TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Offline Approximate String Matching for Information Retrieval: An experiment on technical documentation

Simon Dubois

MASTER THESIS 2013
INFORMATICS

Postadress: Box 1026 551 11 Jönköping
Besöksadress: Gjuterigatan 5
Telefon: 036-10 10 00 (vx)

TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

EXAMENSARBETETS TITEL

Offline Approximate String Matching for Information Retrieval:An experiment on technical documentation

Simon Dubois

Detta examensarbete är utfört vid Tekniska Högskolan i Jönköping inom ämnesområdet informatik. Arbetet är ett led i masterutbildningen med inriktning informationsteknik och management. Författarna svarar själva för framförda åsikter, slutsatser och resultat.

Handledare: He Tan
Examinator: Vladimir Tarasov

Omfattning: 30 hp (D-nivå)

Datum: 2013

Arkiveringsnummer:

Postadress: Box 1026 551 11 Jönköping
Besöksadress: Gjuterigatan 5
Telefon: 036-10 10 00 (vx)

Abstract

Approximate string matching consists in identifying strings as similar even if there is a number of mismatch between them. This technique is one of the solutions to reduce the exact matching strictness in data comparison. In many cases it is useful to identify stream variation (e.g. audio) or word declension (e.g. prefix, suffix, plural).

Approximate string matching can be used to score terms in Information Retrieval (IR) systems. The benefit is to return results even if query terms does not exactly match indexed terms. However, as approximate string matching algorithms only consider characters (nor context neither meaning), there is no guarantee that additional matches are relevant matches.

This paper presents the effects of some approximate string matching algorithms on search results in IR systems. An experimental research design has been conducting to evaluate such effects from two perspectives. First, result relevance is analysed with precision and recall. Second, performance is measured thanks to the execution time required to compute matches.

Six approximate string matching algorithms are studied. Levenshtein and Damerau-Levenshtein computes edit distance between two terms. Soundex and Metaphone index terms based on their pronunciation. Jaccard similarity calculates the overlap coefficient between two strings.

Tests are performed through IR scenarios regarding to different context, information need and search query designed to query on a technical documentation related to software development (man pages from Ubuntu). A purposive sample is selected to assess document relevance to IR scenarios and compute IR metrics (precision, recall, F-Measure).

Experiments reveal that all tested approximate matching methods increase recall on average, but, except Metaphone, they also decrease precision. Soundex and Jaccard Similarity are not advised because they fail on too many IR scenarios. Highest recall is obtained by edit distance algorithms that are also the most time consuming. Because Levenshtein-Damerau has no significant improvement compared to Levenshtein but costs much more time, the last one is recommended for use with a specialised documentation.

Finally some other related recommendations are given to practitioners to implement IR systems on technical documentation.

Acknowledgements

My first thanks are for He Tan who supervised this study. Her patience and teaching skills allow me to explore the field of Information Retrieval. I really appreciated her patience and tolerance towards my English level.

Thanks to Vladimir Tarasov who welcomed me at the University of Jönköping two years ago. I very appreciated his lectures and short talks about Linux.

A special thanks to Anders Wadell for the warm welcome and the help to join the University and to fulfil administrative requirements.

Finally, a warm thank to my partner Delphine who came with me to Sweden, and encouraged me during the Master program and thesis.

Key words

Algorithm comparison

Approximate string matching

Information retrieval

Offline string matching

Overlap coefficient

Phonetic indexation

String distance

String metric

String searching algorithm

Contents

List of figures

List of abbreviations

Ave.: Average

IR: Information Retrieval

Max.: Maximum

Min.: Minimum

OCR: Optical Character Recognition

sec.: Seconds

1 Introduction

1.1 Background

Approximate string matching involves identifying strings similar to a particular string with a limited number of inaccuracy [1]. The goal is to identify strings that might not be exactly equals, but that are enough similar to refer to the same word family, that is, "a base word and all its derived and inflected forms" [2]. For example, "beater", "downbeat", "unbeaten", "beatable" and "beating" belongs to the same family of the base word "beat".

Approximate string matching is used in a large set of different applications, wherever there is a need for more flexibility for data processing [3]. Flexibility in the sense that approximate matching expands the rigidity of exact matching with additional results (Figure 1.1). Traditionally, it is used on user input to correct a text (spell checker [4] and OCR [5]) or to increase relevance of request results (query expansion). In back-end programs, approximate string matching is used, for example, in data integration to increase the number of matches between data set from different sources [4]. The use of the technique has been extended to other domain such as biology (for DNA comparison [6]) and music (for track recognition [7]).

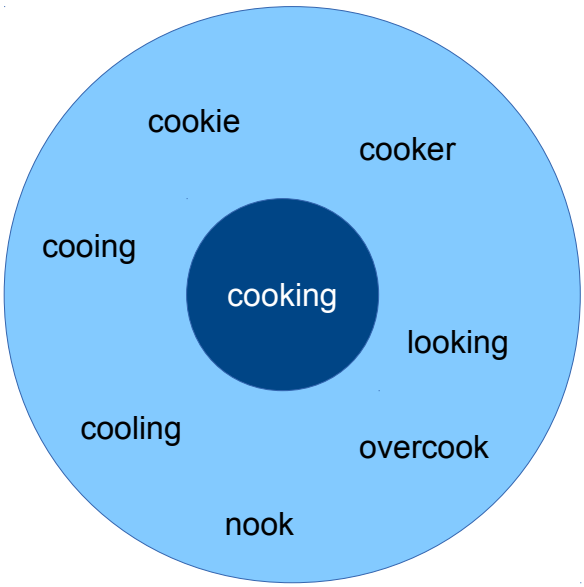

Figure 1.1: Approximate matches expanding exact matches
Darker blue: exact matches; **Lighter blue:** approximate matches

Information Retrieval involves finding all and only relevant information in a collection [8]. Approximate string matching increases the number of matches compared to exact string matching. It is a key technique for IR systems to extend a request with similar terms. However, approximate string matching does not systematically imply more relevant results as neither the meaning nor the context are considered, only characters. So two strings identified as similar might have no other characteristics in common than their orthographic signature (e.g. "chafe" and "chaff").

As approximate string matching addresses problems encountered years before computer science's birth (e.g. spelling errors [9]), multitude of algorithms have been developed and improved since the fifties. Some are compilations of existing algorithms (e.g. agrep [10]), others have been developed in different flavours (e. g. the Levenshtein distance [1]).

This research aims at identifying the strengths and the weaknesses of some of the main approximate string matching algorithms in IR situations. The algorithms are implemented and tested against software documentation in different information needs. Some criteria are defined to measure how each algorithm behaves. Then, the results provide conclusions to guide IR system development in the choice of a relevant string matching algorithm for searches on technical documentation.

1.2 Purpose/Objectives

The aim of this study is to investigate the effects of approximate string matching in IR situations. The final outcome giving explicit recommendations in the choice of an adapted algorithm for main IR scenarios. This study assesses two algorithms attributes:

- relevance of results to a query (are the matches related to the query terms).
- performance (how fast is the approach in different tasks).

The research questions investigated in this paper are :

- which algorithms provide more relevant matches?
- which algorithms run faster?

1.3 Limitations

This study does not claim to explore every string matching technique and their variations but only 6 of them. They are chosen because they are standard techniques commonly used to match strings. Moreover, they process strings with different methods like simulation of pronunciation or comparison of characters, so comparing their results is instructive (see section 3.1.3).

The experiment is conducted on an inverted representing the documentation of open source software (also called manpages). This documentation is widely used by practitioners like users comfortable with terminal, developers and system administrators. Results of this scientific research can be generalised to any documentation focused on a specialised (technical) domain (see section 2.3).

The benchmark is conducted on a regular desktop computer in order to obtain real world results in the situation where IR scenarios are often performed locally. Such choice is justified by the use of approximate string matching in system development and production environment [11][12]. When users search for information stored locally (e.g. only installed software), use of a dedicated server is helpless.

1.4 Thesis outline

In chapter 2, the **theoretical background** provides a summary of knowledge generated last fifty years concerning approximate string matching. This chapter develops the concept of approximate string matching and presents the main algorithms. In the chapter we also introduce the notion of information retrieval in order to understand the issues of approximate matching in the domain.

In chapter 3, the **method** followed in the research is presented. This section explains research method choices (examined IR components, chosen algorithms, IR scenarios) and details the implementation of the experiment (inverted index construction, benchmark implementation and run).

In chapter 4, the **results** of the experiments are presented to the reader. The attributes measured during the tests are assessed for each IR scenario and algorithm.

In chapter 5, the **conclusion** answers the research questions and provides suggestions for future researches in the domain.

2 Theoretical Background

This section explains the theory behind the research domains. First, approximate string matching is introduced through different approaches. Then, Information Retrieval is presented with its related concepts. Finally, the document collection used in the research is introduced.

2.1 Approximate string matching concept

This section gives knowledge regarding approximate string matching algorithms and their background (e.g. discovery context, principle). Only algorithms considered during this research are presented (listed in Table 2.1).

Algorithm family	Algorithm	Time complexity	Space complexity
Exact matching	Character comparison	O(min(n, m))	O(1)
Edit distance	Levenshtein distance	O(nm)	O(mn)
	Damerau-Levenshtein distance	O(nm)	
Phonetic indexation	Soundex	O(n+m)	O(1)
	Metaphone		O(n+m)
Overlap coefficient	Jaccard similarity	O(n+m)	O(1)

Table 2.1: String matching algorithms and their complexity
n: query length; **m:** indexed term length

2.1.1 Edit distance family

Originally, approximate string matching was a way to deal with errors when large-scale computers were standard. At the first single error, the machines were totally stuck until the problem was identified and fixed [13]. A metric called edit distance has been introduced in this context to find (expected) correct output from the returned value. Then, it was possible to identify sources of system failures thanks to position of mismatches between expected (correct) value and returned value. The edit distance is defined as "the minimal cost to transform one string into the other via a sequence of edit operations (usually insertions, deletions and replacements), which each have their own weight or cost" [14].

One of the first computed algorithms addressing this problem is the Hamming distance. This distance is the number of positions at which the corresponding symbols mismatch. In other words, it is the minimal number of character substitutions to transform one string to another. For example, the Hamming distance between "eaten" and "laden" is 2 (mismatches at first and third positions). This method is only valid for strings of equal length. [13]

The Levenshtein distance extends the Hamming distance principle with two more operations. Indeed, the Levenshtein distance considers character insertion and deletion in addition to character substitution. Therefore, this metric applies to strings of different lengths. [15]

Damerau and Levenshtein add character transposition to the set of operations considered by Levenshtein's edit distance. With these four operations (insertion, deletion, replacement, transposition), the edit distance is particularly efficient in natural language processing [16].

2.1.2 Phonetic indexation family

This technique encodes a string into a representation of its pronunciation. The goal is not to encode the exact sound but an approximation of the pronunciation. So two nearly homophone strings have equal indexes. Many variations exist depending on the targeted language or accent. Two methods, Soundex and Metaphone are still standards nowadays and natively implemented in many computing languages [17][18][19].

Soundex was the first attempt to deal with large name records for census. It encodes a word in a four digit code. This code is made with the name's first letter, and the index of the three first remaining consonants as defined in Table 2.2. If the code is incomplete (less than 4 digits), it is completed with '0'. For example, “Department” is encoded as “D-163”: 'D' as first letter, '1' for 'p', '6' for 'r', '3' for 't'. [20]

Index	Represents the letter
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Table 2.2: Soundex consonant classification [20]

Soundex has largely been improved by Metaphone which refines interpretation of sounds. Indeed, this method considers successive consonants, and vowels adjacent to consonants [21]. While Soundex deals with 6 sounds, Metaphone can handle 21 different sounds [22]. For example, “Department” is encoded as “TPRTMNT”: vowels are ignored, 'd' is replaced by 'T', and other consonants are retained. Metaphone has been declined in different versions, including the Double Metaphone for more flexible comparisons.

