSTER idx_edge_name ON edges
          ;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void clusterUsers() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("CLUSTER user_vertices_pkey ON
          user_vertices;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void clusterTweetsByAuthor() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("CLUSTER idx_tweet_author ON
          tweet_vertices;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void clusterTweetsById() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("CLUSTER tweet_vertices_pkey ON
           tweet_vertices;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void createAndClusterEdgesTweetIds() {
    try {
      dbConnection.setAutoCommit(false);

      PreparedStatement st = dbConnection.prepareStatement("CREATE TABLE edges_tweetids AS
           (SELECT * FROM EDGES);");
      st.execute();
      st.close();

      dbConnection.commit();
      dbConnection.setAutoCommit(true);

      st = dbConnection.prepareStatement("CREATE INDEX idx_edges_tweetids_tweetids ON
          edges_tweetids USING btree (tweetid);");
      st.execute();
      st.close();

      st = dbConnection.prepareStatement("CREATE INDEX idx_edges_tweetids_names ON
          edges_tweetids USING btree (name);");
      st.execute();
      st.close();

      st = dbConnection.prepareStatement("CREATE INDEX idx_edges_tweetids_types ON
          edges_tweetids USING btree (type);");
      st.execute();
      st.close();

      st = dbConnection.prepareStatement("CLUSTER idx_edges_tweetids_tweetids ON
          edges_tweetids;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }


  public void analyzeTables() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("ANALYZE user_vertices;");
      st.execute();
      st.close();
      st = dbConnection.prepareStatement("ANALYZE tweet_vertices;");
      st.execute();
      st.close();
      st = dbConnection.prepareStatement("ANALYZE edges;");
      st.execute();
      st.close();
      st = dbConnection.prepareStatement("ANALYZE edges_tweetids;");
      st.execute();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  private int countTweetVertices() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("SELECT COUNT(*) FROM
          tweet_vertices;");

      ResultSet result = st.executeQuery();
      result.next();
      int numRows = result.getInt(1);

      st.close();

      return numRows;
    } catch (SQLException e) {
      e.printStackTrace();
    }

    return 0;
  }

  private int countUserVertices() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("SELECT COUNT(*) FROM
          user_vertices;");

      ResultSet result = st.executeQuery();
      result.next();
      int numRows = result.getInt(1);

      st.close();

      return numRows;
    } catch (SQLException e) {
      e.printStackTrace();
    }

    return 0;
  }

  public int getNumTweetVertices() {
    if (numTweetVertices == 0) {
      numTweetVertices = countTweetVertices();
    }

    return numTweetVertices;
  }

  public int getNumUserVertices() {
    if (numUserVertices == 0) {
      numUserVertices = countUserVertices();
    }

    return numUserVertices;
  }

  public void acquireCursorForTweetsOfUsers(Set<String> users) {
    try {
      closeCursor();

      String nameParams = "";
      for (int ii = 0 ; ii < users.size() ; ii++) {
        if (ii == 0) {
          nameParams += " name=?";
        }
        else {
          nameParams += " OR name=?";
        }
      }

      curStatement = dbConnection.prepareStatement("SELECT * FROM tweet_vertices WHERE" +
          nameParams + ";",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(10000);

      int curPos = 1;
      for (String user : users) {
        curStatement.setString(curPos++, user);
      }

      curResults = curStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void acquireCursorForAllFollowerEdgeTweets(String user, int followerEdgeType) {
    try {
      closeCursor();
      curStatement = dbConnection.prepareStatement("SELECT T.* FROM edges E,
          tweet_vertices T WHERE E.name=? AND E.type=? AND E.tweetid=T.tweetid;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(10000);
      curStatement.setString(1, user);
      curStatement.setInt(2, followerEdgeType);

      curResults = curStatement.executeQuery();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void acquireCursorForTweets(String user) {
    try {
      closeCursor();
      curStatement = dbConnection.prepareStatement("SELECT * FROM tweet_vertices WHERE
          name=?;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(10000);
      curStatement.setString(1, user);
      curResults = curStatement.executeQuery();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void acquireInternalCursorForTweets(String author) {
    try {
      closeInternalCursor();
      curInternalStatement = dbConnection.prepareStatement("SELECT * FROM tweet_vertices
          WHERE name=?;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curInternalStatement.setFetchSize(10000);
      curInternalStatement.setString(1, author);
      curInternalResults = curInternalStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }


  public void acquireInternalCursorForFolloweesEdges(String retweetee, String retweeter,
      int edgeTypeAuthorship, int edgeTypeFollowees) {
    try {
      closeInternalCursor();
      curInternalStatement = dbConnection.prepareStatement("SELECT T.tweetid FROM
          namenetwork_followers F, tweet_vertices T WHERE T.name=F.username AND F.
          followername=?;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curInternalStatement.setFetchSize(10000);
      curInternalStatement.setString(1, retweetee);
      curInternalResults = curInternalStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void acquireInternalCursorForFollowersEdges(String retweetee, String retweeter,
      int edgeTypeAuthorship, int edgeTypeFollowers) {
    try {
      closeInternalCursor();

      curInternalStatement = dbConnection.prepareStatement("SELECT T.tweetid, U.name FROM
          (SELECT tweetid FROM tweet_vertices WHERE name=?) T, (SELECT U.name FROM
          user_vertices U, namenetwork N WHERE N.username=? AND N.followername=U.name) U;"
          ,
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curInternalStatement.setFetchSize(10000);
      curInternalStatement.setString(1, retweetee);
      curInternalStatement.setString(2, retweeter);
      curInternalResults = curInternalStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public long getNextTweetIdFromInternalCursor() {
    try {
      if ((curInternalResults == null) || curInternalResults.isClosed()) {
        return -1;
      }

