A Search Interface For Health

Paul Dalziel

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Abstract

This project is based on the User-Centred Health Information Retrieval Task 2 at the CLEF eHealth 2015 workshop, which seeks to investigate the effectiveness of information retrieval systems when searching for medical information on the web, with the aim to support the research and development of search engines designed to aid health information seeking.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ______________________

Contents

Chapter 1

Introduction

1.1 Motivation and Context

This project was inspired by 2015 CLEF eHealth Lab Task 2 which aims to evaluate the effectiveness of information retrieval systems when searching for health content on the web, with the objective to foster research and development of search engines tailored to health information seeking[2].

1.2 Motivation

Lab Task 2 investigated the problems faced by ordinary people (e.g. not medical experts), while searching to self-diagnose when confronted with a sign, symptom or medical condition that they, or a family member may have. For example, when confronted with signs of jaundice, a non-expert user may use a query such as ”white part of eye turned green” when searching for information [1].

This type of information need and corresponding seeking activity has not been previously considered in CLEF eHealth tasks, nor in other information retrieval evaluation campaigns such as the Text REtrieval Conference (TREC) despite evidence which shows that 72% of internet users in the United States of America look online frequently for information on a disease or medical issue[cite]. This particular type of information seeking behaviour is poorly supported by general purpose search engines [9] which can lead to incorrect self-diagnosis and self-treatment[6].

1.2.1 Challenges

One of the main difficulties faced by search engines stems from a discrepancy between the vocabularies of the users, and the authors of high quality resources. Ordinary users typically do not know the correct medical terminology to accurately name or describe the symptoms they are presented with. This lack of vocabulary results in queries which are often circumlocutory, substituting medical terminology with long, descriptive and ambiguous statements [10]. The problem is further complicated by the possibility that even if the search returns relevant documents, the user may not recognise them as such as a result of their poor understanding of medical terminology, effectively rendering the documents useless.

1.3 Background of CLEF

The CLEF Initiative (Conference and Labs of the Evaluation Forum, formerly known as Cross-Language Evaluation Forum), is a self-organized group set up to facilitate and promote research, innovation and development of information access systems. The CLEF Initiative is structured in two main parts, a series of Evaluation Labs and an annual peer-reviewed conference. Since 2000 the CLEF has become a leader in the international IR research community, with research covering a wide range of areas such as multilingual and multimodal system testing, tuning and evaluation, investigation of data structures and semantical enrichment of data, the creation of benchmarking test collections and the exploration of new evaluation methodologies[2].

1.4 Aims

To be successful in this task the design of the retrieval system should have the following features.
1. To effectively deal with the circumlocutory and ambiguous nature of the queries
2. Account for the discrepancies between the query and document vocabulary
3. Incorporate some form of readability metric to ensure returned documents are useful to the user.