2.1.3 Overlap coefficient family

The overlap coefficient is not an algorithm, but an approach that is directly inspired from the IR field. The goal is to calculate a similarity coefficient of the compared strings based on their shared digits. Such statistics are simpler than identifying pertinent edit operations, but they do not consider strings' character layout (e.g. order and position). [23]

The Jaccard index computes, for two strings, the ratio of their union size (e.g. number of characters occurring in both) and their intersection size (e.g. sum of number of characters occurring in each) (Formula 2.1). For example,

$$J("eaten","laden")=\frac{|\{'e','a','t','n'\}\cap\{'l','a','d','e','n'\}|}{|\{'e','a','t','n'\}\cup\{'l','a','d','e','n'\}|}=\frac{3}{6}=0.5 \quad .$$

$$Jaccard(A,B)=\frac{|A\cap B|}{|A\cup B|}$$

Formula 2.1: Jaccard index [4]

2.2 Information Retrieval issues

2.2.1 Information Retrieval concepts

IR is a field concerned with ”finding material of an unstructured nature that satisfies an information need from within large collections”. It is one of the Information Logistics discipline focusing on assessing relevance of results to match information need and available content. The information need is defined as ”the topic about which the user desires to know more”. To request information, users express search query, often composed with key words. It is important to notice that queries do not always correspond to the real information needs. It is the case when query terms differ from collection terms. For example, consulting a numeric catalogue, a user looking for some chairs for the living room is most likely to query “chair” on the search engine. If the catalogue uses “seat” instead of “chair” to name and describe its products, the system will not match user query and indexed vocabulary. Thereby, no relevant results will be returned and the user's information need will not be satisfied. [23]

The key concept of IR is relevance: “the ultimate goal of IR is to retrieve all and only the relevant documents for a user's information need” [8]. To reach relevance, IR systems have to deals with numerous issues [23], including

- to process large collections of documents (e.g. distributed databases, the growing Web).
- to rank results in relevance order so more relevant results are presented first.
- to increase flexibility in matching (e.g. approximate term matching).

A fundamental aspect of IR system is the inverted index [23]. An inverted index is a dictionary of terms and, for each term, a posting list of documents in which terms occur. Because fetching the complete document collection's content at each research query is too long (time complexity), it is necessary to pre-process documents and extract terms. But an incidence matrix with one dimension for terms, one for documents and the number of occurrence at intersections would cost too much memory (space complexity) and mainly contain '0' (absence of term in document). It has been found that building an inverted index solves both time and space complexity problems.

How to evaluate relevance in IR systems is an old question. The difficulty is to define criteria as relevance is sorely influenced by subjectivity as "different users may differ about the relevance or non-relevance of particular documents to given questions " [24]. However, such fact does not prevent to define standards metrics [24]:

- precision: the fraction of retrieved documents that are relevant (Formula 2.2).
- recall: the fraction of relevant documents that have been retrieved (Formula 2.3).
- f-measure: the harmonic mean of precision and recall (Formula 2.4).

$$P = \frac{retrieved\ documents \cap relevant\ documents}{retrieved\ documents}$$

Formula 2.2: Precision [23]

$$R = \frac{retrieved\ documents \cap relevant\ documents}{relevant\ documents}$$

Formula 2.3: Recall [23]

$$F = \frac{2PR}{P+R} \quad \text{with } P \neq 0 \text{ or } R \neq 0$$

Formula 2.4: F-Measure [23]

2.2.2 Approximate string matching in IR

This research focuses on the flexibility provided by the approximate matching of information need formulation (e.g. a user input query) against a data set (e.g. database, files, indexes). This dimension becomes more and more considered with the growing web and the distribution of data repositories through distinct systems. "The challenge is to integrate them into coherent digital libraries that let users have unimpeded access" [25].

There are two categories of string matching contexts. In-line matching occurs when the text can be processed before the search to build an index of the terms, so algorithms have to consider each terms. At the opposite, offline matching happens when the text can not be processed before the search, so algorithms have to deal with the full text as one block [14]. As inverted index is part of IR systems, it is not useful to address in-line string matching in the context of this research.

Studies about approximate matching for music and voice [7][26][27] and biology [6][28][29] are popular. But few researches have addressed the problem of approximate text matching from a general IR perspective. There are interesting comparisons of approximate string matching algorithms, but they only consider the case of identifying similar records in different databases (also called record linkage) [30][31].

This study aims at increasing knowledge in the field by evaluating approximate string matching in IR approach. Algorithms are run against a documentation collection and results are evaluated as answers to an information need.

2.3 Document collection examined

In this research, we carefully choose the open source software documentation as test data. In the domain, man pages is a standard since the birth of UNIX system [32] and is still used by developers and practitioners as documentation of system commands and functions. Designing IR situations and assessing relevance of results require high knowledge from the researcher [24]. Man pages are both easily accessible and well known by the researcher. The choice has been made to consider the 92 764 man pages provided by the latest long-term support version of Ubuntu, a popular Linux distribution.

Man pages are formatted text files describing use and behaviour of commands and functions. They are spread into sections based on topic attributes like usage (regular program or administration tool) or implementation (shell command or C function). (Figure 2.1).

Usually, man pages respect the following layout (Appendix 7.1.1):
- header: function/command name and section identifier.
- NAME: function/command name and a short description.
- SYNOPSIS: syntax for command and parameters for functions.
- DESCRIPTION: text explaining the purpose of the function/command and the role of each parameter individually.
- RETURN VALUE: value returned by the function.
- AUTHOR: authors' names.
- BUGS: urls and/or links to submit bugs.
- COPYRIGHT: licenses concerning the program use, modification and distribution.
- EXAMPLES: common usage of the function/command.
- SEE ALSO: related commands and functions with their section identifier.

The particularity of such a documentation from an IR perspective is both vocabulary syntax and vocabulary variety. Indeed, some characters usually used as word separators in natural language (e.g. '-' or '_') are parts of the documentation vocabulary (e.g. 'mk_modmap' or 'sane-usb'). So the indexation method has to consider this particularity to not split words on such characters.

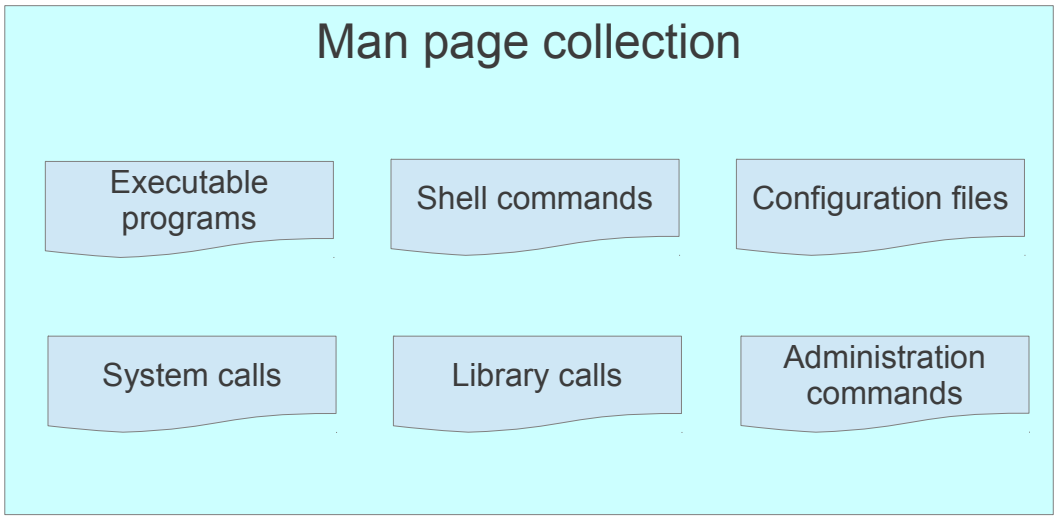

Figure 2.1: Man page sections [33]

The lexicon diversity is also a particularity. Due to the technical nature of the documentation topic, one can suppose a homogeneous vocabulary and a low variability of natural language. Indeed, as documentation aims at reducing ambiguity, so the use of synonyms is usually avoid. For examples, each function has a unique name. At the opposite, in natural language texts (e.g. newspapers, blogs), there is a large use of synonyms and words variation (e.g. “ad”, “advert”, “advertisement” all can be used to reference advertisement). In such context, measuring benefits and costs of approximate string matching compared to exact string matching is pertinent.

3 Research method

3.1 Design

During the research, the holistic perspective (Figure 3.1) has been followed.

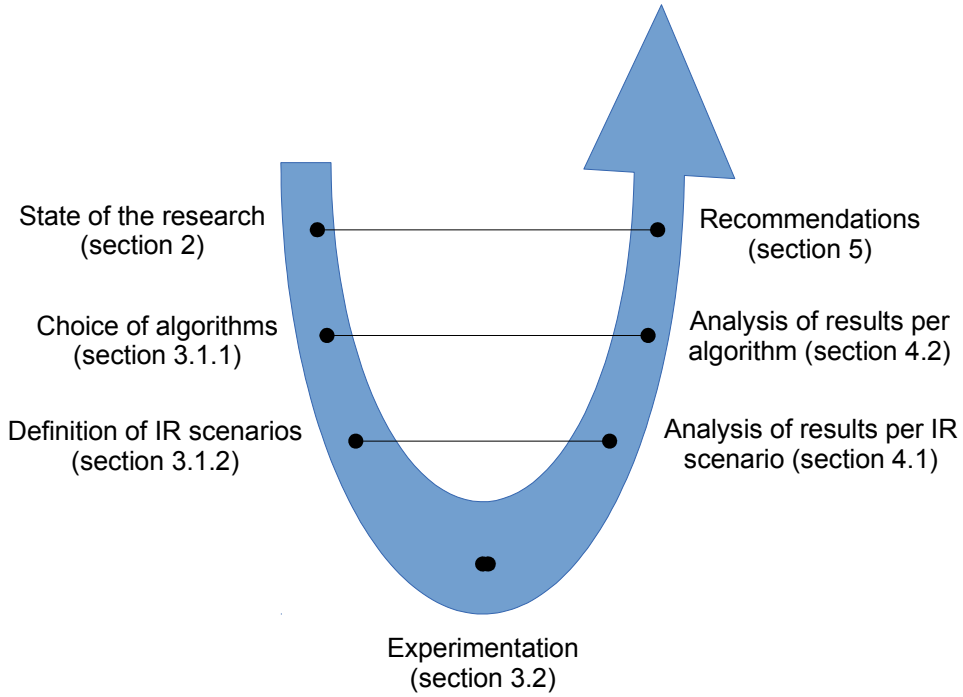

Figure 3.1: Holistic perspective of the research

The research method chosen for this study is the experimental research design. The use of this method is justified "to explore possible causal links between variables". Indeed, the purpose of this research is to consider the effects of different approximate string matching algorithms on reply to an information need. So system development is not suitable because this method analyses impacts of IT in organisations which is not the scope of this research. Survey research is not relevant because here the goal is to study computed data, not human opinion or observation in real situations. [34]

Section 3.1.1 gives the hypotheses the experiment tries to verify. Section 3.1.2 identifies the variables of the experiment and draws the IR system designed to conduct the tests. Section 3.1.3 explains the choices of string matching algorithms while 3.1.4 justifies the choices of IR scenarios. Section 3.1.5 addresses criteria used to compare and evaluate the algorithms.

3.1.1 Experiment hypotheses

Experimental research design usually starts with one or several hypotheses verified (or not) by a study [34]. Here are the hypotheses tested during this research:

- H0: Approximate string matching increases result recall but decreases precision compared to exact string matching.
- H1: Computing approximate string matching takes more time compared to exact matching.

H0 addresses effects of approximate string matching from the IR perspective while H1 focuses on a more technical aspect that is the execution time.

3.1.2 Experiment variables

Considering IR system architecture, different resources and processes are used to match search results (Figure 3.2). First, there is a document collection, composed of text files, music tracks, video clips, web pages, etc. The collection is indexed to make faster searches on terms (see section 2.2.1). When a user submits a query, a term scoring system compares indexed terms to query terms. In this study, approximate string matching algorithm is used to grade each term according to its similarity with query terms. Then documents are scored based on their terms score and frequency. The returned results are documents sorted by score.

IR system components can be viewed as variables influencing search results. It is possible to change only one component (or variable), and compare search results obtained with different implementation of such component. Then it is possible to conclude on an objective efficiency of the different implementations.

The independent variable is defined as "the factor manipulated by the researcher to see what impact it has on another variable(s)" [34]. String matching algorithm is considered as the independent variable because it is subject to variation during the experiment (e.g. different algorithms are tested). Then it is possible to appreciate the effects of each string matching algorithm change on the returned results and thus assess the relevance to the information need.

