University “Politehnica” of Bucharest
Faculty of Automatic Control and Computers Science

Content Recommendation
for the
Adobe Community Engine

Supervisors
Prof.Dr.Eng. Valentin CRISTEA
University “Politehnica” of Bucharest
<valentin.cristea@cs.pub.ro>

Dr.Eng. Paul-Alexandru CHIRIȚĂ
Adobe Systems Inc.
<pchirita@adobe.com>

Author
Alexandru MOȘOI
<alexandru.mosoi@gmail.com>

Bucharest
June 2009

to my parents, Vasile *and* Cristina Moșoi

Abstract

This thesis describes the implementation of a *personalised adaptive content-based filtering system* for the *Adobe Community Engine*. The system suggests users a list of relevant articles based on their browsing history. See figure 1.1 for a picture with the results.

The dissertation introduces a very fast way to estimate the similarity between documents by the means of *fingerprints*. Fingerprints are compared to the classical approach and the error is shown to be in a controllable limit while they are more than two hundreds times faster.

Further, the thesis shows how fingerprints can be used to model the *user behavior*. The speed of the operations available are exploited to build a *real-time profile* of the user which is updated when he or she visits a new page and which is used to find relevant articles. Updating the profile using fingerprints is found to be closely related to the classical approach. The results in appendix B suggest that the filtering system improves its precision after user's first visited article.

The filtering system developed — also referred as the *recommendation engine* — will be integrated in a near future release of the Adobe Community Engine. It will improve the user experience and it will stimulate the growth of the available content.

Keywords: *information retrieval*, *fingerprint*, *keywords*, *recommendation engine*, *ADOBE*, *natural language*, *cosine similarity*

Acknowledgements

I would like to thank to my colleagues at Adobe Systems Inc. who helped me during the development of this project with ideas, comments, suggestions and testing.

Special thanks go to Mihaela Barbu who gave me the original idea to develop a content recommendation engine and to dr.eng. Paul-Alexandru Chiriță who guided me over the last few months.

Contents

1 Introduction **7**
1.1 Overview . . . 7
1.2 Use Cases . . . 8
1.3 Contributions . . . 9
1.4 Outline . . . 9

2 Related Work **11**

3 Algorithm **13**
3.1 Representing Documents in the Vector Space . . . 13
3.1.1 Overview . . . 13
3.1.2 Extracting Keywords . . . 14
3.1.3 Computing Keyword Relevances . . . 16
3.2 Documents Similarity . . . 18
3.3 Approximating the Similarity Between Two Documents . . . 20
3.3.1 Overview . . . 20
3.3.2 Estimating the Cosine Distance . . . 22
3.3.3 Reliability of the Relative Frequency . . . 23
3.3.4 Generating Random Hyperplanes . . . 24
3.3.5 Fingerprints . . . 25
3.3.6 Comparison of Fingerprints with Shingles . . . 25
3.4 Approximating the Similarity Between Users and Documents . . . 26
3.4.1 User Profile Definition . . . 26
3.4.2 Storing the Visited Articles . . . 27
3.4.3 Using Vectors to Represent the User Profile . . . 27
3.4.4 Using Fingerprints to Represent the User Profile . . . 29

4 Design 34

4.1 Overview . . . 34

4.2 User Interface . . . 34

4.3 Back-end . . . 36

4.4 Preprocessing . . . 38

5 Implementation Details 39

5.1 Combining Two Fingerprints . . . 39

5.2 Fast Computation of the Angle . . . 39

5.3 Storing the Visited Articles . . . 40

5.4 Tracking the Followed Recommendations . . . 41

5.5 Handling Keywords . . . 42

5.6 Used Technologies . . . 43

5.6.1 Apache HTTP Server and Mod_python . . . 43

5.6.2 Pycrypto . . . 44

5.6.3 RPyC . . . 44

5.6.4 Beautiful Soup . . . 44

5.6.5 Natural Language Toolkit . . . 45

6 Future Work 46

7 Conclusions 48

A Analysis of the Adobe Community Articles 53

B Results 56

B.1 Overview . . . 56

B.2 Document versus Document . . . 57

B.3 User versus Document . . . 59

B.4 Response Time . . . 61

C Code 62

C.1 Splitting Paragraphs into Sentences and Sentences into Words . . . 62

C.2 Extracting Keywords . . . 63

C.3 The Backend . . . 69

C.4 The Front-end . . . 71

List of Figures

1.1 Screenshot of some suggested articles 8

3.1 English most common words frequency [21] 17
3.2 ρ intersecting γ 21
3.3 Plane and vectors 24
3.4 Updating the user profile 28
3.5 Speed comparison: fingerprinting vs cosine product. Note that fingerprinting includes $2.9ms$ the time required to fingerprint a vector (eg. user profile or a query) 33

4.1 Flow 35
4.2 JavaScript code to insert the generated HTML code 35
4.3 Snippet of the generated HTML code 36
4.4 An example of a suggested article returned by the back-end (presented as a Python dictionary) 37

5.1 Fingerprints Bitwise XOR 40
5.2 Fast sideway-sum for 32bits integers 40
5.3 Probability of a false positive based on the number of elements inserted for $m = 768$ and $k = 9$ (see equation 3.21). 41
5.4 Generating suggestion IDs 42
5.5 Logging suggestion IDs 42
5.6 A PATRICIA trie 44

A.1 Histogram of the numbers of keywords in a vector. 54
A.2 Histogram of all keywords relevances 54
A.3 Graph of the most relevant keyword from each document 55
A.4 Cumulative number of distinct words (documents shuffled). 55

B.1 Engine is better than a monkey 58
B.2 Precision when the relevance of the title is adjusted. 58
B.3 Keyword of a single word vs keyword of multiple words 58
B.4 Precision when the relevance of the code is adjusted 59
B.5 Number of clicks for both fingerprint based and vector based update algorithm depending on the index of the page visited 60
B.6 Evolution of precision with the rank of the page visited 60
B.7 Average queries per second (qps) given the number of clients 61

Chapter 1

Introduction

1.1 Overview

With the growing amount of content available on the Internet it is important to help users to access faster the information they are searching by filtering the irrelevant content and suggesting the content the users might look for.

Recommending entertaining content differs from recommending educational content in that the users seeking entertaining content want more quantity, while the others want more quality and less quantity.

The *Adobe Community Engine* [17] is an integrated web environment for Adobe customers that provides community-based instruction, inspiration, and support in a dynamic collaboration environment.

There is no question that the problem of finding relevant articles for a user is hard. Currently Adobe Community Engine provides a search engine which users can use to find articles based on an arbitrary user query.

The goal of this project is to extend the search functionality with a fast, personalised, adaptive and high precision content-based filtering system. When a user visits a page he or she will be suggested a list of relevant articles based on the past visited articles. The recommendation engine should try to guess the user intent and should refine its results as long as the user continues to browse the site for more information.

Good and fast suggestions are expected to increase the number of satisfied users which hopefully will attract more users and will encourage them to expand and improve the available content on Adobe Community engine.

The recommendation engine should be a stateless web-service easy to integrate

Posted by Cryptodude 2006-10-18 11:06:17.0
Avg. rating of 3.72

Title

A 3-state checkbox in a TreeItemRenderer

Problem

Trees are commonly used to represent file systems. Often the user needs to select several items within several folders and take an action (copy, delete, print, ...) on them, so, there needs to be some visual mechanism for indicating that a node is selected. A checkbox is typically used to represent selection. What we need is a tree of 3-state checkboxes.

Solution

There are three main aspects to the solution: 1. A TreeItemRenderer is created to place a CheckBox control at each node in the tree. 2. An image of a tiny black box is painted on top of the CheckBox when the CheckBox is in the third state. 3. The underlying data model for the tree needs to contain an attribute representing the state of the CheckBox.

Detailed explanation

Selection/de-selection of a parent node should cause children nodes to be selected/de-selected. In the case that a parent node has some children that are selected and some that are not selected, a simple 2-state check box won't do - this third state (where some of the children of a node are selected and some are not) cannot be represented by the selected property of a check box, which is a Boolean.
The solution lies in the implementation of the TreeItemRenderer class. I have created an ActionScript class called CheckTreeRenderer for this purpose.

The first thing to do is override the createChildren method. This method is responsible for creating each node in the tree. Here we create a CheckBox and an Image.

```
override protected function createChildren():void
{
    super.createChildren();
    myCheckBox = new CheckBox();
    myCheckBox.setStyle( "verticalAlign", "middle" );
    myCheckBox.addEventListener( MouseEvent.CLICK, checkBoxToggleHandler );
    addChild(myCheckBox);
    myImage = new Image();
    myImage.source = inner;
    myImage.addEventListener( MouseEvent.CLICK, imageToggleHandler );
    myImage.setStyle( "verticalAlign", "middle" );
    addChild(myImage);
}
```

Each child control, CheckBox and Image, needs to handle mouse clicks, so we create an EventListener for each:

Technologies

FLEX AIR FLASH COLDFUSION

Tags

COMPONENTS

Related recipes

Checkbox 3 state
sephiroth
Rating: 3.15
Similarity: 1.77

Delete a node from XML / XMLListCollection
Born2code
Rating: 2.53
Similarity: 1.70

Tree.getDisplayIndex().
sengung1234
Rating: 2.91
Similarity: 1.68

[Handle the tree items]
Giorgio Natili
Rating: 3.31
Similarity: 1.59

Change the icon of a tree node
Fedele Marotti
Rating: 2.88
Similarity: 1.55

Figure 1.1: Screenshot of some suggested articles

with Adobe Community Engine.

1.2 Use Cases

There are many uses cases of a recommendation engine some of which are listed below:

- The user visits the Adobe Cookbooks sites searching for an article. On the right side of the page he or she is presented with a list of related articles based on his or her session browsing history.
- The previous use case can be extended to suggest articles from online journals like IEEE (http://www.ieee.org/) or ACM (http://portal.acm.org/), developers' weblogs or other technical sites.
- When a user starts a new thread on a forum he or she is suggested with a list of articles matching the topic.

- The user visits the Adobe Community Engine site searching for a solution to a particular problem. Together with the relevant articles the engine presents suitable conversations from the forums. The user can check the comments from other user or provide a better one.

- The engine can be adapted to work as a search engine or a filter: the user input is expanded and modeled as a small document which is used to retrieve only documents relevant to the user and related to the query.

1.3 Contributions

This thesis makes the following contributions:

- Introduces a fast way to estimate the cosine product of two document vectors by the means of fingerprints. The approximation error is shown to be in a controllable limit.

- Describes how to generate document fingerprints in linear time and with constant additional memory.

- Shows how the fingerprints can be reverted to the vector representation using the cosine product.

- Presents a way to combine two fingerprints and relates it to the linear combination of the corresponding vectors.

- Shows how the fingerprints can be used to profile the user behavior.

Section 3.4.4 lists the main advantages of using the fingerprints and presents a speed comparison against the dot product.

1.4 Outline

- The *Introduction* chapter introduces the recommendation engine, the use cases and the contributions of the dissertation.

- The *Related Work* chapter presents some related work and other recommendation engines available.

- The *Algorithm* chapter explains the theoretical details of the recommendation engine, describes how the users and the documents are modeled and defines the fingerprint.
- The *Design* and *Implementation Details* chapters describe how the recommendation engine was implemented.
- The *Future Work* chapter shows some possible improvements and extensions to the recommendation engine.
- In the *Analysis of the Adobe Community Articles* appendix a statistical analysis of the data set used by the recommendation engine can be found.
- Finally, the *Results* shows the ability of the recommendation engine to find relevant articles.

Chapter 2

Related Work

There are many systems that tries to solve the problem of finding relevant documments from a huge collection. Most recommendation systems can be classified in *content-based* and *collaborative* filtering systems.

The content-based filtering systems analyse the documents' content employing techniques from information retrieval theory and try to find similar documents. For example, rollSense (http://www.rollsense.com) analyses blog posts and suggests similar articles from other sources. The suggested articles are pre-computed and presented to the user when he or she visits the site. Other examples of systems taking this approach include Ringo [30] and GroupLens [27].

Youtube (http://www.youtube.com) is a good example of a collaborative filtering system. It uses the *user-video graph to provide personalized video suggestion for users* [5]. For each video another set of videos viewed by other users who also viewed the former video is suggested.

Google (http://www.google.com), the most popular search engine, uses a combination of content-based and collaborative filtering to find relevant web pages on the Internet. The pages are ranked on many factors including the hyperlinks between them, users' behavior, query's language, geographical location of the user, etc.

Balabanović describes in [4] *an adaptive recommendation service which seeks to adapt to its users, providing increasingly personalized recommendations over time*. His engine is a hybrid system that uses both the collaborative approach and the content-based one.

Another work on *automatically generating hypertext links both within and between documents on the basis of semantic similarity* was made by Green in [16].

His article describes how to build and analyze *lexical chains* — sequences of semantically related words — to find *semantic links* — links that connect parts of documents on basis of their semantic similarity.

News@hand *is a news recommender system which makes use of semantic technologies to provide several online news recommendation services.* Its authors, Cantador et al., evaluate in [10] *a model that personalises the order in which news articles are shown to the user according to his long-term interest profile.*

Chapter 3

Algorithm

This chapter presents how the recommendation engine computes the similarity between documents and/or users.

The first section describes how the document is represented and how it's representation is constructed. The next section defines the similarity between two documents. The third section shows how to approximate the similarity and the last section extends it to the similarity between a user and the documents.

3.1 Representing Documents in the Vector Space

3.1.1 Overview

This section introduces how the documents are processed and the mathematical model used to describe a document.

