The University of Manchester

University of Manchester
School of Computer Science

Social Search

Third Year Project Report 2013
B.Sc. (Hons) Computer Science

Freddy Kelly
May 2, 2013

Supervised by Dr. Gavin Brown

Abstract

Web search has seen increased research and development in recent years due in part to the popularity of social and mobile technologies. The way in which we consume and generate content has changed completely with the advent of social networking. This project aims to utilise this new source of 'social data' in web search, using various Machine Learning techniques to produce structured search results. The task is approached from two angles: firstly, the mechanics required to implement a working prototype and secondly, finding effective performance measures for such a system. This report aims to provide a foundation for applications of this class; it describes the fundamentals of a capable system, and provides a set of recommendations for anyone wishing to utilise this new data in search.

Acknowledgements

I would like to thank my supervisor, Dr. Gavin Brown, for his continued support and guidance throughout the project. I also owe a lot to Dr. Caroline Jay of the *Web Ergonomics Lab* for her advice during development of my usability tests, as well as my peers who volunteered to participate and offered their input and support throughout.

Contents

Contents **3**

List of Figures **6**

List of Tables **7**

1 Introduction **8**
1.1 Elevator Pitch . . . 8
1.2 Web Search . . . 8
1.3 Social Networking . . . 8
1.4 Existing Work . . . 9
1.5 Approach . . . 9
1.6 Report Structure . . . 10

2 Background and Literature Survey **12**
2.1 Social Data . . . 12
2.1.1 Facebook . . . 12
2.1.2 Twitter . . . 12
2.1.3 App.net . . . 14
2.2 Ranking Mechanisms . . . 14
2.3 Machine Learning . . . 15
2.3.1 Algorithm Types . . . 16
2.4 Representation & Distance . . . 16
2.4.1 Representation . . . 16
2.4.2 Distance Measures . . . 18
2.5 Clustering Techniques . . . 20
2.5.1 Hierarchical . . . 20
2.5.2 Partitional . . . 23
2.6 Natural Language Processing . . . 25
2.6.1 Tokenisation . . . 25
2.6.2 Stemming . . . 26
2.6.3 Named Entity Recognition . . . 27
2.6.4 Sentiment Analysis . . . 28

3 Prototyping **30**
3.1 Implementation . . . 30
3.2 'Gold Standard' Evaluation . . . 31

3.3 User Study . . . 32
3.3.1 Design . . . 35
3.3.2 Evaluation . . . 35

4 Application Design **37**
4.1 Requirements Analysis . . . 37
4.1.1 Use Cases . . . 37
4.1.2 Formal Requirements . . . 40
4.2 Environment . . . 41
4.3 Specification . . . 42
4.3.1 Class Hierarchy . . . 42
4.3.2 Data Flow . . . 44
4.4 Methodologies . . . 46
4.4.1 AGILE . . . 46
4.4.2 TDD . . . 47
4.5 User Experience . . . 47
4.5.1 Wireframes . . . 47
4.6 Architecture . . . 50
4.6.1 Language . . . 50
4.6.2 Application . . . 51
4.6.3 Interface . . . 52
4.6.4 Storage . . . 52
4.6.5 Testing . . . 52
4.6.6 Deployment . . . 53

5 Implementation **54**
5.1 Clustering . . . 54
5.1.1 DistanceMeasure . . . 55
5.1.2 Tokeniser . . . 56
5.1.3 Document . . . 58
5.2 Back End . . . 59
5.2.1 Models . . . 59
5.2.2 Controllers . . . 60
5.3 Front End . . . 62
5.3.1 Web Views . . . 62
5.3.2 CSS . . . 64
5.3.3 JavaScript . . . 65
5.4 Administration . . . 68
5.4.1 Statistics . . . 68

6 Results and Evaluation **71**
6.1 Application . . . 71
6.2 Analysis . . . 76
6.2.1 Statistics . . . 76
6.2.2 KPIs . . . 76
6.2.3 Crowdsourcing . . . 77
6.2.4 Example Searches . . . 79

6.2.5 Improvements 81

7 Conclusions **83**
7.1 Reflection 83
7.2 Lessons 83
7.3 Possible Improvements 84
7.3.1 Crowdsourced Evaluation 84
7.3.2 Detection and Labelling 84
7.3.3 User Interface 84
7.4 Recommendations 85
7.4.1 Clustering 85
7.4.2 Topic Labelling 85
7.4.3 User Experience 86

Bibliography **87**

Glossary **91**

A Appendix **94**
A.1 Participant Form 95
A.2 Initial Plan 100

List of Figures

1.1 Google's Twitter Integration . . . 11
1.2 Bing's Twitter Integration . . . 11

2.1 Google's PageRank Algorithm . . . 15
2.2 Facebook's EdgeRank Score . . . 15
2.3 Dendrogram Example . . . 20
2.4 Token Comparison . . . 27
2.5 Stemmed Token Comparison . . . 27
2.6 Social Language . . . 28

3.1 Early Prototypes . . . 30
3.2 Manual Clustering . . . 32
3.3 Search Behaviours . . . 34

4.1 Application Environment . . . 41
4.2 Class Diagram . . . 43
4.3 Sequence Diagrams . . . 45
4.3 Wireframes . . . 48
4.2 Code Climate Example . . . 53

5.1 Asynchronous Searching . . . 65
5.3 Query Completion . . . 66
5.2 Modal Image . . . 67
5.4 Mobile vs Desktop . . . 67
5.5 Session View . . . 69
5.6 Dashboard . . . 70

6.1 Search Request . . . 71
6.1 Search Feedback . . . 73
6.2 Data Retrieval . . . 74
6.2 Time taken to select a Result . . . 77
6.3 Returning Visitors . . . 77
6.4 Microworkers Campaign . . . 78
6.5 Completed HITs . . . 79
6.6 Successful Searches against Results/Search . . . 82

7.1 Socially Augmented Search . . . 85
7.2 Hashtag Labelling Example . . . 86

List of Tables

2.1 Tweet Schema 13
2.2 Tweet Entities Schema 13
2.3 Document Clustering Algorithms 20
2.4 Natural Language Processes 25

4.1 Use Cases 38

6.1 Application Statistics 76
6.2 Search Examples 80

1 Introduction

Despite being well established, development in web search is still heavily pursued by names like *Microsoft* and *Google*. Among other motivations, this continued development can be attributed to the evolving ways we create and search information. New data is constantly becoming available, partially thanks to the explosion in social networking. The abundance of this real-time data cannot be ignored; yet currently we are unable to leverage it as an additional source. The aim of this project is to produce a system that enables users to search social data in real-time, using Machine Learning (ML) techniques to produce user-friendly representations of topical discussion.

1.1 Elevator Pitch

> *"Generate search results for a query by clustering Twitter statuses containing that term(s) and extracting URLs from the messages."* [1]

1.2 Web Search

Although search has been around since the early days of the web, it has undergone many changes since its inception. The earliest search engines relied on the simple *crawl-index* model: using crawlers/spiders to search out new web pages, generating an *index* which maps search terms to relevant documents. Modern search engines operate an advanced array of techniques to provide higher relevance to users. There has been a vast amount of investment into researching such techniques, but the majority of existing work has only considered the indexing of *documents*. As we consume and generate more and more content via mobile devices like smartphones and tablets, the need to differentiate this new type of data has become apparent.

1.3 Social Networking

It is no secret that social networks have changed the way we communicate, but increasingly they are changing the way we consume information entirely. Now, when browsing, instead of evaluating content based on author, source etc. we can interact on a totally personal level, making choices based on recommendations from peers, and other social weightings. Social networks provide us with a level of customisation that was previously unachievable with filter bubbling[2] and other similar techniques. More significant, perhaps, is the content we create using

[1] `http://www.businessweek.com/stories/2007-06-18/the-perfect-elevator-pitchbusinessweek-business-news-stock-market-and-financial-advice`

[2] `http://en.wikipedia.org/wiki/Filter_bubble`

social platforms: in a recent Mintel survey 53% of people asked said they use social networks to "Share general news or articles I have read" (Liao 2012). The ability to instantly publish information to the world means (among other things), that information has become instant; information is being published within seconds of an event, from multiple independent sources. It was exactly this kind of social information that spearheaded the rescue operation when US Airways Flight 1549 crash-landed into the Hudson River on January 15, 2009 (Beaumont 2009). Despite its abundance, the use of this social data in search is yet to be extensively explored. It is my belief that developing systems that can capitalise on this new type of information will be key to the future of search.

1.4 Existing Work

Document clustering is by no means a new area of Information Retrieval (IR). Indeed, numerous approaches have been extensively developed and discussed, many used in various commercial applications today (including search). Most applications of document clustering involve large corpora which need to be categorised into groups of relevant documents.

We will consider specific algorithms later (see chapter 2). At this point it is important only to distinguish the defining characteristics of social network data that are significant to the task of clustering. Whilst some work has been focused on clustering this kind of data (Khot 2010), the focus has been on isolated samples, rather than 'live' streams.

In addition to development of clustering algorithms suitable to social networking applications, there have also been a number of commercial projects targeted at incorporating social with search. Prior to July 2011, an agreement between Twitter and Google had allowed users of the latter to view relevant tweets alongside search results (fig. 1.1). The agreement (worth $20m according to Barnett 2011), saw Twitter permit Google access to its 'firehose' API endpoint [3]; providing them all tweets in real-time. The integration was rather minimal, Google simply displayed recent tweets containing the user's query. More recently, a similar contract has seen Bing providing its own Twitter search feature. Unlike Google's effort, Bing's search (fig. 1.2) is completely separate from their primary offering. The application provides users with a list of trending topics, alongside a search field (which appears to be defective at the time of writing). While both products present Twitter's data in new ways, neither attempts to enhance their existing search, instead opting to show tweets as a separated 'stream' of data.

1.5 Approach

When the project was conceived, my focus was on developing effective clustering methods for social network data. However, as development advanced, it became apparent that being able to define 'effectiveness' and accurately measure it would be critical to evaluating the performance of any proposed system. As such, many of the key milestones centred on developing suitable performance metrics. Below is a breakdown of key milestones relating to development and evaluation:

- **Investigate ML Methods** An initial literature survey and evaluation of suitable algorithms.

[3] `https://dev.twitter.com/docs/api/1.1/get/statuses/firehose`

- **Prototype Development** Establishing a Minimum Viable Product (MVP).
- **Exploration of Performance Measures** Investigating suitable Key Performance Indicators (KPIs).
- **Qualitative User Study** Testing and evaluating KPIs with respect to an automated quantitative study.
- **Quantitative Online Study** Implementing KPIs via a distributed online study.
- **Evaluation** Evaluation of methods via analysis of both qualitative and quantitative measures.

Whilst these milestones provide a coarse-grained roadmap for the project, it should be noted that in many cases multiple avenues were explored before deciding on a final outcome. To give an example, whilst developing suitable performance measures, I first considered automated tests against a so-called 'gold standard' (section 3.2), before later ruling out this method.

1.6 Report Structure

Although the project centres on evaluating the feasibility of the proposed system, design is equally emphasised throughout the report, with focus on best practices like Test-driven Development (TDD) as well as key design factors including *scalability*, *deployment* etc. This report is structured as follows:

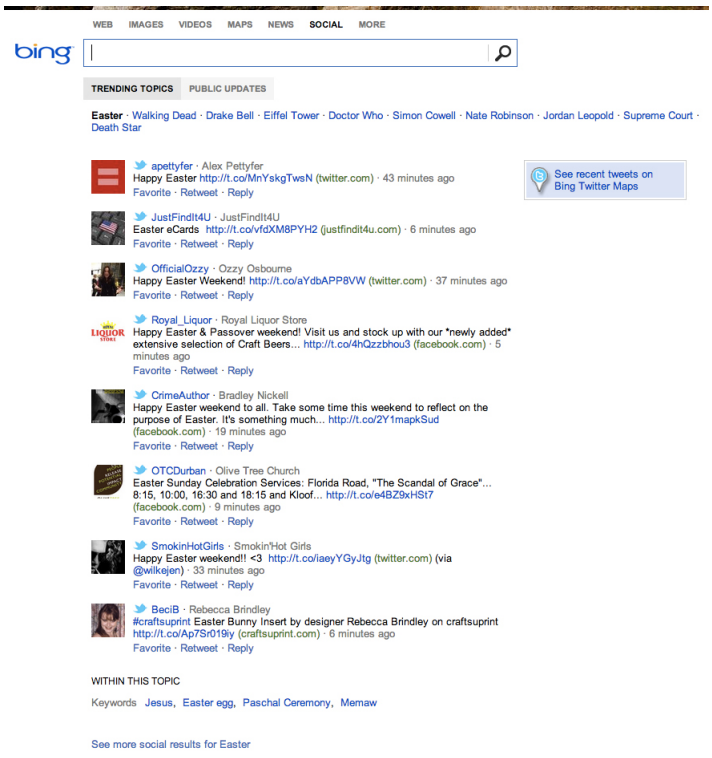
1. **Introduction**
2. **Background and Literature Survey**
   I will discuss the various sources I consulted, drawing conclusions on applicable methods.
3. **Design**
   This chapter discusses the design of each software component at a high level, reflecting the project's life cycle.
4. **Implementation and Deployment**
   This chapter will focus on more low-level aspects of implementation, covering key considerations such as scalability, fault-tolerance etc.
5. **Results and Evaluation**
   This chapter describes a demonstration of my proposed software solution as well as sample data collected during its operation.
6. **Conclusions**
   Here I will present recommendations for incorporating social data in search as well as reflecting on my own learning.

Figure 1.1: Google's Twitter Integration

`http://digitalmarketingengineer.com/2010/11/evolution-of-googles-integration-of-twitter-into-search-results/`

Figure 1.2: Bing's Twitter Integration

`http://www.bing.com/social/`



2 Background and Literature Survey

In this chapter I will review various sources I consulted both prior to, and throughout development. Technical documents, scientific papers and a number of online sources provide a knowledge base I will employ throughout this report. In most cases I have foregone in-depth description in favour of focus on application to the proposed system.

2.1 Social Data

Before fixing on the task of document clustering, we must first establish exactly what we mean by 'social data', and consider the various forms such information can take. The vast majority of social networking platforms offer the ability to communicate with other (authenticated) third party applications via some form of Application Programming Interface (API). We will next consider the types of information currently available from some of the major platforms. At this stage we will focus on *what* information is available, putting implementation specifics aside (see chapter 5).

2.1.1 Facebook

By far the most established social networking platform with well over a billion[1] active users, Facebook exposes data via its 'Graph API'. Whilst it is possible to retrieve a user's status updates, this is only possible once that user has granted an application access. Facebook does not expose any kind of site wide public stream, prohibiting any kind of document clustering applications such as the one I propose.

2.1.2 Twitter

The *microblogging* service Twitter has a more simplistic model compared to Facebook; users post messages (tweets) of 140 characters or fewer, optionally containing attachments such as images or URLs. Whilst Twitter restricts access to its 'catch-all' `statuses/firehose`[2] endpoint to selected partners and resellers[3], it does provide a `search/tweets`[4] method, which returns any tweets matching a supplied query. The latter call returns an array of JavaScript Object Notation (JSON) documents, each representing a matched status update. Table 2.1 shows the (condensed) schema for each `Tweet` instance.

[1] `https://www.facebook.com/zuck/posts/10100518568346671`
[2] `https://dev.twitter.com/docs/api/1.1/get/statuses/firehose`
[3] Gnip, Topsy and Datasift
[4] `https://dev.twitter.com/docs/api/1.1/get/search/tweets`
[5] `http://tools.ietf.org/html/bcp47`

Table 2.1: Tweet Schema

https://dev.twitter.com/docs/platform-objects/tweets

Field	Type	Description
text	String	The actual UTF-8 text of the status update.
id	int64	The integer representation of the ID of the user who contributed to this Tweet.
coordinates	float (pair)	Represents the geographic location of this Tweet as reported by the user or client application.
created_at	String	UTC time when this Tweet was created.
entities	Entities (see table 2.2)	Entities which have been parsed out of the text of the Tweet.
favorite_count	int	Indicates approximately how many times this Tweet has been "favorited" by Twitter users.
retweet_count	int	Number of times this Tweet has been retweeted.
lang	String	When present, indicates a BCP 47[5] language identifier corresponding to the machine-detected language of the Tweet text.
place	Place	When present, indicates that the tweet is associated (but not necessarily originating from) a Place (e.g., a town, city or venue).
user	User	The user who posted this Tweet.

Table 2.2: Tweet Entities Schema

https://dev.twitter.com/docs/platform-objects/entities

Field	Type	Description
hashtags	JSON Array	Represents hashtags which have been parsed out of the Tweet text.
media	JSON Array	Represents media elements (e.g., images or video) uploaded with the Tweet.
urls	JSON Array	Represents Universal Resource Locators (URLs) included in the text of a Tweet.
user_mentions	JSON Array	Represents other Twitter users mentioned in the text of the Tweet.

Twitter provides a variety of useful metadata as well as the actual `text` value of each tweet. Simple counters like `favourite_count` and `retweet_count` are a good way to gauge the popularity of a tweet (e.g., for ranking). Additionally, attributes such as `coordinates` and `lang` provide good potential features for clustering and filtering.

The Twitter API exposes this data for all publicly available accounts (the vast majority), providing a rich source of real-time data.

2.1.3 App.net

App.net is one of the first social networks to employ a subscription style revenue model (cf. advertising based). In its infancy, App.net offers a microblogging service similar to Twitter. App.net provides an API to (fee-paying) developers which offers a multitude of endpoints to access various data produced by users of the service. The `stream` endpoint returns an array of all recent activity on the platform, including both `post`[6] and `message`[7] objects. Whilst perhaps not the best candidate for this project due to its restricted access, App.net merits inclusion to provide an exemplar of other JSON APIs.

Whilst I have only referenced a few examples of the types of data available, it is clear that there is certainly scope to retrieve the real-time data required to produce a working system. Despite nuances, the majority of platforms function in a similar way, opting for RESTful APIs returning JSON. Exploiting this similarity by developing a modular system for interacting with such APIs, I hope to ensure the application remains extendable, with potential to integrate additional platforms in the future.

2.2 Ranking Mechanisms

I have already hinted how social traction measures like retweets might be used to gauge popularity of a topic or story. Before exploring this avenue (and others) any further, I will first consider how existing search engines apply rankings to their results. Returning to the 'baseline' search model (discussed in section 1.2), we now consider the original design proposal for Google. Brin and Page 1998 describe a system which "[...] makes use of the link structure of the Web to calculate a quality ranking for each Web page." known as 'PageRank'. PageRank uses a 'citation graph' to measure importance based on the number of incoming links for each result. This objective measure is described as "correspond[ing] well with people's subjective idea of importance." (Brin and Page 1998, p.109). Figure 2.1 provides an approximation of the PageRank mechanism. Despite the secrecy surrounding Google's actual algorithms, it is clear that the major contributor to a positive ranking is the number of external sites linking to a given page. This idea is perhaps less applicable for social references, as often resources are too 'fresh' to have gained many citations. Despite this, the concept of a weighted ranking is certainly apropos in a social context. Indeed 'EdgeRank' (2.2), described during Facebook's 2011 developer conference, *f8*, uses weightings based on platform interactions such as *likes*, *comments* etc. EdgeRank governs which stories, and perhaps most importantly, in what order, appear on a user's newsfeed. Perhaps most interesting in the context of this project are EdgeRank's weighted scores for each interaction type (denoted w). This prioritisation allows lower friction

[6] `http://developers.app.net/docs/resources/post/#post-fields`
[7] `http://developers.app.net/docs/resources/message/#fields`

Figure 2.1: Google's PageRank Algorithm
http://www.sirgroane.net/google-page-rank/

$$PR(A) = (1-d) + d(PR(T_1)/C(T_1) + \cdots + PR(T_n)/C(T_n)) \tag{2.1}$$

where:

$$\begin{aligned} PR(T_n) &= \text{importance, e.g. } PR(T_1) \text{ for } \texttt{index.html} \text{ and so on} \\ C(T_n) &= \text{count of outgoing links from page} \\ PR(T_n)/C(T_n) &= \text{share of vote for page } A \text{ for a given backlink from page } n \\ d &= \text{dampening factor} \end{aligned}$$

Figure 2.2: Facebook's EdgeRank Score
http://www.whatisedgerank.com/

$$\sum_{e \in E} u_e w_e d_e \tag{2.2}$$

where:

$$\begin{aligned} e &= \text{Affinity score between viewing user and edge creator} \\ w &= \text{Weight for this edge type (status, comment, like, tag, etc.)} \\ d &= \text{Time decay factor based on how long ago the edge was created} \end{aligned}$$

actions, such as liking a story, to carry less significance than, say, posting a comment (which usually requires more work by the user). This idea easily translates to other platforms too, e.g. retweets *vs* replies on Twitter.

There is clear parity between the core concepts of search engine ranking algorithms such as PageRank and newer social measures like EdgeRank. Evaluating sources based on a citation graph as originally described by Brin and Page (1998) can be compared to rating according to the number of *likes* or *retweets* amassed. Twitter recently announced they would be adding similar mechanisms to their API (Laird 2013). As we have seen (section 2.1), the necessary metadata are available to provide 'social rankings', allowing results to be organised by what people think is important at any given time.

2.3 Machine Learning

Machine Learning (ML) is the key ingredient that will allow for sensible categorisations of the data retrieved from social networks. Before evaluating possible approaches, we first provide a formal definition of document clustering problems:

Definition 1 (Document Clustering)**.**
Given: *a set of n documents, $X = \{x_1, x_2, \ldots, x_n\}$*
along with a (normalised) distance measure $d : X \times X \to [0, 1]$

Return: *a partition of* X *into* k *clusters,* $\Pi_k = \{\pi_1, \pi_2, \ldots, \pi_k\}$

In this section I will investigate various algorithms (and varying implementations), with a focus on the task of clustering the types of data previously examined (section 2.1).

2.3.1 Algorithm Types

Prior to exploring implementations, it is important discuss the taxonomy of ML algorithms. Depending on required outcome and available inputs to an algorithm, we can make the following categorisations[8]:

- **Supervised Learning**
  Supervised learning is based on some known training set; e.g. *classification* problems. The learning algorithm approximates classes for new examples based on proximity to known input/output examples.
- **Unsupervised Learning**
  Unsupervised learning describes algorithms in which the possible labels are unknown; the task of the algorithm is to determine how the data is organised. e.g. categorising scientific papers.

As we will be working with no prior knowledge of possible class labels for the data we will be retrieving, the task lies predominantly in the latter category. More specifically, the task of document clustering lies in the larger field of data mining as it involves a wide range of disciplines. Although the real-time nature of the task prohibits *supervised* methods from being viable for clustering, certain elements are still applicable, for example training sets could be generated using smaller samples of data.

2.4 Representation & Distance

In order to cluster documents we intuitively rely on some notion of similarity or difference. We represent this comparison as a function, $d : X \times X \to [0, 1]$ (definition 1), that is, given two documents, return a difference score between 0/1 (0 if they are identical, 1 if there is no similarity). Before we can develop this function, we need to establish how the arguments will be represented. This section considers various approaches for both representation and comparison of strings.

2.4.1 Representation

A number of representations exist for textual data in data mining applications, I will consider a number of techniques, using the following (minimal) example to demonstrate each:

```
a = "UC San Diego Computer Scientists Develop First-person
     Player Video Game that Teaches How to Program in Java
     http://buff.ly/16Pn8BU"
b = "A video game designed to teach elementary students and
     above how to program in Java! @UCSDJacobs Research Expo
     http://t.co/EqGQQRSvBK #ucsd"
```

[8] Other (less relevant) algorithm types such as *semi-supervised* and *reinforcement* have been omitted for brevity.

Tokens

The process of tokenisation (section 2.6.1), breaks each `String` into an array of terms. It is possible to directly compare these arrays e.g. using Jaccard Index (section 2.4.2). This simple solution provides a quick and easy comparison, but, can cause problems for some clustering techniques. For example, *K-means* (section 2.5.2) generates new centroids using the mean value of the cluster. Although we might substitute some analogous alternative (e.g., combining terms from the cluster), this is likely to be sub-optimal.

Vector Space Model

In order to represent each document as a vector, we first construct a dictionary of all the terms in the corpus:

```
D = { 0 => "uc",
      1 => "san",
      2 => "diego",
      3 => "comput",
      4 => "scientist",
      5 => "develop",
      6 => "first-person",
      7 => "player",
      8 => "video",
      9 => "game",
     10 => "teach",
     11 => "program",
     12 => "java",
     13 => "http://buff.ly/16pn8bu",
     14 => "design",
     15 => "elementari",
     16 => "student",
     17 => "@ucsdjacob",
     18 => "research",
     19 => "expo",
     20 => "http://t.co/eqgqqrsvbk",
     21 => "#ucsd" }
```

We can then represent each document as a vector, each entry containing the number of occurrences of the corresponding term in the dictionary, $f(t, d)$:

```
a = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
     1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
b = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
     1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
```

It is important to note that this model does not preserve the ordering of the terms (which could provide additional information). We can now measure the distance between these two vectors simply by taking the cosine of the angle between them:

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} \tag{2.3}$$

where the norm, $\|\mathbf{x}\|$, is calculated,

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Instead of counting occurrences (known as a *bag-of-words*), we can use alternative term weightings. Term frequency-inverse document frequency (TF-IDF) reflects a term's importance within a given corpus, and is calculated as follows:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D) \tag{2.4}$$

where normalised term frequency, tf, is calculated,

$$\text{tf}(t, d) = \frac{\text{f}(t, d)}{\max\{\text{f}(w, d) : w \in d\}} \tag{2.5}$$

and inverse document frequency, idf, is calculated,

$$\text{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \tag{2.6}$$

where $|\{d \in D : t \in d\}|$ is the number of documents containing term t.

A possible downside to vector representations is the so-called 'curse of dimensionality'; in this example we considered a corpus containing only two documents, as we expand this our vectors become increasingly sparse, wasting large portions of memory to store zeros. We can overcome this using the 'hashing trick'[9] which eliminates the need for a dictionary, storing each document as a hash from term to weighting (e.g., {`"uc":1, "san":1, "diego":1, ...`}).

N-grams

It is worth noting that if we convert each document into a set of *N-grams* (see section 2.6.1) we reduce the possible terms to a finite set, thus avoiding the 'curse of dimensionality' described above.

2.4.2 Distance Measures

We have already seen how we can compare vector representations by calculating the angle between them. Next we discuss other distance metrics for comparing documents.

Jaccard Index

A simple similarity metric, the *Jaccard index* or *Jaccard coefficient* is defined as the ratio of the size intersection divided by the size of the union:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{2.7}$$

[9] `http://en.wikipedia.org/wiki/Hashing_trick`

We convert this to a *Jaccard distance* simply by subtracting from 1:

$$J_\delta(A, B) = 1 - J(A, B) \tag{2.8}$$

We can use this measure to compare token arrays or strings themselves (e.g., as an array of characters).

Levenshtein

Levenshtein distance, often referred to as *edit distance*, computes the difference between two strings as the minimum number of *insert*, *delete* and *replace* operations required to transform one string to the other. We can recursively define $lev_{a,b}(i, j)$ as follows:

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & , \text{ if } \min(i,j) = 0 \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} & , \text{ else} \end{cases} \tag{2.9}$$

where a_i and b_i is the token/character at position i in documents a and b respectively.

We can subsequently define a normalised distance as follows:

$$\text{norm-lev}_{a,b}(i,j) = \frac{\text{lev}_{a,b}(i,j)}{\max(i,j)} \tag{2.10}$$

Levenshtein distance is usually used for shorter strings, often with marginal differences (e.g., for spell checking applications). Comparison of longer strings is possible, however usually prohibitive in computation cost (roughly proportional to the product of the two string lengths).