However, testing only one case of information need would not be enlightening. Indeed, it would not be possible to generalize experiment results to other search queries. Therefore, it would not be possible to provide serious recommendations for IR system development in the choice of an approximate string matching algorithm. That is why one more independent variable is required: algorithms have to be tested in different information need (e.g. IR scenarios).

Other variables (yellow boxes in Figure 3.2) are not addressed, which means that are intended to not vary during the experiment. Indeed, if document collection, term indexation technique and document scoring method are same for all tests, it is possible to make a comparison of effects produced by different search queries or string matching algorithms on search results.

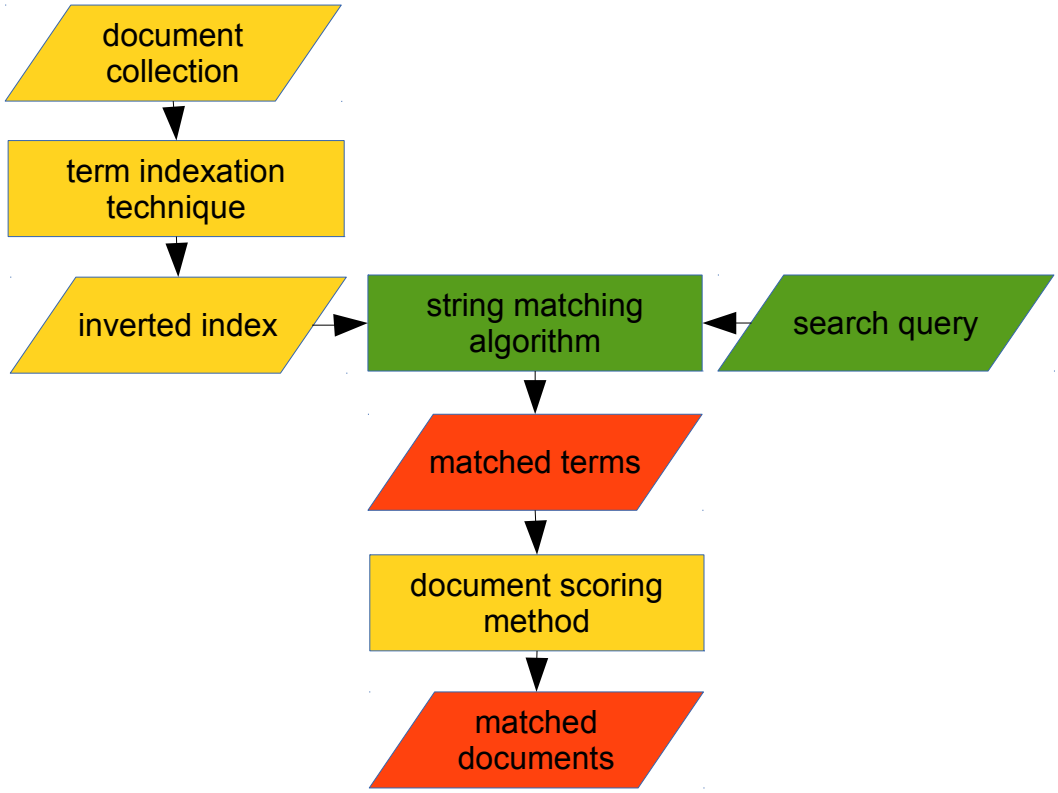

Figure 3.2: IR system components as experiment variables

Rectangle: process ; **Parallelogram:** data ;
Green: independent var. ; **Red:** dependent var. ; **Yellow:** frozen var.

3.1.3 Selected string matching algorithms

Section 2.1 briefly presents the functioning of the main approximate string matching algorithms. Hamming distance is not part of the experiment as it requires strings to have the same length, which is too restrictive for most IR systems.

Comparing Levenshtein and Dameraud-Levenshtein distance algorithms is pertinent to measure the cost (execution time) and benefit (additional relevant matches) of computing character transposition on terms from specialized documentation (at the opposite of personal/individual documents where typing error is more likely to occur). Comparing Soundex and Metaphone methods is suitable. Indeed, even if both compute a code representing words pronunciation, Metaphone algorithm includes much more tests than Soundex. Are these tests benefit on such data? The experiment conducted in this study answers such question.

For an objective evaluation of relevance and performance, a basic exact string matching algorithm is also included. The idea is, on one hand to measure the average execution time of each algorithm. On the other hand, relevance of matches and non-matches is analysed in order to recognized the usefulness of algorithms.

3.1.4 Defined IR scenarios

IR scenarios are simulation of real world situations where a user has an information need and expresses it to an IR system which return answer (e.g. in the form of a document set). The goal is to define scenarios that are the most likely to happen and that would play with approximate matching. The researcher personality also influences definition of scenarios as returned documents will be appreciated based on his experience.

IR scenarios are divided into two groups (Table 3.1 and Table 3.2) based on search approach. In browsing tasks, user does not precisely know what he/she is looking for, and the query is usually intended to match a large set of documents. At the opposite, when user is fully aware about his/her information need, when he/she can express it explicitly, the action is called "searching". Such distinction are necessary as user expectations are different (large document set versus precise document).

N°	Context	Information need	Search query
1	User is trying to open a file	Parameters and returned values of the function open	open file
2	User is monitoring information about computer	Function returning hostname	get hostname
3	User is monitoring time required to write data to file	Functions returning current timestamp	get timestamp
4	User is manipulating file descriptor	Parameters and returned values of the function fcntl	fntcl (typing error)
5	User has heard about a helpful function "youmask"	Information about the function named "youmask"	youmask

Table 3.1: IR scenarios: searching tasks

N°	Context	Information need	Search query
6	User is developing a POSIX C application	All POSIX functions	posix
7	User is getting the error EOVERFLOW	All functions returning the error EOVERFLOW	eoverflow
8	User developing a distributed application	All functions dealing with sockets	socket
9	User is porting a Linux application to Windows platform	All functions using the exclusive Linux library unistd	unistd.h
10	User is going to collaborate with or succeed to a developer and would like to know functions he/she has created	All functions implemented by Andreas Gruenbacher	Andreas Gruenbacher

Table 3.2: IR scenarios: browsing tasks

IR scenarios focus on system calls in C in order to fit with information in sample. Each scenario is defined by:
- a context: case in which the information need could emerge.
- an information need: what documentation is expected.
- a search query: terms employed by the user to describe his/her information need.

In any IR scenario, different queries can be used. For example, for IR scenario 6, other possible queries are "standard posix", "conforming to posix", "conforms posix". For this study, search queries used during experiment are the first that naturally come to the user mind in the related context, based on the researcher experiment in the domain. Testing natural queries is necessary to provide useful conclusion for real world situations. For example, testing "conforms posix" is naturally avoided as user does not expect all documents about posix C functions to contain the world "conforms".

Assessing document relevance to a query has to be done for each IR scenario. As relevance is a subjective notion, the researcher has to immerse himself into each scenario to well understand the information need. Doing so, it becomes possible, for each document, to ask the question "in this situation, is the document useful?". Degree of usefulness in searching tasks differs to browsing tasks. Indeed, in searching task, assessing relevance is more obvious as the information need is precise. However, in browsing tasks, information need is wide and can place great importance on interpretation. In case of doubt, for documents where proves of relevance were slight, marking document as relevant has been favoured.

3.1.5 Evaluation criteria

Two criteria are evaluated to measure advantages and drawbacks in the use of approximate string matching.

First, to evaluate the benefits brought by approximate string matching, the F-Measure is computed and assesses relevance of results (Section 2.2.1). Considering documents as relevant or not (based on researcher's experience in the domain) is a long process. It is necessary to consider individually each document and decide whether or not it is relevant for IR scenarios. Then, it is possible to compute precision and recall.

A purposive sample is used in this work. Two reasons justify the use of such sample [34]. First is because the document collection is very large (92 764 files). Second is because specific documents are required for some IR scenarios: their absence would make the experiment pointless. Indeed, when no relevant document exists, precision and recall used to evaluate algorithms and verify H0 are equal to 0 (see Formula 2.2 and 2.3). So all algorithms have the same results and it is not possible to identify which one gives the best results. So it is necessary to ensure that every IR scenario has relevant documents in the sample. Benefit of such sample is to include these interesting elements and reduce the size of the studied population. The sample is made up with 689 man pages based on two criteria: (1) system calls, (2) C and C++ functions.

Second, to evaluate the cost of using approximate string matching, the duration required to execute algorithms is monitored. Time necessary to initialise term list, sort matched terms, score and sort documents is excluded as these operations are not linked to algorithms and are sorely implementation dependent. Time monitoring is performed for each IR scenarios on the complete document collection.

3.2 Implementation

The experiment has been conducted on a regular desktop computer, similar to the ones in offices. The software environment used during the process is strictly open-sourced:
- Archlinux version 32bits March 2013 (operating system).
- MySQL version 5.5 (database management system).
- PHP version 5.5 (scripting language).
- GCC version 4.8 (C compiler).
- Valgrind version 3.8 (memory debugging tool).

Here is a description of the hardware environment:
- CPU: Intel Core i5 – 460M - 2.53 GHz – 3 MB L3 cache
- RAM: 2.5 Go - DDR3
- HDD: 5400 rpm - SATA

3.2.1 Building the inverted index

A simple database (Figure 3.3) is designed to store the inverted index:
- `documents` stores document url, a unique identifier and a boolean identifying the document as part of the sample.
- `terms` stores term value, a unique identifier, and the inverse document frequency to score documents (see section 3.2.2).
- `term_distribution` stores triplet term (as identifier), document (as identifier) and term frequency in document.
- `term_scoring` temporary stores terms and their scores for document scoring (see section 3.2.2).

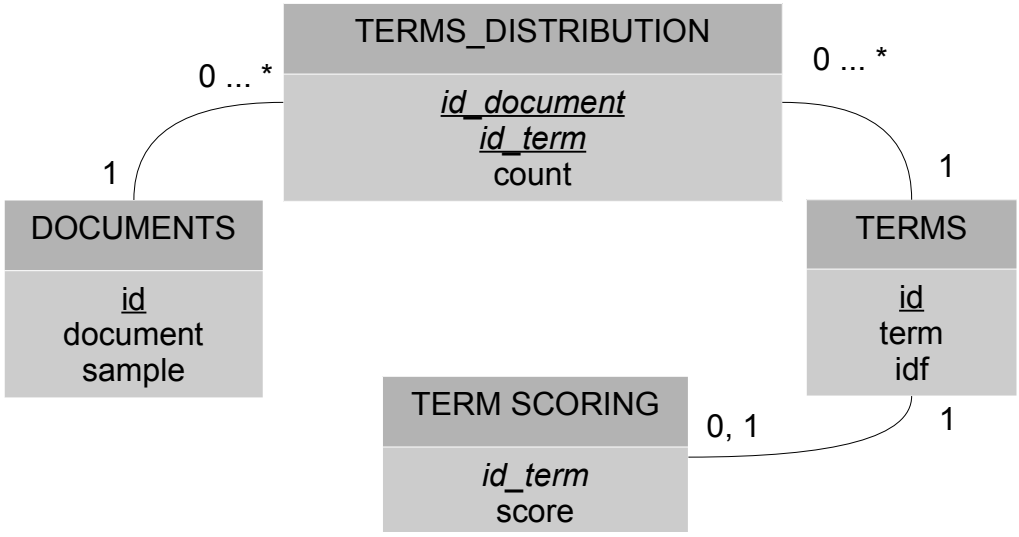

Figure 3.3: Inverted index in database (Appendix 7.2.1)
Underline: primary key; **Italic:** foreign key

The complete documentation is reachable at [35] in HTML format. To index the collection, a PHP script (Appendix 7.2.2) loops through each document and:
- downloads the HTML file and extracts the text in lower case from the HTML element with id "content".
- looks for the string "#include" (a C/C++ directive) in section SYNOPSIS and set the document as part of the sample if found.
- splits the full text in terms with a regular expression.
- inverts the term list and counts the number of term occurrences in the document.
- stores the document url in the database and gets its identifier.
- for already stored terms, retrieve their identifiers.
- for non already stored terms, inserts them and gets their identifiers.
- for each term, inserts the triplet(id_document, id_term, nbr_occurrences) in the database.