      if (!curInternalResults.next()) {
        return -1;
      }

      return curInternalResults.getLong("tweetid");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  public String getNameFromCurrentInternalCursorPos() {
    try {
      if ((curInternalResults == null) || curInternalResults.isClosed()) {
        return null;
      }

      if (curInternalResults.isBeforeFirst()) {
        return null;
      }

      return curInternalResults.getString("name");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }

  public void acquireCursorForTweetVertices() {
    try {
      closeCursor();
      //This is ASC because of the way that it is used for creating content edges
      curStatement = dbConnection.prepareStatement("SELECT * FROM tweet_vertices ORDER BY
          name ASC;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(10000);
      curResults = curStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public String getNextTweetFromCursor() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return null;
      }

      if (!curResults.next()) {
        return null;
      }

      return curResults.getString("tweet");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
```

```java
    }
  }

  //The ORDER BY statement here is essential because of the way the algorithm is set up.
  public void acquireCursorForUpdatingTweetScores() {
    try {
      closeCursor();

      dbConnection.setAutoCommit(false);
      curStatement = dbConnection.prepareStatement("SELECT E.*, U.score as user_score FROM
           edges E, user_vertices U WHERE E.name=U.name ORDER BY E.name;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(100000);
      curResults = curStatement.executeQuery();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  //The ORDER BY statement here is essential because of the way the algorithm works.
  public void acquireCursorForUpdatingUserScores() {
    try {
      closeCursor();

      dbConnection.setAutoCommit(false);

      curStatement = dbConnection.prepareStatement("SELECT E.*, T.score as tweet_score
          FROM edges_tweetids E, tweet_vertices T WHERE E.tweetid=T.tweetid ORDER BY E.
          tweetid;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(100000);

      curResults = curStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void acquireCursorForInitializingUserScores() {
    try {
      closeCursor();
      curStatement = dbConnection.prepareStatement("SELECT * FROM user_vertices;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curStatement.setFetchSize(10000);
      curResults = curStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public List<Integer> getFollowerCountsOfOverlappingUserSet(String distinguishedUser,
      String otherUser) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("SELECT C.count FROM ((SELECT
          username AS name FROM namenetwork_followers WHERE followername=?) INTERSECT (
          SELECT followername AS name FROM namenetwork WHERE username=?)) S, followercount
           C WHERE S.name=C.username;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      st.setFetchSize(1000000);
      st.setString(1,distinguishedUser);
      st.setString(2,otherUser);

      ResultSet result = st.executeQuery();

      ArrayList<Integer> followerCounts = new ArrayList<Integer>();

      while (result.next()) {
        followerCounts.add(new Integer(result.getInt(1)));
      }

      st.close();

      return followerCounts;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }


  public void resetScores() {
    try {
      PreparedStatement st;

      st = dbConnection.prepareStatement("UPDATE user_vertices SET score=original_score;");
      st.executeUpdate();
      st = dbConnection.prepareStatement("UPDATE tweet_vertices SET score=original_score;");
      st.executeUpdate();

      st.close();
    } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Couldn't normalize user scores!");
    }
  }

  //Should only be done once, after the initial user scoring method is undertaken
  public void normalizeUserScores() {

    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE user_vertices SET original_score=(original_score / S.sum) FROM (SELECT SUM(original_score) FROM user_vertices) S;");
      st.executeUpdate();

      st.close();
    } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Couldn't normalize user scores!");
    }
  }


  public void normalizeTweetScores() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE tweet_vertices SET original_score=(original_score / S.sum) FROM (SELECT SUM(original_score) FROM tweet_vertices) S;");
      st.executeUpdate();

      st.close();
    } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Couldn't normalize tweet scores!");
    }
  }

  public boolean originalUserScoreCalculated() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("SELECT SUM(original_score)
          FROM user_vertices;");

      ResultSet result = st.executeQuery();
      result.next();
      double sum = result.getDouble(1);

      st.close();

      return (Math.rint(sum) == 1);
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }

  public boolean originalTweetScoreCalculated() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("SELECT SUM(original_score)
          FROM tweet_vertices;");

      ResultSet result = st.executeQuery();
      result.next();
      double sum = result.getDouble(1);

      st.close();

      System.out.println("original_score for tweets sum was " + sum);

      return (Math.rint(sum) == 1);
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }

  private int executeUpdateScoresBatch(PreparedStatement st) {
    int updates = 0;

    try {
      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        updates += results[ii];
      }

      st.close();
      return updates;

    } catch(Exception e) {
      e.printStackTrace();
      return 0;
    }
  }

  //The scores passed in already have the lambdaTweet factor included in them
  public int updateTweetScores(Map<Long,Double> scores) {
    PreparedStatement st = null;

    int updates = 0;
    int curBatchCount = 0;

    try {
      dbConnection.setAutoCommit(true);

      for (Long tweetId : scores.keySet()) {
        if (null == st) {
          st = dbConnection.prepareStatement("UPDATE tweet_vertices SET score=(score + ?)
              WHERE tweetid=?;");
          curBatchCount = 0;
        }

        double score = scores.get(tweetId);

        st.setDouble(1, score);
        st.setLong(2, tweetId);

        st.addBatch();
        curBatchCount++;

        if (curBatchCount > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
          updates += executeUpdateScoresBatch(st);
          st = null;
        }
      }