In order to get the relevant documents from a collection of documents a value to estimate the similarity between two documents is required. First the documents are modeled as a feature vector which is a collection of (keywords, relevance) pairs:

$$\vec{v} \triangleq \{\kappa_1 : v_{\kappa_1}, \kappa_2 : v_{\kappa_2}, \ldots\} \tag{3.1}$$

A *keyword* κ can be anything characteristic to the document — eg. a tag, a word or a phrase. The *relevance* of a keyword κ (ie. v_κ) is a real positive number specifying the degree of importance the keyword has in describing the document. If a keyword, κ is unrelated to a document then its relevance is $v_\kappa = 0$ and can be omitted from the document representation. In the rest of the thesis *document* will be used to mean the document feature vector, unless inferred from the context.

Following a common approach the smaller the distance between two documents, the more similar they are. The distance can be any function like Hamming or Euclidean distance, but one that behaves particularly well for text is the cosine similarity of the two documents which is described in section 3.2.

3.1.2 Extracting Keywords

To model a document as a feature vector like in 3.1, first the keywords are extracted from the document using the next steps:

- First the document is split into sentences using simple regular expressions. For example the paragraph:
  - *The second half and in 2009, we're going to see a lot more deterioration. This is just the beginning. There are some clear-cut trends among companies that are struggling the most, including the ones below.*

  is split into the following sentences:
  - *The second half and in 2009, we're going to see a lot more deterioration.*
  - *This is just the beginning*
  - *There are some clear-cut trends among companies that are struggling the most, including the ones below.*
- Then each sentence is split into lowercase words delimited by spaces. For example from the above sentences the word list result (spaces are used as delimiters):
  - *the second half and in 2009 we're going to see a lot more deterioration.*
  - *this is just the beginning*
  - *there are some clear-cut trends among companies that are struggling the most including the ones below*
- Next punctuation, numbers and noise words are removed. The above lists are reduced to:
  - *second half deterioration*
  - *beginning*

  - *clear-cut trends companies struggling including*

- Because a word can have different forms (like gerund or plural) which all refer to the same concept, the words resulted from the previous step are stemmed using the Porter stemming algorithm [26].

  - *second half deterioar*
  - *begin*
  - *clear-cut trend compani struggl includ*

- Because consecutive words give a better representation of the concept of the document the keywords extracted are all 1-gram, 2-gram, 3-gram, 4-gram and 5-gram words from each sentence. For example *chang_background* is more relevant for a document than only *chang* or *background* and helps discriminate between documents about *changing the background* and documents about *changing the contents of the window in the background.* The table B.3 gives an experimental comparision between single and multiple words keywords and shows that the latter gives better results. The keywords extracted from the example paragraph are:

  - *second half deterioar second_half half_deterioar second_half_deterioar*
  - *begin*
  - *clear-cut trend compani struggl includ compani_struggl* ...

- The last step is to remove all keywords that are present in only very few documents improving both the performance and the quality. Many sentence fragments aren't actually meaningful concepts (eg. *second_half_deterioar*) but some random association which are unlikely to appear anywhere else. Removing such keywords increases the similarity between documents.

Text versus Code

Many articles from the Adobe Community database contain fragments of code as examples. Because code is more verbose than text and carries less semantics it's important to treat it differently.

The code fragments are detected knowing that they usually appear inside < *pre* > or < *code* > HTML tags. These tags are used to display the text with a fixed-size font which is often used by programmers when they write and read code.

The engine treats code differently from text: a sentence is considered a single line of source and no stemming is done. After the processing of the code fragment usually only the function names remain.

Because the keywords resulted from the code carry less semantics their relevance is adjusted to one third. The code is not eliminated because there are some articles with no text so the recommendation is done solely on the function names. The table B.4 summarizes the results for different coefficients of the code keywords.

Other approaces: Lexical Compounds

Allan and Raghavan studied in [2] many part-of-speech patterns and found that the lexical compounds of the following form:

$$\{adjective?\ noun+\}$$

give a very good representation of the document.

In spite of that, this approach failed for Adobe Community articles giving terrible results. One explanation that the author came up with is that most posts contain solutions to different problems and the relevant phrases are actions like *changing the background* or *using a checkbox to filter items* which don't match the above pattern. Nevertheless, other part-of-speech patterns may be investigated.

3.1.3 Computing Keyword Relevances

The function of keyword relevance is to decide which keywords describe best the document.

The simplest idea is to consider the relevance of a keyword κ as the ***t****erm* ***f****requency of κ in the document*, tf_κ.

$$tf_\kappa \triangleq \frac{\text{\# of occurences of } \kappa \text{ in the document}}{\text{\# of keywords in the document}} \tag{3.2}$$

Using the frequency to compute the keywords relevance has some drawbacks. Most languages including English and Romanian have stop words, or noise words, like *the, I, it, me, and, although.* Many applications to natural language processing (NLP) maintain a list of such words which are removed before processing the text. A list of stop words can be found at [1].

Many of the noise words are the most common words in the language. Adam Kilgarriff analyzed the *British National Corpus* [7], *a 100 million word collection*

frequency	word	frequency	word
6.18%	the	4.23%	is, was, be, are, ’s (= is), were, been, being, ’re, ’m, am
2.94%	of	2.68%	and
2.46%	a, an	1.80%	in, inside
1.62%	to	1.37%	have, has, have, ’ve, ’s (= has), had, having, ’d (= had)
1.27%	he, him, his	1.25%	it, its
1.17%	I, me, my	0.91%	to
0.86%	they, them, their	0.86%	not, n’t, no
0.83%	for	0.83%	you, your
0.70%	she, her	0.65%	with
0.64%	on	0.62%	that

Figure 3.1: English most common words frequency [21]

of samples of written and spoken language and produced the statistics from in 3.1. The first 20 most common words in English represent more than one third of the words used. The engine uses a list of the most common 332 English words.

However, ignoring noise words is not enough. For example, if all documents in a collection talk about *memory* the word *memory* brings little to no information to distinguish between any two documents. Nevertheless, because of the high frequency of the word *memory* within documents it will falsely appear as an important concept increasing the similarity between all documents.

The solution is to decrease the weighting of the keywords which are very frequent in the collection of documents. Salton and Buckley describe in [28] various attempts to eliminate indiscriminating words. The paper shows that the formula which gives the best empirical results, called *term frequency - inverse document frequency (tf-idf)*, is

$$d_\kappa = tf_{document,\kappa} * \log \frac{1}{df_\kappa} \tag{3.3}$$

where the ***d**ocument **f**requency* of the keyword κ, df_κ, is

$$df_\kappa \triangleq \frac{\#\text{ of documents } \kappa \text{ occurs in}}{\#\text{ of documents in the collection}} \tag{3.4}$$

The tf-idf formula has two advantages

- The weight of the stop words will be (almost) 0. If κ is a very frequent word then df_κ is close to 1 and $\log \frac{1}{df_\kappa}$ is close to 0.

Even though it is still a good idea to remove the noise words. Doing so, will improve the overall performance of the process.

- The weight of the indiscriminating words will be low.

The only disadvantage of tf-idf is that for each keyword its document frequency must be computed beforehand which requires an initial pass over the entire collection of documents.

Keyword's Position in the Document

Chiriță et al. show in [11] how to improve the keyword's relevance formula based on the keyword's position in text. The current implementation of the engine currently does not use their approach.

Nevertheless, the relevances of the keywords extracted from the title are increased to reflect the fact that they contain more semantic information than keywords from the document's content. When a user visits a page and sees the recommendations he or she matches the titles with what he or she is currently searching. The side effect of the increased relevance of the title keywords is a better match between the user profile and the titles of the articles suggested.

3.2 Documents Similarity

A distance frequently used in text information retrieval is the *cosine similarity* between the two documents represented by the vectors $\vec{u}$ and $\vec{d}$:

$$\cos\angle(\vec{u},\vec{d}) \in [-1,+1] \tag{3.5}$$

A cosine distance of 1 means that the articles are (almost) identical and a cosine distance of 0 or lower means that the articles are totally unrelated. The recommendation engine uses the cosine distance to measure the similarity of two documents or between the user profile and a document.

Definition 3.2.1. The dot product, or the scalar product, of two vectors $\vec{u}$ and $\vec{v}$ denoted by $\vec{u}\cdot\vec{v}$ is:

$$\vec{u}\cdot\vec{v} \triangleq \sum_{\kappa} u_\kappa \cdot v_\kappa$$

Theorem 3.2.2. *If $\vec{u}$ and $\vec{v}$ are two vectors then*

$$\cos\angle(\vec{u},\vec{v}) = \frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\cdot\|\vec{v}\|}$$

Proof. From the law of cosines

$$\|\vec{v}-\vec{u}\|^2 = \|\vec{v}\|^2 + \|\vec{u}\|^2 - 2\cdot\|\vec{v}\|\cdot\|\vec{u}\|\cdot\cos\angle(\vec{u},\vec{v})$$

which leads to

$$\begin{aligned}
\cos\angle(\vec{u},\vec{v}) &= \frac{\|\vec{v}\|^2 + \|\vec{u}\|^2 - \|\vec{v}-\vec{u}\|^2}{2\cdot\|\vec{v}\|\cdot\|\vec{u}\|} \\
&= \frac{\sum_\kappa v_\kappa^2 + \sum_\kappa u_\kappa^2 - \sum_\kappa (v_\kappa - u_\kappa)^2}{2\cdot\|\vec{v}\|\cdot\|\vec{u}\|} \\
&= \frac{\sum_\kappa v_\kappa^2 + u_\kappa^2 - (v_\kappa - u_\kappa)^2}{2\cdot\|\vec{v}\|\cdot\|\vec{u}\|} \\
&= \frac{\sum_\kappa v_\kappa \cdot u_\kappa}{\|\vec{v}\|\cdot\|\vec{u}\|} \\
&= \frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\cdot\|\vec{v}\|}
\end{aligned}$$

□

If an article $\vec{d_1}$ is a about *memory leaks* words like *memory* and *leak* will be very frequent. When compared to another article $\vec{d_2}$ about *memory leaks* the values of the terms $d_{1,memory}\cdot d_{2,memory}$ and $d_{1,leak}\cdot d_{2,leak}$ from the dot product (3.5) will be high increasing the value of the cosine similarity.

Corollary 3.2.3. *Let $\vec{u} = \{\kappa : u_\kappa, \ldots\}$ be an arbitrary vector and $\vec{q} = \{\kappa : 1\}$ a query vector. Then:*

$$\frac{u_\kappa}{\|\vec{u}\|} = \cos\angle(\vec{u},\vec{q}) \tag{3.6}$$

Proof.

$$\begin{aligned}
\cos\angle(\vec{u},\vec{q}) &= \frac{\vec{u}\cdot\vec{q}}{\|\vec{u}\|\cdot\|\vec{q}\|} \\
&= \frac{\sum_\kappa u_\kappa \cdot q_\kappa}{\|\vec{u}\|\cdot\|\vec{q}\|} \\
&= \frac{u_\kappa}{\|\vec{u}\|}
\end{aligned}$$

□

3.3 Approximating the Similarity Between Two Documents

3.3.1 Overview

The online computation of the cosine similarity with all Adobe Cookbooks articles is time prohibitive. The Wikipedia [31] article on *Local Sensitive Hashing* describes a neat idea to approximate the cosine similarity between two vectors.

Definition 3.3.1. Let γ be a plane. The binary relation $\sim_\gamma$ exists between two vectors iff they lie on the same side of γ.

Theorem 3.3.2. $\sim_\gamma$ *is an equivalence relation. For any three vectors* $\vec{u}$, $\vec{v}$ *and* $\vec{t}$ *all the following holds true:*

- *Reflexivity:* $\vec{u} \sim_\gamma \vec{u}$
- *Symmetry: if* $\vec{u} \sim_\gamma \vec{v}$ *then* $\vec{v} \sim_\gamma \vec{u}$
- *Transitivity: if* $\vec{u} \sim_\gamma \vec{v}$ *and* $\vec{v} \sim_\gamma \vec{t}$ *then* $\vec{u} \sim_\gamma \vec{t}$

Definition 3.3.3. Let $Pr[\vec{u} \sim_\gamma \vec{d}]$ be the probability that two vectors $\vec{u}$ and $\vec{d}$ to lie on the same side of an arbitrary plane γ.

Theorem 3.3.4. *If* $\vec{u}$ *and* $\vec{d}$ *are two vectors then*

$$Pr[\vec{u} \sim_\gamma \vec{d}] = 1 - \frac{\angle(\vec{u}, \vec{d})}{\pi}$$

Proof.

1. Case $\vec{u} = \vec{d}$.

   If $\vec{u} = \vec{d}$ then $\angle(\vec{u}, \vec{d}) = 0$ and $Pr[\vec{u} \sim_\gamma \vec{d}] = 1 = 1 - \frac{0}{\pi} = 1 - \frac{\angle(\vec{u}, \vec{d})}{\pi}$

2. Case $\vec{u} \neq \vec{d}$.

   ρ be the plane induced by the vectors $\vec{u}$ and $\vec{d}$ and the origin O.

   ρ and γ have at least one point in common, the origin O, so they must have at least one line in common. Let's suppose that the two planes do not coincide and let $\delta = \rho \cap \gamma$ be the common line.

   The line δ divides the plane ρ in two half-planes. The upper half-plane is included in the upper half-space induced by γ and the lower half-plane is included in the lower half-space.