Terms are defined as any combination of letters, numbers, and the following symbols if preceded or followed by letters and/or numbers: '-', '_', '+', '.', '@' and '/'. Finally, the inverse document frequency is computed and stored for each term (Formula 3.1).

$$idf_t = \log_{10}\left(\frac{collection\,size}{nbr\,of\,documents\,containing\,t}\right)$$

Formula 3.1: Inverse document frequency

For the complete documentation, the resulting inverted index contains:
- 92 764 documents.
- 783 878 terms.
- 19 514 684 pairs (id_document, id_term, nbr_occurrences).
- a mean of 210 unique term per document.
- a mean of 695 occurrence per document.

For the purposive sample, the resulting inverted index contains:
- 689 documents.
- 9 409 terms.
- 165 365 pairs (id_document, id_term, nbr_occurrences).
- a mean of 240 unique term per document.
- a mean of 649 occurrence per document.

3.2.2 Implementing the benchmark

The benchmark is a program designed to measure algorithm efficiency. The goal is to provide information about what is being done and how long it takes. For comparative results, each algorithm has to be executed in similar conditions: same parameters, same data, without memory leak. Moreover, the application has to deal with large data in memory to avoid costly requests to database during monitored run time.

To deal with such constraints, the C language has been chosen:
- data types and low-level memory management (more control on memory usage).
- low-level language (no resources spent on hidden routines).
- time measurement in microsecond.
- compatibility with memory debugging tool (Valgrind certify the absence of any memory leak)
- MySQL API.

The program has been designed to be modular (Figure 3.4). A core component is responsible for initialization, terms and documents scoring. Term scoring is done with respect to the algorithm given as parameter. Algorithms return a value, either a similarity coefficient (a decimal value between 0 and 1) or a boolean for match/mismatch (Table 3.3). A threshold is a limit value below which terms are considered as not approximatively similar and thus not taken into account to score documents. Edit distance algorithms and Jaccard similarity return a decimal value, so it is possible to use threshold to filter results. There is no one explicitly optimal threshold value for the approximate string matching algorithms. The threshold has to be set to reach the optimal balance precision/recall for the concerned vocabulary. The optimal balance between precision and recall is achieved with the highest F-Measure value for representative test cases. For this study, threshold value tests have been conducted against indexed terms from the documentation sample based on IR scenarios. A threshold of 0.5 for edit distance algorithms and 0.75 for Jaccard similarity have been chosen.

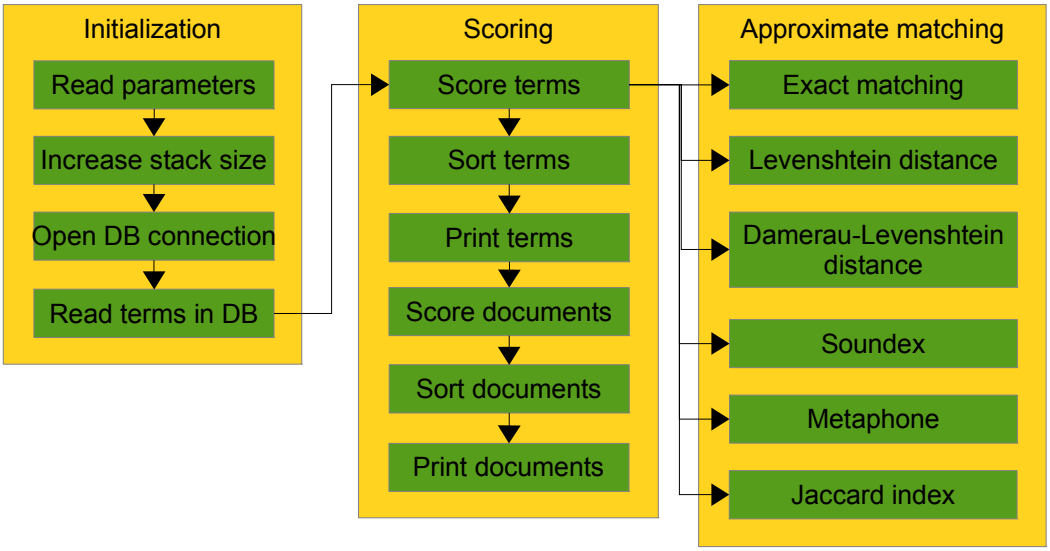

Figure 3.4: Benchmark flow-chart
Yellow: function collection ; **Green:** function

Algorithm family	Algorithm	Returned value	Implementation
Exact matching	Character comparison	0 for character mismatch 1 for character matches	Appendix 7.3.1
Edit distance	Levenshtein distance	Ratio of edit operations (Formula 3.2) 0 if the edit distance is greater than half its upper bound Between 0.5 and 1 otherwise	Appendix 7.3.2
	Damerau-Levenshtein distance		Appendix 7.3.3
Phonetic indexation	Soundex	0 for key mismatch 1 for key match	Appendix 7.3.4
	Metaphone		Appendix 7.3.5
Overlap coefficient	Jaccard similarity	0 for less than half overlap Between 0.75 (¾ of characters overlap) and 1 (complete overlap)	Appendix 7.3.6

Table 3.3: Researched string matching algorithms and their returned value

$$edit\_distance\_ratio(s_1, s_2) = 1 - \frac{edit\_distance(s_1, s_2)}{MAX(length(s_1), length(s_2))}$$

Formula 3.2: Edit distance normalisation to similarity coefficient
s_1 and s_2: terms

Documents are scored with the tf-idf weighting calculation (Formula 3.3) in order to consider term informativeness as well as similarity coefficient. To speed up such calculation, an intermediate table "term_scoring" (Figure 3.3) is filled with pairs (id_term, term_score) and used to compute the final document score.

$$score(d, q) = \sum_{t \in q} (tf_{t,d} \times idf_t \times sc)$$

Formula 3.3: Document scoring (adapted from [23])

d: a document ; **q:** matched terms occuring in d;
$\mathbf{tf_{t,d}}$**:** number of occurences of t in d ; $\mathbf{idf_t}$**:** Formula 3.1 ; **sc:** similarity coefficient

3.2.3 Running the benchmark

The benchmark has been built with various options (see Appendix 7.3.7) in order to achieve the different targets of the research: use of different algorithms, and use of different search queries.

Tests have been conducted as follow : for each algorithm, each query runs once. So with 5 algorithms (Table 2.1) and 10 tests cases (Table 3.1 and Table 3.2), 50 tests have been conducted. For test, each document has been marked as relevant or not to the information need. Then recall, precision and F-Measure and have been computed.

To evaluate time cost for each approximate matching method, the benchmark has been run with the same queries, but matches have been performed against the complete documentation in order to provide real world statistics. For these tests, only durations spent on string matching algorithms have been considered.

4 Results

This section presents and analyses results obtained during the experiments. Figures are obtained based on tables in Appendix 7.4

4.1 Result relevance

From an IR perspective, relevance is measured with two metrics precision and recall, and their harmonic mean called F-Measure. While high precision means that few non relevant documents have been retrieved, high recall signifies that most of the relevant documents have been retrieved.

4.1.1 Result relevance on searching tasks

Searching tasks are situations where a user is looking for few precise documents. Figure 4.1 gives an overview of results relevance on searching tasks. For any algorithm, precision is always quasi-null (below 0.1) while recall is either medium (around 0.5 for exact matching, Soundex and Metaphone) or very high (above 0.8 for edit distance and Jaccard similarity). Such results is due to the nature of searching tasks. As there is only few relevant documents, a few non relevant retrieved documents are enough to decrease the precision, while high recall is easily reachable if the few relevant documents are retrieved.

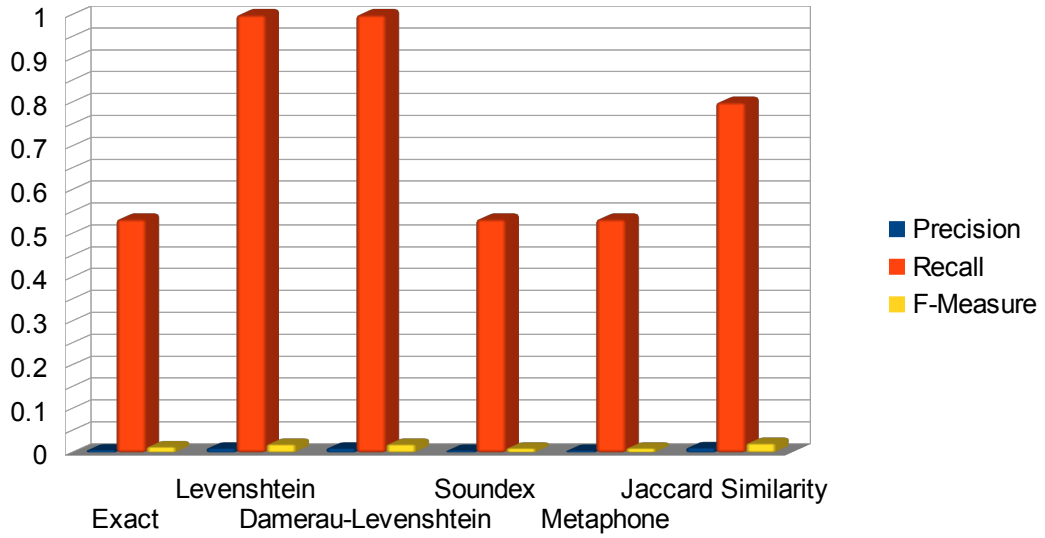

Figure 4.1 : Statistics about result relevance during searching tasks (data from Table 4.1, columns "Ave.")

Algorithm	Precision			Recall			F-Measure		
	Min.	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**
Exact	0	0.014	0.006	0	1	0.533	0	0.028	0.013
Levenshtein	0.002	0.017	0.010	1	1	1	0.004	0.033	0.019
Damerau-Levenshtein	0.002	0.017	0.010	1	1	1	0.004	0.033	0.019
Soundex	0	0.011	0.005	0	1	0.533	0	0.022	0.010
Metaphone	0	0.011	0.005	0	1	0.533	0	0.022	0.010
Jaccard Similarity	0	0.034	0.011	0	1	0.800	0	0.066	0.021
All algorithms (average)	0.001	0.017	0.008	0.333	1	0.733	0.001	0.034	0.015

Table 4.1: Statistics about result relevance during searching tasks

In most IR scenarios, precision is very low regardless of the algorithm used. Such result is caused by vocabulary homogeneity. For example, query for IR scenarios 1 is "open file". Sample has 248 documents containing the term"open", 406 documents containing the term "file" and 233 containing both terms "open" and "file". However, only 6 documents are considered as relevant. Other files contains these terms (or their approximations) but are not any help to open documents. For example, files containing texts like "[the function] performs one of the operations described below on the open file" [36] or "A file descriptor argument was out of range, referred to no open file" [37] are useless for a developer looking for a function to open a file. Moreover, man pages usually contain a section "SEE ALSO" [33] suggesting names of related functions. So if a function name A is given as query, the IR system also matches any document B suggesting A in "SEE ALSO" section, even if B is not relevant for the information need. That is why precision is relatively low independently of the string matching algorithm.

In IR scenarios 1 and 2 (see Appendix 7.4.1 and 7.4.2) , all methods get ridiculously low precision and perfect recall. Indeed, query terms ("open file" and "get hostname") correspond to documentation vocabulary. In IR scenarios 3 to 5, only edit distance algorithms and Jaccard similarity return all relevant documents. In IR scenario 3 (see Appendix 7.4.3), these algorithms succeed because they can match terms that belongs to the same word family "time": "timestamp" (from query), "timezone", "time.h", "time_t" (from relevant documents). In IR scenarios 4 (see Appendix 7.4.4), the query has transposition of characters as typing error ("fntcl" for "fnctl") that only these algorithms can identify. In IR scenarios 5 (see Appendix 7.4.5) query is an approximation of a term pronunciation ("youmask" for "umask"), that Soundex and Metaphone failed to approximate because first letters 'y' and 'u' does not match. Here again edit distance algorithms and Jaccard similarity succeed because only 2 characters differ ('y' and 'o').