1.5 Outline

1. Introduction
2. Background and Related work
3. Implementation
4. Results and Analysis
5. Discussion and Conclusion

Chapter 2

Background

2.1 Basics of an Information Retrieval System

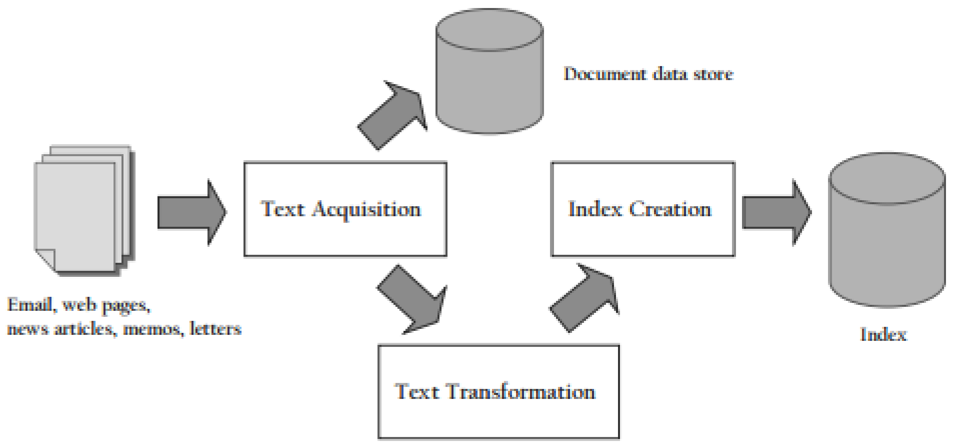
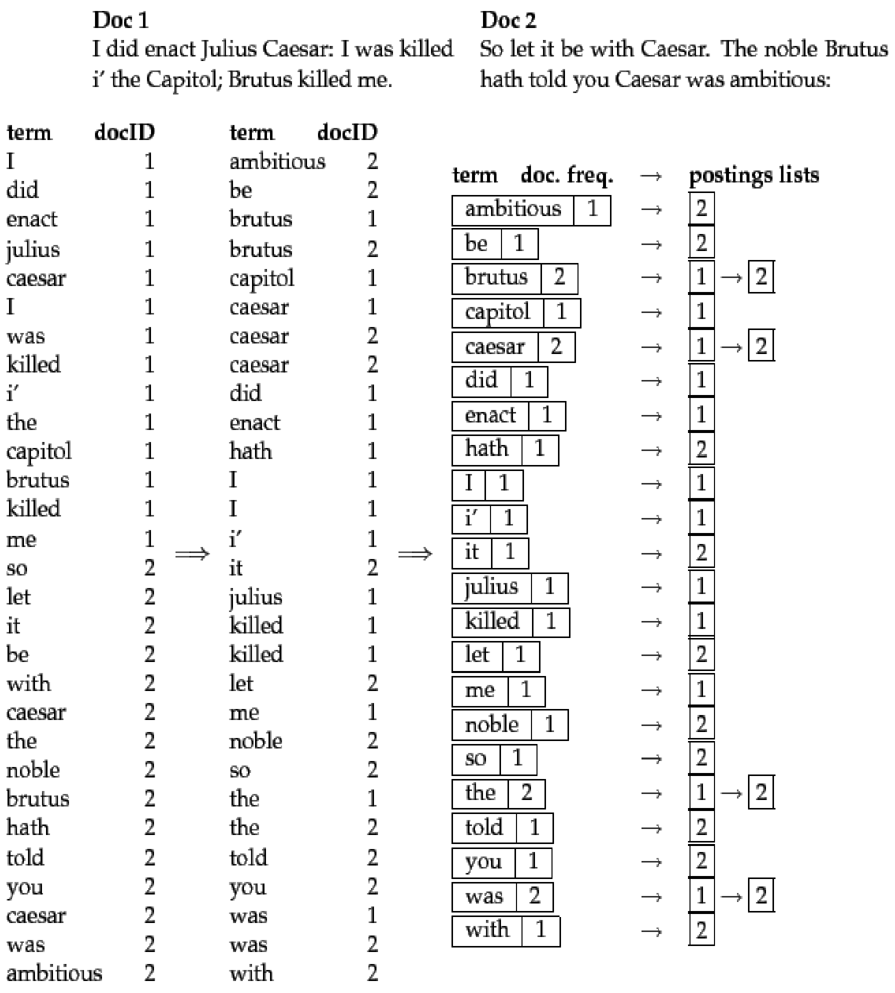


Figure 2.1: An IR System

2.2 Indexing

An Information Retrieval (IR) system takes formal statements of information needs, in the form of queries and matches them to data objects which are referred to as documents, although they are not necessarily text. Unlike a database system, which returns only exact matches, an Information Retrieval system will return all queries which match to some specified degree. The returned documents are classed as relevant and the remainder of the collection will be classed as non-relevant. The Information Retrieval system will use ranking to sort the retrieved documents, ensuring the most relevant are show to the user first.



Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

$\Longrightarrow$

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Figure 2.2: Inverted indexing process

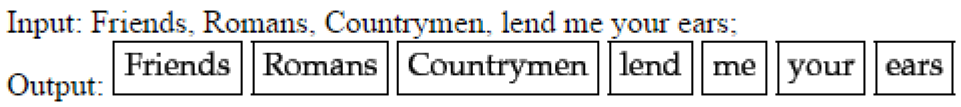


Figure 2.3: Processing the text into tokens allows the representation of each document as a bag-of-words, disregarding grammar and word order but keeping multiplicity.

2.3 Tokenization

Tokenization is the process of forming words from the character sequences in a document by splitting them into individual tokens. Each token is an instance of a character sequence in the document which is grouped in such a way as to have sematic meaning within the document. Typically tokenization involves some degree of normalisation such as the removal of punctuation and tokens, known as stopwords, which are so common as to appear in almost every query. Tokens will usually not match exactly to the corresponding words in the document due to the normalisation process of stemming which reduces words to their word stem, base or root form.

In order to efficiently search these representations, the Information Retrieval system will normally use the inverted index data structure. The index is created by assigning a document identifier to each document in the collection then linking the normalised tokens from the document as in show in the example figure 2.3. The index is then sorted so that the terms are alphabetical, as seen the middle column of the example index figure 2.2 and multiple instances of tokens in each document are merged. The next step is to group instances of the same token and divide into dictionary and postings structure as in the right column in the figure 2.2. The dictionary may also record some statistics such as the document frequency, which is the number of times the token appears in the document and the number of documents which contain the token. This process greatly reduces the storage requirements of the index and improves the performance of the Information Retrieval system[5].

2.4 Evaluation of an Information Retrieval System

Evaluation is essential to understanding whether an information retrieval system or search engine is effective in a specific application or task, and whether it offers any improvements over existing strategies. There are numerous different techniques s

2.4.1 Measures

Recall and Precision

One of the main distinctions made when evaluating search engines is between effectiveness and efficiency. Effectiveness can be thought of as a measure of the ability of the system to retrieve the correct information and efficiency measures how quickly this is done. The most common effectiveness measures are recall and precision. Recall is a measure of how well the system finds all the relevant documents for a query, and precision measures its ability to reject non-relevant documents.

	Relevant	Non-Relevant
Retrieved	$A \cap B$	$\overline{A} \cap B$
Not Retrieved	$A \cap \overline{B}$	$\overline{A} \cap \overline{B}$

$$
\begin{aligned}
Recall &= \frac{|A \cap B|}{|A|} \\
Precision &= \frac{|A \cap B|}{|B|}
\end{aligned}
$$

Figure 2.4: Recall and Precision

In the figure 2.4, A is the relevant set of documents for the query, $\bar{A}$ is the non-relevant set, B is the set of retrieved documents, and $\bar{B}$ is the set of documents that are not retrieved. The operator ∩ gives the intersection of two sets. For example, A ∩ B is the set of documents that are both relevant and retrieved This table can be used to define recall and precision where recall is the proportion of relevant documents that are retrieved for a query, and precision is the proportion of retrieved documents which are relevant[5].

Normalized Discounted Cumulative Gain

In information retrieval, Normalized Discounted Cumulative Gain (NDCG) is often used to measure effectiveness of a search engine or retrieval system. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities by usefulness (gain) of a document based on its position in a graded relevance scale (ranking) of the documents in a result set.

In the NDCG the cumulative gain (CG) Figure 2.5 is sum of the scores of from each result in a given result set. However, the CG does not take into account the ordering of the search results so a method is needed to ensure higher ranked results should be weighted according to their position in the set of results, penalising highly relevant documents which have a low ranking and non-relevant documents which are high ranking. The discounting process adds in this weighting by summing the score divided by the log of the rank. Figure 2.6 In

$$\mathrm{CG_p} = \sum_{i=1}^{p} rel_i$$

Figure 2.5: Cumulative Gain where rel_i is the graded relevance of the result at position i.