      if (null != st) {
        updates += executeUpdateScoresBatch(st);
      }

      System.out.println("there were " + updates + " tweet updates out of " +
          scores.keySet().size() + " tweets this iteration");

      return updates;
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }

  //The scores passed in already have the lambdaUser factor included in them
  public int updateUserScores(Map<String,Double> scores) {
    PreparedStatement st = null;

    int updates = 0;
    int curBatchCount = 0;

    try {
      dbConnection.setAutoCommit(true);

      for (String userName : scores.keySet()) {
        if (null == st) {
          st = dbConnection.prepareStatement("UPDATE user_vertices SET score=(score + ?)
              WHERE name=?;");
          curBatchCount = 0;
        }

        double score = scores.get(userName);

        st.setDouble(1, score);
        st.setString(2, userName);

        st.addBatch();
        curBatchCount++;

        if (curBatchCount > DatabaseInterface.MAX_DATABASE_BATCH_SIZE) {
          updates += executeUpdateScoresBatch(st);
          st = null;
        }
      }

      if (null != st) {
        updates += executeUpdateScoresBatch(st);
      }

      System.out.println("there were " + updates + " user updates out of " +
          scores.keySet().size() + " users this iteration");
```

```java

      return updates;
    } catch (Exception e) {
      e.printStackTrace();
      return 0;
    }
  }

  public void updateBaseTweetScores(double lambdaTweets, double totalAmount) {

    double originalScoreFactor = 1 - lambdaTweets;

    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE tweet_vertices SET
          score=(? + (original_score * ?));");
      st.setDouble(1, (lambdaTweets * (totalAmount / (double)getNumTweetVertices())));
      st.setDouble(2, originalScoreFactor);
      st.executeUpdate();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void updateBaseUserScores(double lambdaUsers, double totalAmount) {

    double originalScoreFactor = 1 - lambdaUsers;

    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE user_vertices SET score
          =(? + (original_score * ?));");
      st.setDouble(1, (lambdaUsers * (totalAmount / (double)getNumUserVertices())));
      st.setDouble(2, originalScoreFactor);
      st.executeUpdate();
      st.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }


  public long getNextTweetIdFromCursor() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (!curResults.next()) {
        return -1;
      }

      return curResults.getLong("tweetid");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  public String getNextNameFromCursor() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return null;
      }

      if (!curResults.next()) {
        return null;
      }

      return curResults.getString("name");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }

  public String getNextNameFromInternalCursor() {
    try {
      if ((curInternalResults == null) || curInternalResults.isClosed()) {
        return null;
      }

      if (!curInternalResults.next()) {
        return null;
      }

      return curInternalResults.getString("name");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }


  public String getNameFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return null;
      }

      if (curResults.isBeforeFirst()) {
        return null;
      }

      return curResults.getString("name");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }


  public long getTweetIdFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (curResults.isBeforeFirst()) {
        return -1;
      }

      return curResults.getLong("tweetid");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  public double getUserScoreFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (curResults.isBeforeFirst()) {
        return -1;
      }

      return curResults.getDouble("user_score");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  public double getTweetScoreFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (curResults.isBeforeFirst()) {
        return -1;
      }

      return curResults.getDouble("tweet_score");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  public int getEdgeTypeFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (curResults.isBeforeFirst()) {
        return -1;
      }

      return curResults.getInt("type");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }

  boolean setOriginalUserScoreBatch(Map<String,Double> scores) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE user_vertices set
          original_score=?, score=? where name=?;");

      for (String userName : scores.keySet()) {
        st.setDouble(1, scores.get(userName));
        st.setDouble(2, scores.get(userName));
        st.setString(3, userName);
        st.addBatch();
      }

      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        if (results[ii] != 1) {
          System.out.println("Couldn't add user " + (ii + 1) + " of " + results.length);
          return false;
        }
      }

      st.close();

      return true;
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }


  void setTweetScoresAndOriginalScoresToZero() {
    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE tweet_vertices SET
          score=0, original_score=0;");
      st.executeUpdate();
      st.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  boolean setOriginalTweetScoreBatch(Map<Long,Double> scores) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("UPDATE tweet_vertices set
          original_score=?, score=? where tweetid=?;");

      for (Long tweet : scores.keySet()) {
        st.setDouble(1, scores.get(tweet));
        st.setDouble(2, scores.get(tweet));
        st.setLong(3, tweet);

        st.addBatch();
      }

      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        if (results[ii] != 1) {
          System.out.println("Couldn't add tweet " + (ii + 1) + " of " + results.length);
          return false;
        }
      }

      st.close();

      return true;
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }

  boolean addTweetBatch(Set<Tweet> tweets) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("INSERT INTO tweet_vertices (
          tweetid, name, date, tweet, score, original_score) VALUES (?, ?, ?, ?, ?, ?);");

      for (Tweet tweet : tweets) {
        st.setLong(1, tweet.getId());
        st.setString(2, tweet.getUser());
        st.setTimestamp(3, tweet.getTimestamp());
        st.setString(4, tweet.getTweet());
        st.setDouble(5, 0);
        st.setDouble(6, 0);

        st.addBatch();
      }

      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        if (results[ii] != 1) {
          System.out.println("Couldn't add tweet " + (ii + 1) + " of " + results.length);
          return false;
        }
      }

      st.close();

      return true;
    }
    catch (SQLException e) {
      e.printStackTrace();
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
        System.out.println("");
      }

      return false;
    }
  }

  boolean addUserBatch(Set<String> userNames) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("INSERT INTO user_vertices (
          name, score, original_score) VALUES (?, ?, ?);");