Another advantage of edit distance algorithms and Jaccard Similarity is the sort of returned information (Table 3.3). Their return value (a decimal number) does not simply inform whether or not there is a match between terms, but also how similar they are. So if terms are similar around 75%, these algorithms consider the match as valid and return 0.75. Such information is not given by Soundex and Metaphone that return a binary value with 1 for approximate phonetic match and -1 for no approximate phonetic match. There is no information to indicate how two phonetic approximations are similar. For example, in IR scenario 5, Metaphone indexes "youmask" as YMSK and "umask" as UMSK, both terms are just not considered as similar thus there is no match.

4.1.2 Result relevance on browsing tasks

Browsing tasks are situations where a user has no precise idea of his/her information need and expects a large set of results before querying more precisely. Figure 4.2 gives an overview of results relevance on browsing tasks that is very different to the one obtained with searching tasks. Indeed, on browsing tasks, precision is either very high (above 0.9 for exact matching and Metaphone) or medium (around 0.6 the other) while recall is always high (above 0.8).

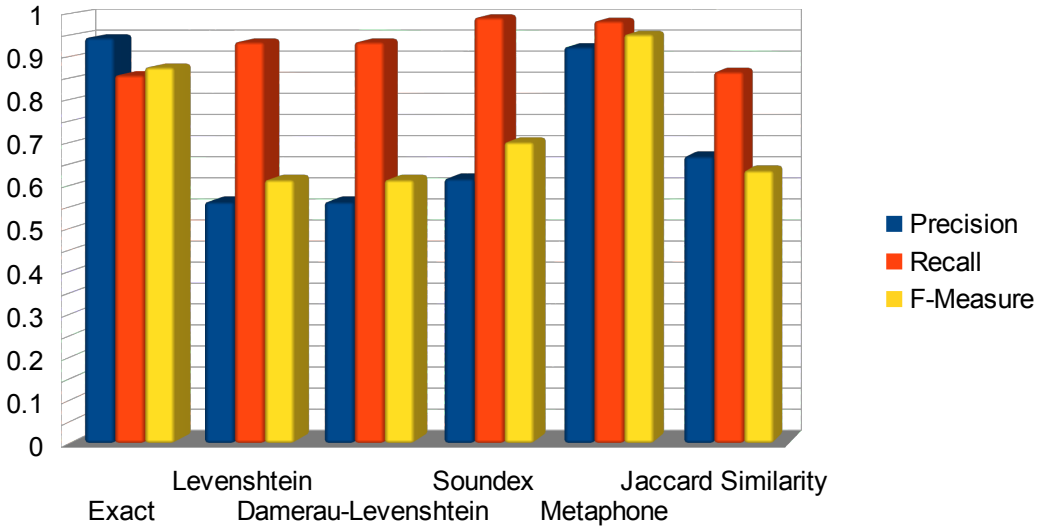

Figure 4.2: Statistics about result relevance during browsing tasks (data from Table 4.2, columns "Ave.")

Algorithm	Precision			Recall			F-Measure		
	Min.	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**
Exact	0.851	1	0.944	0.371	1	0.857	0.531	1	0.875
Levenshtein	0.026	0.931	0.562	0.668	1	0.934	0.051	0.921	0.613
Damerau-Levenshtein	0.026	0.931	0.562	0.668	1	0.934	0.051	0.921	0.613
Soundex	0.082	0.938	0.616	0.977	1	0.991	0.152	0.968	0.701
Metaphone	0.851	1	0.923	0.938	1	0.983	0.903	1	0.952
Jaccard Similarity	0.024	1	0.668	0.371	1	0.865	0.047	0.988	0.635
All algorithms (average)	**0.310**	**0.967**	**0.713**	**0.666**	**1**	**0.927**	**0.289**	**0.966**	**0.732**

Table 4.2: Statistics about result relevance during browsing tasks

In IR scenarios 6 to 10, query terms correspond to the documentation vocabulary (no mismatch no approximation), so recall is generally high. In IR scenario 6 (see Appendix 7.4.6) the query term ("posix") is the name of an international standard for UNIX software. As regular standards, POSIX has different version and is often used with version number (e.g. "POSIX.1-2001"). On such case, edit distance algorithms and Jaccard similarity have low or average recall because version number is longer than "posix" alone, so edit distance or similarity coefficient are below threshold. At the opposite, Soundex and Metaphone succeed because they ignore non letter characters.

In IR scenario 7 (see Appendix 7.4.7), the user is looking for all functions returning the error "eoverflow", which is a code used when a value is beyond its expected data type. Some other functions also handle this error, but without using the specific code "eoverflow". As edit distance algorithms and Jaccard similarity accept few differences in characters, they match variations like "ioverflow" or "overflow" and return non-relevant documentation. Soundex and Metaphone does not follow this error as removing the first letter ('e') change the phonetic index that does not match with query term.

In IR scenario 8 (see Appendix 7.4.8), query term ("socket") is a well-spread file-type used in network connections. The low precision obtained by edit distance is caused by other popular (but non relevant) terms occurring in the documentation, like "locked" or "blocked". Because there is only few letters different between query term and "locked" or "blocked", the edit distance is low and terms as considered as similar. This IR scenario also emphasises differences between Soundex and Metaphone. While Metaphone gets a high precision by means of a precise phonetic index, Soundex has a low precision due to the simplicity of its phonetic index. For example,"socket" as a Soundex index of "S230", like "sized", "sight", "squashed" or "succeed". Consequently, Soundex retrieves 291 documents while only 128 are relevant for this IR scenario.

In IR scenario 9 (see Appendix 7.4.9), query term ("unistd.h") is a file name and all methods handle this scenario successfully.

In IR scenarios 10 (see Appendix 7.4.10), search is made on proper names ("Andreas Gruenbacher"). In the documentation, proper names mainly refer to authors of functions. Only exact matching and Metaphone (designed to deal with name records) get high precision. Because of long query terms, edit distances with high value are accepted to validate matches. For example, at least 5 edit operations (add / remove / replace / transpose character) are tolerated to transform any term into "gruenbacher". So edit distance methods identify more matches between terms that have fewer similarities (and thus decrease precision). Soundex matches too much terms too, because it only considers the first letter and the three next consonants, so the length of the terms does not reduce the number of candidate.

4.1.3 Result relevance on all tasks

Figure 4.3 draws a global trend of approximate string matching algorithms. Each of these methods gets higher recall at the cost of precision. Indeed, approximate matching increases number of matched terms and thus increases number of matched documents. So compared to exact matching, more documents are retrieved, relevant (increasing recall) as non relevant (reducing precision).

Looking at result relevance globally (Appendix 7.4), F-Measure reveals that approximate string matching is neither a perfect solution nor a total mess. When search query is correct (IR scenarios from 1, 2 and 6 to 10), no method totally failed (both low precision and recall). However, on search queries with errors (IR scenarios from 3 to 5), only edit distance algorithms and Jaccard similarity gives relevant results (very high recall with low or very low precision).

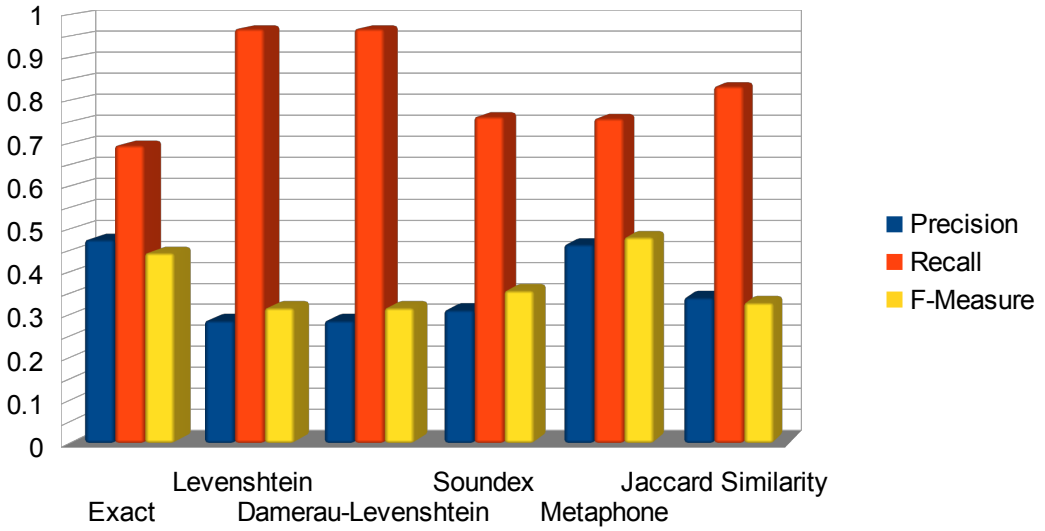

Figure 4.3 : Statistics about result relevance on all tasks
(data from Table 4.3 columns, "Ave.")

Levenshtein distance and Damerau-Levenshtein distance get the highest recall and lowest precision. Both algorithms return similar value so computing character transposition is unnecessary with such documentation. Soundex and Metaphone get similar recall, but Metaphone has higher precision thanks to a more elaborated phonetic index construction. Jaccard similarity has very good results in some IR scenarios, but this method matches too much non relevant terms as it does not take into account position of characters in terms.

Algorithm	**Precision**			**Recall**			**F-Measure**		
	Min.	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**	**Min.**	**Max.**	**Ave.**
Exact	0	1	0.475	0	1	0.695	0	1	0.444
Levenshtein	0.002	0.931	0.286	0.668	1	0.967	0.004	0.921	0.316
Damerau-Levenshtein	0.002	0.931	0.286	0.668	1	0.967	0.004	0.921	0.316
Soundex	0	0.938	0.311	0	1	0.762	0	0.968	0.356
Metaphone	0	1	0.464	0	1	0.758	0	1	0.481
Jaccard Similarity	0	1	0.340	0	1	0.833	0	0.988	0.328
All algorithms (average)	**0.001**	**0.967**	**0.360**	**0.223**	**1**	**0.830**	**0.001**	**0.966**	**0.374**

Table 4.3: Statistics about result relevance on all tasks

4.2 Performance

Performance is measured by execution time in seconds (Figure 4.4). One can observe that algorithms sorted by execution time are in the same order than sorted by time complexity (increasing order):
- exact matching (O(min(n, m))).
- phonetic indexation and similarity coefficient (O(n+m)).
- edit distance (O(nm)).

Damerau-Levenshtein algorithm is the only one method with an execution time above 1 second on average. Both edit distance methods have important execution time variations (respectively 2 sec. and 1 sec.). However, no pre-computing on terms in database is possible to optimize these method.

Metaphone consumes 0.5 seconds more per search on average, but this method (like Soundex) can be greatly optimised. Indeed, it is possible to compute Metaphone (or Soundex) indexes during documentation indexation and store it in database. Then, on searches, it is necessary to compute only once Metaphone (or Soundex) indexes of query terms and then perform exact matching between pre-encoded indexes in database and encoded indexes from query terms.

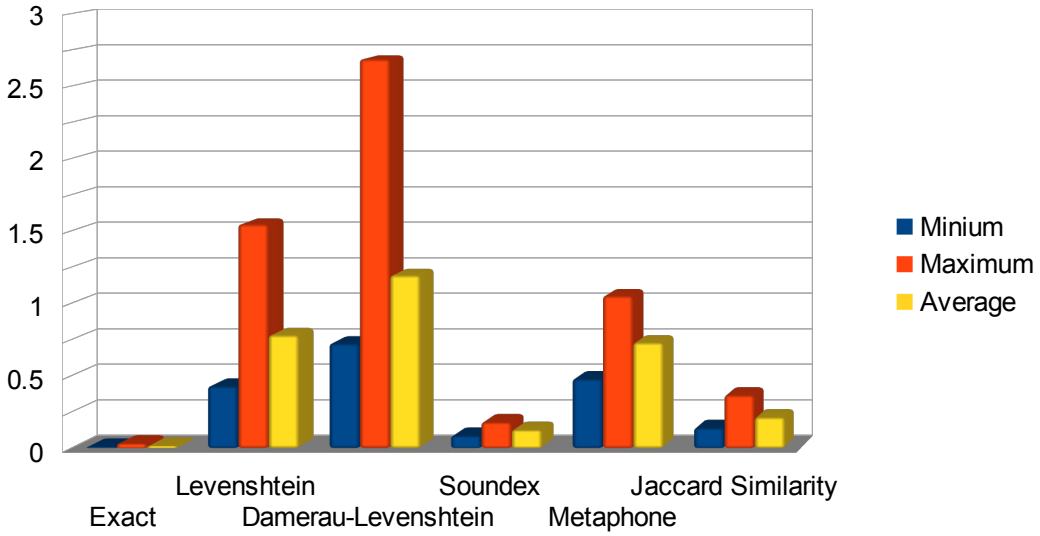

Figure 4.4 : Execution time in seconds
(data from Table 4.4)

Algorithm	**Minimum**	**Maximum**	**Average**
Exact	0.009	0.032	0.016
Levenshtein	0.429	1.547	0.781
Damerau-Levenshtein	0.724	2.687	1.195
Soundex	0.086	0.180	0.127
Metaphone	0.480	1.053	0.730
Jaccard Similarity	0.140	0.367	0.214
All algorithms (average)	**0.311**	**0.978**	**0.511**

Table 4.4: Execution time (in seconds)

5 Conclusion and recommendations

This section provides final conclusion for hypothesises and answers the research questions. Additional recommendations are provided for practitioners to implement approximate term matching in IR systems and for researchers to conduct further researches in the domain.