$$\mathrm{DCG_p} = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i)}$$

Figure 2.6: Discounted Cumulative Gain accumulated at a particular rank position p

order to provide a fair method of comparison the DCG should be normalised to account for differences in the difficultly of queries. A normalisation process is required as performance of one query to another is not directly comparable in this form due to potential differences between queries. Some queries will be 'harder' and produce less results, resulting in a lower overall DCG, yet may not necessarily be worse performing than a system with high DCG. This normalisation is achieved by taking the DCG and dividing by the ideal DCG for that result. Once the NDCG is know for query, the results can be averaged allowing for comparison with other systems. See Figure 2.7

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$$

Figure 2.7: Normalized Discounted Cumulative Gain: IDCG$_k$ is the ideal DCG for a given set of queries.

Chapter 3

Implementation

This section focuses on the design choices made throughout the project and how they were implemented. It will cover the underlying structure of the implementation and show the reasoning behind the decisions that were made during the implementation of the application.

3.1 Tools

3.1.1 Test Collection

To measure the effectiveness of an information retrieval system a test collection is required. A standard test collection consists of three elements:

1. A document collection
2. A test suite of information needs expressed as queries
3. A set of relevance judgements, which can be either graded or a binary assessment

The test collection dataset and the test suite of information needs must be of a sufficiently large size to enable calculations of average performance. A large set of at least 50 information needs encoded as test queries are required, as results are generally highly variable over different documents and information needs.

3.1.2 Dataset

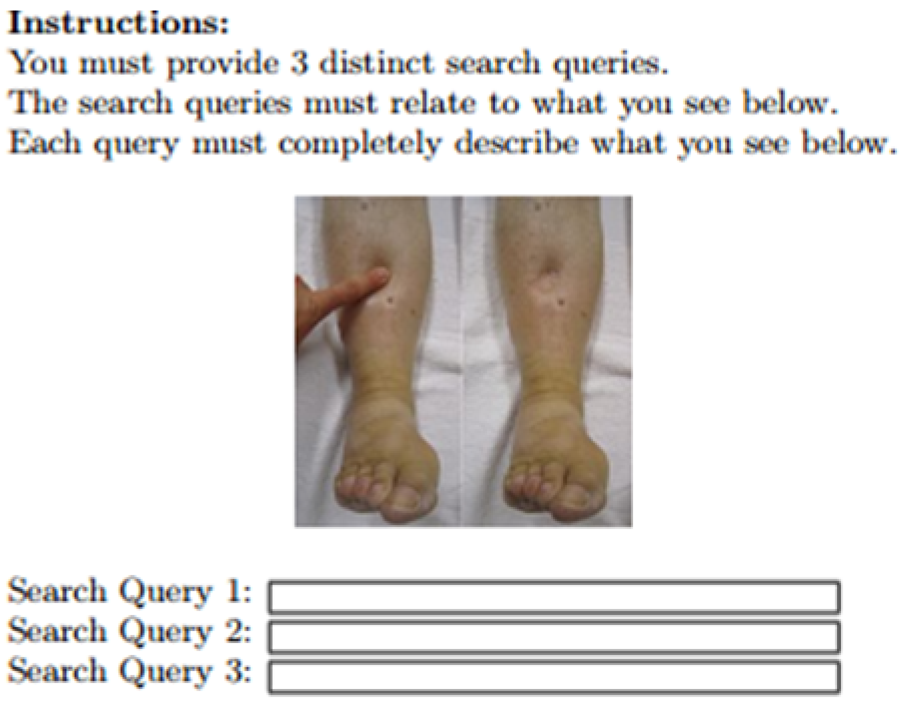
The dataset used for 2015 CLEF eHealth Lab Task 2 is a document corpus consisting of raw HTML documents resulting from the a crawl of approximately one million predominantly medical and health-related websites. The majority of the websites added to the corpus have been certified by the Health on the Net (HON) Foundation as adhering to the HONcode principles see table 3.1. In addition to the documents adhering to HONcode principles, approximately $60-70\%$ of the collection, other commonly used health and medicine websites such as Drugbank, Diagnosia and Trip Answers have also been added to the dataset. The resulting dataset provides a large variety of website which cover a broad range of health topics and are targeted at both the general public and healthcare professionals.