      for (String user : userNames) {
        st.setString(1, user);
        st.setDouble(2, GraphManager.DEFAULT_ORIGINAL_SCORE);
        st.setDouble(3, GraphManager.DEFAULT_ORIGINAL_SCORE);

        st.addBatch();
      }


      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        if (results[ii] != 1) {
          System.out.println("Couldn't add user " + (ii + 1) + " of " + results.length);
          return false;
        }
      }

      st.close();

      return true;
    }
    catch (SQLException e) {
      e.printStackTrace();
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
        System.out.println("");
      }

      return false;
    }
  }

  boolean addEdgeBatch(Set<Edge> edges) {
    try {
      PreparedStatement st = dbConnection.prepareStatement("INSERT INTO edges (name,
          tweetid, type) VALUES (?, ?, ?);");

      for (Edge edge : edges) {
        st.setString(1, edge.name);
        st.setLong(2, edge.tweet);
        st.setInt(3, edge.type);

        st.addBatch();
      }

      int[] results = st.executeBatch();
      for (int ii = 0 ; ii < results.length ; ii++) {
        if (results[ii] != 1) {
          System.out.println("Couldn't add edge " + (ii + 1) + " of " + results.length);
          return false;
        }
      }

      return true;
    }
    catch (SQLException e) {
      e.printStackTrace();
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
        System.out.println("");
      }

      return false;
    }
  }

  //This wouldn't be safe to use if the user could set which table to look into
  boolean tableHasRows(String tableName) {
    PreparedStatement st;
    try {
      st = dbConnection.prepareStatement("SELECT COUNT(*) FROM " + tableName + ";");

      ResultSet result = st.executeQuery();
      result.next();
      int numRows = result.getInt(1);

      st.close();

      if (numRows > 0) {
        return true;
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }

    return false;
  }

  public void close() {
    try {
      closeCursor();
      dbConnection.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }


  public void closeCursor() {
    if (null != curStatement) {
      try {
        curStatement.close(); //Closes the result set, too
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }

  public void closeInternalCursor() {
    if (null != curInternalStatement) {
      try {
        curInternalStatement.close(); //Closes the result set, too
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
  }

  public void acquireInternalCursorForFollowersInUserVertices(String userName) {
    try {
      closeInternalCursor();
      curInternalStatement = dbConnection.prepareStatement("SELECT U.name FROM
          user_vertices U, namenetwork N WHERE U.name=N.followername AND N.username=?;",
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
      curInternalStatement.setFetchSize(100000);
      curInternalStatement.setString(1, userName);
      curInternalResults = curInternalStatement.executeQuery();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public String getNextFollowerNameFromCursor() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return null;
      }

      if (!curResults.next()) {
        return null;
      }

      return curResults.getString("followername");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }

  public String getNextFolloweeNameFromCursor() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return null;
      }

      if (!curResults.next()) {
        return null;
      }

      return curResults.getString("username");
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    }
  }

  //Note: this doesn't advance the cursor as the others do
  public long getUserIdFromCurrentCursorPos() {
    try {
      if ((curResults == null) || curResults.isClosed()) {
        return -1;
      }

      if (curResults.isBeforeFirst()) {
        return -1;
      }

      return curResults.getLong("userid");
    } catch (SQLException e) {
      e.printStackTrace();
      return -1;
    }
  }
}
```

Listing A.4: LuceneIndexManager.java

```
package com.wyeknot.serendiptwitty;

import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.Field.TermVector;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.Term;
import org.apache.lucene.index.TermEnum;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopScoreDocCollector;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.store.LockObtainFailedException;
import org.apache.lucene.util.Version;

import com.wyeknot.serendiptwitty.StanfordTweetIndexer.StanfordParser;

public class LuceneIndexManager {

  public static final String USER_NAME_FIELD_NAME = "user";
  public static final String DATE_FIELD_NAME = "date";
  public static final String TWEET_FIELD_NAME = "tweet";

  Directory index;

  IndexReader reader;
  IndexSearcher searcher;

  private IndexWriter writer = null;
  Analyzer gAnalyzer = null;

  private Map<String,Integer> terms;

  public LuceneIndexManager(String path) throws IOException {
    index = FSDirectory.open(new File(path));
  }

  public boolean indexExists() {
    try {
      return IndexReader.indexExists(index);
    } catch (IOException e) {
      e.printStackTrace();
```

```java
      return false;
    }
  }

  public Directory getIndex() {
    return index;
  }

  public IndexReader getReader() {
    return reader;
  }

  public static IndexWriter getIndexWriter(Directory index, Analyzer analyzer) throws
      CorruptIndexException, LockObtainFailedException, IOException {
    IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_35, analyzer);
    return new IndexWriter(index, config);
  }

  public static IndexWriter getIndexWriter(Directory index) throws CorruptIndexException,
      LockObtainFailedException, IOException {
    Analyzer analyzer = getDefaultAnalyzer();
    return getIndexWriter(index,analyzer);
  }

  public static Analyzer getDefaultAnalyzer() {
    return new StandardAnalyzer(Version.LUCENE_35);
  }

  public Analyzer getOrInitializeAnalyzer() {
    if (null == gAnalyzer) {
      gAnalyzer = LuceneIndexManager.getDefaultAnalyzer();
    }

    return gAnalyzer;
  }

  private void initializeIndexForWriting() throws CorruptIndexException,
      LockObtainFailedException, IOException {
    if (null == writer) {
      writer = getIndexWriter(index, getOrInitializeAnalyzer());
    }
  }

  public void addDocument(Document doc) throws CorruptIndexException,
      LockObtainFailedException, IOException {
    initializeIndexForWriting(); //This function only initializes it if needed
    writer.addDocument(doc);
  }

  public void closeIndexForWriting() {
    try {
      if (null != writer) {
        writer.close();
      }

      writer = null; //So that it will get initialized if it's needed again
    } catch (CorruptIndexException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void initializeForReading() throws CorruptIndexException, IOException {
    if (null == reader) {
      reader = IndexReader.open(index);
      searcher = new IndexSearcher(reader);
    }
  }

  public void createDocumentFrequency() throws CorruptIndexException, IOException {

    initializeForReading();