5.1 Research hypotheses

5.1.1 Hypothesis H0

H0: Approximate string matching increases result recall but decreases precision compared to exact string matching.

This hypothesis is verified for browsing tasks as well as searching tasks. Indeed, the highest average of precision and the lowest average of recall are reached with exact string matching (see Table 4.1 and Table 4.2). So any of the tested approximate string matching methods get higher recall and lower precision than exact string matching.

5.1.2 Hypothesis H1

H1: Computing approximate string matching takes more time compared to exact matching.

This hypothesis is verified in all the scenario (see Section 7.4). Exact string matching is always the fastest to perform matches.

5.2 Research questions

5.2.1 Which algorithms provide more relevant matches ?

Searching or browsing a technical documentation is a basic activity for practitioners. Contrary to the web, regular specialised documentations are not excessively large and cover only their domain. Such distinction is important as it makes absence of relevant documents from search results much more problematic than presence of non relevant document. That is why recall is more important than precision.

Results show that edit distance algorithms is a good replacement to exact string matching inside IR system working on technical documentation. Indeed, the Levenshtein distance increases recall by 32% on average compared to exact matching. Moreover, this method is the most constant, without any IR scenario (including queries with errors) with no or too few matches.

The cost of recall augmentation (through approximate matching) is a significant reduction of precision. Indeed, approximate matching increase the range of matched terms with non-relevant terms also. Even if edit distance has the lowest precision on average, the conclusion of this research is to recommend this method for similar IR system where precision is not a priority. If precision is still considered, Metaphone is a good choice as it provides a subtle improvement of both precision and recall compared to exact matching.

5.2.2 Which algorithms run faster ?

Before concluding this hypothesis, it is important to remind that monitored time only consider algorithm execution. Following steps are excluded from time monitoring:
- to query terms in database.
- to score and retrieve documents based on term scores.
- to sort documents.
- to print results.

Soundex and Jaccard similarity are faster (less than 0.25 seconds in average). Metaphone takes more time, but can be optimised with pre-computed indexed term and reach the same performance. Damerau-Levenshtein distance is the longest method (more than 1 second), but can be ignored in favour of Levenshtein distance (see section 5.3) that consume 0.781 seconds in average.

5.3 Recommendations for practitioners

Among approximate string matching methods, edit distance algorithms proves to be more useful than others. Damerau-Levenshtein can be avoided as it does not make difference with Levenshtein but takes much more time to compute. Jaccard similarity and Soundex methods are not recommended. If performance is a real issue, Metaphone can be considered despite its failures on query containing errors (IR scenarios 4 and 5). Practitioners has to remember that Metaphone only consider letters, and such constraint might not be compatible with some domain or tasks (e.g. search on version number or history).

Other aspects are determinant to obtain more relevant results:
- inverted index construction. Splitting full texts into terms vary according to documentation vocabulary and syntax. Moreover, for optimisation purpose, some calculations have to be done during document indexation (e.g. term idf).
- in specialised documentations, texts are usually divided into sections. Considering sections and document titles is determinant for some usages (e.g. with IR scenario 7, search in "ERROR" section only).
- term scoring threshold. This value is critical as it decides on whether or not indexed terms are used to score documents. If this value is too low, indexed terms very different to query terms will be used to score (non relevant) documents and precision will decrease. If too high, more relevant terms will be ignored and recall will decrease.

5.4 Future researches

This research can be easily extended to other approximate string matching methods, IR scenarios and documentation type. Indeed, the method has no such constraint and its implementation can easily be adapted (e.g. the benchmark tool is modular).

This study reveals that one of the limitations of phonetic approximation methods is the value they return. Contrary to edit distance and Jaccard similarity, Soundex and Metaphone only return true (for match) or false (for mismatch) but not the degree of match or mismatch. Such information is necessary to weight matched terms in IR systems. So a measure of similarity between two Soundex indexes or two Metaphone indexes would increase the interest for these methods in IR systems.

About result relevance, this research only considers presence of documents in the retrieved set. Document score or position in the sorted list of results have not been addressed. Result position is determinant with large document collections (e.g. the Web) as users are reluctant to search in long result lists. Moreover, studying the effects of different approximate matching methods on document scores would provide complementary recommendations for practitioners.

This study focuses on improving relevance results at the term scoring level. However, real IR systems are more elaborated and involve algorithms and precise calculations at different level (e.g. to process query terms or to score documents). It is necessary to explore interactions between approximate term matching and other IR aspects. By considering IR system as a whole, one can identify methods that, used with approximate term matching, increase precision and/or recall of results. Possible aspects can be: relevance feedback and query expansion to weight approximate matches, use of thesaurus or ontology to validate approximate matches.

6 References

[1] G. Navarro, ”A guided tour to approximate string matching”, *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, March 2001.

[2] L. Bauer and P. Nation, ”Word families”, *International Journal of Lexicography*, vol. 6, no. 4 , pp. 253-279, 1993.

[3] P. A. Hall and G. R. Dowling, ”Approximate string matching ”, *Computing Surveys*, vol. 2, no. 4, pp. 381-402, December 1980

[4] C. Li, J. Lu and Y. Lu, ”Efficient merging and filtering algorithms for approximate string searches ”.

[5] T. A. Lasko and S. E. Hauser, ”Approximate string matching algorithms for limited-vocabulary OCR output correction ”.

[6] W. I. Chang and E. L. Lawler, ”Sublinear approximate string matching and biological applications”, *Algorithmica*, no. 12, pp. 327-344, 1994.

[7] C. C. Liu, J. L. Hsu and A. L. Chen, ”An approximate string matching algorithm for content-based music data retrieval”.

[8] F. Crestani, M. Lalmas and C. J. Van Rijsbergen, *Information Retrieval Uncertainty and Logics: Advanced models for the representation and retrieval of information*, USA: Kluwer Academic Publishers, 1998.

[9] H. Masters, *A study of spelling errors*, Iowa: University of Iowa Studies in Education, 1927.

[10] Z. Ahmad, M. Umer Sarwar and M. Y. Javed, ”Agreo – a fast approximate pattern-matching tool”, *Journal of Applied Sciences Research*, vol. 2, no. 10, pp. 741-745, 2006.

[11] P. Stephenson, ”Chapter 6: completion, old and new”, *A User's Guide to the Z-Shell*, Mar. 23, 2003, [Online]. Available: zsh.sourceforge.net/Guide/zshguide06.html . [Accessed Mar. 24, 2013].

[12] ”Features in ubuntu 12.04.2”, *PrecisePangolin ReleaseNotes UbuntuDesktop - Ubuntu Wiki*, Feb. 2, 2013, [Online]. Available: wiki.ubuntu.com/PrecisePangolin/ReleaseNotes/UbuntuDesktop. [Accessed Mar. 24, 2013].

[13] R. W. Hamming, ”Error detecting and error correcting codes”, *The Bell System Technical Journal*, vol 29, no. 2, 1950.

[14] N. Jacobs, ”Relational sequence learning and user modelling”, 2004.

[15] V. I. Levenshtein, ”Binary codes capable of correcting deletions, insertions and reversals”, *Soviet Physics – Doklady,* vol. 10, no. 8, pp. 707-710, 1966.

[16] F. J. Damerau, ”A technique for computer detection and correction of spelling errors”, *Communications of the ACM*, vol. 7, no. 3, pp. 171-176, 1964.

[17] ”soundex”, *PHP soundex - Manual*, Apr. 12, 2013, [Online] php.net/soundex [Accessed Apr. 14 2013].

[18] ”metaphone”, *PHP metaphone - Manual*, Apr. 12, 2013, [Online] php.net/metaphone [Accessed Apr. 14 2013].

[19] ”fuzzystrmatch”, *PostgreSQL Documentation 9.2 fuzzystrmatch*, [Online] postgresql.org/docs/8.3/static/fuzzystrmatch.html [Accessed Apr. 15 2013].

[20] ”The soundex indexing system”, *Soundex System*, May 30, 2007, [Online] archives.gov/research/census/soundex.html. [Accessed Mar. 20, 2013].

[21] L. Philips, ”Hanging on the metaphone”, *Computer Language*, vol. 7, no. 12, 1990.

[22] ”MetaphoneEn_v1_1”, *Formal specification for the Metaphone algorithms in different languages. - Google Project Hosting*, Jan. 4, 2012, [Online] code.google.com/p/metaphone-standards/wiki/MetaphoneEn_v1_1 [Accessed Apr. 20 2013].

[23] C. D. Manning, P. Raghavan and H. Schütze, *An Introduction to Information Retrieval*, Cambridge: Cambridge University Press, 2009.

[24] C. J. Van Rijsbergen, *Information Retrieval*, Second Edition, Glasgow: University of Glasgow, 1979.

[25] J. C. French, A. L. Powell and E. Schulman, ”Applications of approximate word matching in information retrieval”, 1997.

[26] A. Ghias, J. Logan, D. Chamberlin and B. C. Smith ”Query by humming”.

[27] R. J. McNab, L. A. Smith, I. H. Witten, C. L. Henderson and S. J. Cunningham, ”Towards the digital music library: Tune retrieval from acoustic input”, 1995.

[28] C. Lok-Iam, ”Approximate string matching in DNA sequences”, 2003.

[29] X. Chen, S. Kwong and M. Li, ”A compression algorithm for DNA sequences”, 2001.

[30] S. J. Grannis, J. M. Overhage and C. McDonald, ”Real world performance of approximate string comparators for use in parient matching”, 2004.

[31] W. E. Winkler, ”Approximate string comparator search strategies for very large administrative lists”.

[32] K. Dzonsons, ”History of UNIX manpages”, *History of UNIX Manpages*, Nov. 11, 2011, [Online] manpages.bsd.lv/history.html [Accessed Apr. 2 2013].

[33] *Ubuntu Manpage man - an interface to the on-line reference manuals*, [Online] manpages.ubuntu.com/manpages/precise/en/man1/man.1.html [Accessed Apr. 19 2013].

[34] K. Williamson, *Research methods for students, academics and professionals*, Second Edition, 2002.

[35] *Ubuntu Manpage Directory Listing*, [Online] manpages.ubuntu.com/manpages/precise/en [Accessed Mar. 15 2013].

[36] *Ubuntu Manpage fcntl - manipulate file descriptor*, [Online] http://manpages.ubuntu.com/manpages/precise/en/man2/fcntl64.2.html [Accessed May 30 2013].

[37] *Ubuntu Manpage intro -- introduction to system calls and error numbers*, [Online] manpages.ubuntu.com/manpages/precise/en/man2/intro.2freebsd.html [Accessed May 30 2013].

[38] *Ubuntu Manpage sched_yield - yield the processor*, [Online] manpages.ubuntu.com/manpages/precise/en/man2/sched_yield.2.html [Accessed May 30 2013].