Table 3.1: HONcode principles

Principle 1	Authoritative	Indicate the qualifications of the authors
Principle 2	Complementarity	Information should support, not replace, the doctor-patient relationship
Principle 3	Privacy	Respect the confidentiality of personal data submitted to the site by the visitor
Principle 4	Attribution	Cite the source(s) of published information, date medical and health pages
Principle 5	Justifiability	Site must back up claims relating to benefits and performance
Principle 6	Transparency	Accessible presentation, accurate email contact
Principle 7	Financial disclosure	Identify funding sources
Principle 8	Advertising policy	Clearly distinguish advertising from editorial content

3.1.3 Topics

A set of training queries are provided part of the CLEF eHeath Task2 which attempt to emulate the circumlocutory style of queries made by non-medically trained users when seeking information to understand symptoms or conditions after being presented with symptoms of disease or illness. Constructing the query set proved to be challenging task. Since the average layperson is unable to effectively self-diagnose it is pointless to ask crowd-sourcing participants to label a set of queries with a description of symptoms resulting from a medical condition. Staton et al.[8] found the solution was to create the queries by using crowd sourcing methods. Instead of asking the crowdsourcing participants to generate the labels for the training data, they were asked to solve the inverse problem, i.e. given a label, generate the data. An example would be created by(showing some medical symptoms with a strong visual component. The participants would be shown three diverse pictures that demonstrated the symptom and asked If you were the patient in this picture, what queries would you issue to find out what is wrong with you? See figure 3.1.



Instructions:
You must provide 3 distinct search queries.
The search queries must relate to what you see below.
Each query must completely describe what you see below.

Search Query 1:
Search Query 2:
Search Query 3:

Figure 3.1: The crowdsourcing task for obtaining training data. The symptom shown is edema.

The CLEF organisers adopted this approach to collect a set of 266 possible unique queries, then a sub set of queries were selected by randomly picking one query to represent each condition, referred to as the pivot query. The pivot query was used to manually select two additional queries per condition, one which was deemed to be the most similar (called most) and the query that was least similar (called least) to the pivot query in appearance.

This resulted in the final training set of 67 queries, with 3 queries for each of the 22 conditions, plus an additional condition with only 1 query[7]. See figure **??** for an example and Appendix for full Topic set with narratives.

Listing 3.1: An example query

```
<top>
        <num>clef2015.test.2</num>
        <query>lump with blood spots on nose</query>
</top>
```

3.1.4 Relevance judgements

Relevance

The standard approach to evaluation of information retrieval systems is based on the concept of relevance. The relevance of a document is determined by whether the information need expressed in the query has been met by the content of the document. Each document in the test collection is given a classification as to its relevance to each query. In a binary collection the relevance of a document is simply classed as either 0 for non relevant, or 1 for relevant. In a graded relevance test collection relevance judgements are provided on a three point scale:

0 - Not Relevant

1 - Somewhat Relevant

2 - Highly Relevant

Relevance is assessed based on whether a document meets the information need expressed in the a query. For example, an information need might be: How are pets or animals used in therapy for humans and what are the benefits? This might be translated into a query such as:

```
pet therapy
```

A relevant document must included details of how pet or animal assisted therapy is used, it is not sufficient for a document just to contain the words used in the query.

Qrel files

Query relevance judgement (qrel) files are a vital component of a test collection, as these provide the 'ground truth judgement' of relevance on which to evaluate the retrieval systems performance. A qrel file is used in conjunction with an output file, called a results file, produced by the retrieval system which details the ranked results of the queries.