There are many different ways in which textual data can be represented and compared. I have covered a few popular techniques, however there are many more[10]. Selecting an appropriate representation will largely depend on the document clustering algorithms in use, as well as various performance considerations (chapter 4).

[10] `http://en.wikipedia.org/wiki/String_metric`

2.5 Clustering Techniques

Having established the background to ML and various types of problems, I now focus on document clustering specifically. I will consider a combination of both *hierarchical* and *partitional* techniques, summarised below:

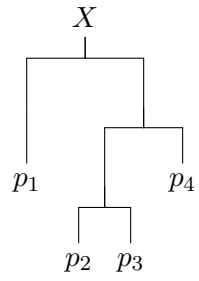
Table 2.3: Document Clustering Algorithms

Algorithm	**Citation**
K-means	Srivastava and Sahami (2009, p.164)
Spectral (using co-occurrence)	Khot (2010)
CoWeb & ClassIt	Sahoo et al. (2006)
Agglomerative (Naive/K-D tree)	Walter et al. (2008)

2.5.1 Hierarchical

Hierarchical techniques produce nested partitions, each new partition (up the hierarchy) comprised of its children. This structure is easily represented as a tree, known as a *dendrogram* (fig. 2.3). There are two basic hierarchical techniques: *agglomerative* or 'bottom-up', whereby each item starts in its own cluster before being progressively merged; *divisive* or 'top-down', all items beginning in a single cluster before being recursively bisected.

Figure 2.3: Dendrogram Example



COWEB & CLASSIT

Both *CoWeb* and its derivative *ClassIt* are incremental *hierarchical* clustering algorithms. *CoWeb* incrementally sorts observations into a classification tree, each node labelled by a probabilistic concept. The application of *CoWeb* to document clustering as described in Sahoo et al. (2006) provides the added advantage of finding the most significant terms in each cluster using various distribution models.

Algorithm 2.1 CoWeb (Sahoo et al. 2006, p.6)

Input: A new sample (item) to cluster, and a reference to the root node.
Output: A cluster hierarchy containing all items.

Update the attribute value statistics at the root.
if root is a leaf node **then**
  return the expanded node that accommodates the new item
else
  Find the best child of the root to host the item and perform the qualifying step (if any) among the following:

1. Create a new node for the item instead of adding it to the best host, if that leads to improved Category Utility.
2. Merge nodes if it leads to improved Category Utility and call *CobWeb*(item, merged node)
3. Split node if it leads to improved Category Utility and call *CoWeb*(item, root)

end if
if none of the above steps are performed **then**
  return *CoWeb*(item, best child of root)
end if

At the centre of the algorithm is the quality measure, *Category Utility*. In order to apply the algorithm to numerical attributes (and thus text), we make the assumption that attribute values (e.g., term frequencies) follow a Normal distribution, (Sahoo et al. 2006, pp.8). Let $C_1, \ldots, C_K$ be the child clusters of cluster C_p. Equation (2.11) computes the Category Utility of $C_1, \ldots, C_K$.

$$CU_p[C_1, \ldots, C_K] = \frac{\sum_k P(C_k) \sum_i (\frac{1}{\sigma_i k} - \frac{1}{\sigma_i p})}{K} \tag{2.11}$$

where:

$$P(C_k) = \text{the probability of a document belonging to cluster } k, \text{ given that it belongs to the parent cluster } p$$
$$\sigma_{ip} = \text{standard deviation of the value of attribute } i \text{ in parent node } p$$
$$\sigma_{ik} = \text{standard deviation of the value of attribute } i \text{ in the child node } k$$

Classit is the algorithm derived from using *CoWeb* with the assumption that attribute values follow a Normal distribution (as in 2.11). One downside to this approach is that attributes commonly used to represent text documents usually follow a *skewed* distribution. Additionally, a Normal distribution assumes real-valued attributes, whereas frequency counts (e.g., bag-of-words) are natural numbers.

Potential advantages of applying such an algorithm to the task of document clustering lie in the application of a more suitable distribution to the Category Utility function. This is useful in judging the information content of a term, and thus inferring the topic of a given cluster. Another more interesting characteristic of *CoWeb* is its ability to cluster new information without repetition of the existing clustering. This is a great property when considering applications

involving real-time data; new information about a topic could be fetched to 'enhance' an existing set of results. Indeed, Sahoo et al. make reference to this incremental property with regard to the tasks of *Retrospective Topic Detection*, *On-line New Event Detection* and *Topic Tracking*: "Retrospective topic detection involves detecting new events in the already collected set of documents. On-line new event detection involves identifying a new event, e.g., an earthquake or a road accident, in a new document. Tracking involves keeping track of the evolution of an event by assigning the incoming news stories to their corresponding events." (Sahoo et al. 2006, p.4).

Agglomerative

Agglomerative methods generate a *hierarchical* partition working from the 'bottom-up'. Here we discuss the naive $O(n^3)$ algorithm, as well as various optimisations giving reduced complexity.

Algorithm 2.2 Naive Agglomerative (Walter et al. 2008)

Input: A dataset $X = \{x_1, \ldots, x_n\}$ to cluster.
Output: A partition of X, $\Pi_k = \{\pi_1, \ldots, \pi_k\}$ into k clusters.

```
active ← ∅
for d ∈ X do
    active ← active ∪ { new Cluster(d) }          ▷ Initially each sample is a cluster.
end for
while |active| > 1 do
    bestDelta ← ∞
    left, right ← NULL
    for a ∈ active do                             ▷ Finds the closest pair (left,right).
        for b ∈ active do
            if a != b and distance(a, b) < bestDelta then
                bestDelta ← distance(a, b)
                left ← a
                right ← b
            end if
        end for
    end for
    active ← active \ {left, right} ∪ { new Cluster(left,right) }   ▷ The 'merge' step.
end while
return active
```

The outer loop invariant ($|\text{active}| > 1$) ensures the algorithm continues merging until the root of the dendrogram is reached. One possible modification to this may be to stop at a given k value, (e.g., $|\text{active}| \geq k$). Advancing this, exiting the loop if some distance threshold is met, (e.g., $|\text{active}| > 1$ **and** bestDelta $<$ threshold) would enable a variable number of clusters to be produced. This naive approach also has the nice property of working on string representations directly, assuming the distance function compares clusters themselves.

Walter et al. (2008) propose two major enhancements over the naive approach; first, a *k-d tree* to accelerate searching for the nearest cluster, second, a *min-heap* to maintain the best match for each 'active' cluster along with its distance value. The former replaces the 'active' set in the naive example, providing analogous operations for insertion/retrieval in $O(log\ n)$ time. The latter also allows us to retrieve the closest pair of (any) clusters in the same time. Walter et al. also proposes a *locally-ordered* version of the algorithm which removes the requirement for

a *min-heap* based on the assumption that *distance* function obeys a *non-decreasing* property, defined as: $d(A, B) \leq d(A \cup C, B)$. This new version of the algorithm performs merges in order of locality, proceeding through each cluster's nearest neighbour.

Both improved versions yield significant speed-up over the naive approach; with empirical performance better than $O(n^2)$, and close proportionality to input size (Walter et al. 2008, p.6).

A potential hurdle in achieving this improvement is the requirement to represent samples in Cartesian space (for insertion into a *k-d tree*). *BK-trees*, proposed by Burkhard and Keller (1973) offer one possible solution, enabling similar search in metric space.

Another approach involves constructing a Finite state automaton recognising precisely the set of strings within a given *Levenshtein distance* (section 2.4.2) of a target word (known as a 'Levenshtein automaton'[11]). By representing our samples in this way we can efficiently search 'nearby' samples by intersecting their automata. Tested on a dictionary of nearly 235,000 words, this approach gave significant reductions in lookup operations (648 vs 58,928 for a string of length 5 with an 'edit range' of 2) (Johnson 2010).

Although the potential exists to implement alternate data structures, allowing more efficient searching/lookup, it is important to offset this cost with that of the initial setup time of such structures (amortisation). Indeed the cost of constructing a Non-deterministic finite automaton (NFA) and then converting to an equivalent Deterministic finite automaton (DFA) is exponential in the worst case[12]. As our proposed system will likely not persist data structures between users/sessions, such long-term savings provide little benefit.

2.5.2 Partitional

Partitional techniques create a non-nested partitioning of the data, e.g. if K clusters are desired, a partitional approach typically finds all K clusters simultaneously (cf. bisecting/merging).

K-means

K-means is one of the most widely documented *partitional* clustering algorithms. First proposed in 1957 (Lloyd 1982), the standard algorithm works by iteratively refining clusters through an updating or 're-centring' step:

Algorithm 2.3 K-means (Srivastava and Sahami 2009, p.164)

Input: A dataset $X = \{x_1, \ldots, x_n\}$ to cluster, k: the number of clusters to find.
Output: A partition of X, $\Pi_k = \{\pi_1, \ldots, \pi_k\}$ into k clusters.

```
repeat
    for i = 1 → |X| do
        Assign x_i to the nearest cluster π_j, where nearness is
        measured in terms of distance from x_i to centroid μ_j.
    end for
    Recalculate centroids μ_1, ..., μ_k according to (2.12)      ▷ The 're-centring' step.
until convergence
return the final partitions
```

[11] `http://en.wikipedia.org/wiki/Levenshtein_automata`

[12] Construction of an NFA in $O(kn)$ where k is the edit distance and n is the length of the target word. Conversion to DFA worst case $O(2^n)$, giving $O(2^{kn})$ (Johnson 2010).

The 're-centring' or *update* step sets each centroid to the mean of the samples contained in that cluster (assuming a vector representation):

$$\mu_j = \sum_{x_i \in \pi_j} x_i / |\pi_j| \tag{2.12}$$

K-means uses a single point (the *centroid*) to represent the entire cluster. This draws parity with application to discussion data; a single tweet containing key subject terms can be used to represent any given topic. One potential issue with using *K-means* in a 'real-time' application (besides its *NP-hard* time complexity) is the 're-centring' step; using a vector representation it is simple to get a mean value (2.12), however, given the potentially infinite and sparsely populated term space seen from social data, finding an efficient alternative may be problematic. Additionally *K-means* requires prior knowledge of the number of clusters (k). Although methods exist to estimate/learn k values (Arunprabha and Bhuvaneswari 2010), this constraint is undesirable.

Spectral

Khot proposes a method which clusters words (contained in tweets) using spectral clustering on word a co-occurrence matrix, maintaining a reverse index from terms to the documents containing them. The process is summarised below (Khot 2010):

1. **Sanitise data** (including: tokenising, stemming & stopword removal)
2. **Get a vocabulary** of unigrams (e.g., all occurring terms), and create features for each tweet
3. Using these features, **create word co-occurrence matrix**, W, such that $W_{ij} = n$ if there are n tweets that contain both the features i and j.
4. **Perform spectral clustering** on the weight matrix, W, to obtain adjacency matrix, D.
5. Perform clustering to **partition resultant graph**, using reverse index to retrieve tweets contained in each term cluster.

$$W = \begin{array}{c} \\ w \\ o \\ r \\ d \\ s \end{array} \begin{array}{c} \begin{array}{ccccc} w & o & r & d & s \end{array} \\ \begin{pmatrix} 0 & \dots & & & \\ \vdots & & & & \\ & & & & \\ & & & & \\ & & & & \end{pmatrix} \end{array} \qquad D = \begin{array}{c} \\ w \\ o \\ r \\ d \\ s \end{array} \begin{array}{c} \begin{array}{cccc} d & o & c & s \end{array} \\ \begin{pmatrix} 0 & \dots & & \\ \vdots & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \end{array}$$

Khot considers a variety of Spectral methods to obtain word clusters, including k-NN and ϵ-NN, but finds using unmodified weights preferable. *K-means* is used to perform the final step, albeit with some modification in order to achieve a soft clustering, thus allowing the same term to belong to multiple clusters. For example, "obama" belongs to both "health care" and "drilling".

The premise is that, in general, the number of documents dominates the average document size (in a social context). Thus, by clustering only the terms contained in each document, we can achieve greater efficiency than standard $O(document^2)$ approaches. Indeed, the sample of

Table 2.4: Natural Language Processes

Process	Citation
Tokenisation	Jiang and Zhai (2007), J. Mayfield and P. McNamee (2005), Paul McNamee and James Mayfield (2004) & Hassler and Fliedl (2006)
Stemming	Rijsbergen, Robertson, and Porter (1980)
Named Entity Recognition	Ritter et al. (2011), Li et al. (2012) & Finin et al. (2010)
Sentiment Analysis	Tsolmon, Kwon, and Lee (2012)

1000 tweets used, yielded a vocabulary of only 1374 terms (˜1.3 new terms/tweet). Additionally, by clustering words, we are able to identify key terms for each cluster, which may be useful in the task of labelling (expanded in section 2.6.4).

In most cases document clustering concerns a fixed dataset, making direct comparisons possible (e.g., with benchmarks via *F-measure*, *Rand measure* or similar). In the context of social data, no document sets are the same, indeed even successive searches for the same query will produce varied results. This means that 'good clustering' becomes a more subjective matter; indeed "clustering is in the eye of the beholder" Estivill-Castro (2002).

2.6 Natural Language Processing

Another discipline with foundations in ML, Natural Language Processing (NLP) concerns the task of enabling computers to process natural language. Much of the work conducted in this field involves computationally intense algorithms designed to parse natural language into machine readable representations to perform inference, so-called 'deep' NLP. In contrast, 'shallow' NLP methods focus on parsing surface forms, often via more efficient approximate operations (e.g., using regular expressions). We next consider a number of useful operations in the latter category with the goal of extracting additional features from social data.

2.6.1 Tokenisation

Tokenisation, the process of splitting a string into parts (usually words), is commonly the first step of any kind of natural language text preparation. Although sounding trivial, it is actually a complex problem, even when limited to a single language. To give an example, if given the string `"Facebook sets up HQ at the old Sun Microsystems office in San Francisco"`, intuitively we see that both `"Sun Microsystems"` and `"San Francisco"` are proper nouns. However, were we to simply tokenise using word boundaries, we would miss this distinction, and also potentially introduce false positives (e.g., matching `"sun"` to the star). Another issue this example highlights, is acronyms like `"HQ"` - ideally we would like to match this with `"headquarters"`, in this case, however, we would require some prior knowledge to perform a substitution. There are many other issues relating to finding word boundaries (e.g., appropriately handling hyphenation). Below is a summary of some common approaches to tokenisation:

1. **Naive Heuristics**
   Most straightforward (as well as computationally efficient), is to harness a number of simple heuristics to determine word boundaries. The most simple (English) example is to use spaces, although more advanced methods (e.g., Jiang and Zhai 2007) also consider punctuation, quotes, parentheses etc.

2. **Rule Based**
   Rule based methods incorporate prior linguistic knowledge (e.g., grammar rules, dictionaries) to improve the quality of tokenisation. Use of dictionaries enables useful expansion, for example substituting `"HQ"` with `"headquarters"`. Hassler and Fliedl propose a system, *JavaTok 1.0*, that uses token definitions with user-defined token types to provide "[...] proper treatment of both general and language-related tokenization difficulties" (Hassler and Fliedl 2006). The requirement for prior knowledge makes rule based methods more complex to implement (requiring language-specific rules), especially when working with the types of language found in a social context.

3. **Character N-grams**
   J. Mayfield and P. McNamee (2005) describe a method which, unlike the rule based approach, uses overlapping character n-grams as indexing terms (tokens). This method removes the requirement for a dictionary, converting strings into n-grams from a (finite) set. For example, `"Facebook sets up HQ at ..."` with $n = 5$ becomes: `["␣face", "faceb", "acebo", "ceboo", ...]` etc. Unlike methods that tokenise using word boundaries, n-grams give the distinct advantage of having a finite set of possibilities, e.g. if we consider just the lowercase alphabet of 26 characters (plus space), with $n = 3$ we get 19,863 possibilities ($|\text{alphabet}|^n$). Paul McNamee and James Mayfield demonstrate that accuracy of this approach actually rivals (in some cases exceeds) that of unnormalised word boundary methods for both monolingual and bilingual applications (Paul McNamee and James Mayfield 2004, pp.77-90). It should also be noted that n-gram methods generate a larger number of tokens, requiring more space as well as making comparisons more computationally expensive.

There are a number of other techniques in use in addition to those discussed here; the task of tokenisation is closely linked to the domain, more specifically the language in use. In the case of social data it is likely (though only conjecture) that domain specific knowledge[13], in combination with a naive approach, may provide best results with relatively minimal time investment.

2.6.2 Stemming

To be able to measure distance between vectors of words we need to be able to identify and count matching terms. Figure 2.4 shows two tokenised tweets matching the query "obama gun control". From this example we can see a number of matching terms (`"Obama", "gun", "control", "in", "to"`), notice however `"push"` and `"pushes"`; ideally we would want to consider these as matches too. We need a canonical form for each term, so that when we intersect two sets of terms, all tokens with an equivalent 'meaning' are matched. *Stemming* is the process of repeatedly removing suffixes from a word to achieve a root form. The most popular stemming implementation is the 'Porter Stemmer' algorithm (Rijsbergen, Robertson, and Porter 1980).

[13] `https://support.twitter.com/articles/166337-the-twitter-glossary`

Figure 2.4: Token Comparison

```
a = ["Obama", "pushes", "gun", "control",
     "in", "CT:", "President", "Barack", "Obama",
     "is", "trying", "to", "boost", "the",
     "chances", "of", "gun"]
b = ["Obama", "in", "Conn.", "to", "push",
     "for", "gun", "control", "as", "critical",
     "week", "begins", "in", "Congress"]
a & b = ["Obama", "gun", "control", "in", "to"]
```

Figure 2.5: Stemmed Token Comparison

```
a = ["Obama", "push", "gun", "control",
     "in", "CT:", "Presid", "Barack", "Obama",
     "is", "try", "to", "boost", "the",
     "chanc", "of", "gun"]
b = ["Obama", "in", "Conn.", "to", "push",
     "for", "gun", "control", "as", "critic",
     "week", "begin", "in", "Congress"]
a & b = ["Obama", "push", "gun", "control", "in", "to"]
```

Figure 2.5 shows the resultant tokens after applying the Porter stemmer to our previous tweets. Note that the stemmer does not necessarily return a word's *morphological root*, however this does not really matter as long as equivalent words map to the same stem.

2.6.3 Named Entity Recognition

Named Entity Recognition (NER) refers to the task of identifying the various proper nouns contained within a given passage. Conventional Part of speech (POS) taggers rely heavily on tagged dictionaries (as well as proper formatting, e.g. capitalisation) to determine a word's tag. This returns us to the issue of vocabulary, more specifically the informal language in use on the majority of social networks. Figure 2.6 shows various abbreviations/misspellings for the word "tomorrow" that have all been used in tweets. It is clear that reliance on a standard dictionary will likely miss a large portion, if not the majority of such cases. We need an alternative. Li et al. propose an *unsupervised* system to recognise all possible entities within a collection of tweets. The system, named 'TwiNER', works by first tokenising each document into *segments*, then assigning probabilities to each, indicating likelihood to represent an entity. The article explores several alternatives for calculating these probabilities, including lookup in a dictionary constructed from a snapshot of *Wikipedia*. The system then uses these values to determine candidate entity names based on a number of probabilistic models (Li et al. 2012). When compared to conventional methods such as *Stanford-NER* and *LBJ-NER*, TwiNER achieves significant improvement in both *precision* and *recall*. This is especially impressive when we note that both are *supervised* methods (requiring prior training), whereas TwiNER is *unsupervised*.

Finin et al. offer an alternate approach, *crowdsourcing* named entity annotations for tweets. Workers were presented with sample tweets and asked to mark each term as *person*, *organisation*, *location* or *none*. Each response was then checked for agreement to remove anomalies (e.g., workers who randomly selected inputs). A 'gold standard' set of annotations was also used to

Figure 2.6: Social Language (Ritter et al. 2011)

```
["2m", "2ma", "2mar", "2mara", "2maro", "2marrow",
"2mor", "2mora", "2moro", "2morow", "2morr", "2morro",
"2morrow", "2moz", "2mr", "2mro", "2mrrw", "2mrw", "2mw",
"tmmrw", "tmo", "tmoro", "tmorrow", "tmoz", "tmr", "tmro",
"tmrow", "tmrrow", "tmrrw", "tmrw", "tmrww", "tmw", "tomaro",
"tomarow", "tomarro", "tomarrow", "tomm", "tommarow",
"tommarrow", "tommoro", "tommorow", "tommorrow", "tommorw",
"tommrow", "tomo", "tomolo", "tomoro", "tomorow", "tomorro",
"tomorrw", "tomoz", "tomrw", "tomz"]
```

check responses as they were made, alerting workers if their answers did not match. The article demonstrates the power of *crowdsourcing*, albeit on a comparatively small sample (Finin et al. 2010). Unlike the *unsupervised* method described above, the *crowdsourced* approach also enabled entity types to be recorded (e.g., *person* vs *location*). Despite being little use in a real-time context, *crowdsourcing* still provides a great asset, for example in training another classifier (e.g., Locke and Martin 2009), or establishing a dictionary of domain-specific terminology.

Despite being an established area of NLP work, NER with application to social data is still relatively young. The work done by Li et al. as well as others (e.g., Ritter et al. 2011) shows promise. Being able to accurately identify entities within a tweet opens many opportunities, for example up-weighting these terms during clustering to improve overall quality. Also, tasks such as labelling of clusters and search filtering might benefit from such analysis.

2.6.4 Sentiment Analysis

Sentiment Analysis describes the task of deriving sentiment from a given passage. For example the tweet, "Failed my driving test" has a negative sentiment. This area is probably one of the most investigated areas with respect to social networking due to its commercial applications (e.g., monitoring reaction to a product or brand). It also has interesting implications for search, for example aggregating overall opinion on a given cluster would allow users to gauge reaction to a given story.

Interestingly, Twitter itself provides this functionality as part of its search API. The search modifiers ':)' and ':(' allow tweets with either a positive or negative sentiment to be retrieved[14]. There are also numerous free and commercial APIs[15] for analysing sentiment of tweets.

Tsolmon, Kwon, and Lee (2012) propose a novel system which enables extraction of events based on sentiment. The concept uses significant term shifts and rapid changes in term frequency distribution to identify user commentary on any given event. When evaluated on a set of 30 events spanning six different topics, the system was able to accurately identify events based on tweets posted during that period. If we parallel this idea to the events of January 15, 2009 and Flight 1549, we can see why this type of event detection is significant. The ability to recognise these patterns in historical data could also enable searching of past events.

[14] `https://dev.twitter.com/docs/using-search`

[15] e.g. `http://help.sentiment140.com/api`, `http://web.viralheat.com/sentiment-api/` available and `http://opani.com/help/twitter-sentiment`

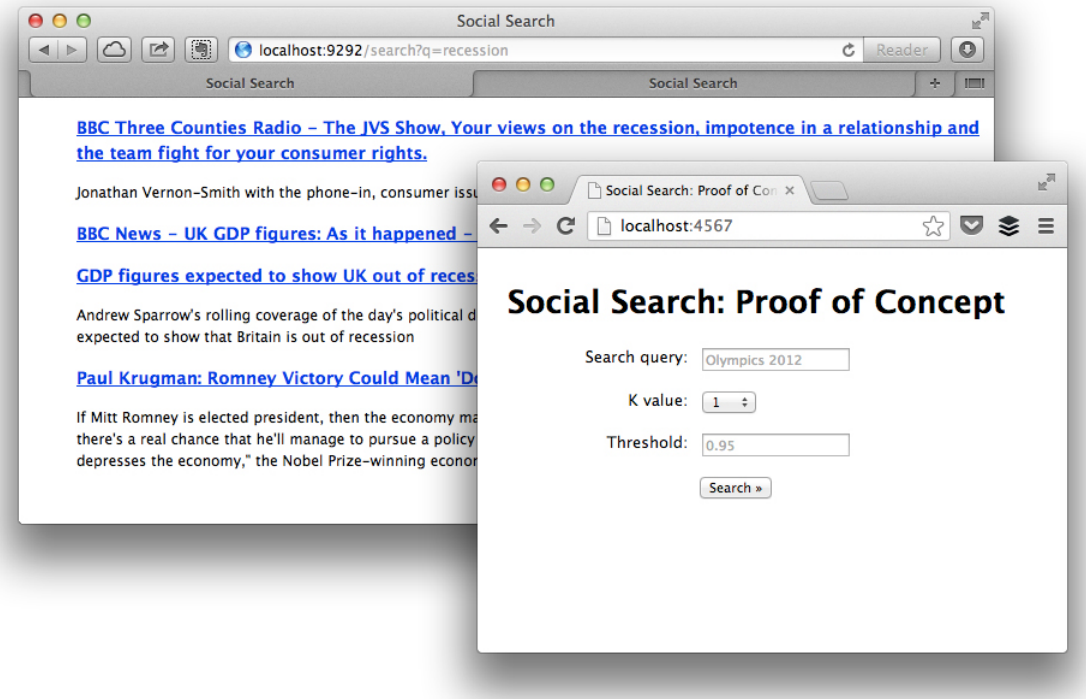
It is clear that in order to extract the maximum amount of information from social data, leveraging of NLP techniques is crucial. It is also apparent that social data presents a wealth of problems, requiring many conventional NLP techniques to be re-evaluated, and new approaches developed. Because users frequently abbreviate their posts in order to fit within a specified character limit, finding a normalised representation of meaning becomes a complex task. In our proposed system the need for efficiency prohibits the application of deep parsing algorithms, however, many shallow NLP techniques will aid normalisation as well providing additional metadata.

3 Prototyping

Every element of the system was developed in an iterative manner; beginning with a minimal proof of concept, built shortly after the project's inception. This initial concept, serving to evaluate user behaviour, also provided a basis for further advancement described in this chapter.

In this section I briefly describe the basic design of my initial prototype, along with the user study, carried out to gauge reaction to the system and direct the subsequent development.

Figure 3.1: Early Prototypes



3.1 Implementation

The prototyping stage provided an opportunity to experiment with a variety of implementation technologies. The basic development stack consisted of:

- **Language:** Ruby (`http://www.ruby-lang.org/en/`) provided fast deployment and extensibility via its large package library (`http://rubygems.org/`).