    // first find all terms in the index
    terms = new HashMap<String,Integer>();
    TermEnum termEnum = reader.terms();

    while (termEnum.next()) {
      Term term = termEnum.term();

      if (!"tweet".equals(term.field())) {
        continue;
      }

      terms.put(term.text(), reader.docFreq(term));
    }
  }

  public Map<String,Integer> getDocumentFrequency() {
    return terms;
  }

  public Document getDocumentFromDocId(int doc) throws CorruptIndexException, IOException
      {
    return searcher.doc(doc);
  }

  ScoreDoc[] runQuery(String queryString) throws ParseException, CorruptIndexException,
      IOException {

    initializeForReading();

    //We search in the tweets themselves if nothing else is specified
    Query query = new QueryParser(Version.LUCENE_35, TWEET_FIELD_NAME,
        getOrInitializeAnalyzer()).parse(queryString);

    TopScoreDocCollector collector = TopScoreDocCollector.create(25000, true);

    searcher.search(query, collector);

    return collector.topDocs().scoreDocs;
  }

  public void closeIndexForReading() {
    if (null != reader) {
      try {
        reader.close();
      } catch (IOException e) {
        e.printStackTrace();
      }

      reader = null;
    }

    if (null != searcher) {
      try {
        searcher.close();
      } catch (IOException e) {
        e.printStackTrace();
      }

      searcher = null;
    }
  }

```

```java
public void indexTweets(String path) {
  parser.indexTweets(path, Recommender.NUM_TWEETS_TO_INDEX);
}

StanfordParser parser = new StanfordParser() {
  @Override
  boolean tweetHandler(Tweet tweet) {
    Document doc = new Document();
    Long curTweet = Long.valueOf(tweet.id);
    doc.add(new Field("tweetid", curTweet.toString(), Field.Store.YES, Field.Index.
        NOT_ANALYZED));
    doc.add(new Field("user", tweet.user, Field.Store.YES, Field.Index.ANALYZED));
    doc.add(new Field("date", tweet.timestamp.toString(), Field.Store.YES, Field.Index.
        NOT_ANALYZED));
    doc.add(new Field("tweet", tweet.tweet, Field.Store.YES, Field.Index.ANALYZED,
        TermVector.YES));
    try {
      addDocument(doc);
    } catch (CorruptIndexException e) {
      e.printStackTrace();
      return false;
    } catch (LockObtainFailedException e) {
      e.printStackTrace();
      return false;
    } catch (IOException e) {
      e.printStackTrace();
      return false;
    }

    return true;
  }
};



public static class DocVector {

  private Map<String,Double> terms;
  private double magnitude = 0;

  /* At creation, the map takes a term to the term frequency
   * in the vector. It will later be modified to be tf*idf
   */
  public DocVector(String[] termList, int[] frequenciesList, Map<String,Integer>
      dfValues, int totalDocs) {
    if (termList.length != frequenciesList.length) {
      throw new RuntimeException("Term list and Frequency list had different lengths!");
    }

    terms = new HashMap<String,Double>();

    for (int ii = 0 ; ii < termList.length ; ii++) {
      if (!dfValues.containsKey(termList[ii])) {
        System.err.println("Something is wrong -- all of the terms should be in the df
            values -- " + termList[ii]);
        continue;
      }

      int df = dfValues.get(termList[ii]).intValue();
      double idf = Math.log10((double)totalDocs / (double)(df + 1)) * (1.0 / (Math.log10
          ((double)(df + 1))));

      terms.put(termList[ii], Double.valueOf((double)frequenciesList[ii] * idf));
    }
  }

  public double cosineSimilarity(DocVector v) {

      double curSum = 0;

      for (String term : terms.keySet()) {
        double otherVal = v.getTfIdfValue(term);
        if (otherVal == 0) {
          continue;
        }

        curSum += otherVal * terms.get(term).doubleValue();
      }

      double magnitudes = v.getMagnitude() * this.getMagnitude();
      if (magnitudes == 0) {
        return 0;
      }

      curSum /= magnitudes;

      return curSum;
    }

    public double getMagnitude() {
      if (magnitude == 0) {
        double curSum = 0;

        for (Double tfIdf : terms.values()) {
          curSum += tfIdf.doubleValue() * tfIdf.doubleValue();
        }

        magnitude = Math.sqrt(curSum);
      }

      return magnitude;
    }

    public double getTfIdfValue(String term) {
      if (terms.containsKey(term)) {
        return terms.get(term).doubleValue();
      }
      else {
        return 0;
      }
    }
  }
}
```

Listing A.5: StanfordTweetIndexer.java

```
package com.wyeknot.serendiptwitty;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;


public class StanfordTweetIndexer {

  private static final String USER_PREFIX = "U";
  private static final String TIMESTAMP_PREFIX = "T";
  private static final String TWEET_PREFIX = "W";

  public static final String STANFORD_TWEETS_FILENAME = "tweets2009-12-pt1.txt";

  public static final String TWEET_NOT_AVAILABLE_STRING = "No Post Title";

  public static final long MAXIMUM_USER_ID = 61578414;
  public static final long TWEET_NUMBER_BASE = 0;
```