The results file has a standardised format which is delimited by spaces:

```
query_id    iter    docno    rank    sim    run_id
```

Query_id is the query number (e.g. clef2015.qtest.1 or similar, depending on the evaluation group). The iter constant, 0, which is required but not used. The document number is a string value taken from the filename of the document, which in turn relates to the original url of the website. The Similarity (sim) is a float value. Rank is an integer from 0 to 1000, which is required but ignored by the program. Runid is a string which is used to differentiate between the results files produced by the information retrieval system. Input is assumed to be sorted numerically by qid and the sim is assumed to be higher for the documents retrieved first.

A qrel file is used to determine relevance for each docno in relation to qid. It consists of text tuples very similar to those of the results file:

```
qid      iter      docno      rel
```

Where qid is the query number, iter is the iteration constant which is nearly always zero and not used, docno is the official document number that corresponds to the docno field in the results file and rel is the graded value of the relevance judgement, 0 for not relevant, 1 for partially relevant and 2 for highly relevant.

3.1.5 The TREC_EVAL evaluation tool

The system uses the TREC_EVAL evaluation tool, available from `http://trec.nist.gov/trec_eval/`, to calculate the values of the NDCG and recall precision measures for each run. TREC_EVAL evaluation tool, commonly referred to as trec_eval, is the most widely used evaluation tool in the information retrieval community. Created by the Text REtrieval Conference (TREC) community, which started in 1992 as part of the TIPSTER Text program, TREC was co-sponsored by the National Institute of Standards and Technology (NIST) and U.S. Department of Defence. Tasked to support research within the information retrieval community by providing the necessary infrastructure such as test collections and evaluation tools which allow large-scale evaluation of information retrieval methodologies. [3]

The TREC_EVAL tool is typically used via the command line for evaluating an information retrieval experiment, commonly referred to as a 'run'. Trec_eval will be passed a results file, produced by the system which is being evaluated, and a qrel file. On a Linux based system, the basic format for running the trec_eval program on the command line is:

```
./ trec_eval trec_qrel_file trec_results_file
```

where trec_eval is the executable name for the code, trec_qrel _file is a file contains for each query the set of all documents judged (qrel) and trec_results_file is the the ranked list of documents produced by an information retrieval system and the end of a run. The trec_eval tool also has a large number of Linux style flags, however in the scope of this project only [ndcg_cut] was used.

3.2 Initial challenges

figuring out trec
- pre-dates the web
- differing terminology
- everybodys been using it for 20 years already
python path issues on mint

3.3 Choice of technologies

3.3.1 Python programming language

Huge standard library Generally good quality documentation for standard library Gobs of third-party modules to rival even Java Platform agnostic, and present in virtually every *nix distribution I'm aware of without even needing to install it (because so many system tools are written using it) Very common use of BSD/MIT-style licensing with third-party modules; GPL licensing gives corporate lawyers a big headache Emphasis on code readability (see PEP-8) and DRY principles without sacrificing readability (kind of a middle ground between perl and java, I guess?) Useful for a really broad range of programming tasks from little shell scripts to enterprise web applications to scientific uses; it may not be as good at any of those as a purpose-built programming language but it can do all of them, and do them well (e.g. you don't see web apps written as bash scripts nor do you see linux package managers written in Java) - some

Easy to learn Increases programmer productivity (significantly) Its syntax rules make yours and other's code easy to read It's concepts of modules (and packages) make code easy to maintain It's cross-platform Batteries included (a lot of standard libraries)

It's designed to be extremely easy to learn - it removes much of the boilerplate and complexity, but not the power, of some of the languages in your list. It's a spiritual successor to Turbo Pascal of the 1980's: cheap (free in this case), easy to learn, taught in schools, plenty of beginners' books and learning materials, and yet powerful enough for real-world work.

For me personally the simplest answer is the very broad range of things you can do with it. Someone here called it "the second best language for everything". Its strength isn't that it does one thing fantastically (rails, PHP), it's that it does so many things very well.