- **Web Server:** a combination of Rack (`http://rack.github.io/`), Thin (`http://code.macournoyer.com/thin/`) and Shotgun (`https://github.com/rtomayko/shotgun`) for fast redeployment.
- **Frameworks:** Sinatra (`http://www.sinatrarb.com/`) was used to provide a Model-view-controller (MVC) architecture.
- **Libraries:** Twitter (`https://github.com/sferik/twitter`) and Levenshtein (`https://rubygems.org/gems/levenshtein`)

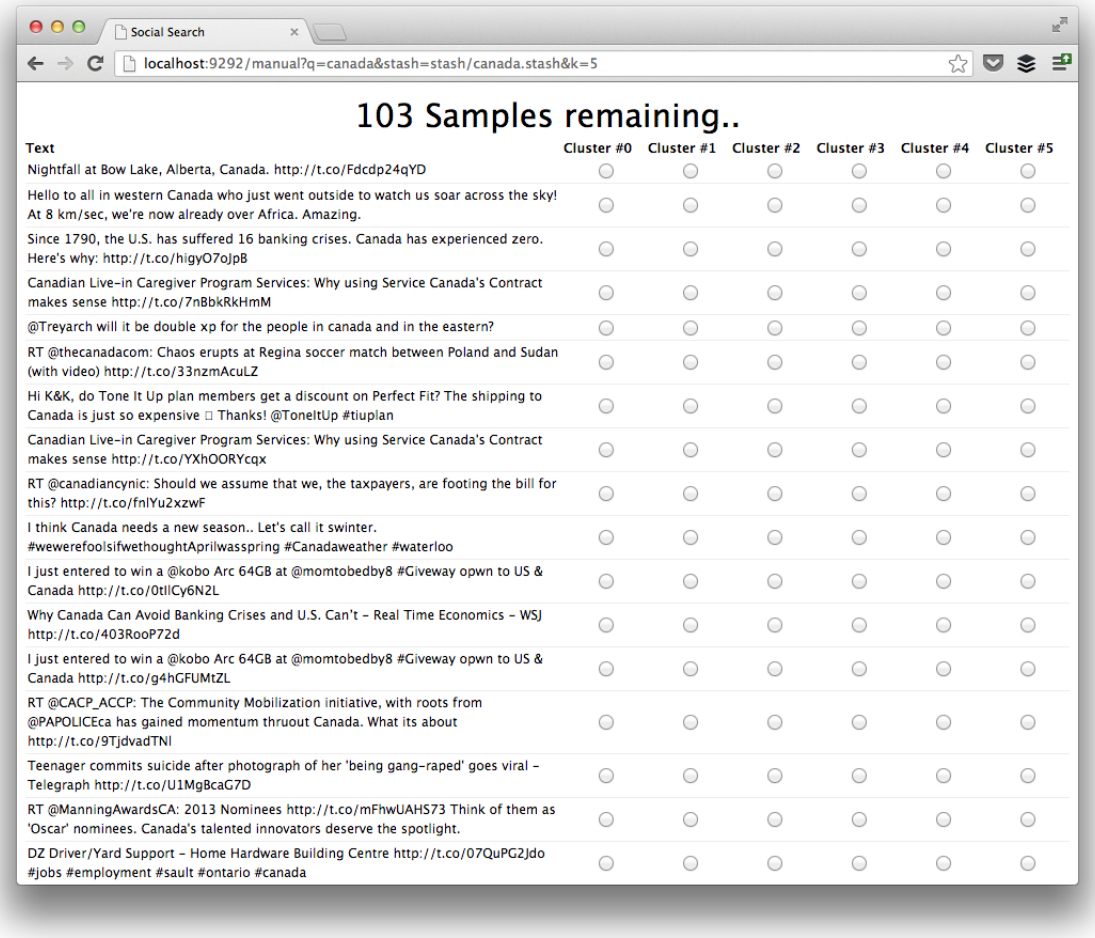
The basic system comprised two views: a *search* (input) and a *results* page. The code was contained within two simple classes: an application class, extending `Sinatra::Base` to provide handlers for *HTTP* requests; and a clusterer class (*K-means* in this example), providing all the document clustering logic.

3.2 'Gold Standard' Evaluation

As part of my efforts to develop KPIs, I developed a system for producing and comparing to 'gold standard' test data (Manning, Raghavan, and Schutze 2008, p.356). Extending the `Clusterer` class I provided a analogous manual clustering interface, allowing a sample of data to be clustered by an expert user. This instance of clustered samples was then saved via object marshalling[1] for later comparison with another clustering algorithm. Comparisons were made by matching clusters based on commonality (Jaccard index), then calculating the percentage of 'incorrectly' classified samples. Figure 3.2 shows the manual clustering interface.

[1] `http://ruby-doc.org/core-2.0/Marshal.html`

Figure 3.2: Manual Clustering
Samples (left) are matched to possible clusters (right).



3.3 User Study

Having investigated a number of KPIs used in search (Manning, Raghavan, and Schutze 2008, pp.151-175), as well as implementing my own 'gold standard', I quickly realised the need for more qualitative measures. After shifting focus I started further research in the areas of Human-computer interaction (HCI) and User Experience (UX). Stanford's online HCI course (Klemmer 2010) describes evaluation via *participant observation*, *interviewing*, *designing studies* and *running web experiments*. Expanding on this, Russell (2007) presents further analysis of user behaviour specific to search, providing three distinct views:

1. **Field studies** (*Meso*: mid-level observations)
   e.g. Getting out to see what reality is.

2. **Eyetracking studies** (*Micro*: low-level details)
   e.g. Studies in the microscopic.

3. **Session analysis** (*Macro*: millions of observations)
   e.g. What are people doing in logs, bring outside behaviour to where we can see signals.

Russell demonstrates (unsurprisingly) that users tend to consider the first two results more thoroughly, trusting the ranking mechanism to provide relevance. More interestingly, through log analysis (*macro*), a number of behavioural patterns emerge (fig. 3.3).

Russell also defines the following *mental model* of a searcher (p.50):

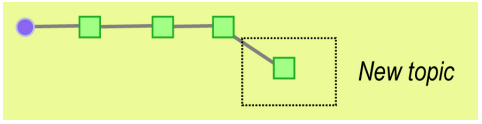
1. **Predictable Behaviour** *Can I predict what will happen when I do X?*
2. **How is content indexed?** *Is it full-text? How are images indexed?*
3. **How does Google look it up?** *Which keywords should I pick?*
4. **How are the results ranked?** *What does the order mean?*
5. **What's in the index?** *What kinds of documents can I search?*

Building upon this model, I devised a qualitative user study to determine equivalent behaviours in a social context. Specifically, I wanted to gain a better insight into how users would use the prototype, and how I can measure performance in subsequent releases (via *macro*-based KPIs).

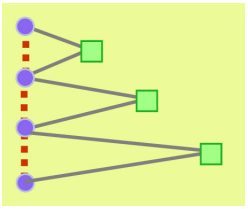
Figure 3.3: Search Behaviours (Russell 2007, p.38)

(a) Topic Exploration

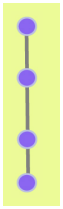
(b) Topic Switch



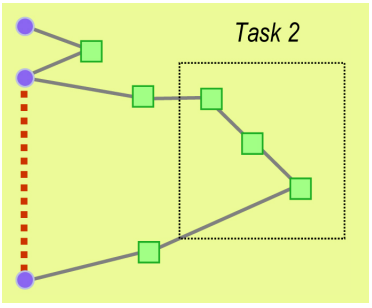
(c) Methodical Exploration of Results



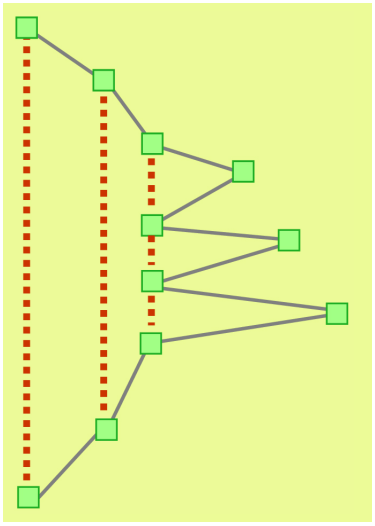
(d) Query Reform



(e) Multitasking



(f) Stacking Behaviour



3.3.1 Design

Before deciding the format of the study, I set out the major motivations in consultation with Dr. Caroline Jay[2], a HCI specialist working at the University's *Web Ergonomics Lab*[3]:

- Understanding user behaviour: types of queries, evaluation of results etc.
- Gauging reactions to the User Interface (UI) to determine possible improvements.
- Defining valid performance measures pertaining to social search engines.

Dr. Jay recommended a combination of informal interviews with a recorded evaluation. I decided to provide participants with a list of *concrete* tasks (Klemmer 2010, Lecture 7.3), broken down into two categories: 'topical', those that referred to recent events (e.g., 'why is X in the news?'); and 'factual', which usually had a single objective (e.g., 'when was X born?'). It is important to note that these were provided in the form of a task, rather than a specific query string; leaving the participant to formulate their own query(s). Each participant was given a list of 10 tasks (five of each query type), and asked to use two candidate systems (the manipulation) to perform their queries: one sourcing social data, the other direct from Google. Both systems had identical 'white label' UIs, and were used in alternate orders to *counter-balance* within subjects (Klemmer 2010, Lecture 7.2).

A questionnaire (appendix A.1) in combination with video/audio recordings was used to collect data from participants. Each participant was encouraged to "think aloud" (Lewis and Rieman 1994, Chapter 5) whilst completing tasks, providing insight into their *mental model*.

The study was carried out with approval from the School's Undergraduate Ethics Committee[4], who provided guidance on matters relating to consent, data handling and participant safety.

3.3.2 Evaluation

Based on a total of five participants I was able to gain a better understanding of how users interacted with the system, as well as their *mental models*. After observing both success and failure of the system, I was able to formulate some simple KPIs for measuring performance:

1. **# of results clicked**
2. **Time to click** i.e. $\Delta(\text{query}, \text{result click})$
3. **Returning Visitors %**
4. **Time between Searches** i.e. query reform (fig. 3.3d)

As well as developing quantitative measures, I was also able to observe a number of patterns in user behaviour. For example, participants regularly failed to get quality results with their first query using the social system, especially without prior knowledge of the project. However, when users understood they were searching social data, they were able to reformulate their queries to achieve their goal. Often this would involve 'orienteering' (Russell 2007, p.39), whereby users reduce specificity of their query to find a result covering a broader subject area (e.g., 'population of China' to 'China'). Interestingly, I also observed many users (usually the more

[2] http://www.cs.man.ac.uk/~jayc/
[3] http://wel.cs.manchester.ac.uk/
[4] http://ethics.cs.manchester.ac.uk/

technically inclined) often attempted to find the answer to their query by inspecting the results pages themselves (looking for key facts in the meta data). Significantly, users exhibiting this behaviour could benefit from a UI that exposes the social data collected during the propagation of results. This behaviour is one that most likely stems from participants previous experience with other search platforms. Other notable behaviours in this category include use of boolean connectives (e.g., 'AND', 'OR' etc.) in queries, as well as queries formatted as a question for factual tasks (e.g., 'Who was the 32nd president of the U.S.?'). It is probable that in the latter case participants where hoping to find a matching question on a Q&A site like *Yahoo! Answers* or *Quora*. The final notable behaviour was participants' grasp on the temporal nature of social data, that is, results are implicitly time scoped to be the most recent when using a social source. In many cases this arose after a participant unsuccessfully used a time modifier (e.g., 'events in *X now*'), subsequently removing the modifier to yield improved results.

The study provided invaluable feedback that shaped the rest of the system design. In many cases I was able to confirm my own assumptions, and, in a number of cases, disprove them. As well as aiding me to define a number of simple KPIs to measure future progress, I was also able to collect valuable qualitative data, providing a reference point for later work, especially in UX design.

Russell (2007, p.42) defines two search types: 'Informational/Directed/Closed' and 'Informational/Locate', in the context of the proposed system I volunteer a third type for consideration: 'Social/Temporally Limited'.

4 Application Design

Since this is a relatively unexplored software area, much of the design focus will be centred on attaining user validation via a more User-centered design (UCD) process. In order to adapt to evolving user requirements, together with constant alteration to core algorithmics, emphasis on a modular and extensible codebase will be key.

In this chapter I will describe the design of the proposed application, with emphasis on providing a platform for further development of systems of this type.

4.1 Requirements Analysis

Having defined the core concept of a 'social search engine', I now formally develop the requirements for such a system from a user perspective. I will describe a standalone system, however it is important to note that adoption of such systems will likely take the form of integration into an existing platform like we have already seen (section 1.4).

The system I propose will provide a platform for testing and evaluating different components in a 'hot-swap' manner. Rather than fixing on a single implementation, I hope to gain greater insight by iteratively improving the system whilst in use by real users. As such I will need to provide a constant feedback loop between users and myself in order to evaluate the effectiveness of each iteration. Additionally I will need a loosely-coupled codebase that emphasises modularity and enables components (e.g., algorithms or APIs) to easily be replaced in isolation, without significant modification.

I will therefore need to consider the requirements of two primary actors, namely the *End-user* and *Administrator*, with the majority of the focus (at least in terms of UX) paid to the former.

4.1.1 Use Cases

To provide context, I first consider the basic search use case, at this point not distinguishing 'social' from conventional web search.

Table 4.1: Use Cases

(a) Search Request

Use Case 1: Search Request	
Scope:	System-wide
Level:	User-goal
Primary Actor:	End-user
Triggers:	User's *information need*.
Preconditions:	User specifies search *query*.
Postconditions:	User is presented *results* view.
Normal Flow:	1. a) User enters their query into *search* field. b) User submits request by either clicking *search* control or via *return* keystroke. c) System displays *results* view: i. User clicks a result, and is directed to appropriate URL. ii. User reattempts search via persistent *search* field.
Alternate/Exceptional Flows:	2. No results: a) System displays 'no matches' failure message: i. User returns to *search* view via *back* control. ii. User reattempts search via persistent *search* field. 3. System error: a) System displays 'system error' failure message: i. User returns to *search* view via *back* control. ii. User reattempts search via persistent *search* field.

In addition to this 'core' use case, there are a number of peripheral cases relating to providing a qualitative feedback loop as well as quantitative KPI tracking.

(b) Feedback Submission

Use Case 2: Feedback Submission	
Scope:	System-wide
Level:	User-goal (subsidiary)
Primary Actor:	End-user
Triggers:	User wishes to leave feedback.
Preconditions:	User has made a *query* (i.e., is on a *results* page).
Postconditions:	User is presented *confirmation* view and response is recorded.
Normal Flow:	1. a) User clicks *feedback* control. b) System displays *feedback* view. c) User completes relevant fields; i. User clicks *submit* control. ii. User clicks *cancel* control (and input is discarded).
Alternate/Exceptional Flows:	2. Validation error: a) System displays 'invalid input' failure message: i. User rectifies erroneous fields and clicks *submit* control again. ii. User clicks *cancel* control (and data is discarded).

(c) Data Retrieval

Use Case 3: Data Retrieval	
Scope:	Isolated
Level:	Exceptional-task
Primary Actor:	Administrator
Triggers:	Admin wishes to access records (i.e., for analysis).
Preconditions:	Administrator has *authenticated* themselves.
Postconditions:	System data is provided (e.g., as a `.sql` file).
Normal Flow:	1. a) Admin selects desired data (e.g., by date range) b) Admin clicks *request* control. c) System initiates download of relevant file.
Alternate/Exceptional Flows:	2. Validation error: a) System displays 'invalid selection' failure message: i. Admin rectifies erroneous selection and clicks *request* control again. ii. Admin clicks *cancel* control (and selection is discarded).

Despite appearing trivial, providing a positive UX throughout the search process is by no means a menial task. Search engines provide a portal to the World Wide Web (WWW) for many users, often serving as their browser homepage. Google's *Enterprise Search* UX goals[1] provide a useful baseline from which to establish a set of requirements:

- **Relevancy** Clearly the most important requirement; yet no quantified measures exist. Google claim to tackle this with constant analysis and testing of user queries. Accurately measuring relevance is crucial to assessing any search engine.
- **No user manual needed** Although Google refers to a wide range of product offerings, the need for *self-describing* interfaces is certainly critical. More specific to search engines is the requirement to follow UI conventions: deviating from the norm could quickly alienate users.
- **Personalised Search Experience** A major focus for many search engines; techniques like filter bubbling are commonplace, however, the potential of 'social customisation' still remains largely unexplored.
- **The Power of Testing** Google describes the importance of ensuring any new feature provides a positive impact on UX. Interestingly this is achieved through a combination of beta releases and production tests (e.g., A/B splits).
- **Fast is Better than Slow** Google emphasises the significance of response times, claiming that even the most marginal differences (≤ 0.2 secs) affect user retention.

4.1.2 Formal Requirements

Having considered the core use cases for the system, as well as a number of other peripheral requirements, I now formulate the initial requirements of the system. I have intentionally kept detail to a minimum at this stage; it would be easy to make assumptions about various functional and non-functional requirements based on my own experiences of web search, however I want to maintain a user-centric design process.

Functional

1. *User* can make a *search request*, producing a *results* page containing zero or more *results*.
2. User can select a *result*, and be passed to the relevant *URL*.

Non-functional

1. The system will be highly *extensible*, providing all functions via an API, with all functionality divided into distinct *units*.
2. The *performance* (in terms of page load) must be comparable to that of a commercial search engine.
3. Data collected from users for analysis must be held in a secure manner, in accordance the Data Protection Act[2].

[1] `http://www.google.com/enterprise/end_user_experience.html`
[2] `http://www.legislation.gov.uk/ukpga/1998/29/contents`

4.2 Environment

Before formalising the components of the system, it is important to consider how the system will integrate with external actors in order to provide the required functionality.

Figure 4.1: Application Environment

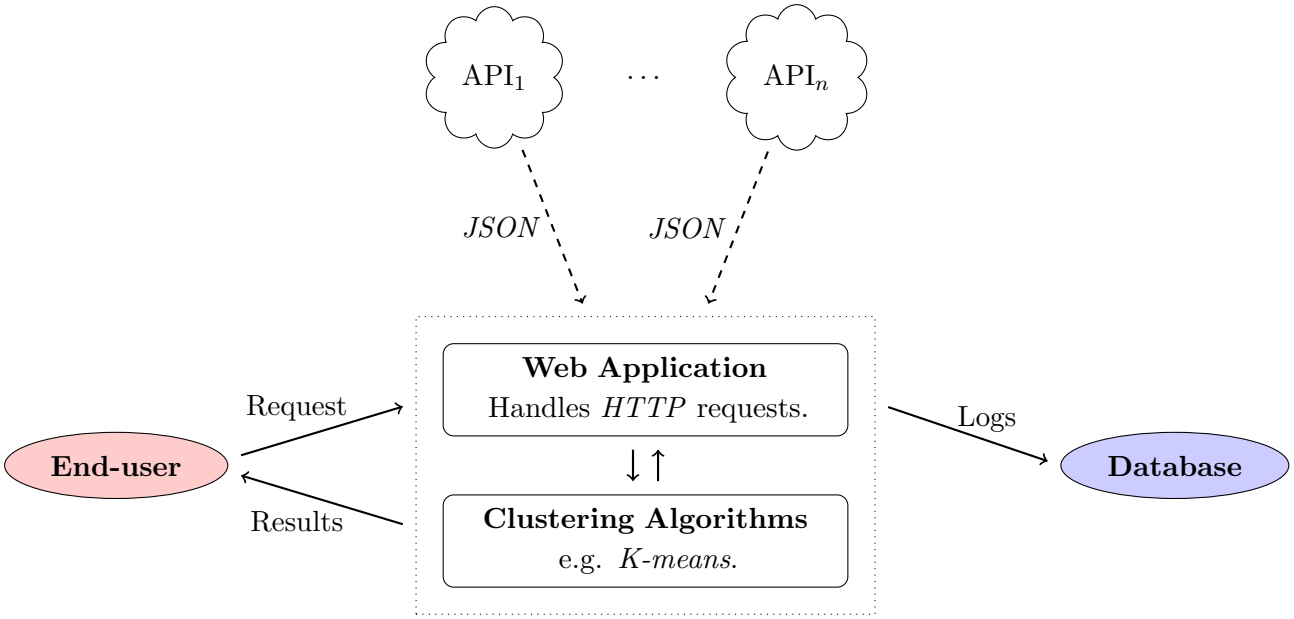


Figure 4.1 provides a coarse-grained view of the major system entities. The responsibilities of each are expanded below:

1. **Web Application**
   - Provides a UI to both the *End-user* and *Admin*.
   - Interacts with third party APIs to request/retrieve data.
   - Connects to *database* to store logging information.
   - Contains *control logic* to provision all use cases.
2. **Clustering Algorithms**
   - Provides abstractions for *Clustering*; *Distance Measures*; and *Tokenisation*.
   - Interacts with application *control logic* via a standard interface.
   - Enables 'hot-swapping' of components via loosely-coupled architecture.
3. **Database**
   - Provides persistent storage for KPI calculation data.
   - Autonomous and decoupled from other components.

In the following section I will dissect these entities into their component parts, providing a formal specification for each.

4.3 Specification

In this section I will describe the core system components in detail, breaking them down into their component classes and defining their interfaces using standard Unified Modelling Language (UML) notation.

4.3.1 Class Hierarchy

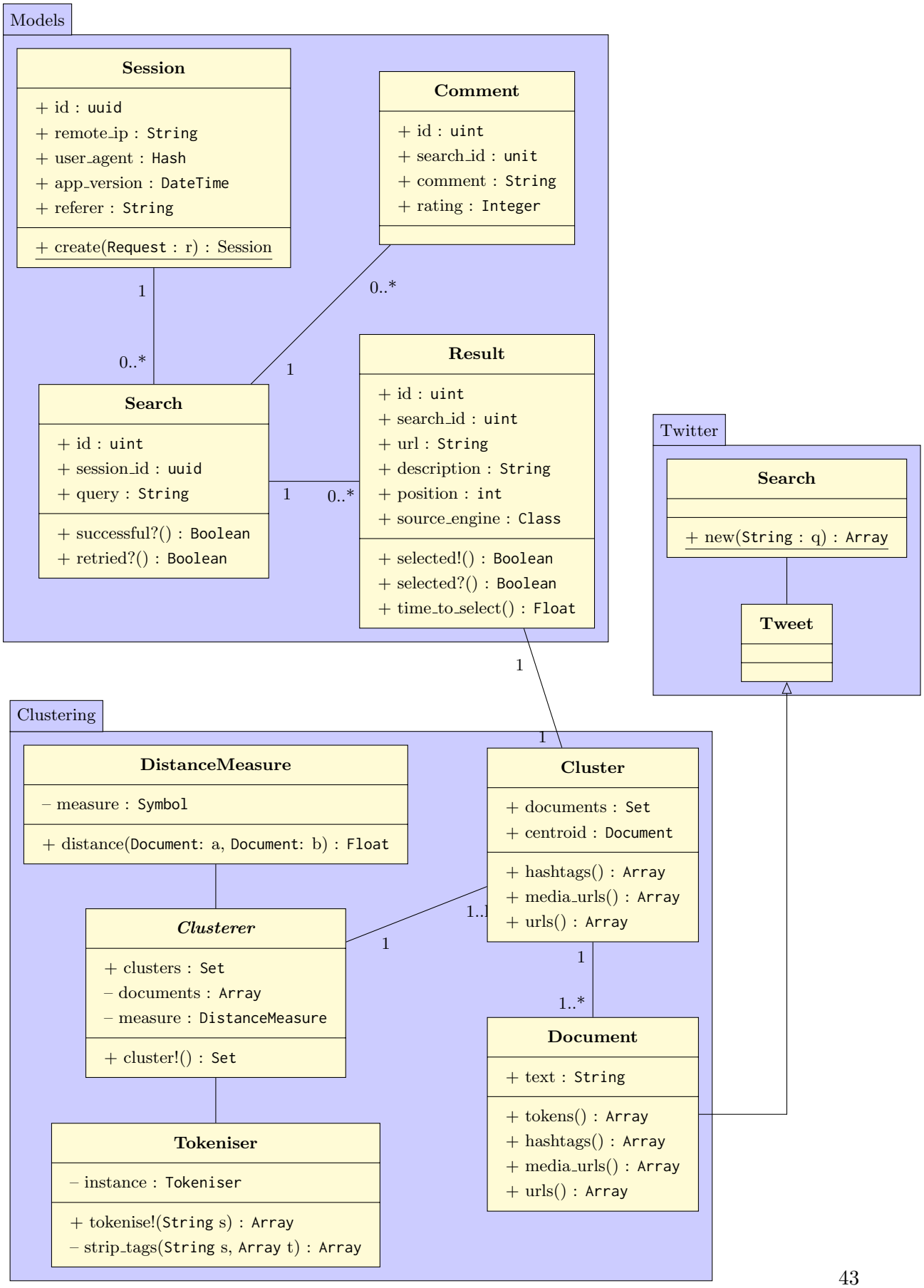
Figure 4.2 describes the overall structure of the system, including interfaces for all the external entities (e.g., APIs and Databases).

The application's classes are divided into two top-level namespaces separating application logic from the ML elements:

1. **Models**
   Map *1:1* to database tables, providing abstraction for storage/logging.
   - **Session**
     Generated from browser `session_id`, uniquely identifies a user of the application.
   - **Search**
     Created every time someone clicks *search* and associated with the current *Session* object.
   - **Result**
     Belonging to a *Search* instance, contains meta data about each retrieved URL.
   - **Comment**
     Records user feedback for a given *Search* instance.
2. **Clustering**
   Contains components required to perform document clustering (i.e., those discussed in chapter 2).
   - **Clusterer**
     Called when a new *Search* is made; performs the clustering. Implemented as an *abstract* class, extended by each implementation (e.g., KMeans `extends` Clusterer).
   - **Cluster**
     A collection of documents, includes helpful aggregator methods for generating corresponding *Result*.
   - **Document**
     Extends status instances retrieved from a social network, mapping to a relevant representation (e.g., *vector space* or *tokenised*, see section 2.4).
   - **DistanceMeasure**
     Encapsulates various distance measures, each implemented as `static` methods. Functions are then called via a *Singleton* pattern from *Clusterer* instances.
   - **Tokeniser**
     A similar implementation to above, however calls to *Tokeniser*, allow combinations of tokenisation strategies to be applied to a single *Document*.

I have also shown the external *Twitter* namespace and *Tweet* JSON wrapper to demonstrate how they interface with the application.

Figure 4.2: Class Diagram

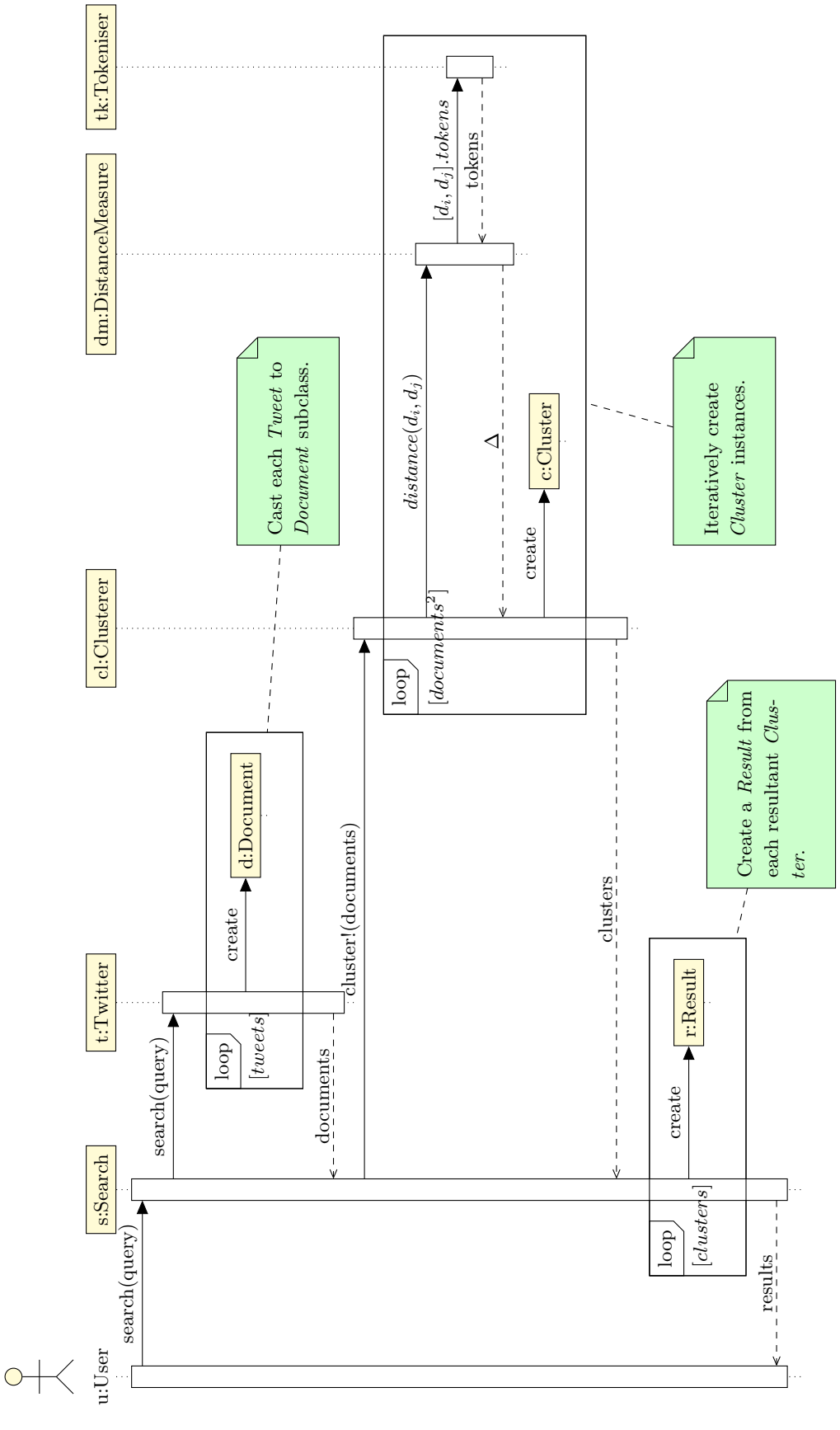


4.3.2 Data Flow

Having established the major application classes, I will now describe data flow within the system with respect to the use cases described in section 4.1.1.