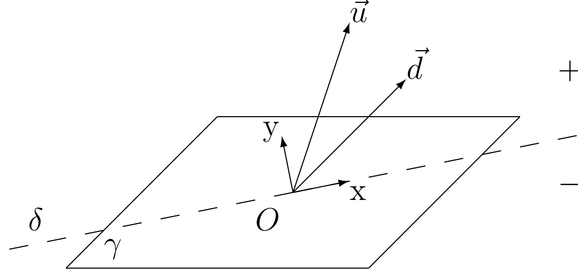

Figure 3.2: ρ intersecting γ

In ρ define a coordinate system xOy such that the support line of the $\vec{Ox}$ axis is δ and $\vec{Oy}$ axis lies in the upper half-space induced by the plane γ.

Using the coordinate system xOy, define $\widehat{aOb}$ as the positive angle (ie. the angle measured anticlockwise) between two vectors $\vec{a}$ and $\vec{b}$. Of course the identities hold true:

(a) $\angle(\vec{a}, \vec{b}) = |\widehat{aOb}| = |\widehat{bOa}|$

(b) $\widehat{aOb} = -\widehat{bOa}$

Let $\alpha = \angle(\vec{u}, \vec{d})$. Two disjunct cases can be identified:

(a) Case $\alpha = \widehat{dOu}$ (see figure 3.2).

  i. if $\widehat{xOu} \in [0, \alpha)$ then $\vec{u}$ is in the upper half-plane of ρ, while $\vec{d}$ is in the lower half-plane.

  ii. if $\widehat{xOu} \in [\alpha, \pi)$ then $\vec{u}$ and $\vec{d}$ are both in the upper half-plane of ρ.

  iii. if $\widehat{xOu} \in [\pi, \pi + \alpha)$ then $\vec{u}$ is in the lower half-plane of ρ, while $\vec{d}$ is in the upper half-plane.

  iv. if $\widehat{xOu} \in [\pi + \alpha, 2\pi)$ then $\vec{u}$ and $\vec{d}$ are both in the lower half-plane of ρ.

Given that the angle $\widehat{xOu}$ is uniform distributed in the interval $[0, 2\pi)$

the following results:

$$\begin{aligned} Pr[\vec{u} \sim_\gamma \vec{d} | \angle(\vec{u}, \vec{d}) = \widehat{dOu}] &= \frac{(\pi - \alpha) + (2\pi - (\pi + \alpha))}{2\pi} \\ &= \frac{2\pi - 2\alpha}{2\pi} \\ &= 1 - \frac{\angle(\vec{u}, \vec{d})}{\pi} \end{aligned}$$

(b) Case $\angle(\vec{u}, \vec{d}) = \widehat{uOd}$. Analogous

$$Pr[\vec{u} \sim_\gamma \vec{d} | \angle(\vec{u}, \vec{d}) = \widehat{uOd}] = 1 - \frac{\angle(\vec{u}, \vec{d})}{\pi}$$

From the law of total probability

$$\begin{aligned} Pr[\vec{u} \sim_\gamma \vec{d}] &= Pr[\angle(\vec{u}, \vec{d}) = \widehat{uOd}] \cdot Pr[\vec{u} \sim_\gamma \vec{d} | \angle(\vec{u}, \vec{d}) = \widehat{uOd}] \\ &+ Pr[\angle(\vec{u}, \vec{d}) = \widehat{dOu}] \cdot Pr[\vec{u} \sim_\gamma \vec{d} | \angle(\vec{u}, \vec{d}) = \widehat{dOu}] \\ &= 1 - \frac{\angle(\vec{u}, \vec{d})}{\pi} \end{aligned}$$

□

Corollary 3.3.5. *The cosine distance between two vectors $\vec{u}$ and $\vec{d}$ is*

$$\cos \angle(\vec{u}, \vec{d}) = \cos(\pi \cdot (1 - Pr[\vec{u} \sim_\gamma \vec{d}]))$$

3.3.2 Estimating the Cosine Distance

The probability that two vectors $\vec{u}$ and $\vec{d}$ lie on the same side of a plane can be estimated using the *relative frequency*. Consider an experiment of choosing a random plane and checking whether the two vectors lie on the same side of the plane. Then $Pr[\vec{u} \sim_\gamma \vec{d}]$ can be estimated as:

$$Pr[\vec{u} \sim_\gamma \vec{d}] \approx p = \frac{\text{\# of times the } \vec{u} \text{ and } \vec{d} \text{ lie on the same side}}{\text{\# of experiments}} \tag{3.7}$$

For example if from 900 planes the two vectors lie on the same side of 600 of them then the angle between the two vectors can be approximated as $\angle(\vec{u}, \vec{v}) \approx \pi \cdot (1 - \frac{600}{900}) = \frac{\pi}{3}$

3.3.3 Reliability of the Relative Frequency

lookup on T Test

The reliability of the relative frequency, p, to estimate the $Pr[\vec{u} \sim_\gamma \vec{d}]$ for any two vectors $\vec{u}$ and $\vec{d}$ can be measured using the confidence interval for the normal distribution

$$Pr[\vec{u} \sim_\gamma \vec{d}] \in p \pm z \cdot \sigma_p \tag{3.8}$$

where z depends on the level of confidence desired and σ_p is the standard error of a proportion.

$$\sigma_p = \sqrt{\frac{Pr[\vec{u} \sim_\gamma \vec{d}] \cdot (1 - Pr[\vec{u} \sim_\gamma \vec{d}])}{N}} \approx \sqrt{\frac{p \cdot (1-p)}{N}} \tag{3.9}$$

where N is the number of the conducted experiments.

The value of z is standard and can be looked up in the *Z-Table* for the normal distribution. For example for a confidence of:

- 68.27% the value of z is 1.00
- 95.45% the value of z is 2.00
- 99.73% the value of z is 3.00

The largest confidence interval is obtained when $p = 0.5$:

$$p \pm z \cdot \sigma_p \subseteq p \pm z \cdot \sigma_{0.5} = p \pm z \cdot \frac{0.5}{\sqrt{N}} = p \pm \frac{z}{\sqrt{4N}} \tag{3.10}$$

Therefore with a probability of 95.45% (corresponding to z equals to 2.00):

$$Pr[\vec{u} \sim_\gamma \vec{v}] \in p \pm \frac{2}{\sqrt{4N}} = p \pm \frac{1}{\sqrt{N}} \tag{3.11}$$

Note that in order to shrink the confidence interval by a factor of 2 the number of experiments must be increased by a factor of 4. In the implementation N, the number of experiments, was chosen to be 3072.

$$Pr[\vec{u} \sim_\gamma \vec{v}] \in p \pm 0.0180, \text{with } 95.45\% \text{ probability} \tag{3.12}$$

The standard deviation and the average error computed over all pairs of Cookbooks articles were 0.0091 and 0.0073.

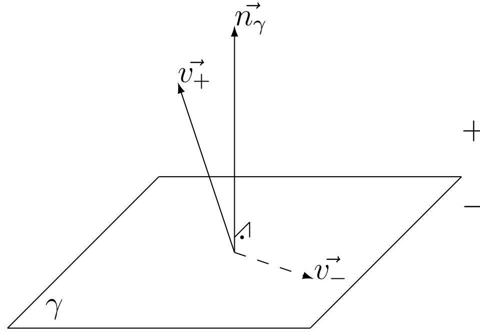

Figure 3.3: Plane and vectors

3.3.4 Generating Random Hyperplanes

Generating a random plane, γ, is equivalent to generating a random vector, $\vec{n_\gamma}$, normal to the plane. The side of γ on which an arbitrary vector $\vec{v}$ lies, $h_\gamma(\vec{v}) = \pm 1$, is induced by the angle between $\vec{v}$ and the normal vector, $\vec{n_\gamma}$. If the angle is lower than $\frac{\pi}{2}$ then $\vec{v}$ lies on the *plus* side of the plane, otherwise it lies on the *minus* side (see 3.3). $h_\gamma(\vec{v})$ can be expressed as:

$$\begin{aligned} h_\gamma(\vec{v}) &= sign(\cos \angle(\vec{n_\gamma}, \vec{v})) \\ &= sign(\|\vec{n_\gamma}\| \cdot \|\vec{v}\| \cdot \cos \angle(\vec{n_\gamma}, \vec{v})) \\ &= sign(\vec{n_\gamma} \cdot \vec{v}) \end{aligned} \tag{3.13}$$

The values of $n_{\gamma,\kappa}$ can be pregenerated and hard coded in the sources as constants. However since the number of random planes (3072 in the implementation) and the number of keywords extracted from the collection of documents are large the memory requirements would be prohibitive. Nevertheless the property of the hash functions to return uniform random distributed numbers can be exploited to compute $n_{\gamma,\kappa}$:

$$n_{\gamma,\kappa} = hash(SEED_\gamma, \kappa) \in (-1, +1) \tag{3.14}$$

where $SEED_\gamma$ is an arbitrary number unique to each plane. Note that since the values of v_κ are always positive it's important that the values of $n_{\gamma,\kappa}$ contain both negative and positive numbers, otherwise every vector will be on the same side of each plane.

Finally the value of $h_\gamma(\vec{v})$ is:

$$h_\gamma(\vec{v}) = sign\left(\sum_{\kappa \in \text{set of keywords from } \vec{v}} hash(SEED_\gamma, \kappa) \cdot v_\kappa\right) \tag{3.15}$$

3.3.5 Fingerprints

Now, computing the angle between two vectors is fairly simple. First generate N random seeds, $SEED_{\gamma_i}(i = 1 \ldots N)$, corresponding to N random planes, $\gamma_i(i = 1 \ldots N)$. Then for a vector, $\vec{v}$, compute the fingerprint of N bits:

$$\begin{aligned} fp(\vec{v}) &= \overline{h_{\gamma_1}(\vec{v}) > 0, h_{\gamma_2}(\vec{v}) > 0, \ldots, h_{\gamma_N}(\vec{v}) > 0} \\ &= \overline{fp_1(\vec{v}), fp_2(\vec{v}), \ldots, fp_N(\vec{v})} \end{aligned} \tag{3.16}$$

(3.7) can be rewritten in terms of counting the number of different bits between the two fingerprints:

$$Pr[\vec{u} \sim_\gamma \vec{v}] \approx fp(\vec{u}) \oplus fp(\vec{v}) \triangleq \frac{\sum_{i=1}^{N} fp_i(\vec{u}) \cdot fp_i(\vec{v})}{N} \tag{3.17}$$

Together with (3.3.4) it results that

$$\begin{aligned} \angle(\vec{u}, \vec{v}) &\approx \pi \cdot fp(\vec{u}) \oplus fp(\vec{v}) \\ \cos \angle(\vec{u}, \vec{v}) &\approx \cos\left(\pi \cdot fp(\vec{u}) \oplus fp(\vec{v})\right) \end{aligned} \tag{3.18}$$

It's important to note that a fingerprint encode only the direction of its vector and not its magnitude.

The recommendation engine uses fingerprints to compute the cosine similarity between documents and user profiles.

3.3.6 Comparison of Fingerprints with Shingles

Broder presents in [9] an algorithm to detect near-duplicate documents. For each document D a *sketch*, $\overline{S_D}$, of N integers is constructed.

$$\overline{S_D} = (S_{1,D}, S_{2,D}, \ldots, S_{N,D}) \tag{3.19}$$

The sketch is used to estimate the *resemblance* $r(A, B)$ (also known as the *Jaccard index* [18]) of two documents A and B:

$$r(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \tag{3.20}$$

where S_D is the set of q-gram words in D. Broder indicates *that a high resemblance (that is, close to 1) captures well the informal notion of "near-duplicate" or "roughly the same"* [9].

The author of this thesis was inspired by Broder's paper to use hash functions to fingerprint documents and calculate the distance between them. However the algorithms are different in the following key points:

- Sketches are used to compute resemblance while fingerprints are used to compute the cosine distance. Two different documents that both talk about *memory leaks* have a low resemblance, but a high cosine similarity.
- Sketches retain the set of q-grams present in the document. Fingerprints store the direction of the vector of most relevant keywords extracted from the document Sketches don't use the notion of how relevant is a keyword to a document.
- It's not possible to combine two sketches to form a new sketch. For combining two fingerprints see 3.4.4.

3.4 Approximating the Similarity Between Users and Documents

This section introduces the user profile describes how to relate documents to the user behavior.

3.4.1 User Profile Definition

The user is represented with two components:

- the set of *visited articles* (section 3.4.2)
- the set of *relevant keywords* from user browsing history (section 3.4.3)

The user profile is stored in user's cookies so it must be encoded using the base64 algorithm [20] and limited to 4 KBytes (see section 4.2).

In the rest of this chapter user profile, or profile, will be used to mean the *vector of the relevant keywords*.

3.4.2 Storing the Visited Articles

As the user browses the Cookbooks articles it is important that he or she is not recommended already visited recipes because he or she is unlikely to visit them again. Some popular entertaining sites suffer from this problem which leads to users frustration because they cannot find new material. The problem can be formulated as follows: *Given an article identified by it's URL did the user visit it?*

The set of possible visited articles can be huge even if it is restricted to the collection known to the recommending engine however, the number of visited URLs in a normal browsing session on Cookbooks was empirically measured to be less than 50.

Burton H. Bloom describes in [6] a space-efficient probabilistic data structure that is used to insert into and test the membership of an element of a set. The Bloom filters were initially used for spell checking to store most frequent words. At the time of the invention of the bloom-filters 64 KBytes of memory was considered a de facto standard while the *Oxford English Dictionary* [24] contains over 600.000 definitions.