```java

static String getUserFromAddress(String address) {
  String[] parts = address.split("/");
  if (parts.length < 3) {
    return null;
  }
  else {
    return parts[parts.length - 1];
  }
}

public static abstract class StanfordParser {
  public static final int ALL_TWEETS_VAL = -2;

  abstract boolean tweetHandler(Tweet tweet);

  private long curTweetId = StanfordTweetIndexer.TWEET_NUMBER_BASE + 1;

  public void indexTweets(String path, int numTweetsRemaining) {
    System.out.println("indexing tweets");

    try {
      FileInputStream fileStream = new FileInputStream(path + STANFORD_TWEETS_FILENAME);
      InputStreamReader in = new InputStreamReader(fileStream, "UTF-8");

      BufferedReader reader = new BufferedReader(in);
      String line = null;

      Tweet tweet = new Tweet(curTweetId);

      boolean keepIndexing = true;

      while (((line = reader.readLine()) != null) && keepIndexing) {
        if (tweet.isComplete()) {

          boolean tweetHandled = tweetHandler(tweet);

          if (numTweetsRemaining != ALL_TWEETS_VAL) {

            if (tweetHandled) {
              //Could combine numTweetsRemaining and curTweetId, obviously, but this is
                  more clear
              numTweetsRemaining--;
              curTweetId++;
              if ((numTweetsRemaining % 500) == 0) {
                System.out.println("Just indexed a tweet with " + numTweetsRemaining + "
                     remaining: " + tweet);
              }

              if (numTweetsRemaining == 0) {
                keepIndexing = false;
              }
            }
          }

          tweet = new Tweet(curTweetId);
        }

        String[] parts = line.split("\t");

        if (parts.length < 2) {
          continue;
        }

        if (parts[0].equals(USER_PREFIX)) {
          String user = getUserFromAddress(parts[1]);

          if (null == user) {
              System.err.println("got a null user: " + parts[1]);
            }

            tweet.setUser(user);
          } else if (parts[0].equals(TWEET_PREFIX)) {

            //This works on the premise that the tweet is always last, which probably isn'
                t especially good to rely on
            if (parts[1].equals(TWEET_NOT_AVAILABLE_STRING)) {
              tweet = new Tweet(curTweetId);
              continue;
            }

            tweet.setTweet(parts[1]);
          } else if (parts[0].equals(TIMESTAMP_PREFIX)) {
            tweet.setTimestampString(parts[1]);
          } else {
            continue;
          }
        }

        reader.close();
        in.close();
        fileStream.close();

      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

Listing A.6: Edge.java

```java
package com.wyeknot.serendiptwitty;

public class Edge {

  public enum Types {
    //Connects a tweet to the user who created it

    EDGE_TYPE_AUTHORSHIP (1, 1, true, true),
    //Connects a tweet to all of the followers of the tweeter
    EDGE_TYPE_FOLLOWER   (2, 1, true, false),
    //Connects the retweet to the original tweeter
    EDGE_TYPE_RETWEET    (3, 1, false, false),
    //Connects the tweets of the followees of the person being retweeted to the retweeter
    EDGE_TYPE_RETWEET_FOLLOWEES (4, 1, true, false),
    //Connects the tweets of the person being retweeted to the followers of the retweeter
    EDGE_TYPE_RETWEET_FOLLOWERS (5, 1, true, false),
    //Connects the tweet to the person being mentioned
    EDGE_TYPE_MENTION (6, 1, false, false),
    //Connects the tweets of the followees of the person being mentioned to the mentioner
    EDGE_TYPE_MENTION_FOLLOWEES (7, 1, true, false),
    //Connects the tweets of the person being mentioned to the followers of the mentioner
    EDGE_TYPE_MENTION_FOLLOWERS (8, 1, true, false),
    //Connects the tweet to the person being @replied to
    EDGE_TYPE_AT_REPLY (9, 1, false, false),
    //Connects the tweets of the person being @replied to to the tweeter
    EDGE_TYPE_AT_REPLY_CONTENT (10, 1, true, false),
    EDGE_TYPE_HASHTAG (11, 1, true, true),
    EDGE_TYPE_CONTENT (12, 1, true, true);

    private final int id;
    private final double probability;
    private final boolean tweetToUserDir;
    private final boolean userToTweetDir;

    private Types(int id, double probability, boolean tweetToUser, boolean userToTweet) {
      this.id = id;
      this.probability = probability;
      this.tweetToUserDir = tweetToUser;
      this.userToTweetDir = userToTweet;
    }

    public int id() { return id; }
    public double probability() { return probability; }
    public boolean tweetToUserDir() { return tweetToUserDir; }
    public boolean userToTweetDir() { return userToTweetDir; }
  }

  public static final Types[] idToEdgeType = {
    null,
    Types.EDGE_TYPE_AUTHORSHIP,
    Types.EDGE_TYPE_FOLLOWER,
    Types.EDGE_TYPE_RETWEET,
    Types.EDGE_TYPE_RETWEET_FOLLOWEES,
    Types.EDGE_TYPE_RETWEET_FOLLOWERS,
    Types.EDGE_TYPE_MENTION,
    Types.EDGE_TYPE_MENTION_FOLLOWEES,
    Types.EDGE_TYPE_MENTION_FOLLOWERS,
    Types.EDGE_TYPE_AT_REPLY,
    Types.EDGE_TYPE_AT_REPLY_CONTENT,
    Types.EDGE_TYPE_HASHTAG,
    Types.EDGE_TYPE_CONTENT
  };


  public String name;
  public long tweet;
  public int type;

  Edge(String name, long tweet, int type) {
    this.name = name;
    this.tweet = tweet;
    this.type = type;
  }
}
```