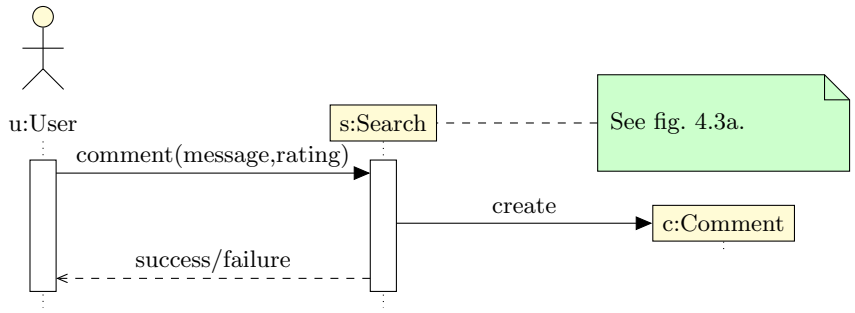
Figure 4.3a shows the interaction between classes when making a search request (table 4.1a). This includes the creation of a couple of model instances: *Search* and, *Result* (also an implicit *Session* for the User). In each case we assume these are persisted to the database via an Object-relational mapping (ORM) (discussed in more detail in chapter 5).

Figure 4.3: Sequence Diagrams

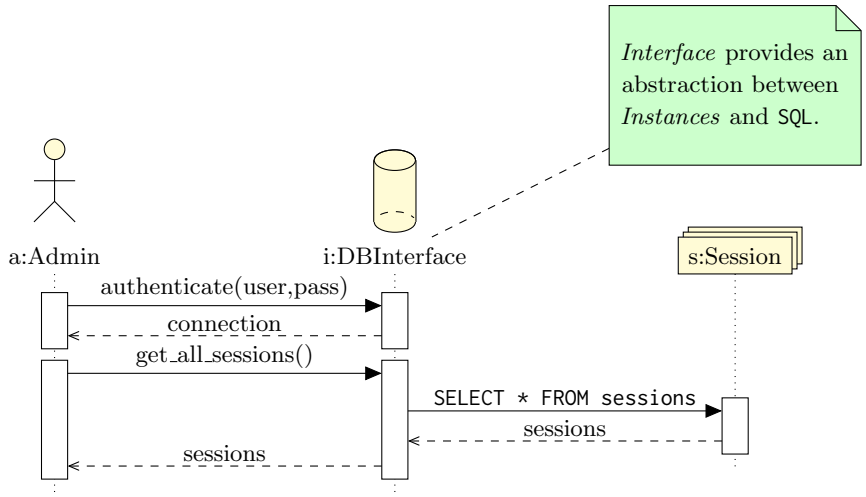


(a) Search Request

(b) Feedback Submission



(c) Data Retrieval



4.4 Methodologies

In order to experiment with a wide variety of approaches, it was important to develop new releases concurrently with those already in production. In this chapter I will describe the methodologies used during development, paying specific attention to rapid provision and constant release cycles.

4.4.1 AGILE

Completion of the initial prototype (chapter 3) was done largely via so-called 'cowboy-coding'[3]. Although great for rapid prototyping, this approach lacks structure, especially with regard to interaction with stakeholders. It was important that design decisions could be based on feedback received from users, either following on from my initial study (section 3.3) or via qualitative feedback loops built into the final application. In order to maintain such flexibility I adopted an *AGILE*[4] approach, with strong focus on the following:

- **Continuous Delivery** Instead of a fixed release date, I put the project in front of users at the earliest opportunity, and made iterative improvements thereafter.
- **Adaptation to Changing Requirements** Feedback loops built into the system allowed users to feedback at any stage (dictating requirements).

[3] http://en.wikipedia.org/wiki/Cowboy_coding
[4] http://agilemanifesto.org/

- **Frequent Releases** In order to try a wide variety of approaches I targeted a minimum of one code push per week.
- **Regular Evaluation** As well as developing in a user-centric fashion, I also used a number of evaluative measures to maintain a quality codebase (section 4.6.6).

4.4.2 TDD

With faster release cycles, the potential for regression errors becomes high. In order to maintain quality it was important to prevent any such errors reaching the production environment. I used Test-driven Development (TDD) to ensure correct functionality before releasing any new code. Each unit of code (e.g., a class or module) had a separate *specification* written **prior** to development. Listing 4.1 shows a simple *RSpec*[5] specification for a calculator application. Writing specifications in this way allows for automated regression testing before a code push, combining with version control enables erroneous commits to be automatically rejected.

```
# spec/calculator_spec.rb
describe Calculator do
  describe '#add' do
    it 'returns the sum of its arguments' do
      expect(Calculator.new.add(1, 2)).to eq(3)
    end
  end
end
```

Listing 4.1: RSpec Example

Using a Test-driven approach also enables design constraints to be quickly realised in the form of a specification, ensuring that development exactly fulfils system requirements.

4.5 User Experience

Despite search interfaces being common to many applications, and in routine use throughout the WWW, it was important not to make any assumptions about design when approaching this task. As my prototype has shown (chapter 3), many users, although familiar with a traditional search UI, struggled to perform effective queries (on social data) based on behaviours learned from such systems. In this section I discuss the task of interface design, with particular emphasis on decisions pertaining to exposing social context.

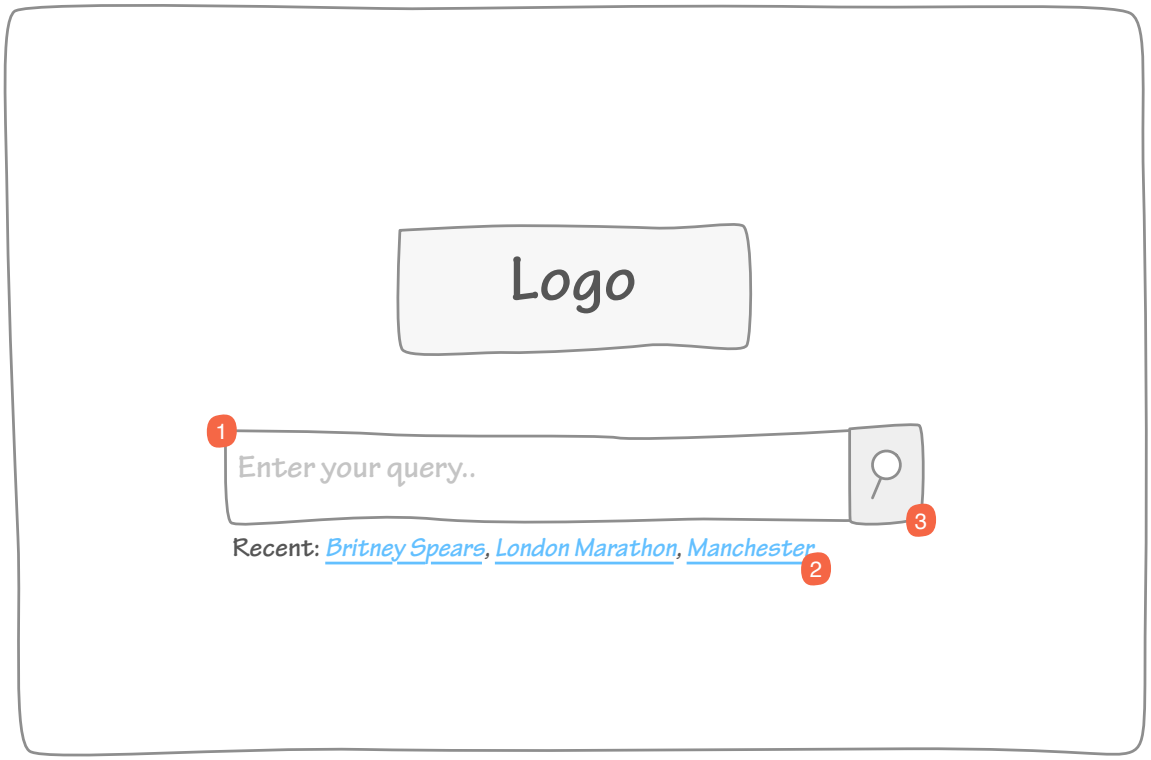
4.5.1 Wireframes

Each *view* was devised individually based on the relevant use case(s). Wireframes enable focus on the data itself, removing aesthetics from the process. Again, as with all development, UX design progressed in an iterative, user-centric fashion, with the following wireframes providing a foundation.

[5] http://rspec.info/

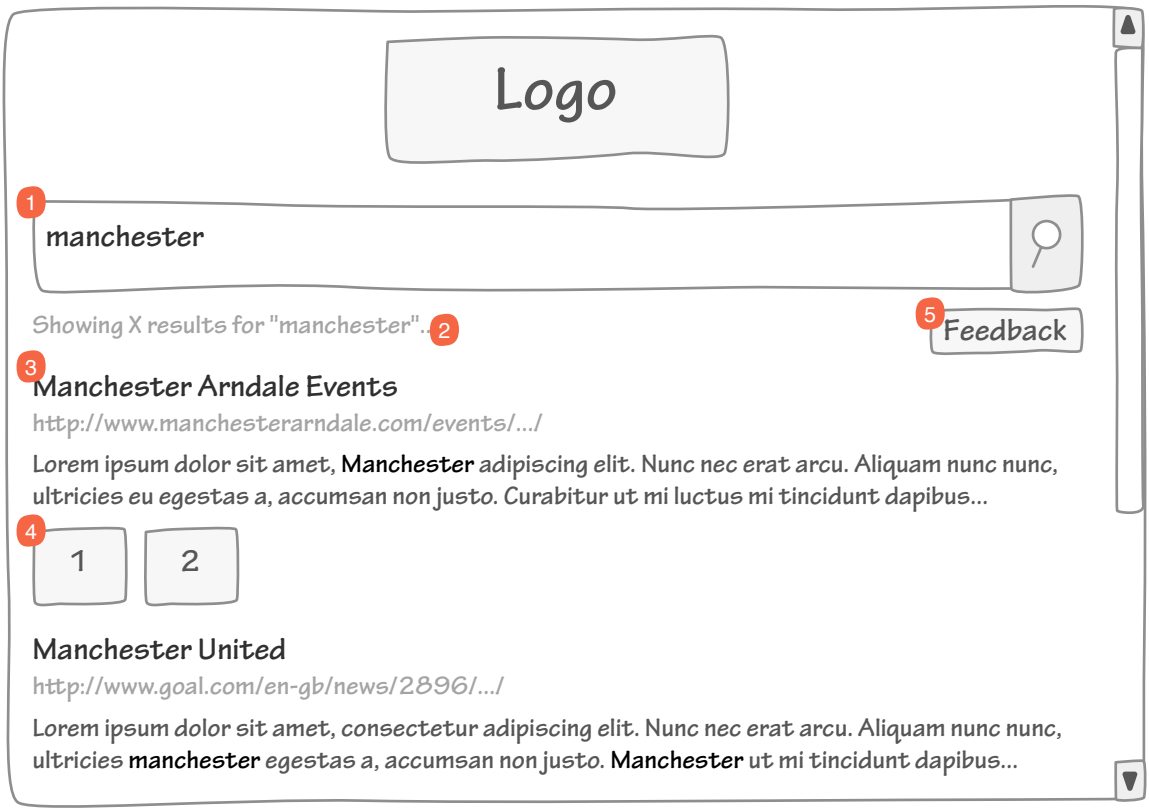
Figure 4.3: Wireframes

(a) Search Request



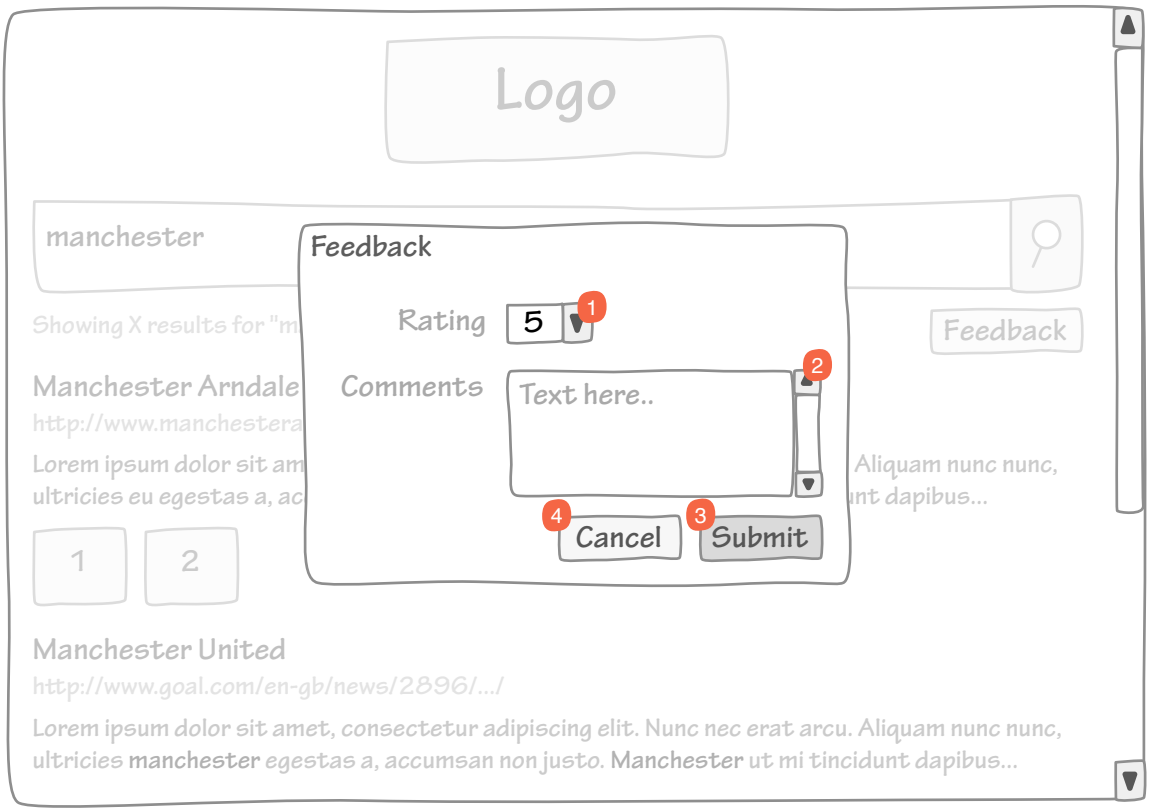
1. **Pre-populated input control** provides direction to users, would initially have *focus*.
2. **Recent searches** links to recent (successful) searches, provides users with example queries to try or aids composition of their own.
3. **Search control** primary call to action, user can submit request by clicking or pressing *return* key.

(b) Search Results



1. **Persistent search input** provides quick reference to the query, and enables refinement/reform (as in fig. 3.3d).
2. **Description** provides a summary of the search, including # of results.
3. **Result** describes result with title and description in conventional format.
4. **Images** retrieved from documents within cluster, adding social context.
5. **Feedback control** allows users to provide feedback on results.

(c) Feedback Form



1. **Rating select** provides a low-friction quantitative feedback method.
2. **Comments input** allows users to contribute detailed feedback.

4.6 Architecture

In this section I detail my choice of development architecture. I have divided the system functionality into three major concerns (roughly corresponding to the namespaces proposed in fig. 4.2). I will additionally consider the systems used to achieve continuous deployment and testing.

4.6.1 Language

Ruby[6], an open source interpreted language, was chosen for the majority of application development. Besides factors such as experience and familiarity, this choice provided a number of benefits:

- **Extensibility:** in Ruby, everything is an object[7], and any class or operation can be easily overridden/extended. The following example shows the `mean` and `mode` convenience methods I added to the Ruby standard library `Enumerable` module:

[6] `http://www.ruby-lang.org/en/`

[7] with a few exceptions: `http://rubylearning.com/blog/2010/09/27/almost-everything-is-an-object-and-everything-is-almost-an-object/`

```ruby
# enumerable_patch.rb
Enumerable.class_eval do
  def mode
    group_by{|e| e}.values.max_by(&:size).first
  end

  def mean
    reduce(&:+).to_f / size
  end
end
```

- **Libraries** as well as being highly extensible, Ruby also benefits from a wealth[8] of packages, or 'Gems' available via its package manager, 'RubyGems'. Gems can be installed in seconds, and offer anything from string distance to API wrappers.
- **Web Frameworks** *Rack*[9] is a lightweight framework for building web-based applications in Ruby. The following example[10] shows a basic 'Hello world!' Rack application:

```ruby
# hello_world.rb
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
end
```

There are a number of other reasons for choosing Ruby, many stemming from its compatibility with the rest of the development stack, discussed below.

4.6.2 Application

The core of the application is powered by Ruby on Rails (Rails)[11], a *Rack* framework developed by *David Heinemeier Hansson*. Rails is a popular MVC framework, well known for streamlining many of the common patterns required in web application development. Used by the likes of Twitter, Basecamp and Github, it is also robust and has a massive development community submitting patches on a daily basis. There are a hundreds of features that make Rails an ideal candidate for any web application. Specific to this project, is its rapid prototyping or 'scaffolding' generators. Scaffolding quickly creates the model, views and RESTful controller for a new resource, providing a working system (with JSON API) in minutes. This is great for quickly testing features during prototyping, as well as providing an initial skeleton for further development. The MVC paradigm provides a nice separation between application logic (models) and presentation (views), glued together by the controller, which determines how to handle each request. Rails provides this structure out of the box, providing scripts to generate each MVC component.

[8] 55,195 gems cut since July 2009 according to `http://rubygems.org/`
[9] `http://rack.github.io/`
[10] `http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html`
[11] `http://rubyonrails.org/`

4.6.3 Interface

As a web application, the interface will be predominantly HyperText Markup Language (HTML), with Cascading Style Sheets (CSS) used to apply styling. A number of client-side enhancements, such as form autocompletion, will be written using JavaScript (JS). To enable faster development in these languages I will be using abstractions written in Ruby which compile to their corresponding language:

- **HAML**, based on *YAML*, uses whitespace to provide hierarchy, removing the need to manually close each HTML tag.
- **SASS** is an extension of CSS which similarly allows rules to be nested to represent hierarchy. It also provides a number of programming constructs such as variables, functions and loops.
- **CoffeeScript** aids JS development by providing syntactic sugar for a number of common operations. It is especially useful when writing Object-oriented code (e.g., using a framework like *Backbone.js*).

4.6.4 Storage

Rails includes *ActiveRecord*, a Object-relational mapping (ORM) compatible with a wide variety of database engines. ActiveRecord provides a database independent abstraction; tables are generated using *migrations* which specify each table's schema (usually mapping to a model). It also provides a number of useful methods for creating/retrieving records, for example `Sessions.where(:created_at, 1.day.ago)` will retrieve all of yesterday's sessions. ActiveRecord migrations also provide a simple way to create and update database schemas, for example, the following creates the *Sessions* table (described in fig. 4.2):

```
# create_sessions.rb
class CreateSessions < ActiveRecord::Migration
  def change
    create_table :sessions do |t|
      t.string    :session_id
      t.string    :remote_ip
      t.string    :user_agent
      t.datetime  :app_version
      t.string    :referer

      t.timestamps
    end
    add_index :sessions, :session_id
  end
end
```

PostgreSQL, an open source SQL database engine was used in combination with ActiveRecord to provide persistent storage. This choice was partly dictated by the choice of deployment platform (section 4.6.6), but also because of its performance and extensibility.

4.6.5 Testing

I have already given an example of TDD using *RSpec*. RSpec, in combination with Rails, provides a number of useful components for TDD. By default, Rails creates a test for every

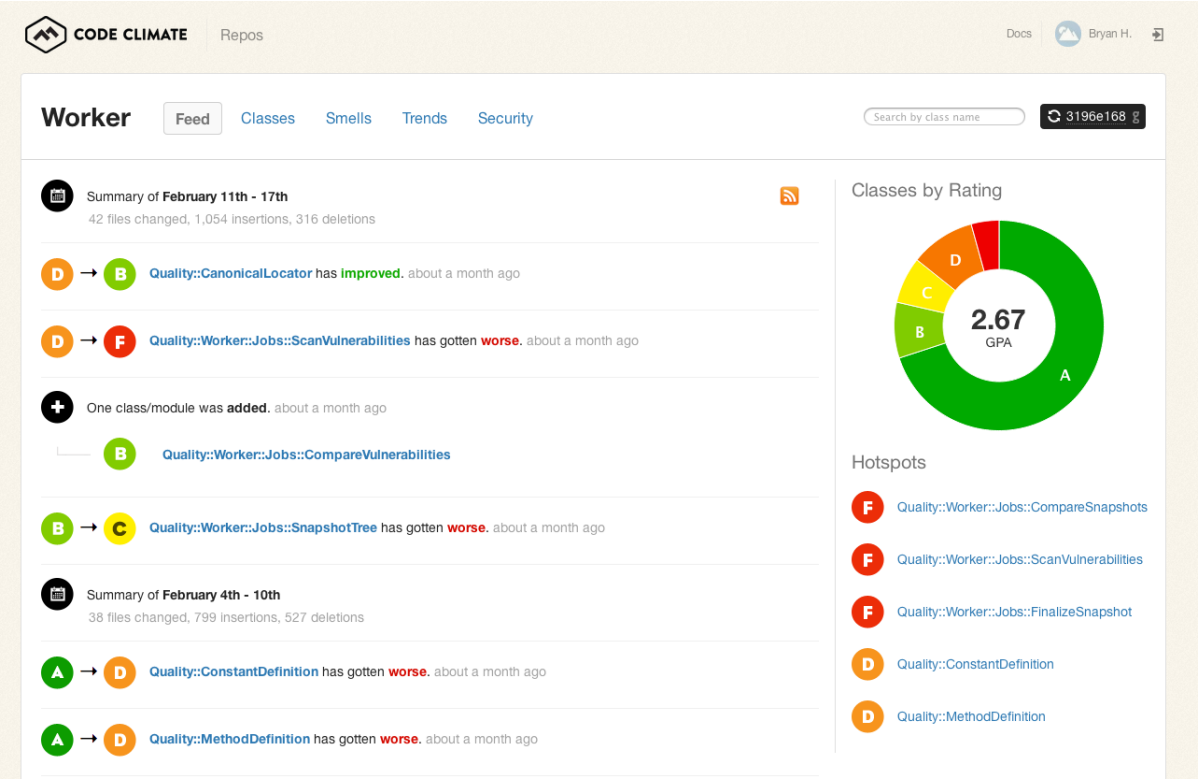
unit/class created. In addition, a set of fixtures are also created, which provide example data for running tests.

4.6.6 Deployment

As discussed in section 4.4, development focused on iterative, test-driven processes, running in parallel with a production release. In order to facilitate this, a streamlined testing/deployment environment is required. Firstly, to track changes to the codebase I used Git+ Version control (VC) combined with *Github*[12] to provide a remote repository. Github also provides an issue tracking system which allows new issues to be raised and then resolved via commit messages. I used issues to track changes in requirements resulting from user feedback, closing them when the corresponding change was pushed live. I used *Code Climate*[13], a system for tracking code quality, to track the overall status of the project. Figure 4.2 shows an example project summary in Code Climate.

The production version of the application was hosted on Amazon Web Services (AWS) via the *Heroku*[14] platform. This provided a number of advantages; firstly, Heroku uses Git+ as a deployment mechanism, meaning updating the production version could be done via single `git push` command. Secondly, Heroku's architecture enables fast and simple scaling: applications run individual processes or 'dynos', which can be quickly increased to handling traffic growth, then shutdown during quieter periods. Finally, being hosting on AWS put the application close to many web services including Twitter itself, greatly reducing latency from API requests.

Figure 4.2: Code Climate Example



[12] http://www.github.com/
[13] https://www.codeclimate.com/
[14] http://www.heroku.com/

5 Implementation

This chapter focuses on the development of the final software artefact. This process was completed in phases, somewhat analogous to 'Sprints' in the *Scrum* methodology. For the sake of brevity, I will focus on the more notable aspects of the implementation.

5.1 Clustering

When a new search request (table 4.1a) is placed, the relevant documents are first retrieved, then passed to the clustering implementation as follows:

```
# search_controller.rb
...

documents = Twitter.search(params[:query], OPTS)
clusterer = Clustering::HAC.new(documents, measure: :jaccard_distance)

# perform clustering
clusterer.cluster!

# generate results
clusterer.clusters.map{|c| Result.create(...)}

...
```

This code polls Twitter for results matching the user's query, then passes these documents to the clustering algorithm (Hierarchical Agglomerative in this case). Each cluster is then mapped to a `Result` object ready to display to the user.

The `Clusterer` implementation then simply implements the `cluster` method, which is passed the `clusters` array. Below shows the `cluster` implementation for Hierarchical Agglomerative (algorithm 2.2) clustering:

```
# hac.rb
def cluster(clusters)

  # start with everything in it's own cluster
  clusters = @documents.map{|d| Cluster.new([d])}

  while clusters.size > 1
    # closest pair of clusters clusters
    left, right, delta = clusters.combination(2).map do |a,b|
      [a, b, distance(a.tokens,b.tokens)]
    end.min_by(&:last) # min by distance

    break unless delta < THRESHOLD

    # merge closest two clusters
    clusters << Cluster.new(left.documents + right.documents)
    clusters.delete(left); clusters.delete(right)
  end

  clusters
end
```

The `cluster` method calls `distance(a,b)` which is delegated (via `Forwardable`[1]) to the selected DistanceMeasure metric via the `Clusterer` superclass:

```
require 'forwardable'
...
class Clusterer
  attr_reader :clusters

  extend Forwardable
  def_delegator :@measure, :distance

  def initialize(documents, options = {})
    @documents = documents
    @clusters  = []
    @measure   = DistanceMeasure.new(options[:measure])
  end
end
```

5.1.1 DistanceMeasure

Each class within the `Clustering` namespace was designed to be loosely-coupled, so that components can be easily replaced. To achieve this, I used a simple superclass for each component to specify its interface, for example, `DistanceMeasure` is implemented as follows:

[1] http://ruby-doc.org/stdlib-1.9.2/libdoc/forwardable/rdoc/Forwardable.html

```ruby
module Clustering
  class DistanceMeasure
    def initialize(measure = nil)
      raise "unknown distance metric, #{measure}" unless self.respond_to?(measure)
      @measure = measure || :normalised_levenshtein
    end

    def distance(*args)
      args.map(&:class).each do |klass|
        raise "#{__method__}: given #{klass}, expected Enumerable" unless klass.include?(Enumerable)
      end
      self.class.send(@measure, *args)
    end

    def normalised_levenshtein
      levenshtein(a, b) / [a, b].map(&:size).max
    end

    def levenshtein
      ...
    end
  end
end
```

This code allows new measures to be added as instance methods (e.g., `levenshtein`), checking if the relevant method exists on object initialisation. Additionally, the arguments passed to `distance(a,b)` are type checked before the relevant implementation is called. This design provides a standard interface to the caller, handling exceptions from a single point.