A Bloom filter is a bit array of m bits and k different hash functions which map each set element to one of the m array positions with a uniform random distribution.

The probability of a false positive when checking for membership is:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-\frac{kn}{m}}\right)^{k} \tag{3.21}$$

where n is the number of inserted elements. The probability of a false positive is minimized when

$$k = \frac{m}{n}\ln 2 \tag{3.22}$$

The section 5.3 details the values of m and k used by the recommendation engine.

3.4.3 Using Vectors to Represent the User Profile

The set of relevant keywords for the user can be represented as a feature vector similar to a document:

$$\vec{u} = \{\kappa_1 : u_{\kappa_1}, \kappa_2 : u_{\kappa_2}, \ldots\} \tag{3.23}$$

After the user visits an article his or her profile is updated to reflect the new set of relevant keywords. A common method to update the user profile is to use the linear combination:

$$\vec{u} \diamond_\xi \vec{d} \triangleq \vec{u} \cdot \xi + \vec{d} \cdot (1 - \xi) \tag{3.24}$$

where $\vec{d}$ is the document and $\xi \in [0, 1]$ is the decay factor. The visual representation of 3.24 can be seen in figure 3.4.

The effect of the decay is to decrease the importance of the earlier visited articles compared to the more recent ones. For example, if $\xi = 0$ the user profile is discarded.

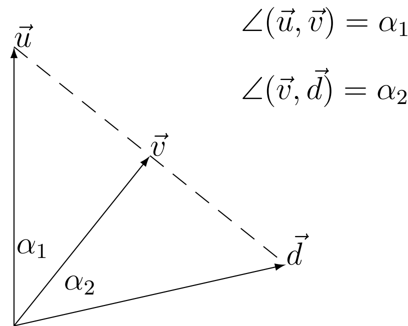

Figure 3.4: Updating the user profile

Advantages and Disadvantages

Keeping the user profile in its vector representation has the advantage of being very easy to work with. The result of the linear combination (3.24) can be manually computed and compared.

Nevertheless this method has several drawbacks:

- Size. On average one (keyword, relevance) pair uses around 16 bytes. To accommodate the size limit of the user cookie (see chapter **??**) the vector is trimmed to the 128 most relevant keywords. The effect of trimming is negligible because empirically the weight of the 100th most relevant keyword is almost two orders of magnitude lower than the weight of the most relevant keyword. Keeping the documents as a list of pairs (keyword, relevance) consumes roughly $16\text{bytes} \cdot 128 = 2078\text{bytes}$ (2730 bytes in base64 [20]).

- Privacy. Nobody should have access to the plain profile of any user consequently the profiles must be encrypted and signed before sent to the user. Encrypting and decrypting user profiles are very slow.

3.4.4 Using Fingerprints to Represent the User Profile

Another method to represent the user profile is to encode the fingerprint of the user feature vector. After visiting a document, $\vec{d}$, the profile is probabilistically updated:

$$\begin{aligned}\vec{u} \circ_{\Psi} \vec{d} \;\triangleq\; & \text{a vector } \vec{\centerdot} \text{ such that} \\ & \quad \text{with probability } \Psi \; h_{\gamma}(\vec{\centerdot}) = h_{\gamma}(\vec{u}) \\ & \quad \text{otherwise } h_{\gamma}(\vec{\centerdot}) = h_{\gamma}(\vec{d}) \\ & \text{where } \gamma \text{ is a random plane}\end{aligned} \tag{3.25}$$

Relation with the Linear Update

The next theorem explores how far $\vec{u} \circ_{\Psi} \vec{d}$ diverges from other documents, $\vec{q}$.

Theorem 3.4.1. *Let $\vec{u}$, $\vec{d}$ and $\vec{q}$ be three arbitrary vectors. If $\vec{v'} = \vec{u} \circ_{\Psi} \vec{d}$ then*

$$\angle(\vec{q}, \vec{v'}) = \Psi \cdot \angle(\vec{q}, \vec{u}) + (1 - \Psi) \cdot \angle(\vec{q}, \vec{d})$$

Proof. From the update algorithm 3.25 it results that:

$$\begin{aligned} 1 - \frac{\angle(\vec{q}, \vec{v'})}{\pi} &= Pr[\vec{q} \sim_{\gamma} \vec{v'}] \\ &= \Psi \cdot Pr[\vec{q} \sim_{\gamma} \vec{u}] + (1 - \Psi) \cdot Pr[\vec{q} \sim_{\gamma} \vec{d}] \\ &= \Psi \cdot (1 - \frac{\angle(\vec{q}, \vec{u})}{\pi}) + (1 - \Psi) \cdot (1 - \frac{\angle(\vec{q}, \vec{d})}{\pi}) \\ &= 1 - \frac{\Psi \cdot \angle(\vec{q}, \vec{u}) + (1 - \Psi) \cdot \angle(\vec{q}, \vec{d})}{\pi} \\ \Longrightarrow & \\ \angle(\vec{q}, \vec{v'}) &= \Psi \cdot \angle(\vec{q}, \vec{u}) + (1 - \Psi) \cdot \angle(\vec{q}, \vec{d}) \end{aligned}$$

□

While both methods to combine two vectors (3.24 and 3.25) are intuitive, a relation between them is not and is provided by the next corollary.

Corollary 3.4.2. *Let $\vec{u}$ and $\vec{d}$ be two vectors. If $\vec{v} = \vec{u} \diamond_{\xi} \vec{d}$ and $\vec{v'} = \vec{u} \circ_{\Psi} \vec{d}$ then*

$$\angle(\vec{v}, \vec{v'}) = \Psi \cdot \angle(\vec{u}, \vec{v}) + (1 - \Psi) \cdot \angle(\vec{d}, \vec{v})$$

With the notations from the notations from the previous corollary, Ψ has to be calculated such that the mean square error

$$Err_\Psi = \sqrt{\frac{\left(\angle(\vec{u},\vec{v}) - \angle(\vec{u},\vec{v'})\right)^2 + \left(\angle(\vec{d},\vec{v}) - \angle(\vec{d},\vec{v'})\right)^2}{2}} \tag{3.26}$$

is minimised. The error Err_Ψ is used to find a vector $\vec{v'}$ that diverges $\vec{u}$ and approaches $\vec{u}$ at the same angles as $\vec{v}$.

Let $\alpha_1 = \angle(\vec{u},\vec{v})$ and $\alpha_2 = \angle(\vec{d},\vec{v})$. Calculating the values of α_1 and α_2 is beyond the scope of this thesis and is left as an exercise for the reader.

From the update algorithm 3.25 it follows that:

$$\begin{aligned}
Pr[\vec{u} \sim_\gamma \vec{v'}] &= Pr[\vec{u} \sim_\gamma \vec{d}] + \Psi \cdot Pr[\vec{u} \nsim_\gamma \vec{d}] \\
&= 1 - \frac{\alpha_1 + (1-\Psi)\cdot\alpha_2}{\pi} \\
\angle(\vec{u},\vec{v'}) &= \pi \cdot Pr[\vec{u} \nsim_\gamma \vec{v'}] = (1-\Psi)\cdot(\alpha_1+\alpha_2) &(3.27)\\
\angle(\vec{d},\vec{v'}) &= \pi \cdot Pr[\vec{d} \nsim_\gamma \vec{v'}] = \Psi\cdot(\alpha_1+\alpha_2) &(3.28)
\end{aligned}$$

The error 3.26 can be rewritten as

$$\begin{aligned}
Err_\Psi &= \sqrt{\frac{(\alpha_1 - (\alpha_1+\alpha_2)\cdot(1-\Psi))^2 + (\alpha_2 - (\alpha_1+\alpha_2)\cdot\Psi)^2}{2}} \\
&= \sqrt{(\alpha_1\cdot\Psi + \alpha_2\cdot(1-\Psi))^2} \\
&= \sqrt{(\Psi\cdot(\alpha_1+\alpha_2) - \alpha_2)^2} &(3.29)
\end{aligned}$$

which is minimized when

$$\Psi = \frac{\alpha_2}{\alpha_1+\alpha_2} \tag{3.30}$$

From 3.27, 3.28 and 3.30 the angles are:

$$\begin{aligned}
\angle(\vec{u},\vec{v'}) &= (\alpha_1+\alpha_2)\cdot(1-\Psi) = \alpha_1 &(3.31)\\
\angle(\vec{d},\vec{v'}) &= (\alpha_1+\alpha_2)\cdot\Psi = \alpha_2 &(3.32)\\
\angle(\vec{v},\vec{v'}) &= \alpha_1\cdot\Psi + \alpha_2\cdot(1-\Psi) = \frac{2\alpha_1\alpha_2}{\alpha_1+\alpha_2} &(3.33)
\end{aligned}$$

The last equations show that Ψ can be used to control the angle between $\vec{u} \circ_\Psi \vec{d}$ and the vectors $\vec{u}$ and $\vec{d}$. There are an infinite number of such vectors located at a fixed angle from the $\vec{u} \diamond_\xi \vec{d}$.

Other Applications

Corollary 3.4.3. *Let $\vec{u}$ and $\vec{d}$ be two vectors and $\vec{v'} = \vec{u} \circ_\Psi \vec{d}$. Let κ be a keyword. If $u_\kappa = d_\kappa = 0$ then $v'_\kappa = 0$.*

Proof. Let $q = \{\kappa : 1\}$. From the corollary 3.2.3

$$\begin{aligned} \cos\angle(\vec{u}, \vec{q}) &= \frac{u_\kappa}{\|\vec{u}\|} = 0 \\ \angle(\vec{u}, \vec{q}) &= \frac{\pi}{2} \end{aligned}$$

Analogous

$$\angle(\vec{d}, \vec{q}) = \frac{\pi}{2}$$

From the theorem 3.4.1:

$$\begin{aligned} \angle(\vec{q}, \vec{v'}) &= \Psi \cdot \angle(\vec{q}, \vec{u}) + (1 - \Psi) \cdot \angle(\vec{q}, \vec{d}) \\ &= \Psi \cdot \frac{\pi}{2} + (1 - \Psi) \cdot \frac{\pi}{2} \\ &= \frac{\pi}{2} \end{aligned}$$

Applying the theorem 3.2.3:

$$\begin{aligned} v'_\kappa &= \|v'\| \cdot \cos\angle(\vec{q}, \vec{v'}) \\ &= \|v'\| \cdot \cos\frac{\pi}{2} \\ &= 0 \end{aligned}$$

□

The previous corollary states that if a keyword is not relevant to the user profile or the document it will not be relevant to the probabilistically updated profile.

Corollary 3.4.4. *Let $\vec{u}$ and $\vec{d}$ be two vectors with $\vec{d} = \{\kappa : -1\}$. If $\Psi = \frac{\pi}{2 \cdot \angle(\vec{d}, \vec{u})} \in [0, 1]$ and $\vec{v'} = \vec{u} \circ_\Psi \vec{d}$ then*

$$v'_\kappa = 0$$

Proof. If $\vec{q} = \{\kappa : 1\}$ is a query vector then $\pi - \angle(\vec{q}, \vec{u}) = \angle(\vec{d}, \vec{u})$. From the theorem 3.4.1:

$$\begin{aligned} \angle(\vec{q}, \vec{v'}) &= \Psi \cdot \angle(\vec{q}, \vec{u}) + (1 - \Psi) \cdot \angle(\vec{q}, \vec{d}) \\ &= \frac{\pi}{2 \cdot \angle(\vec{d}, \vec{u})} \cdot \angle(\vec{q}, \vec{u}) + (1 - \frac{\pi}{2 \cdot \angle(\vec{d}, \vec{u})}) \cdot \pi \\ &= \pi + \frac{\pi}{2 \cdot (\pi - \angle(\vec{q}, \vec{u}))} \cdot (\angle(\vec{q}, \vec{u}) - \pi) \\ &= \frac{\pi}{2} \end{aligned}$$

Applying the theorem 3.2.3 for $\vec{q}$ and $\vec{v'}$

$$v'_\kappa = \|\vec{v'}\| \cdot \cos \angle(\vec{q}, \vec{v'}) = 0$$

□

The last corollary provides means to erase keywords from a vector when only its fingerprint is known.

Advantages and Disadvantages

There are a few advantages to using fingerprinting:

- Small fixed size. The engine's implementation uses fingerprints of 3072 bits (512 bytes when encoded using the base64 algorithm [20]) to represent almost any document of any size. The fingerprint is more than 5 times smaller than the encoded user profile vector.

  Additionally, having a fixed size fingerprint is important when estimating the memory requirements of the back-end (see chapter 4). The size of the fingerprint also makes it very easy to fit in the user's cookie.

- It's fast. On a 2.4GHz processor fingerprinting a vector takes approximately $2.9ms$, while computing the angle between two documents takes just under $0.9us$. Computing the cosine product of two vectors takes about $250us$.

  For one time only operation, computing the similarity using fingerprints is much slower. However, it's important to note that fingerprinting of the documents is done offline in the preprocessing phase and so it can be neglected from the response time.

  A speed comparison between approximation using fingerprints and cosine product is presented in table 3.5.

- No privacy issues. In order to extract any meaningful information from the fingerprint one needs the random plane seeds which are not publicly available. The corollary 3.2.3 provides a method to reverse-engineer the fingerprint.

At the moment of writing this thesis the Adobe Community database contains just over 600 articles and is expected to increase to over 10000 in the following year. This translates in a current increase in speed of more than 40 times when using

fingerprints which leads to much faster responses and more satisfied users. The memory requirements are almost negligibly at current storage size availability, but it's important to consider when extending the collection of documents to include other technical sites, webblogs or forums posts.