7 Appendix

7.1 Documentation

7.1.1 Yes man page

```
NAME
       yes - output a string repeatedly until killed

SYNOPSIS
       yes [STRING]...
       yes OPTION

DESCRIPTION
       Repeatedly output a line with all specified STRING(s), or `y'.

       --help display this help and exit

       --version
              output version information and exit

AUTHOR
       Written by David MacKenzie.

REPORTING BUGS
       Report yes bugs to bug-coreutils@gnu.org GNU coreutils home
page: <http://www.gnu.org/software/coreutils/>
       General help using GNU software: <http://www.gnu.org/gethelp/>
       Report yes translation bugs to
<http://translationproject.org/team/>

COPYRIGHT
       Copyright (C) 2011 Free Software Foundation, Inc.  License
GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>.
       This is free software: you are free to change and redistribute
it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       The  full  documentation for yes is maintained as a Texinfo
manual.  If the info and yes programs are properly  installed  at
your  site,  the command
              info coreutils 'yes invocation'
       should give you access to the complete manual.
```

7.2 Inverted index

7.2.1 Database structure (MySQL)

```
CREATE TABLE IF NOT EXISTS `documents` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `document` tinytext CHARACTER SET latin1 COLLATE latin1_bin NOT
NULL,
  `sample` tinyint(4) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `document` (`document`(255))
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `terms` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `term` tinytext CHARACTER SET latin1 COLLATE latin1_bin NOT NULL,
  `idf` decimal(10,9) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `term` (`term`(255))
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `term_distribution` (
  `id_term` bigint(20) NOT NULL,
  `id_document` bigint(20) NOT NULL,
  `count` tinyint(4) NOT NULL,
  UNIQUE KEY `id_term` (`id_term`,`id_document`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS `term_scoring` (
  `id_term` bigint(20) NOT NULL,
  `score` decimal(10,9) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

7.2.2 Import script (PHP)

```
$terms_list = array();
getFilelistFromServer( "manpages.ubuntu.com/manpages/precise/en/" );
computeIdf();



//  Fetch the given url and analyse links to folders or files
function getFilelistFromServer( $url )
{
    $d = load_html_file( $url );
    $content = $d->getElementById( "content" );

    foreach ( link in $content as $el )
        if ( match_pattern "/\w\/$/" on $el )
            getFilelistFromServer( $url . $el );
        else if ( match_pattern '/.html$/' on $el )
            getInvertedIndexFromFilename( $url . $el );
}
```

```
//  Parse the given file to compute the inverted index of the terms
function getInvertedIndexFromFilename( $filename )
{
    $d = load_html_file( $url );
    $c = $d->getElementById( "content" );
    $matches = match_pattern "/\b(?:[\w-_+.@\/])+\b/" on $c;

    $invertedIndex = array();
    foreach ( $matches as $word )
        if ( isset( $invertedIndex[$word] ) )
            $invertedIndex[$word]['count'] += 1;
        else if ( strlen( $word ) >= 2 AND strlen( $word ) < 255 )
            $invertedIndex[$word]['count'] = 1;

    $sample = 0;
    $match=match_pattern '/SYNOPSIS.*(DESCRIPTION|OPTIONS)/Us' on $c;
    if ( "/man2/" in $filename AND "#include" in $match )
        $sample = 1;

    storeInvertedIndexInDatabase($filename, $invertedIndex, $sample);
}



//  Store the inverted index in the database for the given document
function storeInvertedIndexInDatabase($file, $invertedIndex, $sample)
{
    mysql_insertinto_documents( $file, $sample );
    $id_document = mysql_last_inserted_id();

    foreach ( $invertedIndex as $term => $term_attr )
    {
        if ( ! isset( $terms_list[$term] ) )
        {
            mysql_insertinto_terms( $term );
            $terms_list[$term] = mysql_last_inserted_id();
        }
        $invertedIndex[$term]['id'] = $terms_list[$term];
    }

    foreach ( $invertedIndex as $t )
        mysql_insertinto_termdistrib($t['id'], $id_doc, $t['count']);
}



//  Set the idf for terms in database
function computeIdf()
{
    $collection_size=mysql_query("SELECT COUNT(id) FROM documents");
mysql_query(
    "UPDATE `terms` as T LEFT JOIN (
        SELECT id, LOG10($collection_size/COUNT(id_document)) as idf
        FROM `terms`
        LEFT JOIN `term_distribution` ON `id` = `id_term`
        GROUP BY `id_term` ) AS U ON `T`.`id` = `U`.`id`
    SET `T`.`idf` = `U`.`idf`");
}
```

7.3 Benchmark

7.3.1 Exact matching

```
float       algo_exact( string s1, string s2 )
{
    float   similarity = 0;
    int     i = 0;
    int     j = 0;

    while ( s1[i] different than end of string AND s1[i] == s2[j] )
    {
        i = i + 1;
        j = j + 1;
    }

    if ( s1[i] == s2[j] )
        similarity = 1;

    return ( similarity );
}
```

7.3.2 Levenshtein distance

```
float       algo_levenshtein( string s1, string s2 )
{
    float   similarity = 0;
    int     length_s1 = strlen( s1 );
    int     length_s2 = strlen( s2 );
    int     matrix[256][256];
    int     i = 0;
    int     j = 0;
    int     cost;

    while ( i <= length_s1 )
    {
        m[i][0] = i;
        i = i + 1;
    }

    while ( j <= length_s2 )
    {
        m[0][j] = j;
        j = j + 1;
    }

    i = 0;
    while ( ++i <= length_s1 )
    {
        j = 0;
        while ( ++j <= length_s2 )
        {
            if ( s1[i - 1] == s2[j - 1] )
                cost = 0;
            else
                cost = 1;
            m[i][j] = MIN(
                            MIN(
                                m[i - 1][j] + 1,
                                m[i][j - 1] + 1
                            ),
                            m[i - 1][j - 1] + cost
                        );
        }
    }

    similarity = 1 - ( m[length_s1][length_s2] /
                        MAX( length_s1, length_s2 ) );
    if ( similarity < 0.5 )
        similarity = 0;
    return ( similarity );
}
```

7.3.3 Damerau-Levenshtein distance

```
float       algo_damerau( string s1, string s2 )
{
    float   similarity = 0;
    int     length_s1 = strlen( s1 );
    int     length_s2 = strlen( s2 );
    int     matrix[256][256];
    int     i = 0;
    int     j = 0;
    int     cost;

    while ( i <= length_s1 )
    {
        m[i][0] = i;
        i = i + 1;
    }

    while ( j <= length_s2 )
    {
        m[0][j] = j;
        j = j + 1;
    }

    i = 0;
    while ( ++i <= length_s1 )
    {
        j = 0;
        while ( ++j <= length_s2 )
        {
            if ( s1[i - 1] == s2[j - 1] )
                cost = 0;
            else
                cost = 1;
            m[i][j] = MIN(
                            MIN(
                                m[i - 1][j] + 1,
                                m[i][j - 1] + 1
                            ),
                            m[i - 1][j - 1] + cost
                        );
            if ( i > 1 AND j > 1 AND s1[i - 1] == s2[j - 2]
                    AND s1[i - 2] == s2[j - 1] )
                m[i][j] = MIN( m[i][j], m[i - 2][j - 2] + cost );
        }
    }

    similarity = 1 - ( m[length_s1][length_s2] /
                        MAX( length_s1, length_s2 ) );
    if ( similarity < 0.5 )
        similarity = 0;
    return ( similarity );
}
```

7.3.4 Soundex

```
float       algo_soundex( string s1, string s2 )
{
    string  code1 = algo_soundex_convert_string( s1 );
    string  code1 = algo_soundex_convert_string( s2 );

    return ( algo_exact( code1, code2 ) );
}

string      algo_soundex_convert_string( string s  )
{
    string  code;
    array   rule[SCHAR_MAX + 1] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, '1', '2', '3',
0, '1', '2', 0, 0, '2', '2', '4', '5', '5', 0, '1', '2', '6', '2',
'3', 0, '1', 0, '2', 0, '2', 0, 0, 0, 0, 0 };
    int     i = 0;
    int     prev;

    while ( s[i] different than end of string AND
            s[i] different than letter )
        i = i + 1;
    append s[i] to code;
    prev = s[i]

    if ( s[i] different than end of string )
    {
        i = i + 1;
        while ( s[i] different than end of string AND
                code[j] different than end of string )
        {
            if (rule[s[i]] different than rule[prev] AND rule[s[i]])
                append rule[s[i]] to code;
            if ( s[i] different than 'h' && s[i] different than 'w' )
                prev = s[i];
            i = i + 1;
        }
    }
    return ( code );
}
```

7.3.5 Metaphone

```
float       algo_metaphone( string s1, string s2 )
{
    string  code1 = algo_metaphone_convert_string( s1 );
    string  code2 = algo_metaphone_convert_string( s2 );

    return ( algo_exact( code1, code2 ) );
}

string      algo_metaphone_convert_string( string s )
{
    string  code;
    int     i = 0;

    s = remove_double_consonant_different_than_g_and_c( s );

    while ( s[i] )
        if ( s[i] different than letter )
            i = i + 1;
        else if (s[i] first letter AND s[i] == 'a' AND s[i+1] == 'e')
        {
            append 'e' to code;
            i = i + 2;
        }
        else if ( s[i] first letter AND s[i + 1] == 'n' AND
                    ( s[i] == 'g' OR s[i] == 'k' OR s[i] == 'p' ) )
        {
            append 'n' to code;
            i = i + 2;
        }
        else if ( s[i] not first letter AND s[i] == 's'
                    AND s[i + 1] == 'c' AND s[i + 2] == 'h' )
        {
            append "sk" to code;
            i = i + 3;
        }
        else if ( s[i] == 't' AND ( s[i + 1] == 'c' OR
                    s[i + 1] == 's' ) AND s[i + 2] == 'h' )
        {
            append 'x' to code;
            i = i + 3;
        }
        else if ( s[i] == 'c' OR s[i] == 's' ) AND s[i + 1] == 'h' )
        {
            append 'x' to code;
            i = i + 2;
        }
        else if (s[i] == 'c' AND s[i + 1] == 'i' AND s[i + 2] == 'a')
        {
            append 'x' to code;
            i = i + 3;
        }
        else if ( s[i] == 'c' AND s[i + 1] == 'k' )
        {
            append 'k' to code;
            i = i + 2;
        }
        else if ( s[i] == 'c' OR s[i] == 'q' )
```

```
        {
            append 'k' to code;
            i = i + 1;
        }
        else if ( s[i] == 'd' )
        {
            append 't' to code;
            i = i + 1;
        }
        else if (s[i] == 'g' AND s[i + 1] == 'h' AND s[i + 2] == 't')
        {
            append "ft" to code;
            i = i + 3;
        }
        else if ( s[i] == 'd' AND s[i + 1] == 'g' AND
          ( s[i + 2] == 'e' OR s[i + 2] == 'i' OR s[i + 2] == 'y' ) )
        {
            append 'j' to code;
            i = i + 2;
        }
        else if ( ( s[i] == 'e' OR s[i] == 'i' OR s[i] == 'y' ) AND
            s[i + 1] == 'g' AND s[i + 2] == 'h' AND s[i + 3] == 't' )
        {
            append "ft" to code;
            i = i + 4;
        }
        else if (s[i] == 'g' AND s[i+1] == 'h' AND s[i+2] not vowel)
            us += 1;
        else if ( s[i] == 'g' AND s[i + 1] == 'n' AND
                    s[i + 2] == 'e' AND s[i + 3] == 'd' )
        {
            append 'n' to code;
            i = i + 4;
        }
        else if ( s[i] == 'g' && s[i + 1] == 'n' )
        {
            append 'n' to code;
            i = i + 2;
        }
        else if ( (s[i] first letter OR s[i - 1] different than 'g')
                    AND s[i] == 'g' AND ( s[i + 1] == 'e'
                    OR s[i + 1] == 'i' OR s[i + 1] == 'y' ) )
        {
            append 'l' to code;
            i = i + 2;
        }
        else if ( s[i] == 'g' )
        {
            append 'k' to code;
            while ( s[i] == 'g' )
                i = i + 1;
        }
        else if (s[i]== 'h' AND s[i-1] is vowel AND s[i+1] not vowel)
            i = i + 1;
        else if ( s[i] == 'p' AND s[i + 1] == 'h' )
        {
            append 'f' to code;
            i = i + 2;
        }
```