5.1.2 Tokeniser

The `Tokeniser` class is implemented in a slightly different manner; rather than having disjoint subclasses, each tokenisation strategy is confined to a single method. This approach allows multiple strategies to be combined depending on the arguments passed to the constructor.

```
class Tokeniser
  ...
  # Singleton
  def self.instance
    @@instance ||= new
  end

  # initializes with (custom) options hash
  def initialize(options = nil)
    @options = options || OPTIONS
    @options[:limit] ||= 5

    if @options.has_key?(:keep_only_tags)
      require 'engtagger'
      @tagger = EngTagger.new
    end
  end

  def tokenise(passage)
    passage = keep_only_tags(passage, @options[:keep_only_tags]) if @options[:keep_only_tags]
    tokens  = passage.scan(/[a-zA-Z]+/).reject{|t| t.size < @options[:limit]}
    tokens  = tokens.map(&:downcase) if @options[:downcase]
    tokens  = tokens - STOP_WORDS if @options[:remove_stopwords]
    tokens  = tokens.map(&:stem) if @options[:stem]

    tokens
  end
  ...
end
```

The default options are as follows:

```
OPTIONS = {
  lang: :en,                      # language, for initialisation of parsers
  limit: 5,                       # minimum token length
  stem: true,                     # apply stemming?
  downcase: true,                 # downcase each token
  remove_stopwords: true,         # remove stopwords?
  keep_only_tags: %w(NN NNP ...)  # remove all but these POS tags
}
```

This configuration can then be fine tuned based on user feedback/KPIs.

A modified version[2] of the *engtagger* Gem[3] provides POS tagging, which `Tokeniser#keep_only_tags` uses to reject tokens with tags not included in the `keep_only_tags` option:

```
def keep_only_tags(passage, tags = %w(NNP NN VB))
  raise "invalid tag/s supplied" unless (tags.map(&:downcase) - EngTagger::TAGS.keys).empty?
  @tagger.get_hashed(passage).delete_if{|tag,_| !tags.include?(tag)}.values.join(' ')
end
```

This method additionally checks that the supplied list of tags are valid based on those supported by the Gem[4].

Below shows an example run of the tokeniser using the default `OPTIONS` Hash:

[2] https://github.com/fredkelly/engtagger
[3] https://rubygems.org/gems/engtagger
[4] https://github.com/yohasebe/engtagger/blob/master/lib/engtagger/tags.yml

```
> t = Clustering::Tokeniser.new
> s = "Concert Window, The Netflix For Live Concerts, Gives Fans A New,
       Mobile-Friendly Venue On The Web http://tcrn.ch/10oupM1 by @ripemp"
> t.tokenise(s).uniq
  => ["concert", "window", "mobile", "friendly", "netflix", "venue", "ripemp", "gives"]
```

N.B. URLs are still accessible via `Document#urls`.

5.1.3 Document

In order to extend the tweets returned by the *twitter* Gem[5], I extended `Twitter::Tweet`, monkey patching[6] the `Twitter::SearchResults` class to map to instances of my own `Clustering::Document` class:

```
# search_results_patch.rb
module Twitter
  class SearchResults
    def statuses
      @results ||= Array(@attrs[:statuses]).map do |tweet|
        Clustering::Document.fetch_or_new(tweet)
      end
    end
  end
end
```

`Document` held additional methods, for example for tokenising the tweet:

```
class Document < Twitter::Tweet

  alias_method :to_s, :text

  # splits tweet text into tokens
  def tokens
    @tokens ||= Tokeniser.instance.tokenise(text)
  end
  ...
end
```

The `tokens` method calls the `Tokeniser#instance` singleton and applies the specified tokenising configuration.

The `Document` class also contains a number of other instance methods used in the generation of `Result` instances, for example the following method parses any *Instagram* images to URLs for use in the results view:

[5] http://rubygems.org/gems/twitter
[6] http://en.wikipedia.org/wiki/Monkey_patch

```
class Document < Twitter::Tweet
  ...
  def instagram_urls(size = :l)
    expanded_urls.map do |url|
      begin
        host, id = url.match(/http:\/\/(instagr\.am|instagram.com)\/p\/(.*)\//).captures
        "http://instagr.am/p/#{id}/media?size=#{size}"
      rescue
        nil
      end
    end.to_a.compact
  end
  ...
end
```

5.2 Back End

Once the `Cluster` instances have been created, they are mapped to `Result` models, which are associated with the user's `Session` via a `Search` record. Each of these instances are persisted to the database via *ActiveRecord* and then displayed back to the user.

5.2.1 Models

ActiveRecord allows properties, relations, validations and scopes to easily be expressed using various helper methods. For example, the `Search` class declares relations with `Session`, `Result` and `Comment` (representing foreign key constraints in the underlying `SQL`):

```
class Search < ActiveRecord::Base
  ...
  belongs_to  :session,   touch: true
  has_many    :results,   dependent: :destroy
  has_many    :comments,  dependent: :destroy
  ...
  scope :with_results,
    joins: :results,
    conditions: 'results.search_id IS NOT NULL',
    group: 'searches.id'
  scope :successful,
    joins: :results,
    conditions: 'results.selected_at IS NOT NULL'
  ...
end
```

The `scope` method defines a couple of useful search types for quick retrieval.

```
class Session < ActiveRecord::Base
  ...
  has_many    :searches,    dependent:  :destroy
  has_many    :comments,    through:    :searches
  validates   :session_id,  presence:   true
  ...
end
```

All the models have a similar preamble containing relations and validations. ActiveRecord adds convenience methods for each relation to make retrieval of associated records easier. For example, given a `Session`, `s`, we can retrieve all their searches by calling `s.searches`. Using the `has_many through:...` relation we can also retrieve all the comments belonging to that session via `s.comments`, this generates the equivalent `JOIN` in the underlying *SQL*.

Each `Comment` is then allocated a foreign key for its parent `Search` using the `belongs_to` property:

```
class Comment < ActiveRecord::Base
  ...
  belongs_to :search
  validates :rating, inclusion: { in: 1..5 }
  ...
end
```

The `Comment` class also contains validation logic for the user-submitted rating to ensure it is within the range $[1, 5]$. Any invalid input will result in a `ActiveRecord::RecordInvalid` exception being thrown (and caught by the calling controller action).

5.2.2 Controllers

When a user makes a request to a given endpoint, it is handled by the corresponding method in the controller. For example a request to `/search?query=foo` calls `SearchController#create` with `params={query:'foo'...}`. The controller than handles the request as follows:

```
class SearchController < ApplicationController
  respond_to :html, :json
  ...
  def create
    # find or init search with matching query
    @search = current_session.searches.where(query: params[:query]).first_or_create
    if @search.empty?
      # ... poll Twitter + cluster
      clusterer.clusters.each_with_index do |cluster, position|
        begin
          @search.results.create!(
            source_engine:  clusterer.class,
            position:       position,
            url:            cluster.url
          )
        rescue ActiveRecord::ActiveRecordError => error
          # handle invalid results
        end
      end
    end

    respond_with @search, include: :results, template: 'search/results'
  end
  ...
end
```

The controller first checks for an existing search with the same query belonging to the user. If none exists, it initialises a new `Search` instance and performs the clustering. In this example the

`Result` metadata (*title*, *description* etc.) is not set, this is instead retrieved from the provided URL:

```
class Result < ActiveRecord::Base
  ...
  before_validation :scrape_page, if: Proc.new { source_engine::SCRAPED }, on: :create
  ...
  def source_engine
    super.constantize
  end

  def page
    @page ||= Page.get(url)
  end

  def scrape_page
    self.url          = page.url          # gives a resolved url
    self.title        = page.title
    self.description  = page.description
  end
  ...
end
```

The `scrape_page` method is selectively called depending on the class constant `SCRAPED` within the `Clusterer` implementation. This design means that each `Clusterer` implementation can toggle this behaviour independently.

```
class MyClusterer < Clusterer
  ...
  SCRAPED = true # enable scraping of result metadata
  ...
end
```

The `Page` class uses the *HTTParty*[7] and *Nokogiri*[8] Gems to scrape the provided URL and parse the title and description, as well providing a resolved URL (cf. shortened URLs[9]).

Once the `@search` object has been populated with associated `Result` instances the view is rendered using the `respond_with` helper. Rails will either render HTML or JSON (**??**) depending on the request headers.

[7] `http://rubygems.org/gems/httparty`
[8] `http://rubygems.org/gems/nokogiri`
[9] `http://en.wikipedia.org/wiki/URL_shortening`

```
{
   "created_at":"2013-04-21T20:32:25Z",
   "id":499,
   "query":"obama",
   "results_count":7,
   "session_id":298,
   "updated_at":"2013-04-21T20:32:48Z",
   "results":[
      {
         "created_at":"2013-04-21T20:32:38Z",
         "description":"Obama sent an envoy to Venezuela dictator Hugo Chavezs ...",
         "id":1314,
         "media_urls":[
            ...
         ],
         "position":0,
         "search_id":499,
         "selected_at":null,
         "source_engine":"HAC",
         "time_delta":"2013-04-21T21:32:02+01:00",
         "title":"Obama Refuses To Send Envoy To Thatcher Funeral ...",
         "updated_at":"2013-04-21T20:32:38Z",
         "url":"http://..."
      },
      ...
   ]
}
```


5.3 Front End

5.3.1 Web Views

When a `Search` is to be rendered as HTML Rails calls the *HAML* template corresponding to the controller action and CRUD verb (e.g., `search#show`). A global layout file (`layouts/application.html.haml`) provides the surrounding layout for the view content:

```
!!!
%html
  %head
    %title= page_title
    = stylesheet_link_tag 'application', media: 'all'
    = javascript_include_tag 'application'
    = csrf_meta_tags
  %body{class: ('splash' if current_page?(root_path))}
    #header
      .container
        %h1#logo
          = link_to root_path do
            = responsive_image_tag 'logo.png'
        = display_flash.html_safe
    #content
      .container
        = yield
    #footer
      .container
        %ul
          %li= link_to 'What is this?', '/modals/about/', class: 'modal-link'
```

The layout `yield`s to the relevant view template, inserting the corresponding content (e.g., search results). The `search/results.html.haml` view loops over the results printing the relevant metadata in an ordered list:

```
#navigation
  = render 'form'
.results
  - if @search.nil? or @search.results.length == 0
    #info
      %p Sorry, I couldn't find that!
  - else
    #info
      %p Found #{pluralize(@search.results.size, 'result')} for "#{@search.query}"...
      = link_to 'Not what you\'re looking for?', "/search/#{@search.id}/modals/comment_form/"
    %ol
      - @search.results.each do |result|
        ...
          %h3
            = link_to result.title, result_path(result), data: { url: result.url }
            %time{datetime: result.time_delta} #{time_ago_in_words(result.time_delta)} ago
          %cite= truncate_url(result.url, length: 100)
          %section
            %p= highlight(truncate(result.description, length: 300), @search.query_tokens)
            - if result.has_media?
              %ul.thumbs
                - result.media_urls.each do |media_url|
                  %li
                    = link_to media_url, rel: "thumbs-#{result.id}" do
                      = image_tag media_url
```

The view also adds controls for sending feedback, as well as rendering the 'form' *partial* containing the search input controls, used across the application.

Each result contains an *anchor* tag linking to the `result_path` for that `Result`. The results controller records the click by setting `selected_at` to the current time before redirecting the user to the selected URL (`@result.url`):

```
# results_controller.rb
class ResultsController < ApplicationController
  def show
    @result = Result.find(params[:id])
    @result.selected! # record the click
    redirect_to(@result.url, status: :moved_permanently)
  end
end
```

5.3.2 CSS

Each HTML interface element is styled using CSS, rendered via the *SASS*[10] preprocessor. SASS extends CSS with a number of useful constructs including variables; functions; and nesting. These additions allow for more efficient, DRY code; the following snippet styles the error dialogues used for presenting exceptions (e.g., invalid input):

```
// search.css.scss
...
.error, .warning, .success, .message {
  color: #ddd;
  padding: 0.5em 0.75em;
  border: 1px solid;
  @include box-shadow(1px 1px 1px #eee);
  margin-top: 1em;
  font-size: 0.8em;
  background: #fcfcfc;

  h3, h4 {
    font-weight: bold;
    margin-bottom: 0.5em;
  }
}
...
```

This nesting compiles to give the equivalent rules (`.error h3, .warning h3,...`). This code also uses the `box-shadow(...)` function provided by the *Bourbon* style library[11] to produce the necessary vendor-prefixes for the `box-shadow` attribute.

SASS' ability to nest rules provides great flexibility, for example the following set of rules target mobile users:

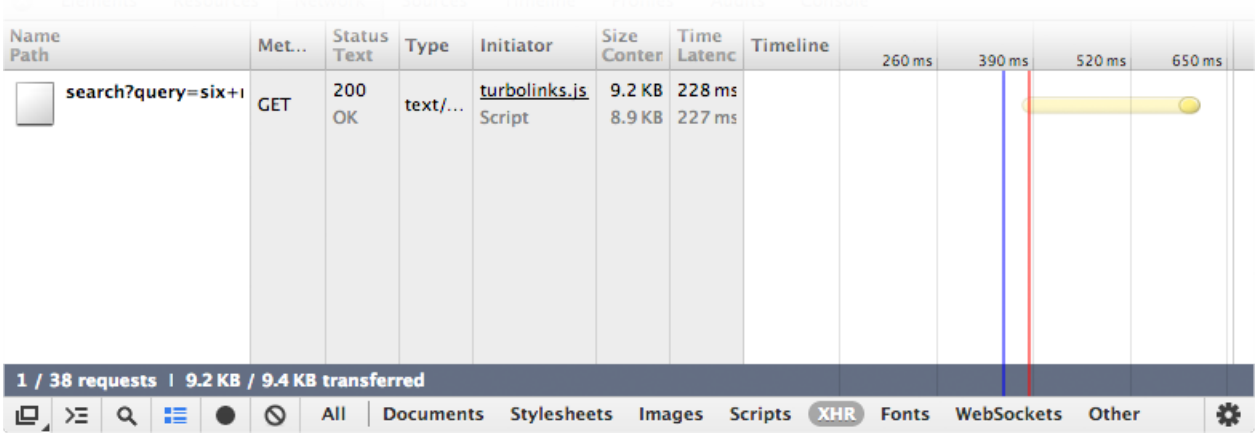
[10] `http://sass-lang.com/`
[11] `http://bourbon.io/`

```
// search.css.scss
...
@media only screen
and (min-device-width : 320px)
and (max-device-width : 480px) {
  body {
    font-size: 300%;
    margin: 1em;
    font-family: sans-serif;

    .container, .splash .container {
      width: 100% !important;

      #recents {
        /* only show 3 recents */
        li:nth-child(n+4) {
          display: none;
        }
      }
    }

    form#search button {
      font-weight: bold;
    }
  }
  .modal {
    max-width: none;
  }
}
...
```

5.3.3 JavaScript

There are a number of JS enhancements built into the front end of the application, enabling a more efficient UX. The application utilises the *turbolinks* Gem[12] (part of Rails 4.0) to speed up page loading by replacing the relevant HTML *asynchronously*. Figure 5.1 shows the HTTP request timeline for a search request (table 4.1a).

Figure 5.1: Chrome's Web Inspector showing an asynchronous search request.



[12] https://github.com/rails/turbolinks/

Instead of re-loading common assets (JS, CSS, images etc.) at the start of each request, the application removes unnecessary HTTP requests delivering a performance boost to the user.

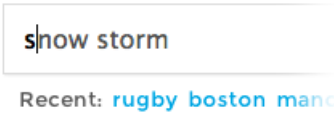
JavaScript is also used to load modal windows for search images as well as the result feedback form. The *fancybox.js*[13] jQuery library provides simple bindings for loading hidden HTML into a modal:

```
# search.js.coffee
...
$('.thumbs a').fancybox(
  type: 'image'
)

$('a.modal-link').fancybox(
  onComplete: ->
    # completed feedback
    $('form').on('ajax:success', (e, data, status, xhr) ->
      $(this).fadeOut('fast', ->
        $(this)[0].reset() # clear form
        $('#comment .success').fadeIn()
      )
    )
)
...
```

The first call binds all the thumbnail controls within the search results page, so that images load above content (fig. 5.2). The second initialises a modal for the search feedback form (fig. 4.1c), binding the `ajax:success` event (i.e., form submission) to show a confirmation message.

Twitter's own *typeahead.js*[14] jQuery library is also in use to provide autocompletion for searches (similar to Google). The library uses a custom JSON endpoint which lists the 100 most recent searches to provide autocompletion as a user types their query (fig. 5.3).

Figure 5.3: Query Completion



In addition to powering a number of third-party libraries, jQuery is also in use to replace assets with high resolution versions on devices that support this feature (e.g., tablets and smartphones):

```
# search.js.coffee
...
if window.devicePixelRatio >= 2
  $('img').each (i, img) ->
    if big = $(img).attr('data-2x')
      $('img').attr('src', big)
...
```

This code iterates over each `<img/>` tag in the HTML, substituting the `src` attribute with the path stored in `data-2x` (if exists). Figure 5.4 shows the same page on both a mobile and desktop browser (Chrome).

[13] http://fancybox.net/
[14] http://twitter.github.io/typeahead.js/

Figure 5.2: Modal Image

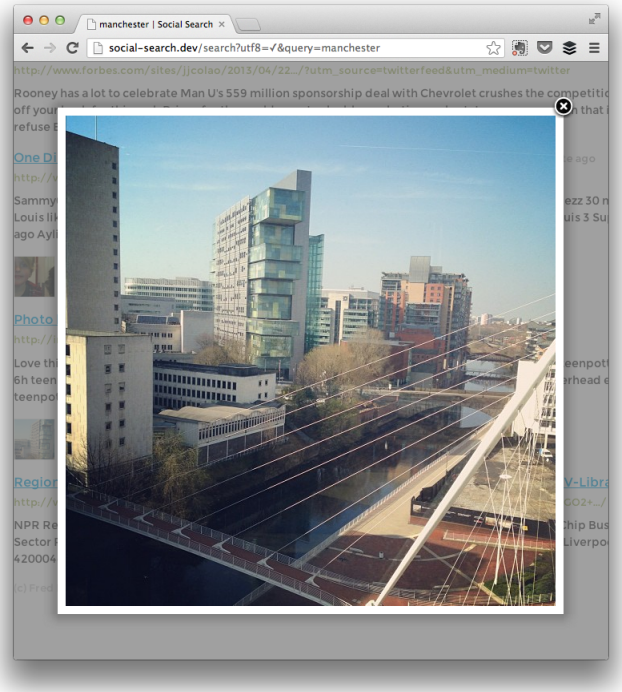
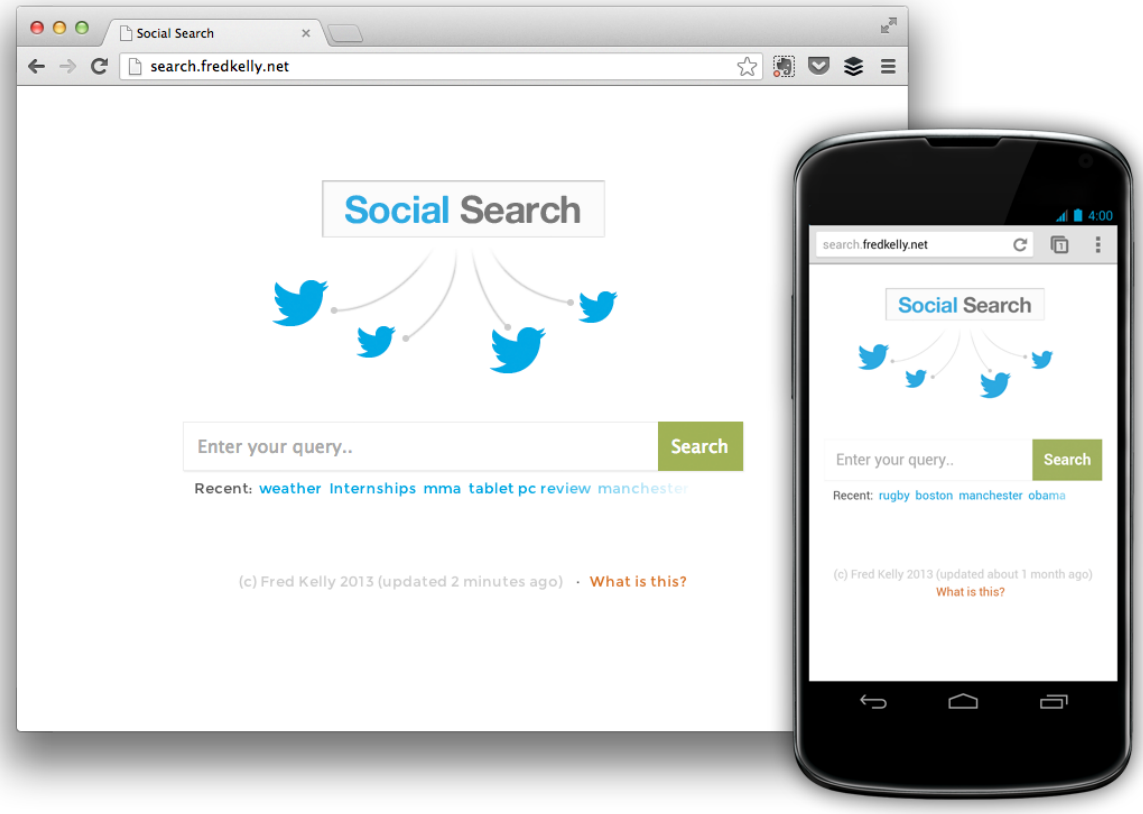


Figure 5.4: Mobile vs Desktop



5.4 Administration

The administration interface provides quick access to the data recorded via the user accessible elements of the site. The interface was built using the *active_admin* Gem[15], which provides a simple Domain-specific language (DSL) for describing interfaces which the system renders. For example the following provides an index of `Comment` instances:

```
# admin/comments.rb
ActiveAdmin.register Comment, as: 'Search Comment' do
  index do
    column 'ID', :id
    column 'Search' do |comment|
      link_to comment.search.query, admin_search_path(comment.search)
    end
    column :comment
    column :rating
    column 'Created', :created_at
  end
end
```

Similar descriptions for each model in the system allowed quick access to all the data. In addition to the standard views provided by the Gem, I also devised a number of custom components, for example a map view for each `Session` (fig. 5.5). This was achieved by using the *geocoder* Gem[16] to reverse geocode each `Session`'s `remove_ip`, these coordinates were then passed to the Google Static Maps API[17] to render the appropriate image.

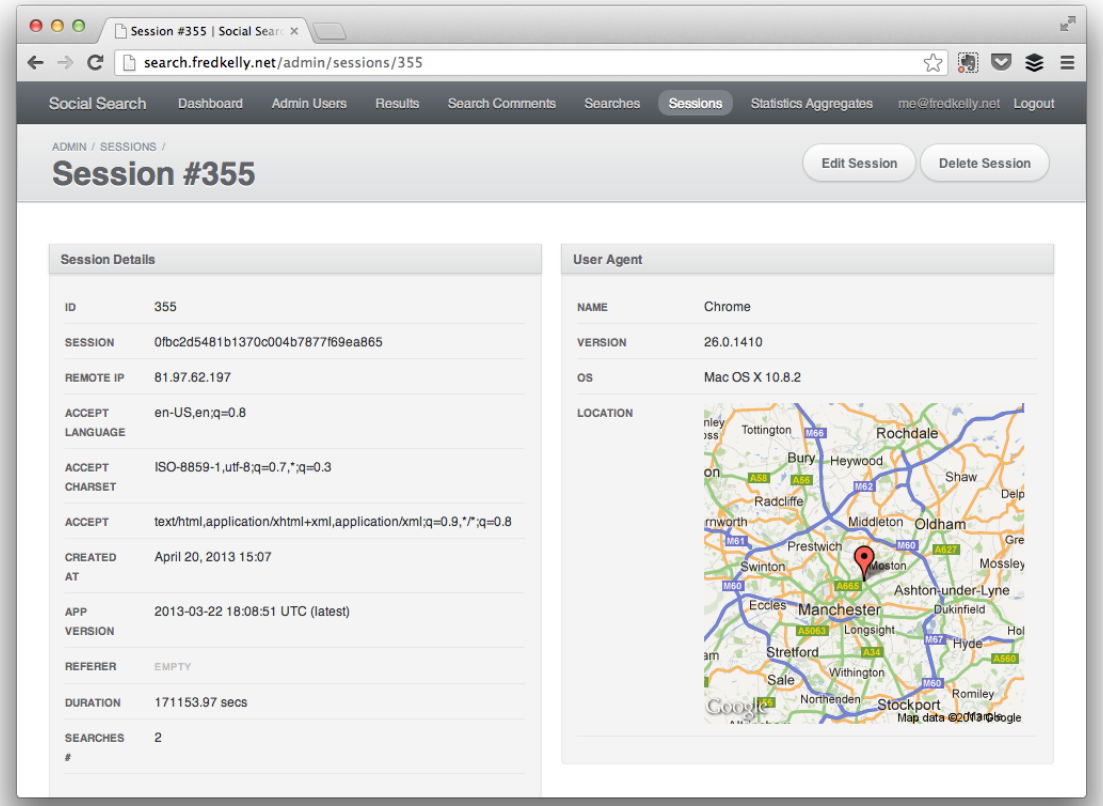
5.4.1 Statistics

In addition to providing quick access to the data recorded by the system, the administration interface also provides a number of quick statistics such as *successful searches/day* etc. In order to record this information over time without requiring computationally expensive *SQL* queries, I created an additional `StatisticsAggregate` model:

```
# statistics_aggregate.rb
class StatisticsAggregate < ActiveRecord::Base
  ...
  scope :this_week, where('created_at >= ?', 7.days.ago)
  default_scope order: 'created_at ASC'
  ...
  # should be relative to created_at if exists?
  def collect_statistics(since = 1.day.ago)
    [Search, Result, Session, Comment].each do |model|
      model.where(created_at: since..0.ago).statistics.each do |key, value|
        write_attribute("#{model.table_name}_#{key}", value.to_f)
      end
    end
  end
  before_create :collect_statistics
end
```

[15] https://github.com/gregbell/active_admin
[16] https://github.com/alexreisner/geocoder
[17] https://developers.google.com/maps/documentation/staticmaps/

Figure 5.5: Session View



The `collect_statistics` method iterates through the models calling `statistics` (which is implemented in each model using the *statistics* Gem[18]). Aggregations are then periodically generated (via `crontab`) using the following Rake[19] task:

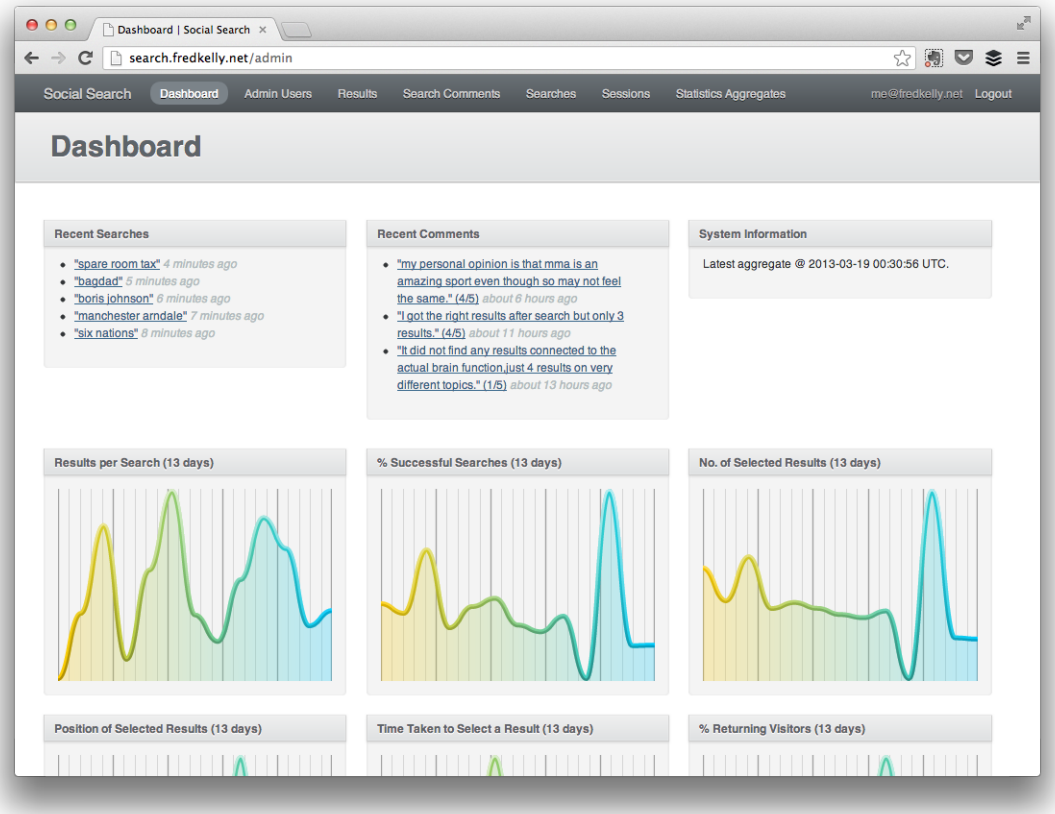
```
# tasks/scheduler.rake
desc "This task generates the application statistics"
task :generate_aggregate => :environment do
  StatisticsAggregate.create
end
```

The administration dashboard then retrieves this data using the `as_time_series` method (via a custom view) to render usage graphs using the *Bobbograph* JS library[20] (fig. 5.6).

```
  def self.as_time_series(scope = :this_week)
    series = {}
    data = send(scope)
    (column_names - IGNORED_COLUMNS).map(&:to_sym).each do |column|
      series[column] = data.map(&column)
    end
    series
  end
```

[18] https://github.com/acatighera/statistics
[19] http://rake.rubyforge.org/
[20] http://bobbograph.robertmesserle.com/

Figure 5.6: Dashboard



6 Results and Evaluation

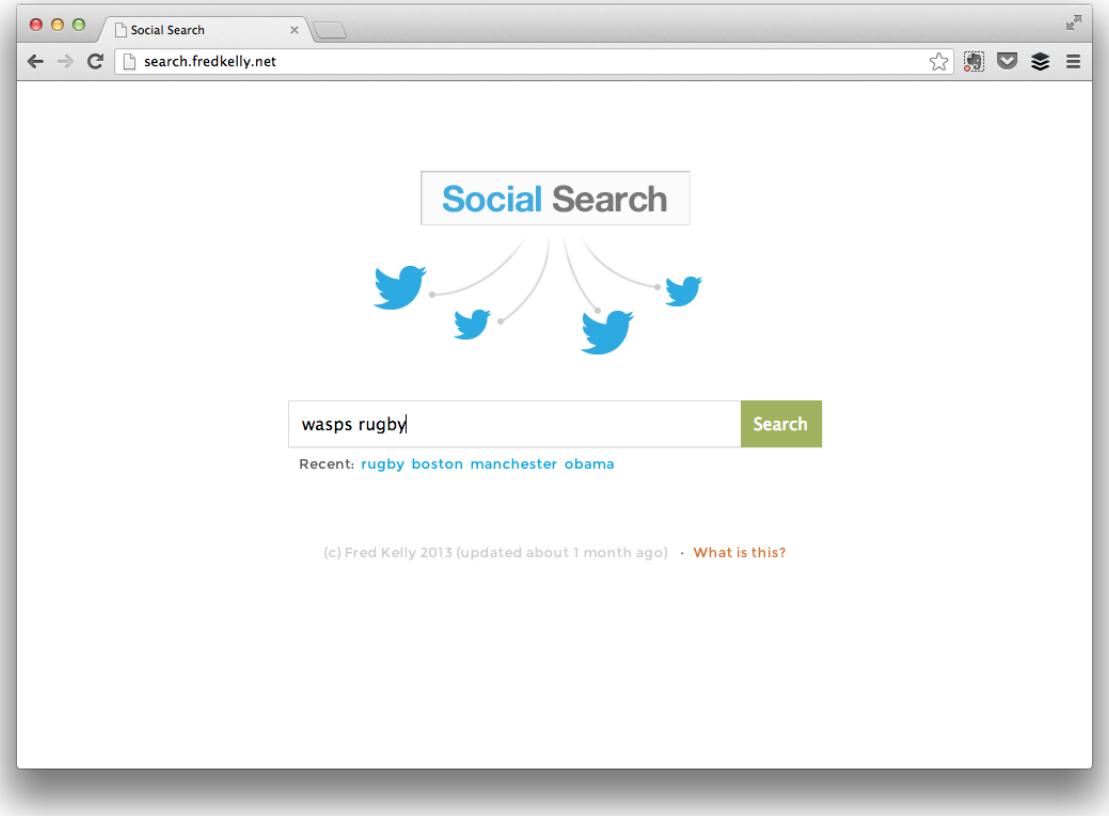
In this chapter I demonstrate the functionality of the final application, as well as presenting the data collected over the course of the project.

6.1 Application

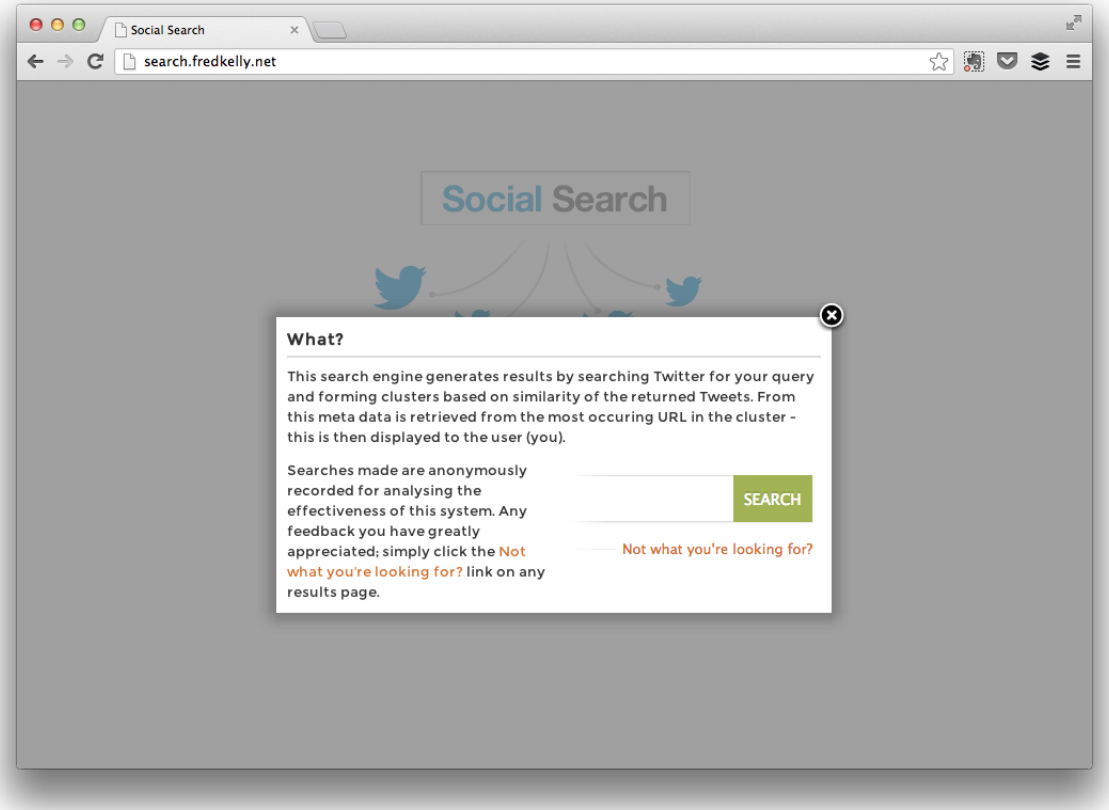
This section provides a demonstration of the final application in typical user flow, as well as exceptional cases.

Figure 6.1: Search Request

(a) Request Input



(b) Application Info Dialog



(c) Search Results

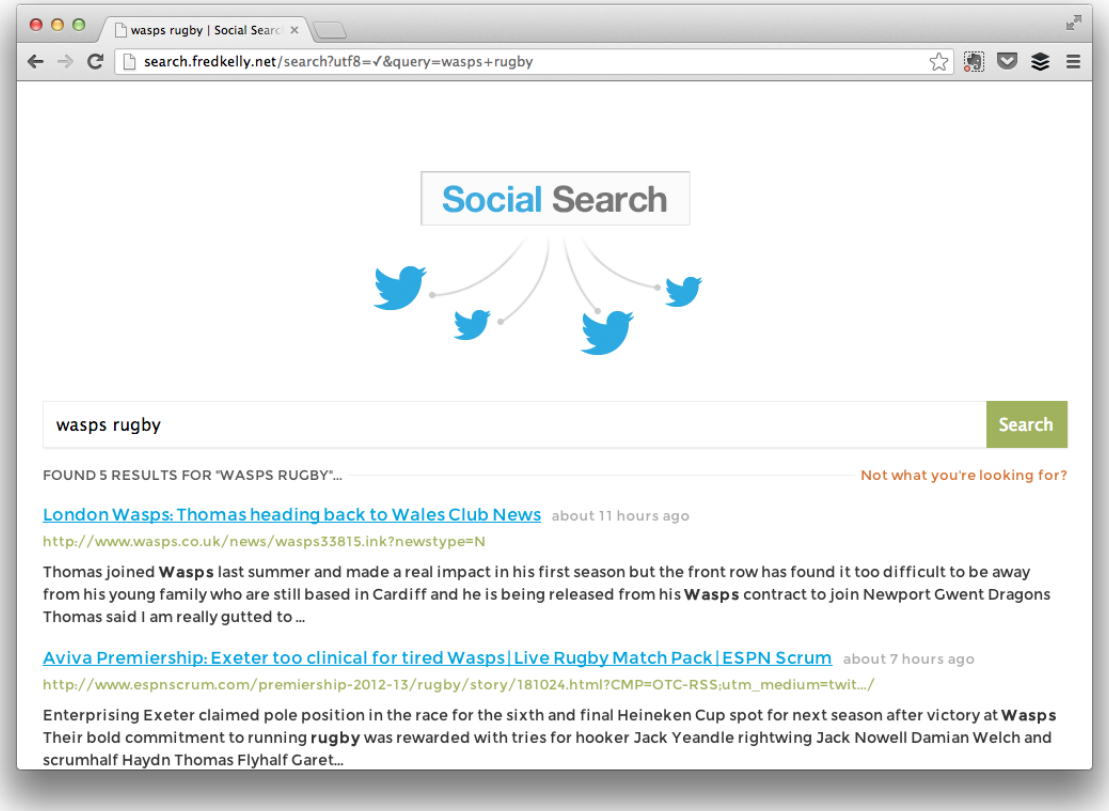
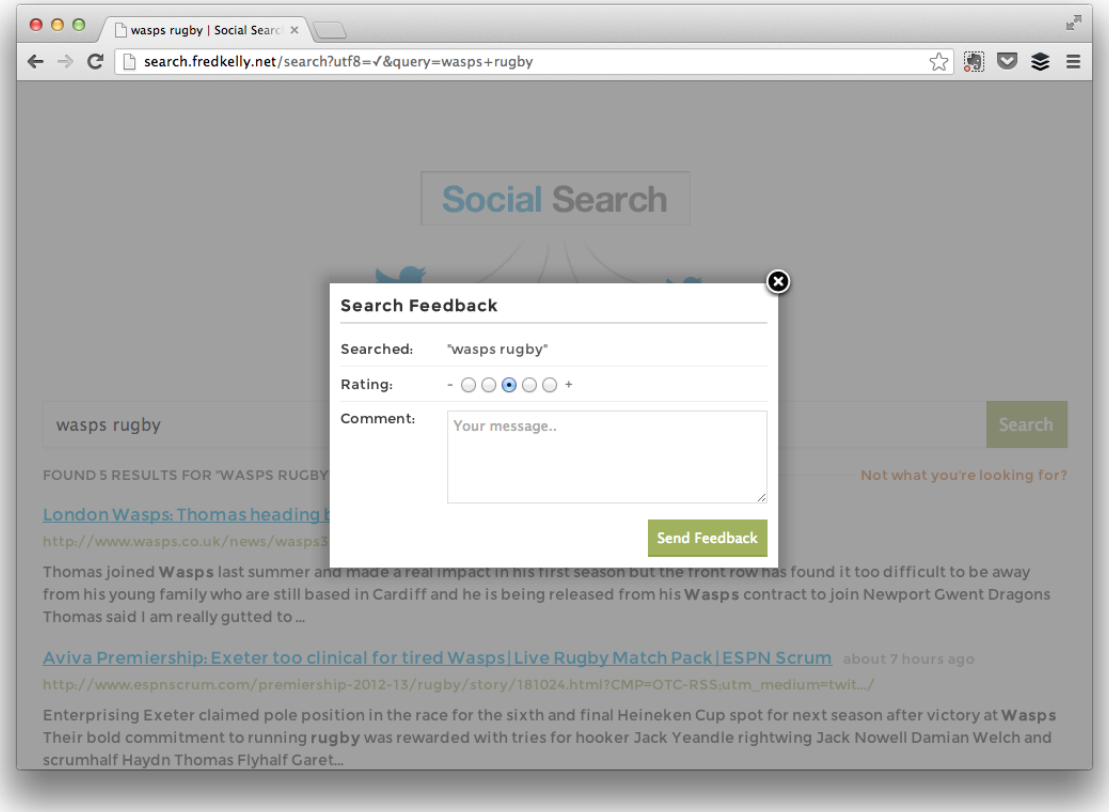


Figure 6.1: Search Feedback

(a) Feedback Modal



(b) Confirmation

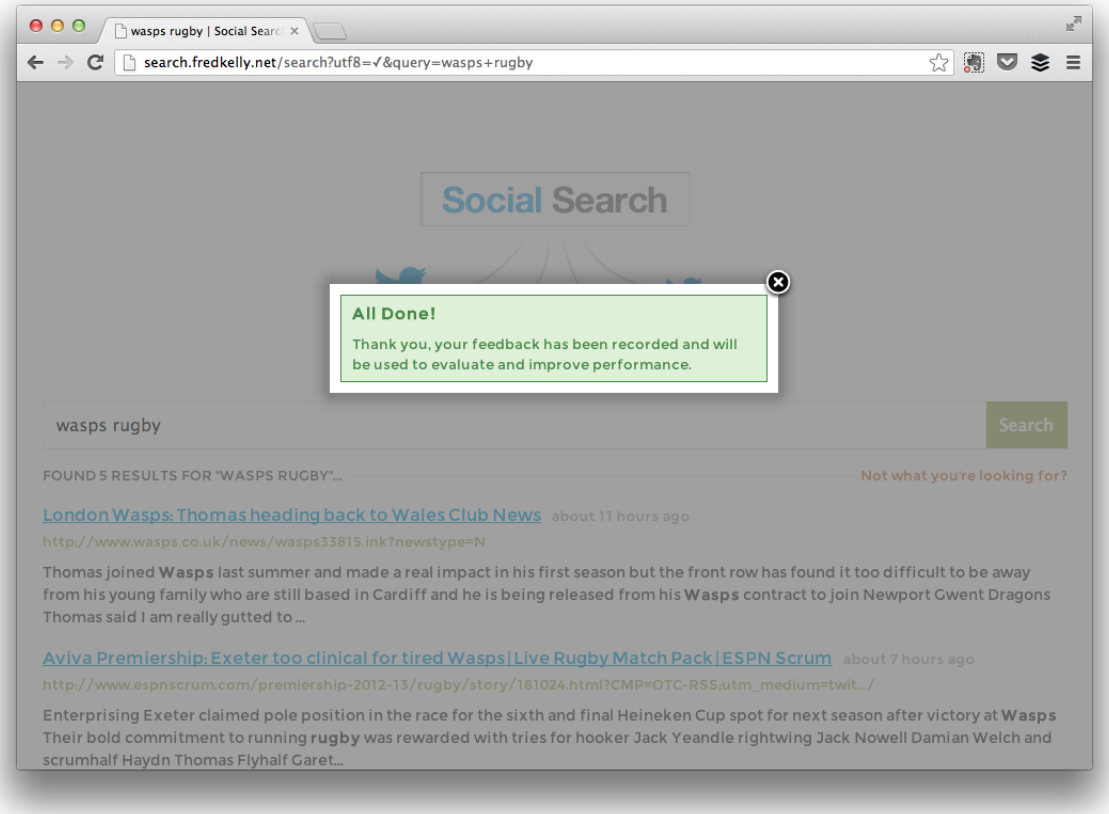
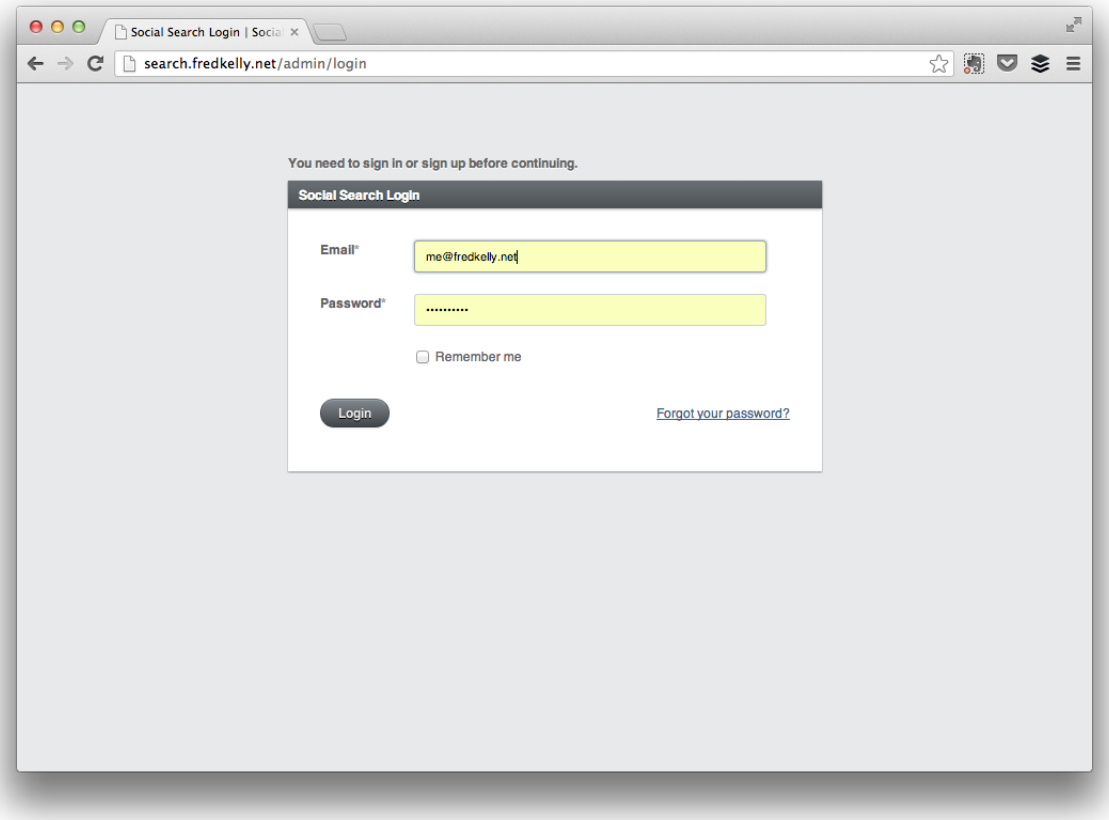
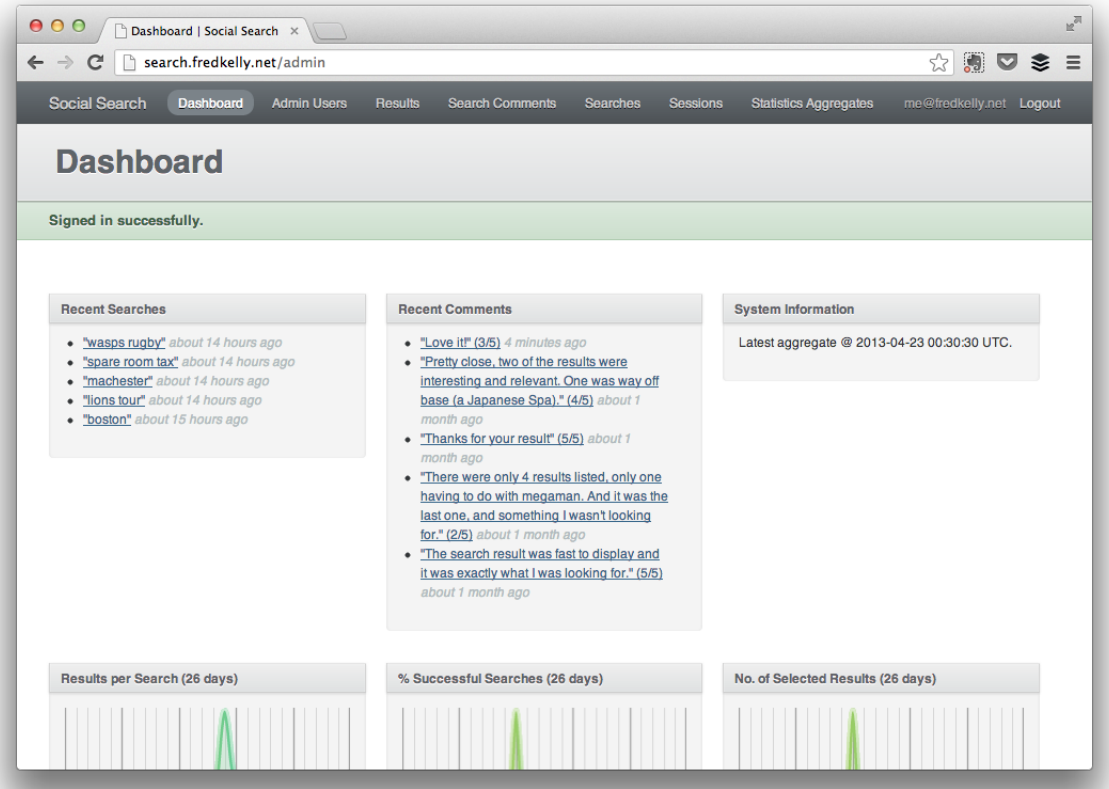


Figure 6.2: Data Retrieval

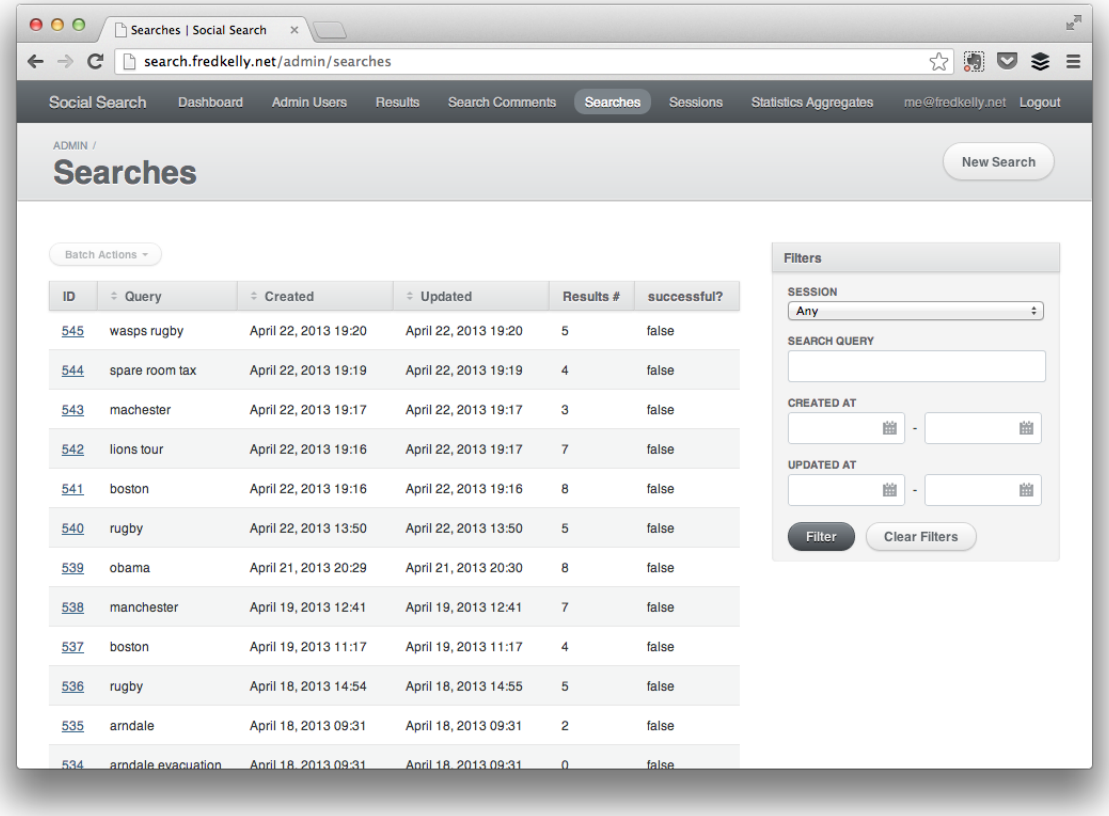
(a) Authentication



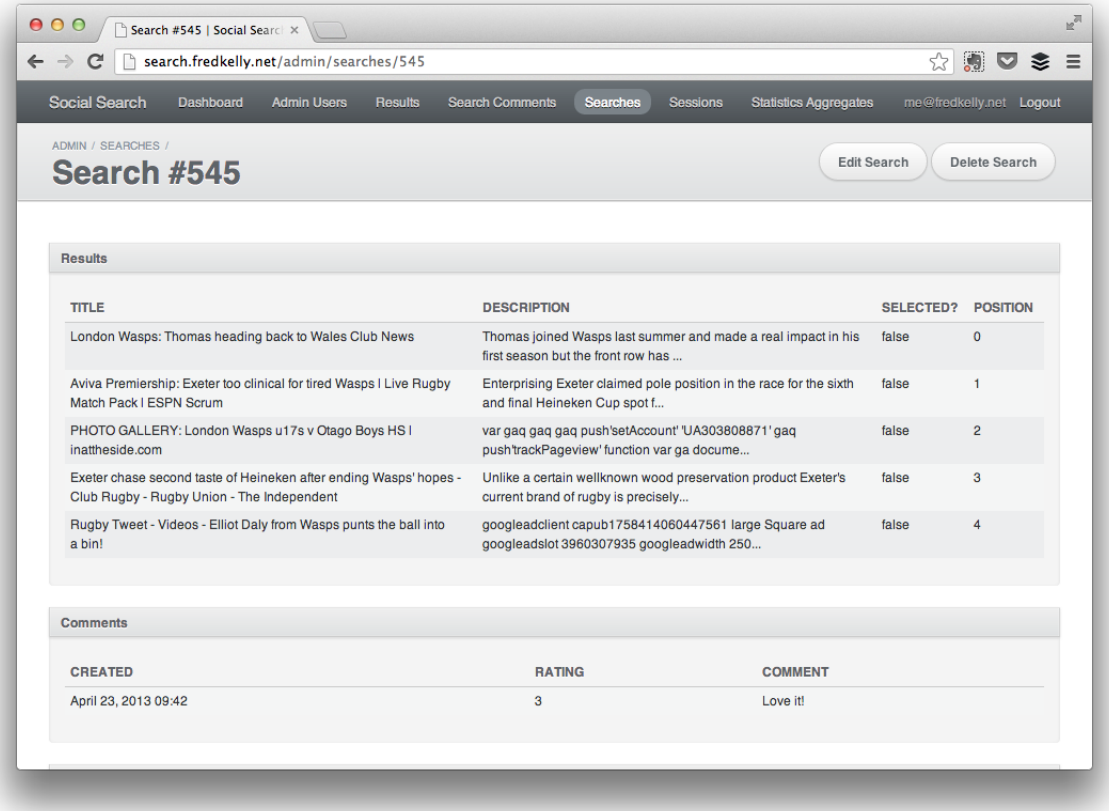
(b) Dashboard



(c) Searches Index



(d) View Search



6.2 Analysis

In this section I present a sample of the data collected from users of the application. I have confined the results to only those collected during the first 20 days the system was online, during which I sent the system to a number of test users (comprised of both peers and paid workers).