# of operations					fingerprinting	cosine product
1	$2.9ms$	+	$0.0009ms$	≈	$2.9ms$	$0.25ms$
10	$2.9ms$	+	$0.009ms$	≈	$2.9ms$	$2.5ms$
100	$2.9ms$	+	$0.09ms$	≈	$3.0ms$	$25ms$
1000	$2.9ms$	+	$0.9ms$	=	$3.8ms$	$250ms$
10000	$2.9ms$	+	$9ms$	=	$11.9ms$	$2.5s$
100000	$2.9ms$	+	$90ms$	≈	$93ms$	$25s$
1000000	$2.9ms$	+	$900ms$	≈	$903ms$	$250s$

Figure 3.5: Speed comparison: fingerprinting vs cosine product. Note that fingerprinting includes $2.9ms$ the time required to fingerprint a vector (eg. user profile or a query)

Chapter 4

Design

4.1 Overview

The feature is composed of three components: the user interface (4.2), the back-end (4.3) and the preprocessing (4.4).

The flow is presented in picture 4.1. The users visits the Adobe Community website (1) which calls the front-end (2) to generate the section displaying the related articles (4). The front-end internally calls the back-end (3) to get the relevant posts.

4.2 User Interface

The user interface is accomplished by the front-end module. The Adobe Community site contains a small JavaScript function (see listing 4.2) which is called every time a page is loaded. The script accesses the front-end service and inserts the returned HTML code into a special section from the original web-page. If for any reason the service fails nothing is displayed.

The front-end is a web-service that generates small HTML snippets (see the example in listing 4.3) which display the recommended articles. The service is available as a regular web-page and provides the following functionality:

- `http://.../service/random_url`. The user is redirected to a random article and his or her profile is cleaned. Mostly used for testing purposes.

- `http://.../service/recommend?url=URL`. After the user visits the article at URL his or her profile is updated and he or she is presented with a list of

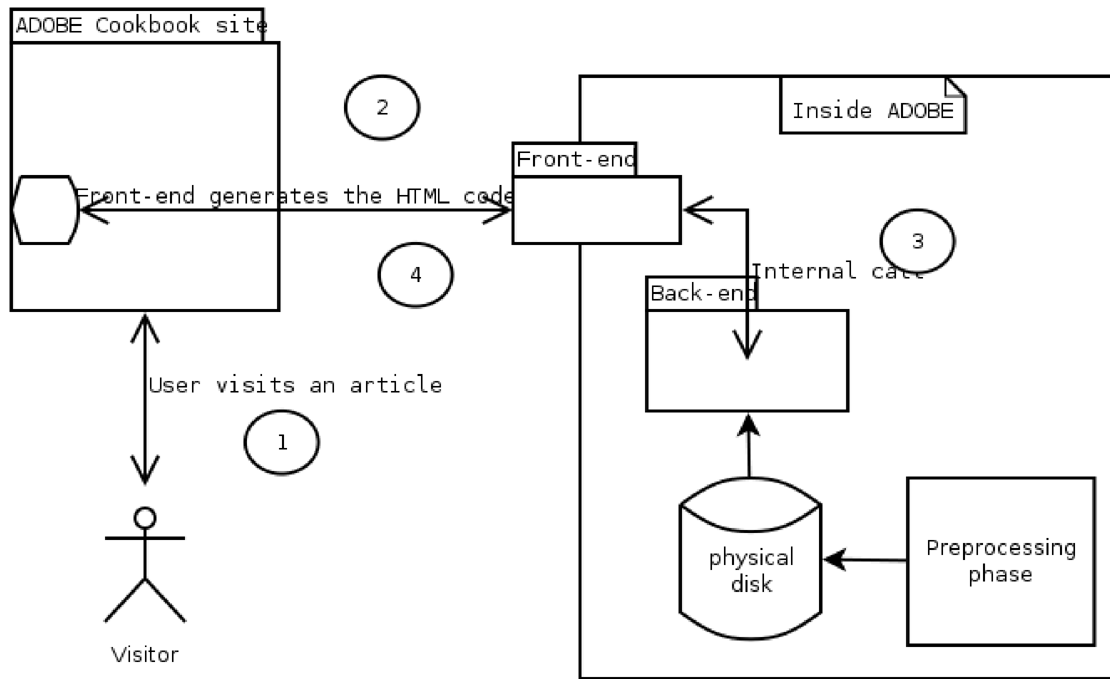

Figure 4.1: Flow

```
1  <script language="javascript">
2  $.ajax({
3      type: "GET",
4      url: "/service/recommend",
5      success: function(html) {
6          document.getElementById('related-recipes')
7                      .innerHTML = html;
8      }
9  });
10 </script>
```

Figure 4.2: JavaScript code to insert the generated HTML code

```
1  <p><a href="http://...">
2    Delaying state change in transitions
3  </a></p>
4  <ul class="meta">
5    <li><a class="ico user"
6              title="SebastianCarlsson"
7              href="http://...">SebastianCarlsson</a></li>
8    <li>Rating: 3.16</li>
9    <li>Similarity: 2.13</li>
10 </ul>
```

Figure 4.3: Snippet of the generated HTML code

relevant articles(as in 4.3). If the *url* parameter is omitted the referrer page is implied.

- http://.../service/redirect?doc_id=ID&dest=URL. Logs that the user followed the recommendation identified by *doc_id* and redirects to URL. This service is used to track the user clicks by replacing the URL to the article with the corresponding redirect service URL.

The front-end stores the user profile in the user cookie [23] with a lifespan of about 15 minutes. Every time the user accesses a page his or her profile is read, sent to the back-end where it's updated to reflect user's action and then stored back in the cookie. The cookie is usually limited to 4 KBytes and the profile must be stored using a limited alphabet, usually using the base64 algorithm [20].

The front-end improves security and privacy by isolation. It does not have access to the plain profile and if it is compromised at worst a malicious hacker will have access to back-end's interface which is already exposed by the front-end as web-services.

The front-end can be extended to use more than one back-end and thus boosting the performance and increasing the failure tolerance of the engine.

4.3 Back-end

The back-end exposes an interface to update the user's profile, suggest articles and log followed recommendations.

```
{
    'doc_id': 0x1234567890abcdef,
    'url': 'http://www.adobe.com/cfusion/'\
           'index.cfm?event=showdetails&postId=322'
    'title': 'Automatic user log out',
    'author': 'Krxtopher',
    'rating': 4.6,
    'similarity': 2.7,
}
```

Figure 4.4: An example of a suggested article returned by the back-end (presented as a Python dictionary)

Similar to many web services the back-end is a stateless service. The advantage is that back-end doesn't have to retain the user's profile between requests which helps improving performance and reliability. However, in order to track the user's behaviour the profile is store in the user's cookie by the front-end.

For privacy reasons the profile is encrypted before it's sent to the front-end Since encrypting and decrypting are slower than other operations involved the profiles are cached on the back-end service to avoid the decryption and therefore decreasing the response time.

The back-end exposes the following API:

- $visit_url(encrypted_profile, url) \rightarrow encrypted_updated_profile$.

  Called when the user visits a new page identified by it's web address *url*. If $encrypted_profile$ is missing the *url* is the first page visited by the user.

- $suggest(encrypted_profile, num_suggestions, ignore_urls) \rightarrow$ a list

  When this service is called, the back-end walks through all articles from Adobe Community database and matches them with the user profile. Then it returns a list of *num_suggestions* posts most relevant for the user. The list doesn't include any articles in *ignore_urls* like the original article. Each item in the list contains the *doc_id* (the suggestion id — see the front-end redirect service, section 4.2), the URL of the suggested article, the *title* to display, the *author*, the *rating* from the users and the *similarity* with the user's profile.

- $log_redirect(doc_id, source, dest) \rightarrow$ None

  Logs a followed recommendation identified by *doc_id*.

4.4 Preprocessing

In the preprocessing phase all Adobe Community articles are parsed. For each document the following data is collected to be used in the back-end:

- the URL (e.g. http://www.adobe.com/cfusion/communityengine/index.cfm?event=showdetails&postId=8924)
- the title (e.g. *Cookbooks and CSS Advisor update 10.3*)
- the author (e.g. *john.doe@adobe.com*)
- the Bayesian rating of the article (a real number between 1 and 5)
- the set of keywords and their relevance to the article (see section 3.1.2)
- the fingerprint of the article (see section 3.3.5)

The first four items are extracted from the database as they are. The rest are computed in the preprocessing phase. This phase works in three steps:

1. Retrieves all the documents from the data base, parse them and extract the keywords.
2. Compute *document frequency* for each keyword and trim redundant keywords (see chapter 3).
3. For each document compute the document vector and calculate the fingerprint.

The data collected is stored on the physical disk and loaded when the back-end starts. The preprocessing phase is run periodically to reflect the changes or the new articles then the back-end is restarted to load the data.

Chapter 5

Implementation Details

This chapter describes some implementation details. The engine was developed under Python 2.6 [14] with the fingerprinting module written in C for speed reasons.

5.1 Combining Two Fingerprints

The algorithm 3.25 can be easily adapted to calculate $fp(\vec{v'}) = fp(\vec{u} \circ_\Psi \vec{d})$. The pseudo-code is provided in 1:

> **for** $i = 1$ to N **do**
>   **if** $random() < \Psi$ **then**
> $fp_i(\vec{v'}) \leftarrow fp_i(\vec{u})$
>   **else**
> $fp_i(\vec{v'}) \leftarrow fp_i(\vec{d})$
>   **end if**
> **end for**

Algorithm 1: Updating the Fingerprint

$random()$ is a function which produces uniform random generated numbers in $(-1, +1)$. Many popular programming languages provide such a function.

5.2 Fast Computation of the Angle

The operation of counting the number of different bits between two fingerprints is equivalent to counting the number of bits set in the bitwise XOR of the two

fingerprints:

$fp(\vec{u})$	1	1	0	1	$\cdots$	1	0
$fp(\vec{d})$	1	0	1	1	$\cdots$	0	0
$fp(\vec{u}) \oplus fp(\vec{d})$	0	1	1	0	$\cdots$	1	0

Figure 5.1: Fingerprints Bitwise XOR

Most modern processors (including x86, x86_64, ARM, PPC) provide bitwise logical instructions like *XOR*, *AND*, *SHL*, *SHR*, *OR* which can operate on *machine-word-size* bits at a time. For 32bit Intel processors the best algorithm known for sideways addition is available from Sean Eron Anderson's website [3] and listed in 5.2. The number of bits in the bitwise XOR of two fingerprints can be calculated by dividing the fingerprints in blocks of 32 bits and summing the sideway-sums of the blocks. On a 2.4GHz processor computing the probability using fingerprints of 3072 bits length was timed at $0.9us$.

```
1 v = v - ((v >> 1) & 0x55555555);
2 v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
3 c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

Figure 5.2: Fast sideway-sum for 32bits integers

In listing 5.2 the first line computes the sideway-sum of groups of two bits. The second line computes the sideway-sum of groups of four bits. The third line computes the sideway-sum of groups of eight bits and adds the four bytes inside the integer v to compute the final sideway-sum. A modified version for 64bits processors is available from Knuth in [22].

From a theoretical point of view this algorithm exploits the parallelism present in the processors to achieve a time complexity of $O(\log(machine_word_size))$ to count $O(machine_word_size)$ bits.

5.3 Storing the Visited Articles

The section 3.4.2 presents how the user visited articles are stored.

The recommending engine uses Bloom filters of $m = 768bits$ ($96bytes$). If a user visits less than 64 articles the probability of a false positive is minimized for $k = 9$ hash functions.

Table 5.3 summarizes the probability of a false positive given the number of inserted URLs. For $n = 64$ visited articles the probability of a false positive is around 0.43%.

# of elements	Pr[false positive]
1	$6 \cdot 10^{-18}$
5	$1 \cdot 10^{-11}$
10	$4 \cdot 10^{-9}$
25	$6 \cdot 10^{-6}$
50	0.0009
64	0.0043
100	0.0457

Figure 5.3: Probability of a false positive based on the number of elements inserted for $m = 768$ and $k = 9$ (see equation 3.21).

5.4 Tracking the Followed Recommendations

Sometimes, in order to test the recall of the recommendation engine the followed suggested articles should be tracked.

There are a few requirements:

- No back-end side storage. The back-end should not maintain a list with all recommendations generated, although it can log (to a physical disk) data.
- Suggestions must expire.
- A malicious hacker should not be able generate false entries.

For each suggestion the engine generates an unique id using the algorithm in listing 5.4.

The URLs of the suggested articles are replaced with the new generated URLs which call the *redirect* web service. The *redirect* service logs the followed suggestion as in 5.5 and redirects to *dest*.

If *ESID* is modified then *SID* will not be a valid suggestion id. If *SID* or *dest* are incorrect then the computed *ID* will be invalid and will not match any *ID* logged in 5.4.