Listing A.7: Pair.java

```java
package com.wyeknot.serendiptwitty;


public class Pair<F, S> {
  private F first;
  private S second;

  public Pair(F f, S s){
    this.first = f;
    this.second = s;
  }

  public F getFirst() { return first; }
  public S getSecond() { return second; }

  public int hashCode() {
    return first.hashCode() + second.hashCode();
  }

  public boolean equals(Object o) {
    if (!(o instanceof Pair<?,?>)) {
      return false;
    }

    @SuppressWarnings("unchecked")
    Pair<F,S> obj = (Pair<F,S>)o;
    if (first.equals(obj.first) && second.equals(obj.second)) {
      return true;
    }

    return false;
  }
}
```

Listing A.8: Tweet.java

```java
package com.wyeknot.serendiptwitty;

import java.sql.Timestamp;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.HashSet;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Tweet {
  long id;
  String user;
  Timestamp timestamp;
  String tweet;

  public static final SimpleDateFormat stanfordTweetDateParser = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");


  private static final Pattern hashTagRegExPattern = Pattern.compile(" #[a-zA-Z0-9]*[a-zA-Z]");

  private static final Pattern atReplyRegExPattern = Pattern.compile("^@[a-zA-Z][a-zA-Z0-9_]*");
  private static final Pattern mentionRegExPattern = Pattern.compile("@[a-zA-Z][a-zA-Z0-9_]*");
  private static final Pattern rtRegExPattern = Pattern.compile("RT @[a-zA-Z][a-zA-Z0-9_]*");
  private static final Pattern rtRegExPattern2  = Pattern.compile("RT@[a-zA-Z][a-zA-Z0-9_]*");
  private static final Pattern rtRegExPattern3  = Pattern.compile("RT: @[a-zA-Z][a-zA-Z0-9_]*");
  private static final Pattern rtRegExPattern4  = Pattern.compile("via @[a-zA-Z][a-zA-Z0-9_]*");
  //This is the starting point of the actual name in each of the RT regex patterns
  private static final int rtPatternNameStart = 4;
  private static final int rtPattern2NameStart = 3;
  private static final int rtPattern3NameStart = 5;
  private static final int rtPattern4NameStart = 5;



  public Tweet(long id) {
    this.id = id;
    user = null;
    timestamp = null;
    tweet = null;
  }

  public long getId() {
    return id;
  }

```

```
public String getUser() {
  return user;
}

public void setUser(String user) {
  this.user = user;
}

public Timestamp getTimestamp() {
  return timestamp;
}

public void setTimestamp(Timestamp timestamp) {
  this.timestamp = timestamp;
}

public void setTimestampString(String timestamp) {
  this.timestamp = getTimestampFromString(timestamp);
}

public static Timestamp getTimestampFromString(String timestamp) {
  try {
    return new Timestamp(stanfordTweetDateParser.parse(timestamp).getTime());
  } catch (ParseException e) {
    e.printStackTrace();
    System.err.println("\nCouldn't parse date " + timestamp + "!!!");
  }

  return null;
}

public String getTweet() {
  return tweet;
}

public void setTweet(String tweet) {
  this.tweet = tweet;
}

public boolean isComplete() {
  return ((null != user) && (timestamp != null) && (tweet != null));
}

public String toString() {
  return user + " on " + timestamp + ": " + tweet;
}


public static Set<String> findRetweetedUsers(String tweet) {
  Set<String> users = new HashSet<String>();

  Matcher m = rtRegExPattern.matcher(tweet);
  while (m.find()) {
    String match = m.group();
    String user = match.substring(rtPatternNameStart).toLowerCase();
    if (user.length() >= 32) {
      continue;
    }
    users.add(user);
  }

  m = rtRegExPattern2.matcher(tweet);
  while (m.find()) {
    String match = m.group();
    String user = match.substring(rtPattern2NameStart).toLowerCase();
    if (user.length() >= 32) {
      continue;
    }
```

```
      users.add(user);
    }

    m = rtRegExPattern3.matcher(tweet);
    while (m.find()) {
      String match = m.group();
      String user = match.substring(rtPattern3NameStart).toLowerCase();
      if (user.length() >= 32) {
        continue;
      }
      users.add(user);
    }

    m = rtRegExPattern4.matcher(tweet);
    while (m.find()) {
      String match = m.group();
      String user = match.substring(rtPattern4NameStart).toLowerCase();
      if (user.length() >= 32) {
        continue;
      }
      users.add(user);
    }

    return users;
  }

  public static String findAtReply(String tweet) {
    Matcher m = atReplyRegExPattern.matcher(tweet);

    String user = null;

    while (m.find()) {
      user = m.group().substring(1).toLowerCase();
      if (user.length() > 32) {
        user = null;
      }
    }

    return user;
  }

  public static Set<String> findMentionedUsers(String tweet, Set<String> retweetedUsers) {
    Set<String> users = new HashSet<String>();

    Matcher m = mentionRegExPattern.matcher(tweet);

    while (m.find()) {
      String match = m.group();
      String user = match.substring(1).toLowerCase();

      if (m.start() == 0) {
        //This is an @reply -- ignore it here
      }
      else if (!retweetedUsers.contains(user) && (user.length() < 32)) {
        users.add(user);
      }
    }

    return users;
  }

  public static Set<String> findHashTags(String tweet) {
    Matcher m = hashTagRegExPattern.matcher(tweet);

    HashSet<String> hashes = new HashSet<String>();

    while (m.find()) {
      hashes.add(m.group().substring(2).toLowerCase());
    }

    return hashes;
  }

}
```