6.2.1 Statistics

Links to the application were distributed via a department email, as well as a number of social networks. The following represents the searches conducted by these users from March 1st to April 1st 2013 (inclusive). Table 6.1 shows the total activities users performed during this period.

Table 6.1: Application Statistics

Sessions	261 (44% returning)
Searches	332
Results	1043

6.2.2 KPIs

Returning to the KPIs I defined during the prototyping stage (section 3.3.2), I now consider how the application fared in terms of these measures.

1. **# of results clicked** 87 (8.3%, representing 21.1% of searches)
   Perhaps the most disappointing statistic, appearing to represent very low user engagement. Inspecting the individual search requests shows that in many cases users provided feedback regarding the content of the results despite not actually visiting any. Similarly, many of these searches lasted over a minute before the user attempted a further query, suggesting that they were in fact engaged with the content. I believe this represents users' demand for *instant* answers to their query in social contexts, indeed this closely reflects the behaviours I observed during my user studies (page 35).

2. **Time to click** $\mu = 27.25,\ \sigma = 25.55,\ \sigma^2 = 652.92$
   The majority of users selected a result within a minute of searching. Unsurprisingly, of the results that were clicked, 75% were at positions 0 or 1, reflecting the findings of Russell (2007, p.28). Figure 6.2 shows how clicks were distributed over time.

3. **Returning Visitors** 114 (44%)
   The portion of sessions whose `remote_ip` matched a previous session ≥ 24 hours old. This value showed encouraging growth over the duration, shown in fig. 6.3.

4. **Time between Searches** $\mu = 46.86,\ \sigma = 86.5,\ \sigma^2 = 7481.63$
   31% of the sessions contained repeated searches, of those, the average time separation was ˜47 seconds, suggesting users examined results before performing additional search/s. This behaviour supports the hypothesis that users expect to glean the answer to their query from results alone.

Figure 6.2: Time taken to select a Result

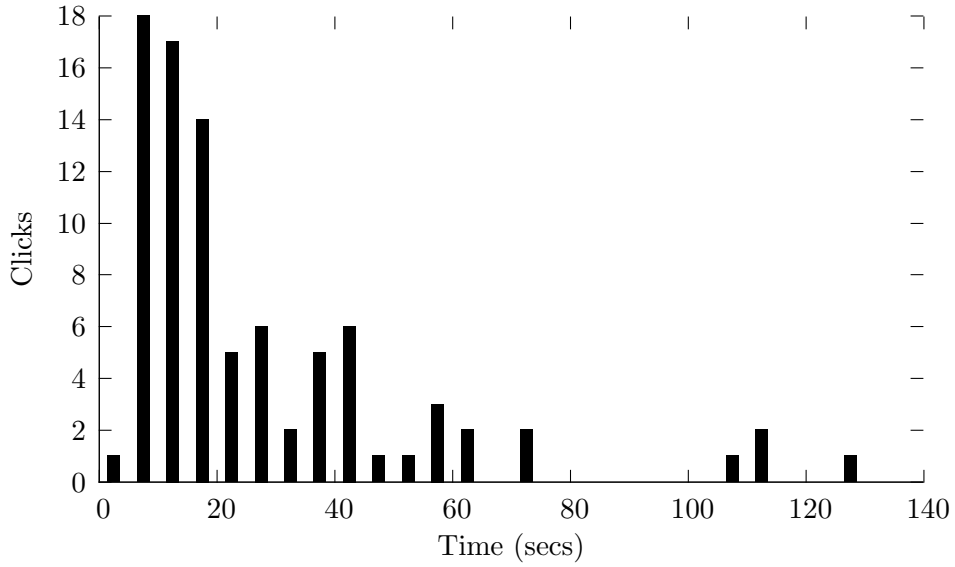
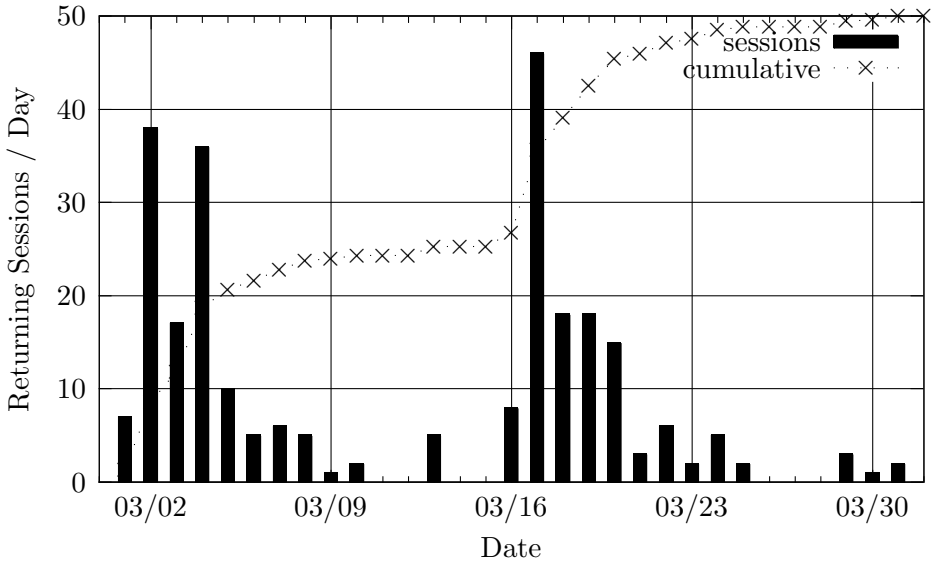


Figure 6.3: Returning Visitors



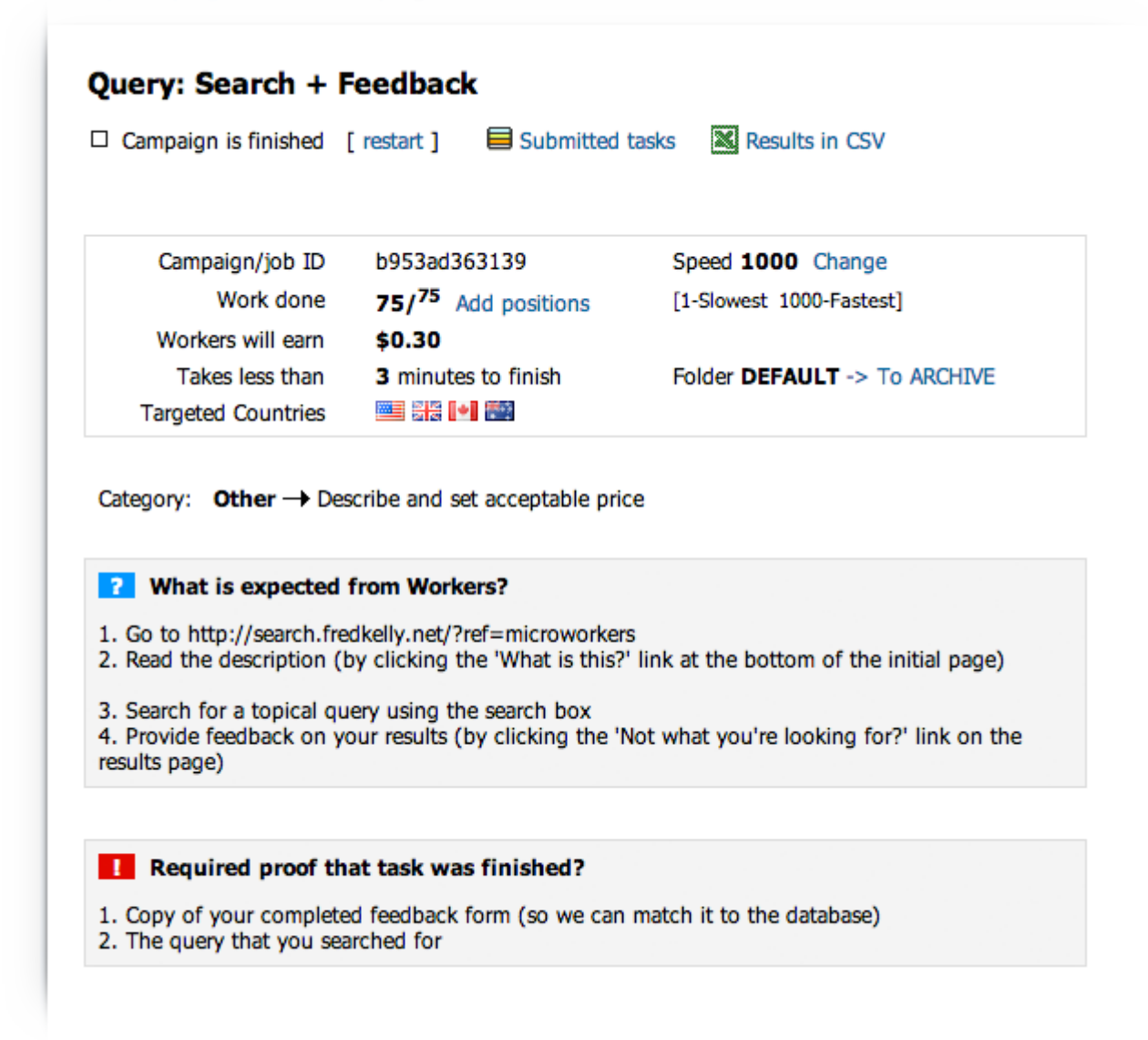
6.2.3 Crowdsourcing

In addition to peers I also distributed the application to a number of crowdsourced workers as a Human Intelligence Task (HIT). A number of platforms exist providing access to human workers, Amazon Mechanical Turk (AMT)[1] being the most well known.

HITs are described as a number of simple steps for each worker to complete, as well as a 'proof' step which demonstrates to you (the 'requester') that the worker completed the task. Figure 6.4 shows the description provided to workers of the *Microworkers* service[2] (an AMT alternative available in the UK).

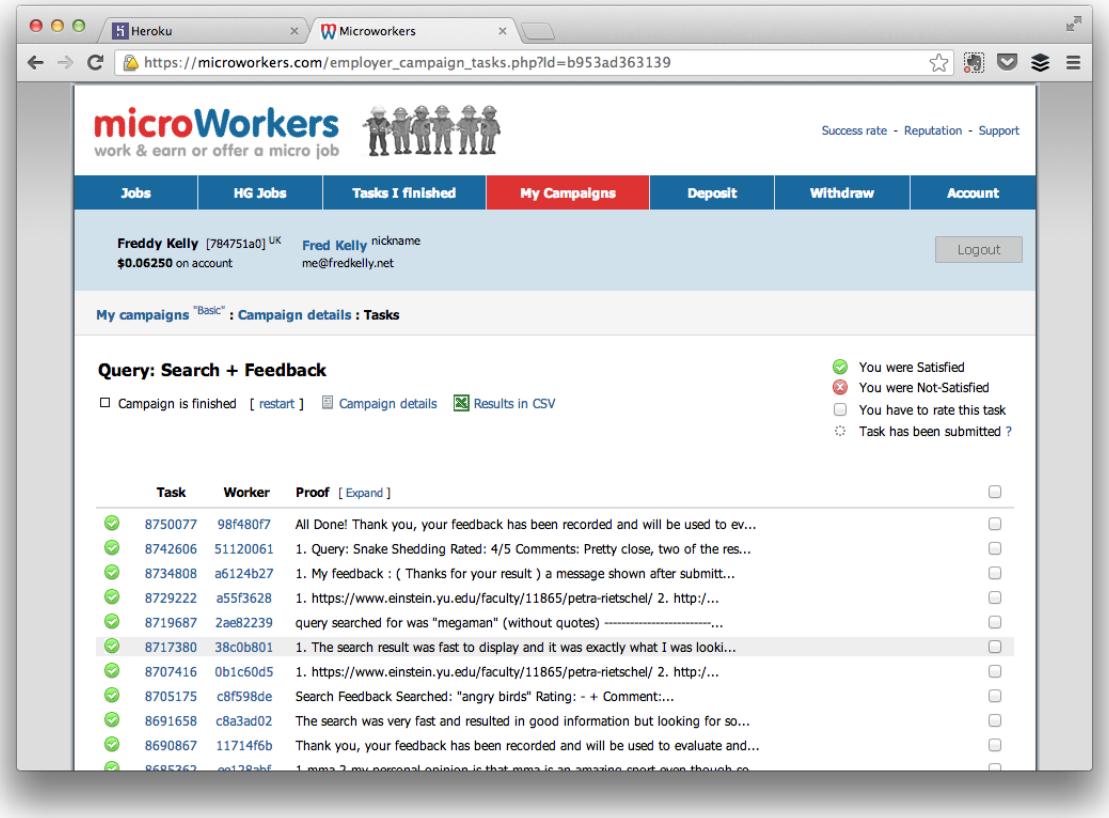
[1] `https://www.mturk.com/mturk/`
[2] `https://microworkers.com/`

Figure 6.4: Microworkers Campaign



Crowdsourced workers were distinguished by the `?ref=microworkers` parameter added to the URL provided. This was stored in the `referer` attribute of the `Session` model. In total 75 workers completed the HIT, each performing a search request and providing feedback. Figure 6.5 shows a listing of the submitted tasks on the Microworkers application.

Figure 6.5: Completed HITs



Sprouse (2011) concludes that use of AMT in the context of quantitative validation studies provides "a viable alternative to laboratory-based acceptability judgment experiments". The majority of the data collected during my own tests would tend to support this view, however in some cases it was hard to draw accurate comparisons. For example, in the context of search experimentation, it's hard to reproduce a real information requirement, workers are focused on a strict set of tasks, tending not to demonstrate exploratory behaviours (e.g., those described in fig. 3.3, page 34).

6.2.4 Example Searches

As well serving as a source for quantitive data, the application also demonstrates the searches users are making, providing an insight into the types of queries that worked well and those which didn't. In the latter cases the addition of a qualitative feedback loop enabled users to described their experience with the system in more detail. Table 6.2 shows some example searches.

Table 6.2: Search Examples

(a) Successful

Query	"syria"
Results (4)	1. United Nations News Centre - Fallen UN official should inspire action to 'transform our world', Ban says in Geneva 2. Syrian war is everybody's problem - CNN.com 3. Syria and Iran condemn U.S. plan to aid anti-Assad rebels | Reuters 4. Syria | Douma | Aftermath of Bombardment | March, 2 2013 - YouTube
Success?	`true`
Comment	n/a

(b) Unsuccessful, possibly due to use of acronym

Query	"ufc"
Results (1)	1. Ronda Rousey to coach Ultimate Fighter against winner of Miesha Tate-Cat Zingano - News — FOX Sports on MSN
Success?	`false`
Comment	"Wanted do see latest info from UFC 158." (3/5)

(c) Successful, despite user not selecting results

Query	"property tax"
Results (15)	1. House suspends rules, takes up personal property tax bill ... 2. BOISE, Idaho: House backs partial personal property tax repeal ... 3. Simon Jenkins: Give London boroughs the freedom to raise their own money ... 4. It's time to abolish stamp duties ... 5. ...
Success?	`false`
Comment	"The search was very fast and brought me to some good information but looking for something a little different." (5/5)

(d) Unsuccessful, too few results

Query	"samsunggalaxy"
Results (3)	1. Samsung Galaxy S III leaked in purple, pegged for April release on Sprint ... 2. Samsung Galaxy S - Toy Story Fans Create Live Action, Shot-For-Shot Remake ... 3. Samsung Galaxy S - Android apps compatibility ...
Success?	`false`
Comment	"Would like to see more results for the galaxy." (3/5)

(e) Unsuccessful, not expected outcome

Query	"britney spears"
Results (2)	1. Britney Spears Goes Tanning With New Guy on Cambio ... 2. Britney Spears Blonde, Britney Spears Brown Hair ...
Success?	`false`
Comment	"Didn't have her twitter or Facebook page." (2/5)

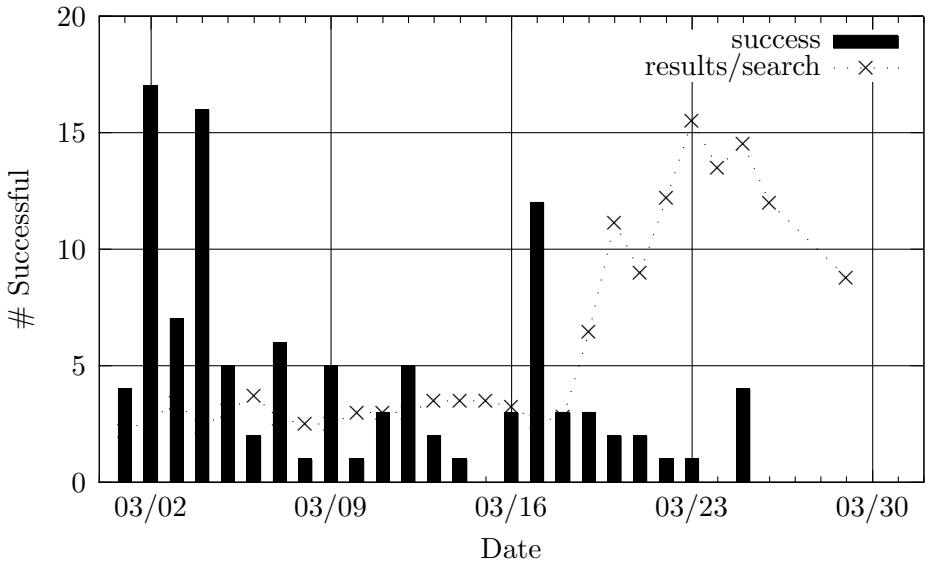
(f) Successful

Query	"nail art"
Results (3)	1. Tetris Nail Art - Worst Nail Art Ever ... 2. MASH Nail Art Stamp Stamping Image Plate No 30 — Nail Art Stamp Plates ... 3. Kiss me, this St. Patrick's Day nail art is Irish ...
Success?	`true`
Comment	"It's very easy to use. However the results are very limited. You have to be very specific with the query to get a result. But the results are exactly what I am looking for so its great." (4/5)

6.2.5 Improvements

As well as serving to collect general validation for the concept, the application also served as a platform for evaluating different ML implementations. Each new `Session` stored a reference to the current `git HEAD`, allowing the precise application version in use to be tracked. The change from *K-means* to *Hierarchical Agglomerative* (HAC) clustering proved most prominent in results, as fig. 6.6 shows.

Figure 6.6: Successful Searches against Results/Search
N.B. HAC was added 03/20.



As results produced by *HAC* were not limited to a specific k value, there tended to be more per search. Interestingly, this actually resulted in a drop in success (i.e., clicked results). This may have been casual, or a result in decreasing engagement towards the end of the experiment, however, it is also possible that users preferred fewer 'curated' results (e.g., they wanted the application to determine importance).

7 Conclusions

In this chapter I will reflect over the duration of the project, considering successes as well as areas to be improved. I will conclude with a set of recommendations for integrating social data with search, and suggestions for future work in this area.

7.1 Reflection

I set out to build a system that used social networks to provide search results. I am pleased with the outcome because the system has stood up well to a large number of users, producing promising results in the process. This project was not only about developing a prototype; I also wanted to build a solid foundation for future work in this area with regard to approach and implementation.

Addressing the task from the user's viewpoint has allowed insight I believe would be missed from a purely scientific standpoint. Indeed, my work has shown users view social in a different light to other media, affecting their behaviours and expectations when searching using this new medium.

An exhaustive approach to developing effective algorithms was out of the scope of this project. However, in developing a extensible platform, I hope to enable such work in the future. It is clear that a successful system will be the result of much refinement in this respect.

7.2 Lessons

This project has provided a great opportunity for me to build knowledge in topics like Machine Learning (ML) and User Experience (UX). I have been able to supplement learning from course modules (e.g., `COMP24111`[1] and `COMP33512`[2]) with invaluable practical experience.

Researching and implementing various ML algorithms has proved very interesting. Although I had worked with some document clustering problems in the past, as well as having experience with the Twitter API, bringing the two together presented interesting challenges I had not anticipated. Most notable was the performance requirement; clustering live data on the fly presents differing problems compared to lab work.

Working with Dr. Caroline Jay I learned to design and implement an effective user evaluation. The results of my initial study proved invaluable in directing much of the project's design, exposing requirements I would otherwise have overlooked.

[1] `http://studentnet.cs.manchester.ac.uk/syllabus/index.php?code=COMP24111&year=2012`
[2] `http://studentnet.cs.manchester.ac.uk/syllabus/index.php?code=COMP33512&year=2012`

7.3 Possible Improvements

As the project developed from a concept to a working application, there were numerous design and implementation choices. As I have already discussed, there are a number of possible refinements just in the document clustering space. In addition to such core 'mechanics', there are a number of other questions yet to be answered, for example, experimentation with varied interfaces and other presentation elements. This section highlights a number of areas I believe are most significant to future development of social search applications.

7.3.1 Crowdsourced Evaluation

Although I found crowdsourcing ineffective in providing valid data during evaluation (section 6.2.3), I believe similar methods could be utilised to provide unbiased 'gold standard' test data as described in section 3.2. Used in combination with qualitative measures this could provide a useful metric for comparing implementations.

7.3.2 Detection and Labelling

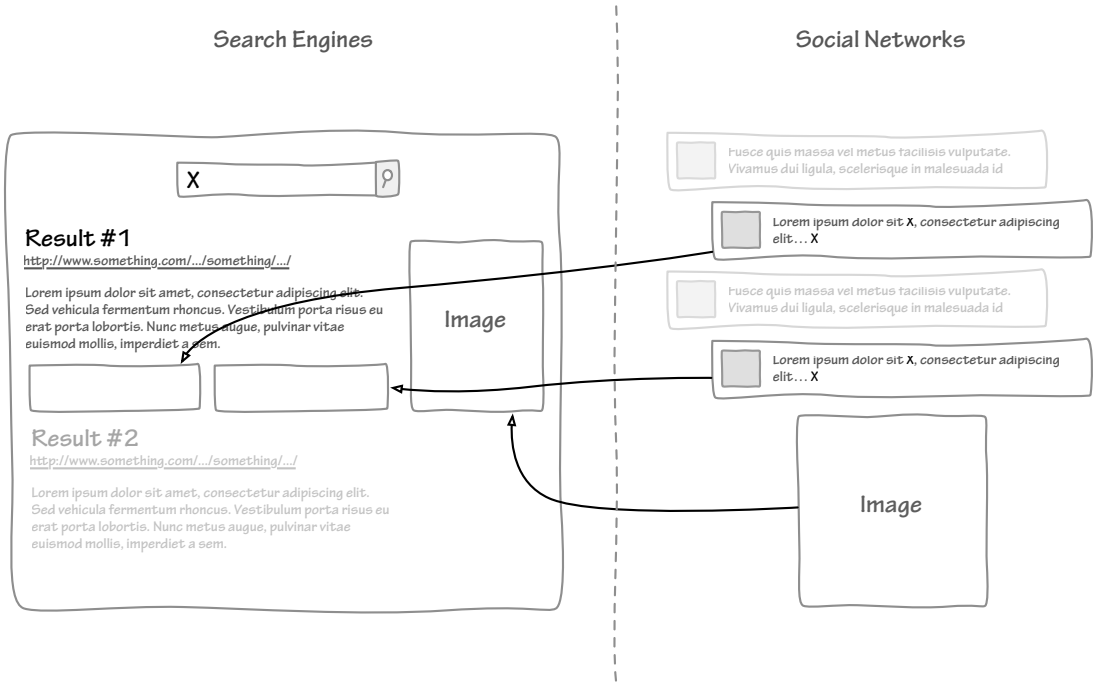
As demonstrated by the events of January 15, 2009 and Flight 1549 Beaumont (2009), social networks like Twitter provide one of the fastest media to transmit new information. Accurately identifying such patterns and in turn generating search results is by no means a trivial task. The work of Tsolmon, Kwon, and Lee (2012) provides promise in this area; though the task of providing adequate descriptions of results still remains. The task of *cluster labelling* in the field data mining is not new, however, producing comparable results in a world of ≤ 140 characters presents a new challenge. Liu et al. (2011) propose a solution that relies on collections of tweets with high similarity to provide role labelling. It is clear from my own work that users have high expectations for results pages, so there is clearly a balance to be struck between providing fresh data (with little description or context), and relying too heavily on cited resources (e.g., URL scraping).

7.3.3 User Interface

Perhaps more significant than the underlying mechanics, is how the content is delivered to users. Currently search is at a crossroads, moving from a document retrieval tool, to a 'personal assistant' (e.g., Google *Now*[3] or *Siri*[4]). Today's users demand more and more from search engines, as echoed in the results of both studies; people want *instant* answers. This makes presentation critical: no longer are results pages just lists of information, they intersect many different sources specifically to answer your query. This vision of search engines "that can make instant applications automatically" (Kobie 2010) has been championed by Conrad Wolfram and *Wolfram |Alpha*. I believe that in this area the existing projects from Google and Bing (figs. 1.1 to 1.2) have failed. Separating social data from traditional results provides no gain for the user, instead creating a confusing visual hierarchy. Based on my own findings, I propose a design that provides 'cross-linking' with social data (fig. 7.1).

[3] `http://www.google.co.uk/landing/now/`
[4] `http://www.apple.com/uk/ios/siri/`

Figure 7.1: Socially Augmented Search



This *augmentation* is something I have attempted to capture in application design (i.e., including media and timestamps). I believe there is a lot of potential to collect new types of metadata in this fashion: opinion (e.g., via sentiment analysis), current trends as well as media (images, video etc.).

7.4 Recommendations

I will conclude by offering a set of recommendations for future work in this area, based on my own experiences during the project.

7.4.1 Clustering

What makes this application uniquely different from other document clustering tasks is the ever-changing nature of the data involved. This makes evaluating approaches difficult. Additionally, the time-critical nature of a web application provides further restriction on implementation. Having said that, I was able to provide satisfactory results using simple Hierarchical methods. Expanding on this, it would be interesting to integrate live data after the initial search request, e.g. using Twitter's streaming APIs[5]. This way new information could be added to results as it becomes available.