```
1 Generate a random ID of D bits
2 Log user's profile and ID
3 For each URL of the suggested articles:
4     SID = D xor digest(URL)  // digest() is a
5                              // cryptographic hash
6                              // function of D bits
7     ESID = cryptographically encrypt (SID, deadline)
8     generate the URL:
9         http://service/redirect?esid=ESID&dest=URL
```

Figure 5.4: Generating suggestion IDs

```
1 http://service/redirect?esid=ESID&dest=URL
2     deadline, SID = decrypt ESID
3     if time < deadline:
4         ID = SID xor digest(URL)
5         Log ID, URL
6         redirect to URL
```

Figure 5.5: Logging suggestion IDs

The advantage of this approach to generating an unique random *ID* for each suggestion is that less logging and tracking is necessary (one for each set of recommended articles versus one for each recommended article).

5.5 Handling Keywords

The section 3.1.2 described what are the keywords and how they are extracted from the text. This section continues and shows how the set of keywords is stored by the preprocessing phase. There are a few things that should be accounted for:

- Accessing a keyword of multiple words should be fast.
- Accessing a keyword which is a prefix of another keyword should be fast.
- Keywords have associated a number like *term frequency*
- Joining two sets of keywords should be relatively easy

- Low memory requirements

The data structure used is the *PATRICIA trie* introduced by Morrison in [25] where each node represents a keywords and each edge is labeled with a word. In figure 5.6 the keywords in the trie are *change*, *change_background*, *change_state*, *user*, *tree* and *tree_item*.

A keyword can be accessed by following the path from root to the corresponding node. The data associated with the keyword is stored in the its node. This data structure has low memory requirements because the paths are shared. The algorithm 2 joins two PATRICIA tries.

function *join(A, B)*
// if A is a trie, A[κ] is a subtree reached by
// following the edge labeled κ from root.
begin
    C $\leftarrow \lambda$
    foreach κ, *a labeled edge from A or B* **do**
        if κ, *edge from A and B* **then**
            C[κ] = join(A[κ], B[κ])
        else if κ, *edge from A* **then**
            C[κ] = A[κ]
        else if κ, *edge from B* **then**
            C[κ] = B[κ]
    end
    return *C*
end

Algorithm 2: A recursive function to join two PATRICIA tries

5.6 Used Technologies

This sections uses some technologies used by the recommendation engine.

5.6.1 Apache HTTP Server and Mod_python

Apacha HTTP Server is an open-source HTTP server for modern operating systems including UNIX, MS-Windows, Macintosh and Netware. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards. (http://www.apache.org)

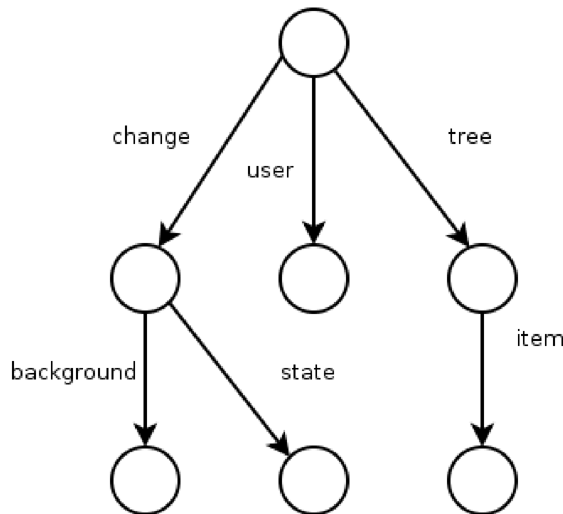

Figure 5.6: A PATRICIA trie

Mod_python is an Apache module that embeds the Python interpreter within the server. (http://www.modpython.org/)

The front-end runs as a web server over Apache HTTP Server using mod_python.

5.6.2 Pycrypto

Pycrypto — The Python Cryptography Toolkit — is a Python library that provides implementations for various cryptographic encryption and digestion algorithms. (http://www.dlitz.net/software/pycrypto/)

The front-end uses pycrypto to generate unique recommendation identifiers and the back-end uses it to encrypt profiles when needed.

5.6.3 RPyC

RPyC, or Remote Python Call, is a transparent and symmetrical Python library for remote procedure calls, clustering and distributed-computing. (http://rpyc.wikidot.com/).

In the server-client scenario the back-end is the server and the front-end, the client. The communication between them is done using the RPyC library.

5.6.4 Beautiful Soup

Beautiful Soup is an HTML/XML parser for Python that can turn even invalid markup into a parse tree. It provides simple, idiomatic ways of navigating, search-

ing, and modifying the parse tree. It commonly saves programmers hours or days of work. (http://www.crummy.com/software/BeautifulSoup/)

Beautiful Soup was used to parse the articles from the Adobe Community database that were written in HTML. This library allowed elimination of tags (eg. $< br >$, $< em >$, $< p >$), detection of code and elimination of comments.

5.6.5 Natural Language Toolkit

NLTK — Natural Language Toolkit — is a Open source Python modules, linguistic data and documentation for research and development in natural language processing, supporting dozens of NLP tasks, with distributions for Windows, Mac OSX and Linux. (http://www.nltk.org/)

NLTK was used for stemming (see section 3.1.2) and part-of-speech tagging (see subsection 3.1.2).

Chapter 6

Future Work

This work was aimed to introduce a new method to compute the similarity between users and documents.

Though, the appendix Results shows that the recommendation engine performs pretty well under the conducted tests, the precession can be improved by including other factors such as the *magnitude* of the document or user vector, the *user rating* of the article or the *hyperlinks* between documents (eg. see the PageRank algorithm [8]). These factors will be combined using the Expectation Maximisation algorithm [12] to compute the similarity.

One area that needs more development is the modeling of documents as vectors. Lexical chains [16], synonym networks [13], lexical patterns [2] or position of the words in the text [11] are only a few areas that need to be investigated.

Currently, the user starts with an empty profile which is filled after the first visited article. How to choose the initial profile to improve precision even nothing is yet known about the user is called the *cold-start* problem. Schein et al. describe some *Methods and Metrics for Cold-Start Recommendations* in [29].

When the recommendation engine goes live the user base will be much larger than the one available during the development, so the engine will be fine tuned to return better results. Moreover, with a larger user base the engine can suggest common articles visited by different users as in collaborative filtering systems.

The engine can be extended on other segments of the Adobe Community Engine like forums. Every time a user visits an article he or she is shown a relevant thread from the forum motivating him or her to contribute additional information on the same topic.

The preprocessing phase analyses all documents from the Adobe Community

Engine database every run. Currently, this is not an issue because the number of articles in the database is not big and the time required to do that is under 2 minutes. Nonetheless, given the projection that many more articles will be added the preprocessing phase should be modified to address only the changes and the new articles.

While the minimal functionality was implemented as the engine was successfully integrated with *Adobe Community Engine* developing a recommendation engine is an endless work of extending, fine tuning, text modeling and testing.

Chapter 7

Conclusions

This thesis introduces a very fast way to estimate the similarity between documents and users by the means of *fingerprints*. Fingerprints are a way to encode the direction of the document/user vector as an array of bits. They are used to compute the cosine similarity between two vectors which are representations of the documents and the users. The estimation error is shown to be negligible.

The profile of the user is obtained by combining the fingerprints of the visited documents such that more recent visited articles have a higher relevance when finding relevant articles. The resulted fingerprint is shown to contain only the features found in the visited documents and nothing more. The user profile is kept in the user cookie.

The fingerprints are used to build a personalised recommendation engine which finds relevant articles based on the user's browsing history. The recommendation engine is implemented as a stateless web-service that generates small HTML snippets which is shown together with the visited articles. A periodic phase is required which models the documents as a vector using traditional information retrieval techniques.

The appendix B — *Results* — shows that empirically the use of fingerprints gives similar results as the typical approach while it is around two orders of magnitude faster.

Bibliography

[1] 1ste Keuze BV. English stopwords. http://www.ranks.nl/resources/stopwords.html.

[2] James Allan and Hema Raghavan. Using part-of-speech patterns to reduce query ambiguity. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 307–314, New York, NY, USA, 2002. ACM. ISBN 1-58113-561-0. doi: http://doi.acm.org/10.1145/564376.564430.

[3] Sean Eron Anderson. Bit twiddling hacks. http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel, 2005.

[4] Marko Balabanović. An adaptive web page recommendation service. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 378–385, New York, NY, USA, 1997. ACM. ISBN 0-89791-877-0. doi: http://doi.acm.org/10.1145/267658.267744.

[5] Shumeet Baluja, Rohan Seth, D. Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 895–904, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. doi: http://doi.acm.org/10.1145/1367497.1367618.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[7] BNC Consortium. The British National Corpus, version 3 (BNC XML Edition). Distributed by Oxford University Computing Services on behalf of the BNC Consortium, 2007. http://www.natcorp.ox.ac.uk/.

[8] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

[9] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, London, UK, 2000. Springer-Verlag. ISBN 3-540-67633-3.

[10] Iván Cantador, Alejandro Bellog, and Pablo Castells. Ontology-based personalised and context-aware recommendations of news items. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 1:562–565, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/WIIAT.2008.204.

[11] Paul Alexandru Chiriță, Claudiu S. Firan, and Wolfgang Nejdl. Personalized query expansion for the web. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 7–14, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7. doi: http://doi.acm.org/10.1145/1277741.1277746.

[12] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977. doi: 10.2307/2984875. URL http://dx.doi.org/10.2307/2984875.

[13] Christiane Fellbaum, editor. *WordNet An Electronic Lexical Database*. The MIT Press, Cambridge, MA ; London, May 1998. ISBN 978-0-262-06197-1. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8106.

[14] Python Software Foundation. Python programming language – official website. http://www.python.org/, 1990-2009.