Listing A.9: User.java

```
package com.wyeknot.serendiptwitty;

import java.util.Comparator;


public class User {
  public long id;
  public String name;

  public static int NO_ID_AVAILABLE = -1;

  static final Comparator<User> NAME_ORDER = new Comparator<User>() {
        public int compare(User u1, User u2) {
          return u1.name.compareToIgnoreCase(u2.name);
        }
    };

    User(long id, String name) {
      this.id = id;
      this.name = name;
    }


  User(String name) {
    this.name = name;
    this.id = NO_ID_AVAILABLE;
  }

  public String toString() {
    return name;
  }
}
```

Bibliography

[1] Lada A. Adamic and Eytan Adar. How to search a social network. *Social Networks*, 25(3):211–230, October 2003.

[2] Marcelo G. Armentano, Daniela Godoy, and Analía Amandi. Recommending Information Sources to Information Seekers in Twitter. In *Proceedings IJCAI: The International Workshop on Social Web Mining*, Barcelona, Spain, 2011.

[3] Sitaram Asur and Bernardo A Huberman. Predicting the Future with Social Media. *Computing*, 1(1):492–499, 2010.

[4] L Backstrom and J Leskovec. Supervised Random Walks: Predicting and Recommending Links in Social Networks. *Proceedings of the fourth ACM international conference on Web search and data mining*, abs/1011.4(6):635–644, 2010.

[5] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, Washington D.C., 2010.

[6] Munmun De Choudhury, Yu-Ru Lin, Hari Sundaram, K. Selcuk Candan, Lexing Xie, and Aisling Kelliher. How does the data sampling strategy impact the discovery of information diffusion in social media. In *Proceedings of the Fourth International AAAI Conference on Weblogs and Social Media*, pages 34–41, 2010.

[7] Hongbo Deng, Michael R. Lyu, and Irwin King. A generalized Co-HITS algorithm and its application to bipartite graphs. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*, page 239, New York, New York, USA, June 2009. ACM Press.

[8] Yajuan Duan, Long Jiang, Tao Qin, Ming Zhou, and Heung-yeung Shum. An Empirical Study on Learning to Rank of Tweets. *Computational Linguistics*, (August):295–303, 2010.

[9] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by Gibbs sampling. *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics ACL 05*, 43(1995):363–370, 2005.

[10] S Hangal, D MacLean, and MS Lam. All friends are not equal: Using weights in social graphs to improve search. In *4th ACM Workshop on SONAM*, volume 10, 2010.

[11] John Hannon, Mike Bennett, and Barry Smyth. Recommending twitter users to follow using content and collaborative filtering approaches. In *Proceedings of the fourth ACM conference on Recommender systems - RecSys '10*, page 199, New York, New York, USA, 2010. ACM Press.

[12] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis - WebKDD/SNA-KDD '07*, pages 56–65, New York, New York, USA, August 2007. ACM Press.

[13] Younghoon Kim and Kyuseok Shim. TWITOBI: A Recommendation System for Twitter Using Probabilistic Modeling. *2011 IEEE 11th International Conference on Data Mining*, pages 340–349, December 2011.

[14] Jon M. Kleinberg. Hubs, authorities, and communities. *ACM Computing Surveys*, 31(4es):5–es, December 1999.

[15] Jerome Kunegis, Ernesto De Luca, and Sahin Albayrak. The link prediction problem in bipartite networks. *Computational Intelligence for Knowledge-Based Systems Design*, 6178:380–389, 2010.

[16] Haewoon Kwak, Changhyun Lee, and Hosung Park. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web - WWW '10*, pages 591–600, 2010.

[17] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

[18] Matthew Michelson and Sofus A Macskassy. Discovering users' topics of interest on twitter. In Roberto Basili, Daniel Lopresti, Christoph Ringlstetter, Shourya

Roy, Klaus Schulz, and L Venkata Subramaniam, editors, *Proceedings of the fourth workshop on Analytics for noisy unstructured text data AND 10*, AND '10, page 73. ACM Press, 2010.

[19] Brendan O'Connor, Ramnath Balasubramanyan, Bryan R Routledge, and Noah A Smith. From Tweets to Polls : Linking Text Sentiment to Public Opinion Time Series. *Most*, 5(May):122–129, 2010.

[20] Ravali Pochampally and Vasudeva Varma. User context as a source of topic retrieval in Twitter. In *Proceeding of the 34rd international ACM SIGIR conference on Research and development in information retrieval*, number Enir, pages 1–3, 2011.

[21] Daniel Ramage, Susan Dumais, and Dan Liebling. Characterizing Microblogs with Topic Models. *International AAAI Conference on Weblogs and Social Media*, 5(4):130–137, 2010.

[22] Michael J. Welch, Uri Schonfeld, Dan He, and Junghoo Cho. Topical semantics of twitter links. In *Proceedings of the fourth ACM international conference on Web search and data mining - WSDM '11*, page 327, New York, New York, USA, February 2011. ACM Press.

[23] Jianshu Weng, EP Lim, and Jing Jiang. Twitterrank: finding topic-sensitive influential twitterers. *Conference on Web Search and Data Mining*, pages 261–270, 2010.

[24] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web search and data mining - WSDM '11*, page 177, New York, New York, USA, February 2011. ACM Press.