```
        else if ( s[i] == 't' AND s[i + 1] == 'h' )
        {
            append '0' to code;
            i = i + 2;
        }
        else if ( s[i] == 'm' AND s[i + 1] == 'b')
        {
            append 'm' to code;
            i = i + 2;
        }
        else if ( ( s[i] == 's' OR s[i] == 't' ) AND s[i + 1] == 'i'
                    AND ( s[i + 2] == 'a' OR s[i + 2] == 'o' ) )
        {
            append 'x' to code;
            i = i + 1;
        }
        else if ( s[i] == 's' AND s[i + 1] == 'c' AND
          ( s[i + 2] == 'e' OR s[i + 2] == 'i' OR s[i + 2] == 'y' ) )
        {
            append 's' to code;
            i = i + 2;
        }
        else if ( s[i] == 'c' AND ( s[i + 1] == 'e' OR
                    s[i + 1] == 'i' || s[i + 1] == 'y' ) )
        {
            append 's' to code;
            i = i + 1;
        }
        else if ( s[i] first letter AND s[i] == 'x' )
        {
            append 's' to code;
            i = i + 1;
        }
        else if ( s[i] == 'z' )
        {
            append 's' to code;
            i = i + 1;
        }
        else if ( s[i] == 'v' )
        {
            append 'f' to code;
            i = i + 1;
        }
        else if (s[i] first letter AND s[i] == 'w' AND s[i+1] == 'r')
        {
            append 'r' to code;
            i = i + 2;
        }
        else if (s[i] first letter AND s[i] == 'w'AND s[i+1] == 'h' )
        {
            append 'w' to code;
            i = i + 2;
        }
        else if ((s[i] == 'w' OR s[i] == 'y') AND s[i + 1] not vowel)
            i = i + 1;
        else if ( ( s[i] == 'a' OR s[i] == 'e' OR s[i] == 'i'
                    OR *us == 'o' ) AND s[i + 1] == 'x' )
        {
            append "ks" to code;
            i = i + 2;
        }
```

```
        else if ( s[i] not first letter AND s[i] is vowel )
            ++us;
        else
        {
            append s[i] to code;
            i = i + 1;
        }
    return ;
}
```

7.3.6 Jaccard similarity

```
float       algo_jaccard( string s1, string s2 )
{
    float   similarity = 0;
    array   inter_set[SCHAR_MAX + 1];
    array   union_set[SCHAR_MAX + 1];
    int     inter_size = 0;
    int     union_size = 0;
    int     i = 0;

    fill_array_with_0( inter_set );
    fill_array_with_0( union_set );

    while ( s1[i] different than end of string )
    {
        if ( union_set[ s1[i] ] == 0 )
        {
            union_set[ s1[i] ] = 1;
            union_size = union_size + 1;
        }
        i = i + 1
    }

    i = 0;
    while ( s2[i] different than end of string )
    {
        if ( union_set[ s2[i] ] == 1 AND inter_set[ s2[i] ] == 0 )
        {
            inter_set[ s2[i] ] = 1;
            inter_size = inter_size + 1;
        }
        if ( union_set[ s2[i] ] == 0 )
        {
            union_set[ s2[i] ] = 1;
            union_size = union_size + 1;
        }
        i = i + 1;
    }

    similarity = inter_size / union_size;
    if ( similarity < 0.75 )
        similarity = 0;
    return ( similarity );
}
```

7.3.7 Usage

```
usage: benchmark     [--print NBR]   Number of printed best results
                                                    (default: all)
                     [--algo ALGO]   Algorithm used to match terms
                                                    (default: all)
                     [--verbose] Print information such as time and
                                          matched terms (default: none)
                     QUERY

Algorithms:
    exact           exact matching (match exactly similar strings)
    levenshtein     levenshtein distance (character insertion,
                                          deletion and substitution)
    damerau         damerau-levenshtein distance (character insertion,
                                 deletion, substitution and transposition)
    soundex         soundex (index name by pronunciation)
    metaphone       metaphone (index word by pronunciation)
    jaccard         jaccard similarity (overlap coefficient
                                          | s1 n s2 | / | s1 u s2 |)
```

7.4 Experiment results

7.4.1 IR scenario 1: ”open file”

Algorithm	Precision	Recall	F-Measure	Execution time (in sec.)
Exact	0.014	1	0.028	0.032
Levenshtein	0.010	1	0.020	0.632
Damerau-Levenshtein	0.010	1	0.020	0.909
Soundex	0.011	1	0.022	0.180
Metaphone	0.009	1	0.018	0.763
Jaccard Similarity	0.010	1	0.020	0.237
Minimun	**0.009** (Metaphone)	**1** (All)	**0.018** (Metaphone)	**0.032** (Exact)
Maximum	**0.014** (Exact)	**1** (All)	**0.028** (Exact)	**0.909** (Damerau-Levenshtein)
Average	**0.011**	**1**	**0.021**	**0.459**

7.4.2 IR scenario 2: ”get hostname”

Algorithm	Precision	Recall	F-Measure	Execution time (in sec.)
Exact	0.006	1	0.012	0.023
Levenshtein	0.002	1	0.004	0.990
Damerau-Levenshtein	0.002	1	0.004	1.392
Soundex	0.005	1	0.010	0.180
Metaphone	0.005	1	0.010	1.053
Jaccard Similarity	0.005	1	0.010	0.367
Minimun	**0.002** (Levenshtein / Damerau-Levenshtein)	**1** (All)	**0.004** (Levenshtein / Damerau-Levenshtein)	**0.023** (Exact)
Maximum	**0.006** (Exact)	**1** (All)	**0.012** (Exact)	**1.392** (Damerau-Levenshtein)
Average	**0.004**	**1**	**0.008**	**0.668**

7.4.3 IR scenario 3: ”get timestamp”

Algorithm	Precision	Recall	F-Measure	Execution time (in sec.)
Exact	0.012	0.667	0.024	0.022
Levenshtein	0.005	1	0.010	1.017
Damerau-Levenshtein	0.005	1	0.010	1.788
Soundex	0.010	0.667	0.020	0.141
Metaphone	0.011	0.667	0.022	0.917
Jaccard Similarity	0.005	1	0.010	0.266
Minimun	**0.005** (Levenshtein / Damerau-Levenshtein / Jaccard)	**0.667** (Exact / Soundex / Metaphone)	**0.010** (Levenshtein / Damerau-Levenshtein / Jaccard Similarity)	**0.022** (Exact)
Maximum	**0.012** (Exact)	**1** (Levenshtein / Damerau-Levenshtein / Jaccard)	**0.024** (Exact)	**1.788** (Damerau-Levenshtein)
Average	**0.008**	**0.834**	**0.016**	**0.692**

7.4.4 IR scenario 4: ”fntcl”

Algorithm	**Precision**	**Recall**	**F-Measure**	**Execution time (in sec.)**
Exact	0	0	0	0.011
Levenshtein	0.014	1	0.028	0.429
Damerau-Levenshtein	0.014	1	0.028	0.743
Soundex	0	0	0	0.086
Metaphone	0	0	0	0.508
Jaccard Similarity	0.034	1	0.066	0.140
Minimun	**0** (Exact / Soundex / Metaphone)			**0.011** (Exact)
Maximum	**0.034** (Jaccard)	**1** (Levenshtein / Damerau-Levenshtein / Jaccard)	**0.066** (Jaccard)	**0.743** (Damerau-Levenshtein)
Average	**0.010**	**0.500**	**0.020**	**0.320**

7.4.5 IR scenario 5: ”youmask”

Algorithm	**Precision**	**Recall**	**F-Measure**	**Execution time (in sec.)**
Exact	0	0	0	0.010
Levenshtein	0.017	1	0.033	0.710
Damerau-Levenshtein	0.017	1	0.033	0.999
Soundex	0	0	0	0.100
Metaphone	0	0	0	0.784
Jaccard Similarity	0	0	0	0.187
Minimun	**0** (Exact / Soundex / Metaphone / Jaccard)			**0.010** (Exact)
Maximum	**0.017** (Levenshtein / Damerau-Levenshtein)	**1** (Levenshtein / Damerau-Levenshtein)	**0.033** (Levenshtein / Damerau-Levenshtein)	**0.999** (Damerau-Levenshtein)
Average	**0.006**	**0.333**	**0.011**	**0.465**

7.4.6 IR scenario 6: ”posix”

Algorithm	Precision	Recall	F-Measure	Execution time (in sec.)
Exact	0.931	0.371	0.531	0.011
Levenshtein	0.931	0.668	0.778	0.558
Damerau-Levenshtein	0.931	0.668	0.778	0.724
Soundex	0.864	1	0.927	0.092
Metaphone	0.955	1	0.977	0.480
Jaccard Similarity	0.931	0.371	0.531	0.152
Minimun	**0.864** (Soundex)	**0.371** (Exact)	**0.531** (Exact / Jaccard)	**0.011** (Exact)
Maximum	**0.955** (Metaphone)	**1** (Soundex / Metaphone)	**0.977** (Metaphone)	**0.724** (Damerau-Levenshtein)
Average	**0.924**	**0.680**	**0.754**	**0.336**

7.4.7 IR scenario 7: ”eoverflow”

Algorithm	Precision	Recall	F-Measure	Execution time (in sec.)
Exact	0.938	1	0.968	0.015
Levenshtein	0.385	1	0.556	0.641
Damerau-Levenshtein	0.385	1	0.556	0.783
Soundex	0.938	1	0.968	0.103
Metaphone	0.938	1	0.968	0.694
Jaccard Similarity	0.536	1	0.698	0.149
Minimun	**0.385** (Levenshtein / Damerau-Levenshtein)	**1** (All)	**0.556** (Levenshtein / Damerau-Levenshtein)	**0.015** (Exact)
Maximum	**0.938** (Exact / Soundex / Metaphone)	**1** (All)	**0.968** (Exact / Soundex / Metaphone)	**0.783** (Damerau-Levenshtein)
Average	**0.687**	**1**	**0.786**	**0.398**

7.4.8 IR scenario 8: ”socket”

Algorithm	**Precision**	**Recall**	**F-Measure**	**Execution time (in sec.)**
Exact	1	0.938	0.968	0.011
Levenshtein	0.612	1	0.759	0.570
Damerau-Levenshtein	0.612	1	0.759	0.816
Soundex	0.430	0.977	0.597	0.113
Metaphone	0.870	0.938	0.903	0.584
Jaccard Similarity	1	0.977	0.988	0.195
Minimun	**0.430** (Soundex)	**0.938** (Exact / Metaphone)	**0.597** (Soundex)	**0.011** (Exact)
Maximum	**1** (Exact / Jaccard)	**1** (Levenshtein / Damerau-Levenshtein)	**0.988** (Jaccard)	**0.816** (Damerau-Levenshtein)
Average	**0.754**	**0.972**	**0.829**	**0.382**

7.4.9 IR scenario 9: ”unistd.h”

Algorithm	**Precision**	**Recall**	**F-Measure**	**Execution time (in sec.)**
Exact	0.851	0.978	0.910	0.009
Levenshtein	0.854	1	0.921	0.719
Damerau-Levenshtein	0.854	1	0.921	1.104
Soundex	0.766	0.978	0.859	0.096
Metaphone	0.851	0.978	0.910	0.518
Jaccard Similarity	0.851	0.978	0.910	0.155
Minimun	**0.766** (Soundex)	**0.978** (Exact / Soundex / Metaphone / Jaccard)	**0.859** (Soundex)	**0.009** (Exact)
Maximum	**0.854** (Levenshtein / Damerau-Levenshtein)	**1** (Levenshtein / Damerau-Levenshtein)	**0.921** (Levenshtein / Damerau-Levenshtein)	**1.104** (Damerau-Levenshtein)
Average	**0.838**	**0.985**	**0.905**	**0.434**

7.4.10 IR scenario 10: ”Andreas Gruenbacher”

Algorithm	**Precision**	**Recall**	**F-Measure**	**Execution time (in sec.)**
Exact	1	1	1	0.019
Levenshtein	0.026	1	0.051	1.547
Damerau-Levenshtein	0.026	1	0.051	2.687
Soundex	0.082	1	0.152	0.180
Metaphone	1	1	1	0.996
Jaccard Similarity	0.024	1	0.047	0.287
Minimun	**0.024** (Jaccard)	**1** (All)	**0.047** (Jaccard)	**0.019** (Exact)
Maximum	**1** (Exact / Metaphone)	**1** (All)	**1** (Exact / Metaphone)	**2.687** (Damerau-Levenshtein)
Average	**0.360**	**1**	**0.384**	**0.953**