[15] Python Software Foundation. Thread state and the global interpreter lock. http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock, 2009.

[16] Stephen J. Green. Building hypertext links by computing semantic similarity. *IEEE Trans. on Knowl. and Data Eng.*, 11(5):713–730, 1999. ISSN 1041-4347. doi: http://dx.doi.org/10.1109/69.806932.

[17] Adobe System Inc. Adobe community engine. http://www.adobe.com/cfusion/communityengine/index.cfm?event=homepage&productId=2, 2009.

[18] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[19] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay. Accurately interpreting clickthrough data as implicit feedback. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 154–161, New York, NY, USA, 2005. ACM. ISBN 1-59593-034-5. doi: http://doi.acm.org/10.1145/1076034.1076063.

[20] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006. URL http://www.ietf.org/rfc/rfc4648.txt.

[21] Adam Kilgarriff. English word frequency list. http://www.kilgarriff.co.uk/bnc-readme.html, March 1996.

[22] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Pre-Fascicle 1a: Bitwise tricks and techniques*. The Art of Computer Programming. Addison-Wesley Professional, 1 edition, April 2008.

[23] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965 (Proposed Standard), October 2000. URL http://www.ietf.org/rfc/rfc2965.txt.

[24] William Little, Jessie Coulson, H. W. Fowler, C. T. Onions, and G. W. S. Friedrichsen. *The shorter Oxford English dictionary on historical principles; prepared by William Little, H. W. Fowler and Jessie Coulson; revised and edited by C. T. Onions*. Clarendon Press, Oxford,, 3rd. ed.; completely reset with etymologies revised by g. w. s. friedrichsen and with revised addenda. edition, 1973. ISBN 0198611161 0198611269 0198611277.

[25] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321479.321481.

[26] M. F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.

[27] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM. ISBN 0-89791-689-1. doi: http://doi.acm.org/10.1145/192844.192905.

[28] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Ithaca, NY, USA, 1987.

[29] Andrew I. Schein, Alexandrin Popescul, Lyle H. Ungar, and David M. Pennock. Methods and metrics for cold-start recommendations. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 253–260, New York, NY, USA, 2002. ACM. ISBN 1-58113-561-0. doi: http://doi.acm.org/10.1145/564376.564421.

[30] Upendra Shardanand and Pattie Maes. Social information filtering: algorithms for automating "word of mouth". In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 210–217, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1. doi: http://doi.acm.org/10.1145/223904.223931.

[31] Wikipedia. Local sensitive hashing. http://en.wikipedia.org/wiki/Locality_sensitive_hashing#Random_Projection.

Appendix A

Analysis of the Adobe Community Articles

This chapter contains a statistical analysis of the collection of Adobe Community articles. The database contains around 600 documents written by users in HTML. The preprocessing phase strips any HTML tag and leaves only the visible text before extracting the keywords.

For the histograms the number of bins were computed using the Sturges' formula:

$$\#\text{ of bins} = \lceil 1 + \log_2 \#\text{ of elements} \rceil$$

The first histogram (A.1) shows the distribution of the number of keywords in the vector representation of the document. The distribution has a bell shape and the average number of keywords in a document is 142.

The second histogram (A.2) shows a histogram of the distribution of keyword relevances as calculated by the recommendation engine. It is interesting that the curve approximates a line.

The third graph (A.3) plots the most relevant keyword (sorted by relevance) from all documents.

The last graph (A.4) can be extrapolated to infer that the addition of new documents will not increase very much the number of unique keywords in the collection.

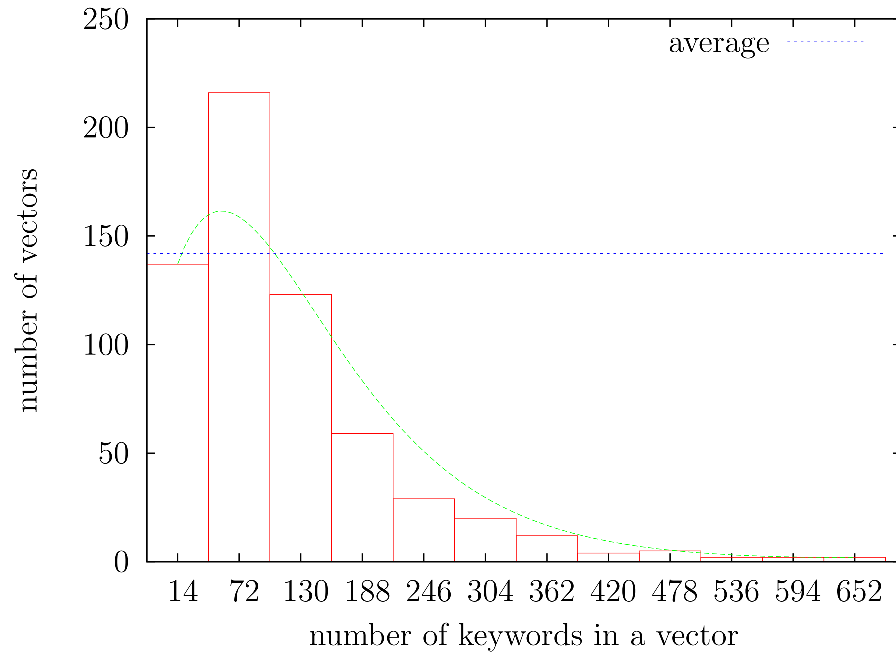

Figure A.1: Histogram of the numbers of keywords in a vector.

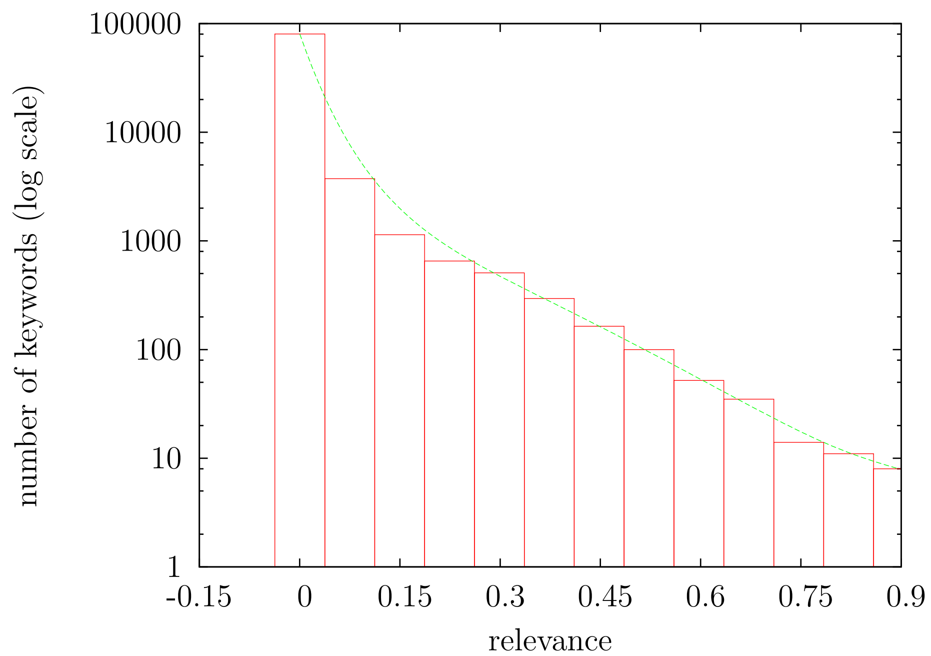

Figure A.2: Histogram of all keywords relevances

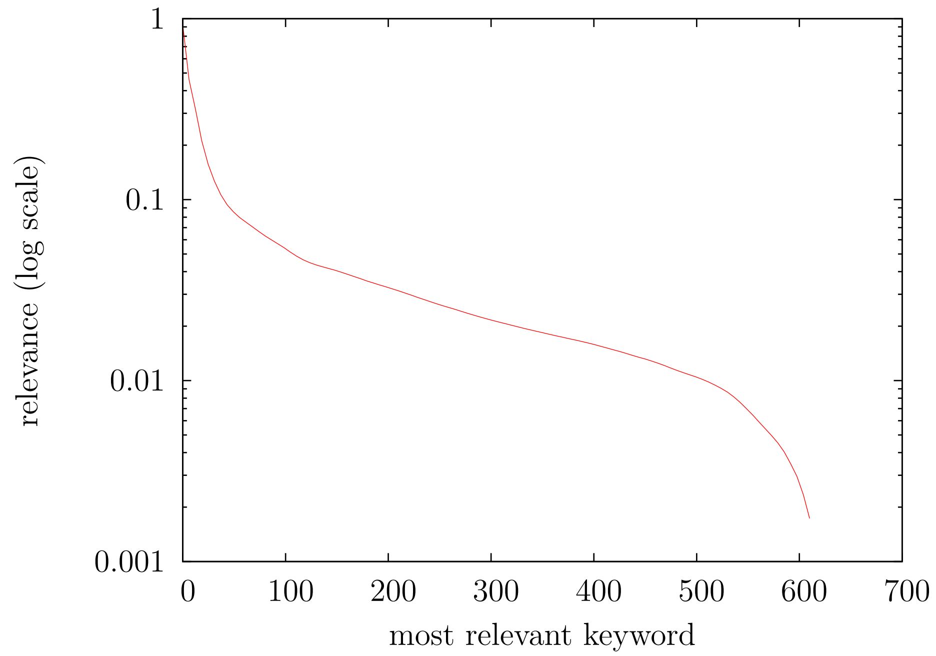

Figure A.3: Graph of the most relevant keyword from each document

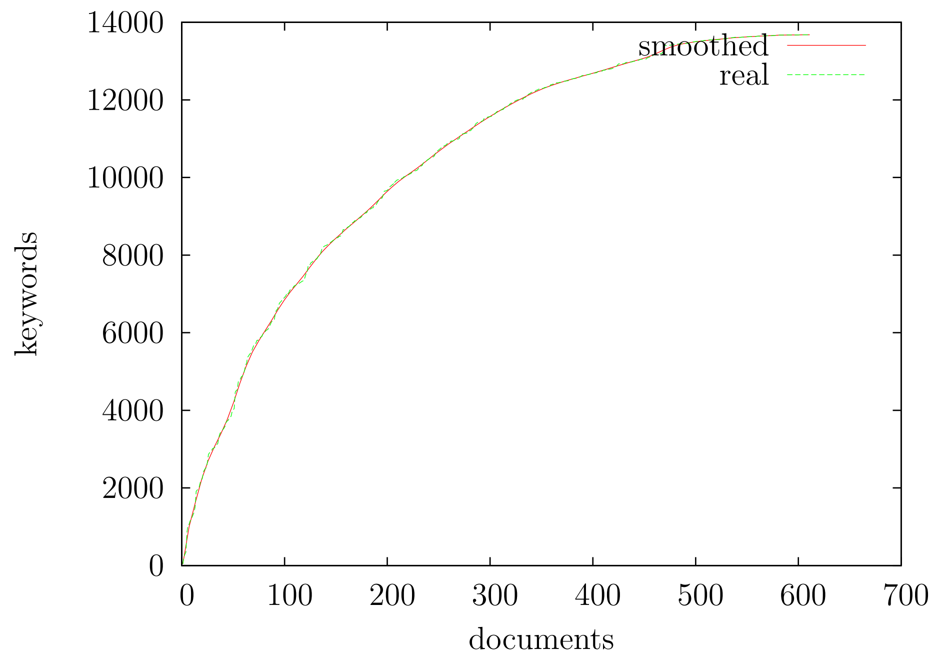

Figure A.4: Cumulative number of distinct words (documents shuffled).

Appendix B

Results

In order to improve the quality of the recommendation engine several tests were conducted which are presented in this chapter

The first section describes the measure used to help decide which features are good and which features aren't.

The second section performs an analysis of the ability of the recommendation engine to measure the similarity between documents. This is equivalent to discarding the user profile each time he or she visits a page.

The last section shows the strength of the engine to understand what the user is looking for.

B.1 Overview

Two statistical classifications are widely used to assess the quality of the recommendation engine: *precision* and *recall*.

- *Precision* is the probability that a retrieved document is relevant

$$\text{Precision} = \frac{|\{\text{relevant documents}\} \bigcap \{\text{documents retrieved}\}|}{|\{\text{documents retrieved}\}|}$$

- *Recall* is the probability that a relevant document is retrieved

$$\text{Recall} = \frac{|\{\text{relevant documents}\} \bigcap \{\text{documents retrieved}\}|}{|\{\text{relevant documents}\}|}$$

Because measuring the *recall* involves determining all relevant documents for user profile which is hard and requires much human resources only the *precision* was measured.

The improvement or regression of different features were tested using A/B tests. To find relevant documents the back-end used an algorithm randomly selected from a list of two or three. Some users, in this case other developers from Adobe, were asked to test the recommendation engine by following some predefine steps. The users didn't posses any internal knowledge on how the engine works.

It's important to note that comparing two different tests is not applicable because bug fixes and other improvements were made between different tests. Also I noticed that some users felt compelled to click one of the recommendations so one feature that is particularly bad had a higher precision when tested together with another poor feature than when tested together with a much better feature. The tested precision is just an approximation of the real precision which will be found when the engine goes live.

B.2 Document versus Document

To test whether the engine can model well the documents the users were asked to follow several steps:

1. User visits a random article and reads its content. His or her profile is cleaned.
2. User checks a list of 5 recommended articles.
3. If the user finds any relevant title he or she follows the link. The recommendation engine logs the click and takes him or her back to the *first* step.
4. Otherwise user starts from the first step.

Joachims et al. show in [19] the probability of a result to be followed depends on it's rank: the lower the rank the higher the probability. Therefore, the test results also include the number of clicks for each rank.

	random	engine
displayed	16	70
clicked	1	22
precision	0.063	0.314
1st article	0	5
2nd article	0	2
3rd article	0	13
4th article	1	1
5th article	0	1

Figure B.1: Engine is better than a monkey

	title ×1	title ×5	title ×8
displayed	70	24	23
clicked	22	8	8
precision	0.314	0.333	0.348
1st article	5	3	2
2nd article	2	5	5
3rd article	13	0	1
4th article	1	0	0
5th article	1	0	0

Figure B.2: Precision when the relevance of the title is adjusted.

	single word	multiple words
displayed	23	30
clicked	12	26
precision	0.522	0.867
1st article	5	15
2nd article	3	4
3rd article	1	2
4th article	3	1
5th article	0	4

Figure B.3: Keyword of a single word vs keyword of multiple words

	code $\times 1$	code $\times 0$	code $\times \frac{1}{3}$
displayed	30	24	27
clicked	19	13	18
precision	0.633	0.542	0.666
1st article	8	8	7
2nd article	3	2	3
3rd article	3	2	5
4th article	4	1	2
5th article	1	0	1

Figure B.4: Precision when the relevance of the code is adjusted

B.3 User versus Document

This section shows the competence of the engine to recommend articles relevant to the user.

The methodology from the previous section was changed:

1. User visits a random article and reads its content. His or her profile is cleaned.
2. User checks a list of 5 recommended articles.
3. If the user finds any relevant title he or she follows the link. The recommendation engine logs the click and takes him or her back to the *second* step.
4. Otherwise user starts from the first step.

The user continues visiting articles as long as he or she sees relevant articles then he or she starts over. The profile also contains already visited pages (see section 3.4.2) and he or she is always presented new articles.

The table B.5 provides a comparison of how the two update algorithms behave when user browses the site (see sections 3.4.3 and 3.4.4). The figure B.6 shows the evolution of the precision. Both update algorithms show an increase in the precision when the user visits the second page.

page number	fingerprint	vector
1	47	17
2	29	8
3	21	5
4	16	5
5	10	3
6	5	3
7	2	3
8	2	3
9	1	3
10	1	2
11	0	1
12	0	1

Figure B.5: Number of clicks for both fingerprint based and vector based update algorithm depending on the index of the page visited

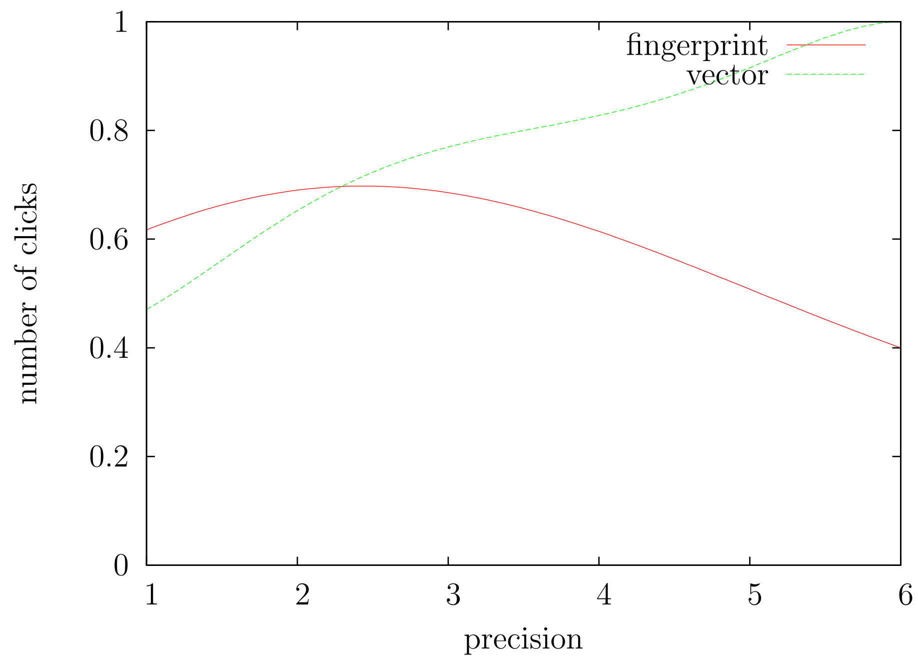

Figure B.6: Evolution of precision with the rank of the page visited

number of clients	qps	response time
1	43.8	23ms
4	20.2	50ms
16	23.7	42ms
64	30.0	33ms

Figure B.7: Average queries per second (qps) given the number of clients

B.4 Response Time

An important aspect of the recommendation engine is the response time. The back-end was designed to use multiple simultaneous threads, but Python has a Global Interpretor Lock [15] so the threads are not concurrent.

To test the engine under heavy load a client that simulates a user that randomly walks the site was written. The client skips the front-end and directly uses the interface from the back-end.

1. The client visits a random page using a clean profile.
2. The client gets a list of five relevant articles.
3. With the probability of $\frac{1}{6}$ the client restarts from step 1.
4. Otherwise, the client chooses a random article from the suggested list and visits it then it goes back to step 2.

The results are summarized in table B.7. At 20qps (queries per second) the recommendation engine performs well beyond the current usage of the Adobe Community Engine.

Appendix C

Code

This chapter provides some sample code from the engine's implementation.

C.1 Splitting Paragraphs into Sentences and Sentences into Words