7.4.2 Topic Labelling

The most time consuming element of the search request logic, scraping URLs for appropriate metadata, seems less than ideal. Providing effective summaries of short documents like tweets

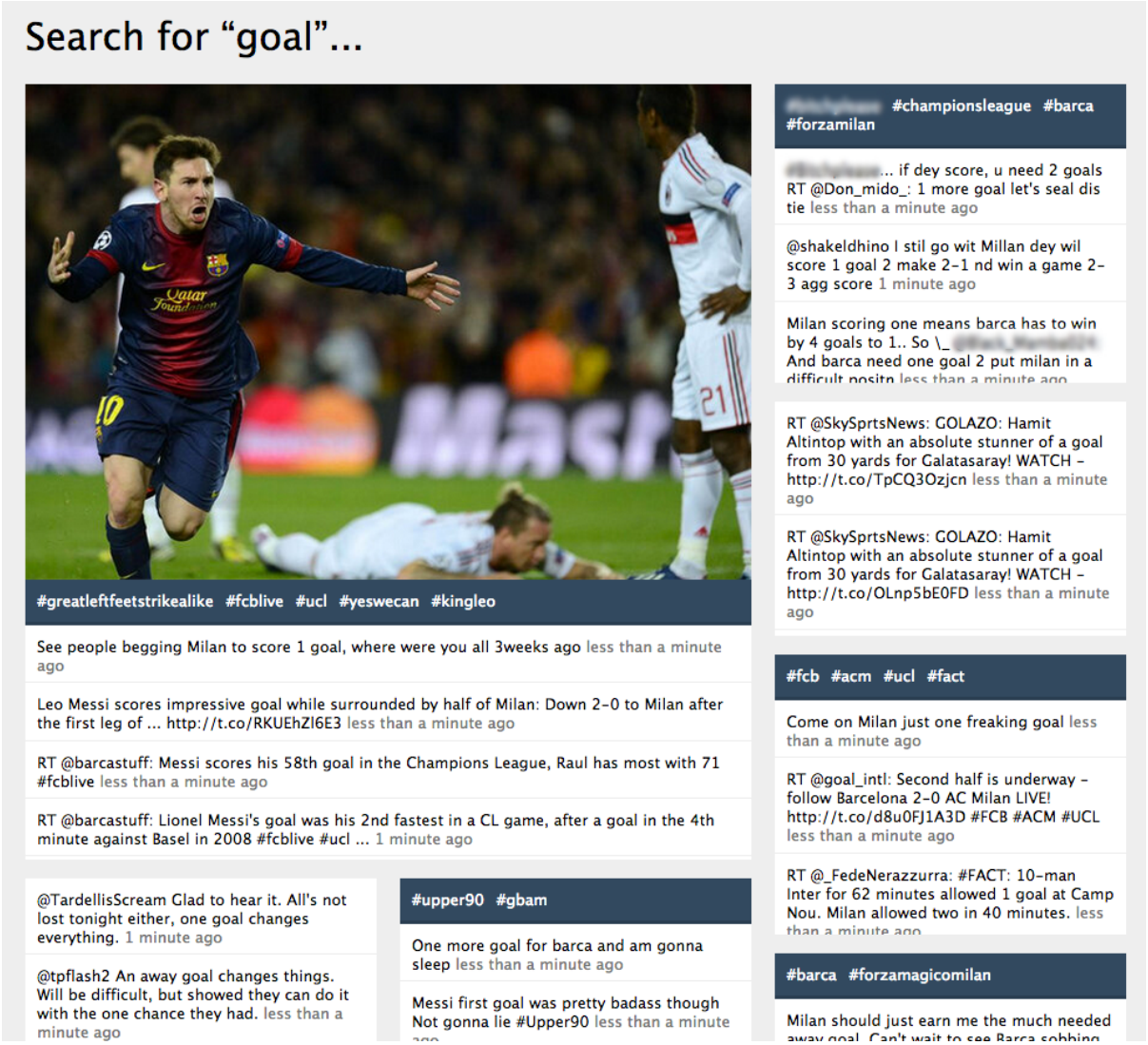
[5] `https://dev.twitter.com/docs/streaming-apis`

is a difficult task; my own experiments using *n-gram* frequency and the *Open Text Summarizer*[6] struggled to produce usable cluster labels. Having said that, URLs do provide a unique identifier that can be used to provide the 'cross-linking' behaviour described above. A possible alternative to traditional labelling could be to use social constructs such as hashtags as described in Xiao, Noro, and Tokuda (2012). Figure 7.2 shows an example of this I implemented using the platform.

7.4.3 User Experience

The importance of visual hierarchy was something I had not anticipated prior to conducting my own studies (section 3.3). Social data has the potential to provide a wealth of useful data that other methods can not access. However, presenting this information in a useful manner presents its own problems. I believe my concept of 'augmented' results (fig. 7.1) provides a good balance between conventional search and social data, without disorientating users.

Figure 7.2: Hashtag Labelling Example



[6] http://libots.sourceforge.net/

Bibliography

Arunprabha, K. and V. Bhuvaneswari (Dec. 2010). "Article:Comparing K-Value Estimation for Categorical and Numeric Data Clustring". In: *International Journal of Computer Applications* 11.3. Published By Foundation of Computer Science, pp. 4–7.

Barnett, Emma (Oct. 2011). *Twitter and Google search talks in deadlock*. URL: http://www.telegraph.co.uk/technology/twitter/8833766/Twitter-and-Google-search-talks-in-deadlock.html (Retrieved 03/30/2013).

Beaumont, Claudine (Jan. 2009). *New York plane crash: Twitter breaks the news, again*. URL: http://www.telegraph.co.uk/technology/twitter/4269765/New-York-plane-crash-Twitter-breaks-the-news-again.html (Retrieved 03/28/2013).

Brin, Sergey and Lawrence Page (Apr. 1998). "The anatomy of a large-scale hypertextual Web search engine". In: *Comput. Netw. ISDN Syst.* 30.1-7, pp. 107–117. ISSN: 0169-7552. DOI: 10.1016/S0169-7552(98)00110-X. URL: http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

Burkhard, W. A. and R. M. Keller (Apr. 1973). "Some approaches to best-match file searching". In: *Commun. ACM* 16.4, pp. 230–236. ISSN: 0001-0782. DOI: 10.1145/362003.362025. URL: http://doi.acm.org/10.1145/362003.362025.

Chung, Tagyoung and Daniel Gildea (2009). "Unsupervised tokenization for machine translation". In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2*. EMNLP '09. Singapore: Association for Computational Linguistics, pp. 718–726. ISBN: 978-1-932432-62-6. URL: http://dl.acm.org/citation.cfm?id=1699571.1699606.

Estivill-Castro, Vladimir (June 2002). "Why so many clustering algorithms: a position paper". In: *SIGKDD Explor. Newsl.* 4.1, pp. 65–75. ISSN: 1931-0145. DOI: 10.1145/568574.568575. URL: http://doi.acm.org/10.1145/568574.568575.

Finin, Tim et al. (2010). "Annotating named entities in Twitter data with crowdsourcing". In: *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*. CSLDAMT '10. Los Angeles, California: Association for Computational Linguistics, pp. 80–88. URL: http://dl.acm.org/citation.cfm?id=1866696.1866709.

Hassler, Marcus and Gunther Fliedl (June 2006). "Text Preparation through Extended Tokenization". In: *Data Mining VII: Data, Text and Web Mining and their Business Applications*. Ed. by A. Zanasi, C. A. Brebbia, and N.F.F. Ebecken. Vol. 37. WIT Press/Computational Mechanics Publications, pp. 13–21. ISBN: 978-1-84564-178-8.

Jiang, Jing and Chengxiang Zhai (Oct. 2007). "An empirical study of tokenization strategies for biomedical information retrieval". In: *Inf. Retr.* 10.4-5, pp. 341–363. ISSN: 1386-4564. DOI: 10.1007/s10791-007-9027-7. URL: http://dx.doi.org/10.1007/s10791-007-9027-7.

Johnson, Nick (July 2010). *Damn Cool Algorithms: Levenshtein Automata*. URL: http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata (Retrieved 04/08/2013).

Khot, Tushar (2010). "Clustering Twitter Feeds using Word Co-occurrence". In: *CS784 Project Report*.

Klemmer, Scott (2010). *Human-Computer Interaction*. URL: https://www.coursera.org/course/hci (Retrieved 04/12/2013).

Kobie, Nicole (Mar. 2010). *Conrad Wolfram on communicating with apps in Web 3.0*. URL: http://www.itpro.co.uk/621535/qa-conrad-wolfram-on-communicating-with-apps-in-web-30 (Retrieved 04/25/2013).

Laird, Sam (Feb. 2013). *Twitter Will Decide the Value of Your Tweets*. URL: http://mashable.com/2013/02/14/twitter-judge-value-tweets/ (Retrieved 04/04/2013).

Levner, Eugene et al. (2007). "Fuzzifying clustering algorithms: The case study of MajorClust". In: *MICAI 2007: Advances in Artificial Intelligence*. Springer, pp. 821–830. URL: http://link.springer.com/chapter/10.1007/978-3-540-76631-5_78.

Lewis, Clayton and John Rieman (1994). *Task-Centered User Interface Design: A Practical Introduction*.

Li, Chenliang et al. (2012). "TwiNER: named entity recognition in targeted twitter stream". In: *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. SIGIR '12. Portland, Oregon, USA: ACM, pp. 721–730. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348380. URL: http://doi.acm.org/10.1145/2348283.2348380.

Liao, Cecilia (May 2012). *Social Media and Networking*. URL: http://academic.mintel.com/display/590164/ (Retrieved 03/25/2013).

Liu, Xiaohua et al. (2011). "Collective semantic role labeling for tweets with clustering". In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, pp. 1832–1837. ISBN: 978-1-57735-515-1. DOI: 10.5591/978-1-57735-516-8/IJCAI11-307. URL: http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-307.

Lloyd, Stuart (1982). "Least squares quantization in PCM". In: *Information Theory, IEEE Transactions on* 28.2, pp. 129–137. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

Locke, Brian and Dr. James Martin (2009). "Named Entity Recognition: Adapting to Microblogging". PhD thesis. University of Colorado Boulder.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schutze (2008). *Introduction to Information Retrieval*. Cambridge University Press. ISBN: 0521865719.

Mayfield, J. and P. McNamee (2005). "The HAIRCUT Information Retrieval System". In: *The Johns Hopkins APL Technical Digest* 26.1.

McNamee, Paul and James Mayfield (2004). "Character N-Gram Tokenization for European Language Text Retrieval". English. In: *Information Retrieval* 7.1-2, pp. 73–97. ISSN: 1386-4564. DOI: 10.1023/B:INRT.0000009441.78971.be. URL: http://dx.doi.org/10.1023/B:INRT.0000009441.78971.be.

Rijsbergen, C.J. van, S.E. Robertson, and M.F. Porter (1980). "New models in probabilistic information retrieval". In: *British Library Research and Development Report, no. 5587*.

Ritter, Alan et al. (2011). "Named entity recognition in tweets: an experimental study". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. EMNLP '11. Edinburgh, United Kingdom: Association for Computational Linguistics, pp. 1524–1534. ISBN: 978-1-937284-11-4. URL: http://dl.acm.org/citation.cfm?id=2145432.2145595.

Russell, Daniel M. (June 2007). "What are they thinking? Searching for the mind of the searcher". In: *Joint Conference on Digital Libraries Vancouver, British Columbia*. URL: http://dmrussell.googlepages.com/JCDL-talk-June-2007-short.pdf (Retrieved 04/13/2013).

Sahoo, Nachiketa et al. (2006). "Incremental hierarchical clustering of text documents". In: *Proceedings of the 15th ACM international conference on Information and knowledge management*. CIKM '06. Arlington, Virginia, USA: ACM, pp. 357–366. ISBN: 1-59593-433-2. DOI: 10.1145/1183614.1183667. URL: http://doi.acm.org/10.1145/1183614.1183667.

Sibson, R. (1973). "SLINK: An optimally efficient algorithm for the single-link cluster method". In: *The Computer Journal* 16.1, pp. 30–34. DOI: 10.1093/comjnl/16.1.30. eprint: http://comjnl.oxfordjournals.org/content/16/1/30.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/16/1/30.abstract.

Sprouse, Jon (2011). "A validation of Amazon Mechanical Turk for the collection of acceptability judgments in linguistic theory". English. In: *Behavior Research Methods* 43.1, pp. 155–167. DOI: 10.3758/s13428-010-0039-7. URL: http://dx.doi.org/10.3758/s13428-010-0039-7.

Srivastava, Ashok and Mehran Sahami (2009). *Text Mining: Classification, Clustering, and Applications (Chapman & Hall/CRC Data Mining and Knowledge Discovery Series)*. Chapman and Hall/CRC. ISBN: 1420059408. URL: http://www.amazon.com/Text-Mining-Classification-Clustering-Applications/dp/1420059408?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1420059408.

Teevan, Jaime, Daniel Ramage, and Merredith Ringel Morris (2011). "#TwitterSearch: a comparison of microblog search and web search". In: *Proceedings of the fourth ACM international conference on Web search and data mining*. WSDM '11. Hong Kong, China: ACM, pp. 35–44. ISBN: 978-1-4503-0493-1. DOI: 10.1145/1935826.1935842. URL: http://doi.acm.org/10.1145/1935826.1935842.

Tsolmon, Bayar, A-Rong Kwon, and Kyung-Soon Lee (2012). "Extracting social events based on timeline and sentiment analysis in twitter corpus". In: *Proceedings of the 17th international conference on Applications of Natural Language Processing and Information Systems*. NLDB'12. Groningen, The Netherlands: Springer-Verlag, pp. 265–270. ISBN: 978-3-642-31177-2. DOI: 10.1007/978-3-642-31178-9_32. URL: http://dx.doi.org/10.1007/978-3-642-31178-9_32.

Walter, Bruce et al. (2008). "Fast agglomerative clustering for rendering". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE, pp. 81–86. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4634626.

Wang, Yongheng and Bo Guo (2008). "Hierarchical Clustering of Large-Scale Short Conversations Based on Domain Ontology". In: *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*. Vol. 1. IEEE, pp. 126–130. DOI: 10.1109/ISCSCT.2008.210. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4731390.

Xiao, Feng, Tomoya Noro, and Takehiro Tokuda (2012). "News-Topic oriented hashtag recommendation in twitter based on characteristic co-occurrence word detection". In: *Proceedings of the 12th international conference on Web Engineering*. ICWE'12. Berlin, Germany: Springer-Verlag, pp. 16–30. ISBN: 978-3-642-31752-1. DOI: 10.1007/978-3-642-31753-8_2. URL: http://dx.doi.org/10.1007/978-3-642-31753-8_2.

Glossary

AMT Amazon Mechanical Turk. 77, 79

API Application Programming Interface. 12, 14, 15, 28, 37, 40–42, 51, 53, 68, 83, 85

AWS Amazon Web Services. 53

corpora A collection of written texts. 9

crawler An automated application that periodically browses the WWW, typically to index documents. 8

CRUD Create, read, update and delete. 62

CSS Cascading Style Sheets. 52, 64, 66

data mining The process of extracting new information by discovering patterns in large data sets. 16, 84

DFA Deterministic finite automaton. 23

document clustering The automated organisation of documents into relevant groups. 9, 12, 15, 16, 19–21, 25, 31, 42, 83–85

DRY Don't repeat yourself. 64

DSL Domain-specific language. 68

filter bubbling The resulting state when a website (often a search engine), makes selective guesses based on learned information about the user. 8, 40

Finite state automaton An abstract model of computation that moves between a finite set of states via conditional transitions. 23

hashtag A word or phrase (prefixed '#') used to group messages relating to the same topic. 13, 86

HCI Human-computer interaction. 32, 35

HIT Human Intelligence Task. 77, 78

HTML HyperText Markup Language. 52, 61, 62, 64–66

IR Information Retrieval. 9

JS JavaScript. 52, 65, 66, 69

JSON JavaScript Object Notation. 12–14, 42, 51, 61, 66

KPI Key Performance Indicator. 10, 31–33, 35, 36, 38, 41, 57, 76

label Denotes which class a particular datum belongs to (e.g., within a training set). 16

metadata Describes other data, 'data about data'. 14, 15

metric space a set with some notion of distance applicable to any two points. 23

microblogging A broadcast medium, usually composed of smaller posts than a traditional blog. 12, 14

ML Machine Learning. 8, 9, 15, 16, 20, 25, 42, 81, 83

MVC Model-view-controller. 31, 51

MVP Minimum Viable Product. 10

NER Named Entity Recognition. 27, 28

NFA Non-deterministic finite automaton. 23

NLP Natural Language Processing. 25, 28, 29

ORM Object-relational mapping. 44, 52

POS Part of speech. 27, 57

proper noun a noun that refers to a unique entity, such as Google, Paris, Dave etc.. 27

Rails Ruby on Rails. 51, 52, 61, 62, 65

REST Representational State Transfer. 14, 51

retweet A repost of a tweet. 13–15

soft clustering Clusters are not necessarily discrete; samples may belong to multiple clusters (cf. hard clustering). 24

stopword Words carrying low information content, e.g. 'the', 'and', 'it'. 24

TDD Test-driven Development. 10, 47, 52

TF-IDF Term frequency-inverse document frequency. 18

third party An external entity (usually an application), with no direct connection to the primary system. 41

training set A set of correctly labelled data examples, used to train a classifier. 16, 92

tweet A single status ($\leq$ 140 chars) posted to the social network Twitter. 9, 12, 14, 24–28, 58, 84, 85, 92

UCD User-centered design. 37

UI User Interface. 35, 36, 40, 41, 47

UML Unified Modelling Language. 42

URL Universal Resource Locator. 12, 13, 38, 42, 58, 61, 64, 78, 84–86

UX User Experience. 32, 36, 37, 40, 47, 65, 83

VC Version control. 53

WWW World Wide Web. 40, 47, 91

A Appendix

A.1 Participant Form

Measuring Search User Study

Freddy Kelly *

April 30, 2013

Contents

1 Participant Information **2**

1.1 Introduction 2

1.2 Consent Form 3

2 Interview **4**

3 Participant Questionnaire **5**

*supervised by Gavin Brown gbrown@cs.man.ac.uk

1 Participant Information

1.1 Introduction

You are being invited to take part in a research study as part of a student project aimed at using Twitter to provide an alternative search engine. Before you decide whether to participate it is important for you to understand why the research is being done and what it will involve. Please take time to read the following information carefully and discuss it with others if you wish. Please ask if there is anything that is not clear or if you would like more information. Take time to decide whether or not you wish to take part.

Who will conduct the research? Freddy Kelly, `fk@cs.man.ac.uk`

Title of Research *Measuring Search*

What is the aim of the research? To explore how users respond to a search engine that uses Twitter as its data source. To collect key metrics which can be used to identify a 'successful search'.

What would I be asked to do if I took part? As part of a group of 10 participants, you will be given a set of search queries to input into two comparable search engines. Your task is simply to try and find relevant results for each query, and document your findings.

What happens to the data collected? The data will be used as a measure for the potential success of the new system, as well as to identify key metrics to be recorded in a similar large-scale online study.

How is confidentiality maintained? Feedback sheets will be completed anonymously; all recordings will be destroyed after analysis.

What happens if I do not want to take part or if I change my mind? It is up to you to decide whether or not to take part. If you do decide to take part you will be given this information sheet to keep and be asked to sign a consent form. If you decide to take part you are still free to withdraw at any time without giving a reason and without detriment to yourself.

What is the duration of the research? The study will take place in a single session lasting no longer than 30 minutes.

Where will the research be conducted? School of Computer Science, University of Manchester, Kilburn Building, Oxford Road, Manchester, M13 9PL

Will the outcomes of the research be published? Data will be included in the final report published on the University's website.

Contact for further information Freddy Kelly, `fk@cs.man.ac.uk` or Gavin Brown (Chief Investigator), `gbrown@cs.man.ac.uk`

If a participant wants to make a formal complaint about the conduct of the research they should contact the Head of the Research Office, Christie Building, University of Manchester, Oxford Road, Manchester, M13 9PL.

1.2 Consent Form

If you are happy to participate please sign and complete the consent form below.

	Initial
I confirm I have read the attached information sheet on the above project and have had the opportunity to consider the information and ask questions and had these answered satisfactorily.	
I understand that my participation in the study is voluntary and that I am free to withdraw at any time without giving reason and without detriment to any treatment/service.	
I understand that screen captures as well as audio/video may be recorded throughout the study.	
I agree to the use of anonymous quotes.	

I hereby agree to take part in the above project.

Name of participant	Date	Signature
Name of person taking consent	Date	Signature

2 Interview

The interview will be conducted pre-study and will cover the following:

1. What services do you currently use to *discover* news articles online?

2. Do you use social networks to discover new stories?

3. When searching do you usually click one of the first *five* results?

4. What do you look for when scanning results for relevance to your query?

5. Given a social context, what attributes would you find useful to aid you in selecting appropriate results?

3 Participant Questionnaire

This study is designed to compare a new type of search engine with a more standard system. You will be given access to two candidate search interfaces and be asked to perform each of the following tasks using each and record your results.

1. Here is a list of **topical** queries..
   - Why has Barack Obama recently been in the news?
   - What is happening in Manchester right now?
   - Which BBC articles are popular at the moment?
   - Who most recently played in the Premiership?
   - What are people saying about George Osbourne?
2. Here is a list of **factual** queries..
   - Who is director general of the BBC?
   - When was Alan Turing born?
   - Who was the 32nd president of the U.S.?
   - How many gold medals did Team G.B. win at the 2012 Olympics?
   - What is the population of China?
3. Please try your own search queries..

A.2 Initial Plan

Social Search
Initial Plan

Freddy Kelly

April 27, 2013

Contents

1 Introduction **2**

1.1 The Elevator Pitch 2

1.2 Basic Specification 2

2 Approach **3**

2.1 Research 3

2.2 Methods 3

2.3 Testing 3

2.4 Contingency 3

2.5 Tracking Progress 3

3 Schedule **4**

3.1 Phased Development 4

3.2 Gantt Chart 5

4 Summary **5**

1 Introduction

1.1 The Elevator Pitch

"I intend to display URLs based on a user's query using tweets matching that query as both a data source and a ranking mechanism."

1.2 Basic Specification

The basic concept of the project is to use social data as a method of sourcing relevant URLs for a given keyword. The core of this is very simple: query an API for a given keyword and parse out a list of links. Whilst this will work, the relevance and value of each URL will be completely random.

The *first phase* in improving this concept will be to group the results into some form of structure. As I plan to source 100s or 1000s of status strings to produce each result, I will need to cluster these by finding relationships between each status. Ideally clusters will be formed around a single resource/URL with all messages related to that resource stored in the cluster (*related does not necessarily mean they contain a matching link).

I will need to investigate various methods of performing this clustering and look at the advantages & disadvantages of each, as well as considering possible combinations to be implemented later on (ensemble learners). I will also investigate what data are available as well as the message strings themselves, for example date/time, geolocation, follower counts etc. - all potential features for my classifier.

2 Approach

2.1 Research

I plan to carry out research throughout the project, covering a broad range of subjects including algorithms, technologies, implementation etc. I will record each of my sources in my logbook so that I can later reference them in the project report. My initial research will look at a selection of relevant academic papers as well as some Machine Learning books. I will use my initial findings to plan my initial classifier and then build iteratively from there.

2.2 Methods

I have chosen to use Ruby as the primary language for developing the classifier. Ruby is an intepreted language written in C with a wealth of Open Source frameworks designed for building Rack based web applications, making it perfect for the project.

In addition to Ruby I will use the RSpec library for testing & the Sinatra framework for handling HTTP requests. The front-end components of the system will be implemented in HTML & CSS with JavaScript, I will include more specific implementation details in the project specification.

2.3 Testing

Sticking with *AGILE* principles I will use test-driven methods throughout the project. As the final system will work with live data, running consistent unit tests will be tricky. I will use a fixed data sample for testing to ensure results are comparable. I will reuse this sample with my *error function* to produce comparable results for the final report.

2.4 Contingency

I have built several weeks of contingency time into my schedule to allow for any unanticipated additional work. I have targeted to complete the majority of the development work before the Christmas break to allow plenty of time to compile the report.

2.5 Tracking Progress

In addition to keeping track using my project plan, I also plan to use the Basecamp project management system to track more fine-grained tasks and ensure all required deliverables are delivered on time. I will use the Git+ version control system to track all code changes and maintain an off site backup of all code.

3 Schedule

3.1 Phased Development

I plan to follow an *AGILE* development strategy focussed on evolving a prototype as the project progresses. To aid me in this I have broken down the goals of the project into smaller *phases*:

1 **Planning & Specification**
Determine what is required, and how work will be prioritised.

2 **Existing Search Systems**
How they work, which algorithms & data structures are common, how they might be improved, what KPI *(Key Performance Indicators)* can I use to draw comparisons.

3 **Data Sources**
Investigate what data is available; implement test environment - unit tests, class stubs etc.

4 **String Clustering**
Focus on the status strings themselves; research possible algorithms and implement a classifier.

5 **Test & Evaluate Performance**
Measure performance using *KPIs*, determine areas for improvement.

6 **Additional Features**
Experiment using additional features to improve performance.

7 **Alternative Classification**
Consider alternative classification algorithms/methods.

8 **Enhanced Clustering**
Formally specify, implement & test an optimised version using improved methods.

9 **Additional Use Cases**
Implement additional use cases such as advanced search parameters.

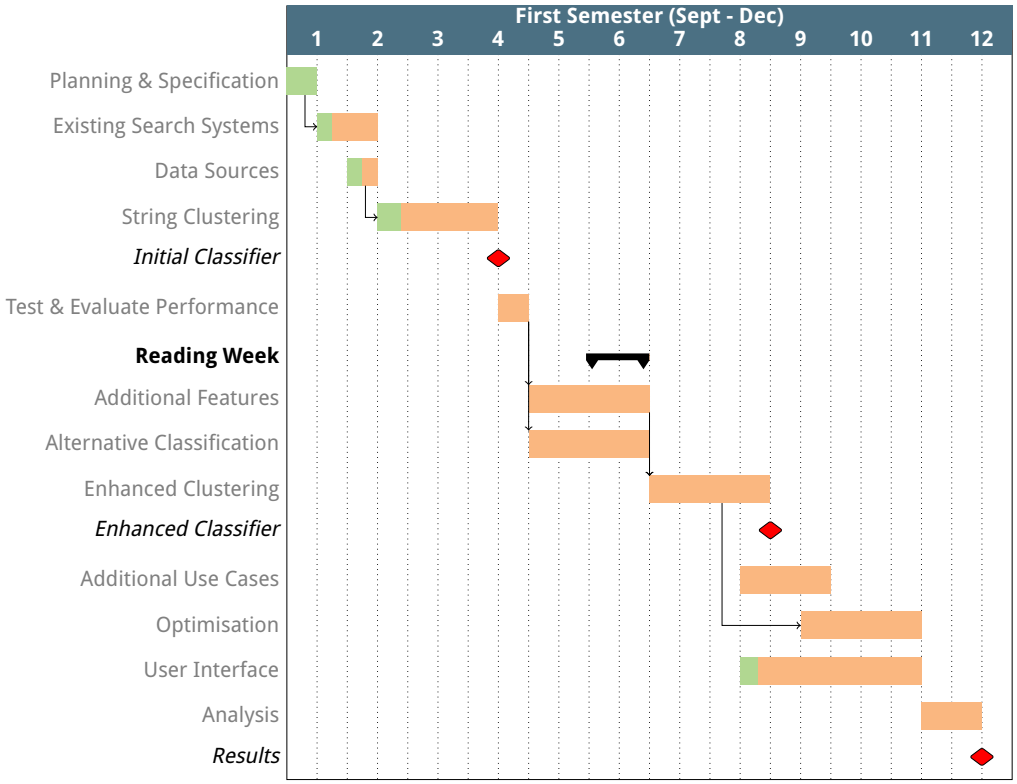
10 **Optimisation**
Optimise existing code for production environment; caching, data structures, etc.

11 **User Interface**
Design, implement & test user interface, consider optimistations based on user feedback.

12 **Analysis**
Collect, analyse & formulate data ready for writing report.

3.2 Gantt Chart

The Gantt chart below shows how the development will progress over the weeks prior to the Christmas break.



Milestones/deliverables are marked by a **red** diamond.

4 Summary

Focussing on smaller iterations will allow me to maintain a working system from an early stage, this is not only useful in ensuring I can deliver a working project but also for analysis: being able to compare and contrast different classification methods will be a key measure of progress.

I have focused all development energy on the 12 week period prior to the Christmas break so that I have plenty of contingency time as well as time to produce the project report.