```
import re
import itertools


_WORD = re.compile('\w+', re.UNICODE)
# Python regexp doesn't support (yet) UNICODE capital letters
_TEXT_SENTENCE_DELIM = re.compile('([.!?]\W+(?=[A-Z]|\n)|$)', re.UNICODE)
_CODE_SENTENCE_DELIM = re.compile('\n|\r|$', re.UNICODE)


def _helper(text, delim):
    """Splits a text into sentences using regexp"""
    sentences = []

    start = 0
    for match in delim.finditer(text):
        end = match.end()
        sentences.append(text[start:end])
        start = end

    return sentences
```

```
def code_to_sentences(text):
    """Splits code into sentences (more like statements)"""
    return _helper(text, _CODE_SENTENCE_DELIM)


def text_to_sentences(text):
    """Splits text into sentences"""
    return _helper(text, _TEXT_SENTENCE_DELIM)


def sentence_to_words(sentence):
    """Splits a sentence into words ignoring punctuation"""
    return _WORD.findall(sentence)
```

C.2 Extracting Keywords

```
"""Various functions to work with the keywords.

The tree representation of keywords has the following recursive definition:
    node = {
        ''          : frequency/relevance,
        word1       : child_node1,
        word2       : child_node2,
        ...
    }

For example:
    {
        ''          : 8,
        'romania'   : {
                ''      : 5,
                'iasi'  : { '': 1 },
                'bacau' : { '': 1 },
                'deva'  : { '': 1 },
            },
        'bulgaria' : { '': 1 },
        'moldova'  : { '': 2 },
    }

The terms in the example are: 'romania'(5), 'romania iasi'(1),
'romania bacau'(1), 'romania deva'(1), 'bulgaria'(1), 'moldova'(2).
```

```
For a document 'keyword' counts represent frequencies.
For a document 'tfidf' counts represent relevance. (see for example:
    http://en.wikipedia.org/wiki/Tf-idf)
For a collection 'df' counts represent document frequency (number
    of documents containing the keyword)
For a collection 'idf' counts represent inverse document frequency.

For simple keywords tfidf formula is similar to the one described on:
    http://en.wikipedia.org/wiki/Tf-idf.
For compound keywords tfidf formula depends on probabilities of the
terms and their parents.

"""

import nltk
from math import log

import splitter
import bjhash


_STEMMER = nltk.stem.PorterStemmer()

# the list of words to ignore
# * http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words
# * some other very common not included above (eg. 'll, 've)
# XXX read from somewhere else
with open('stop_words') as f:
    _STOP_WORDS = f.read().split()

# how long can a compound keyword be
_NGRAM = 4


def tree():
    """Returns an empty tree"""
    return {'': 0}


def extract_keywords(sentence):
    """Extracts keywords from sentence.
```

```
    @return: a tree containing compound keywords from sentence

    """
    words = splitter.sentence_to_words(sentence)
    # lowercase and remove 1 character length words
    words = [w.lower() for w in words if len(w) > 1]
    # removes stopwords
    words = [w for w in words if w not in _STOP_WORDS]
    # stems
    words = [_STEMMER.stem(w) for w in words]
    # removes numbers
    words = [w for w in words if not w.isdigit()]

    # deletes duplicate words
    temp, last = [], ''
    for word in words:
        if word != last:
            temp.append(word)
            last = word
    words = temp

    # generates n-gram keywords
    root = tree()
    for start in xrange(len(words)):
        node = root
        node[''] += 1

        for index in xrange(start, min(start + _NGRAM, len(words))):
            node = node.setdefault(words[index], tree())
            node[''] += 1

    return root


def update_keywords(keywords, new, coef=1):
    """Updates 'keywords' tree with keywords from 'new' tree."""
    keywords[''] += new[''] * coef
    for keyword, child_new in new.iteritems():
        if keyword:
            child_keywords = keywords.setdefault(keyword, tree())
            update_keywords(child_keywords, child_new, coef)
    return keywords
```

```
def update_df(df, keywords):
    """Updates a ‘df’ tree with keywords from ‘keywords’ tree.

    This function is similar to ‘update_keywords’ but counts
    from ‘keywords’ are ignored.

    @param keywords: a tree containing keywords from one document
    @param df: a tree representing document frequency
    @return: the updated ‘df’

    """
    df[’’] += 1
    for keyword, child_keywords in keywords.iteritems():
        if keyword:
            child_df = df.setdefault(keyword, tree())
            update_df(child_df, child_keywords)
    return df


def convert_to_idf(df):
    """Given a ‘df’ tree converts it inplace to an ‘idf’ tree."""
    # if df has at most one son, removes it
    if len(df) <= 2:
        temp = df[’’]
        df.clear()
        df[’’] = temp
        return

    # Deletes all keywords present in all documents where parent is
    # present (ie. idf = 0). For example if ‘foo’ and ‘foo bar’ are
    # both present in 100 documents, then there is no need for ‘foo bar’.
    # Also, deletes all keywords found only in one document.
    for keyword, child in df.items():
        if keyword:
            if child[’’] == df[’’] or child[’’] == 1:
                del df[keyword]

    # Recurses and computes final df values for its sons
    for keyword, child in df.iteritems():
        if keyword:
```

```
            convert_to_idf(child)
            child[''] = log(1. * df[''] / child[''])
            #child[''] = log(1. * df[''])


def compute_tfidf(keywords, idf, _tfidf=None, _sum=None):
    """Computes tfidf for each keyword in document.

    Words not in idf are ignored because they may be too rare
    and not relevant for the document relations.

    """
    if _tfidf is None:
        idf[''] = 0.
        _sum = keywords['']
        _tfidf = tree()

    for keyword, child_keywords in keywords.iteritems():
        if keyword:
            child_idf = idf.get(keyword)
            if child_idf is None:
                continue

            relevance = 1. * child_keywords[''] / _sum * (
                    log(1. * _sum / keywords['']) + child_idf[''])
            #relevance = 1. * child_keywords[''] / _sum * child_idf['']
            child_tfidf = _tfidf.setdefault(keyword, { '': relevance })
            compute_tfidf(child_keywords, child_idf, child_tfidf, _sum)

    return _tfidf



def flatten_tfidf(tfidf, _flat=None, _word=''):
    """Given a `tfidf' tree returns a flat representation.

    For each node (except root) path is concatened and a pair
    (path, relevance) is generated

    For example, given the following tree (copied from module's docstring):
    {
        ''          : 8,
        'romania'   : {
```

```
            ''      : 5,
            'iasi'  : { '': 1 },
            'bacau' : { '': 1 },
            'deva'  : { '': 1 },
        },
    'bulgaria' : { '': 1 },
    'moldova'  : { '': 2 },
}

the result will be the list (possible in a different order):
[
    ('romania',       5),
    ('romania iasi',  1),
    ('romania bacau', 1),
    ('romania deva',  1),
    ('bulgaria',      1),
    ('moldova',       2),
]

"""
if _flat is None:
    _flat = []

for keyword, child in tfidf.iteritems():
    if keyword:
        if _word:
            word = _word + ' ' + keyword
        else:
            word = keyword

        _flat.append((word, child['']))
        flatten_tfidf(child, _flat, word)

return _flat


def compute_fingerprint(tfidf):
    """Computes fingerprint of the document.

    @param tfidf: a flat representation of keywords in a document
    @return computed fingerprint of keywords vector
```

```
    """
    return bjhash.fingerprint(tfidf)
```

C.3 The Backend

```
def exposed_suggest(self, profile_sign, encrypted_profile,
                    num_suggestions, ignore_urls):
    """Suggests ‘num_suggestions‘ documents based on user’s profile.

    @param profile_sign user’s profile signature
    @param encrypted_profile the encrypted user’s profile
    @param num_suggestions number of suggestions to return
    @param ignore_urls a tuple of URLs to ignore
    @return tuple({
            ’doc_id’    : doc_id,      # a suggestion id
            ’url’       : url,         # sugested URL
            ’title’     : title,       # sugested title
            ’similarity’: similarity,  # a coefficient. The higher the better.
            ’author’    : author,      # author of the document
            ’rating’    : rating,      # whatever rating was
                                       # retrieved from cookboks
        })

    """
    profile = self.__get_profile(profile_sign, encrypted_profile)
    ignore_urls = frozenset(ignore_urls)

    # computes similarity between user’s profile and all documents
    docs = []
    suggestion_id = random.getrandbits(crypto.DIGEST_BITS_SIZE)
    self.log.suggestion_id = suggestion_id
    self.log.profile_sign = profile_sign

    profile_fingerprint = profile.fingerprint()
    for doc in self.docs.itervalues():
        url, fingerprint = doc[’url’], doc[’fingerprint’]

        if url not in ignore_urls:
            similarity = bjhash.cosine(fingerprint,
                    profile_fingerprint)
            similarity *= 1 - 0.9 * profile.has_visited(url)
            docs.append((url, similarity))
```

```
    # leaves only the best ‘num’ documents
    docs = heapq.nlargest(num_suggestions, docs, key=lambda d: d[1])

    # computes suggestion ids
    for index in xrange(len(docs)):
        url, similarity = docs[index]

        doc_id = suggestion_id ^ crypto.digest_int(url)
        doc_id = ('%%0%dX' % (2 * crypto.DIGEST_SIZE)) % doc_id
        doc_id = crypto.encrypt_block(self.key, doc_id)

        docs[index] = _dict_doc_to_tuple(self.docs[url],
                                         doc_id=doc_id,
                                         similarity=similarity)

    # makes docs immutable (and thus dumpable)
    docs = tuple(docs)
    self.log.docs = docs
    return docs

def exposed_log_redirect(self, doc_id, source, dest):
    """Logs a redirect from source and dest."""
    try:
        doc_id = crypto.decrypt_block(self.key, doc_id)
        doc_id = int(doc_id, 16)
    except (crypto.DecryptError, ValueError):
        # ignores invalid doc_ids (somebody is messing with our ids)
        _logger.error('Cannot decode doc_id. Malicious url?')
        return

    suggestion_id = doc_id ^ crypto.digest_int(dest)
    self.log.suggestion_id = suggestion_id
    self.log.source = source
    self.log.dest = dest

def exposed_random_urls(self, num_urls):
    """Returns num_urls random URLs from the loaded set"""
    num_urls = min(num_urls, len(self.docs))
    urls = random.sample(self.docs, num_urls)
    return tuple(_dict_doc_to_tuple(self.docs[url]) for url in urls)
```

C.4 The Front-end

```
@_handle_client_errors
def redirect(req, doc_id, dest):
    """Implements a redirection to destination"""
    try:
        client = _connect(req)
        client.root.log_redirect(base64.urlsafe_b64decode(doc_id), _referer(req), dest)
    except Exception:
        # ignores any server error
        pass

    # redirects to destination
    util.redirect(req, urllib.unquote(dest))


@_handle_client_errors
def random_page(req):
    """Redirects to a random page"""
    client = _connect(req)
    urls = client.root.random_urls(1)

    cookie = Cookie.Cookie('u', '', discard=True, expires=0)
    Cookie.add_cookie(req, cookie)

    cookie = Cookie.Cookie('s', '', discard=True, expires=0)
    Cookie.add_cookie(req, cookie)

    if urls:
        util.redirect(req, dict(urls[0])['url'])
    raise apache.SERVER_RETURN, apache.HTTP_INTERNAL_SERVER_ERROR


@_handle_client_errors
def recommend(req, url=None):
    """Generates HTML code with the suggestions"""
    req.content_type = 'text/html; charset=utf-8'

    if url is None:
        url = _referer(req)
    sign, profile = _load_profile(req)

    client = _connect(req)
```

```
sign, profile = client.root.visit_url(sign, profile, url)
docs = client.root.suggest(sign, profile, NUM_RECOMMENDATIONS, (url,))

# transforms url from
# from http://host/.../recommend
#   to http://host/.../redirect
redirect_url = urllib.url2pathname(req.parsed_uri[6])
redirect_url = os.path.join(os.path.dirname(redirect_url), 'redirect')
redirect_url = urllib.pathname2url(redirect_url)

# generates the html code
result = SUGGESTION_HEADER
for doc in docs:
    doc = dict(doc)
    doc['doc_id'] = base64.urlsafe_b64encode(doc['doc_id'])
    doc['url'] = urllib.quote(doc['url'])
    doc['similarity'] = '%.2f' % (doc['similarity'] * 4 + 1)
    result += SUGGESTION_ELEMENT.format(redirect_url=redirect_url, **doc)
result += SUGGESTION_FOOTER

_save_profile(req, sign, profile)
return result
```