3.3.2 Whoosh

The core of the system is the python Whoosh search engine library. Whoosh was created by Matt Chaput as search server for the online documentation of high-end 3D animation software Houdini[4]. Open sourced in 2007, it has since grown in popularity as a result of its ability to easily add search functionality to applications and websites. As Whoosh is pure Python, it does not require compilation, binary packages or a Java Virtual Machine (JVM) allowing it run anywhere Python can, for example all Linux operating systems have Python installed. This offered a major advantage over other search engine libraries such as Lucene, Sphinx, Solr and ElasticSearch in terms of suitability for this project.

Like one of its ancestors, Lucene, Whoosh is not really a search engine, its a programmer library for creating a search engine [1].

Practically no important behavior of Whoosh is hard-coded. Indexing of text, the level of information stored for each term in each field, parsing of search queries, the types of queries allowed, scoring algorithms, etc. are all customizable, replaceable, and extensible.

Introduce whoosh
What is it?
How does it work?

3.3.3 BeatifulSoup

Beautiful Soupis a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work. These instructions illustrate all major features of Beautiful Soup 4, with examples. I show you what the library is good for, how it works, how to use it, how to make it do what you want, and what to do when it violates your expectations. The examples in this documentation should work the same way in Python 2.7 and Python 3.2. You might be looking for the documentation forBeautiful Soup 3. If so, you should know that Beautiful Soup 3 is no longer being developed, and that Beautiful Soup 4 is recommended for all new projects. If you want to learn about the differences between Beautiful Soup 3 and Beautiful Soup 4, seePorting code to BS4. This documentation has been translated into other languages by Beautiful Soup users:

Making the soup To parse a document, pass it into the BeautifulSoup constructor. You can pass in a string or an open filehandle:

from bs4 import BeautifulSoup

soup = BeautifulSoup(open("index.html"))

soup = BeautifulSoup("¡html¿data¡/html¿") First, the document is converted to Unicode, and HTML entities are converted to Unicode characters:

BeautifulSoup("Sacr&eacute; bleu!") ¡html¿¡head¿¡/head¿¡body¿Sacr bleu!¡/body¿¡/html¿ Beautiful Soup then parses the document using the best available parser. It will use an HTML parser unless you specifically tell it to use an XML parser. (See Parsing XML.)

Kinds of objects Beautiful Soup transforms a complex HTML document into a complex tree of Python objects. But youll only ever have to deal with about four kinds of objects: Tag, NavigableString, BeautifulSoup, and Comment.

Tag A Tag object corresponds to an XML or HTML tag in the original document:

soup = BeautifulSoup('¡b class="boldest"¿Extremely bold¡/b¿') tag = soup.b type(tag) # ¡class 'bs4.element.Tag'¿ Tags have a lot of attributes and methods, and Ill cover most of them in Navigating the tree and Searching the tree. For now, the most important features of a tag are its name and attributes.

Name Every tag has a name, accessible as .name:

tag.name # u'b' If you change a tags name, the change will be reflected in any HTML markup generated by Beautiful Soup:

tag.name = "blockquote" tag # ¡blockquote class="boldest"¿Extremely bold¡/blockquote¿ Attributes A tag may have any number of attributes. The tag ¡b class="boldest"¿ has an attribute class whose value is boldest. You can access a tags attributes by treating the tag like a dictionary:

tag['class'] # u'boldest' You can access that dictionary directly as .attrs:

tag.attrs # u'class': u'boldest' You can add, remove, and modify a tags attributes. Again, this is done by treating the tag as a dictionary:

tag['class'] = 'verybold' tag['id'] = 1 tag # ¡blockquote class="verybold" id="1"¿Extremely bold¡/blockquote¿

del tag['class'] del tag['id'] tag # ¡blockquote¿Extremely bold¡/blockquote¿

tag['class'] # KeyError: 'class' print(tag.get('class')) # None Multi-valued attributes

HTML 4 defines a few attributes that can have multiple values. HTML 5 removes a couple of them, but defines a few more. The most common multi-valued attribute is class (that is, a tag can have more than one CSS class). Others include rel, rev, accept-charset, headers, and accesskey. Beautiful Soup presents the value(s) of a multi-valued attribute as a list:

css_soup = BeautifulSoup('¡p class="body strikeout"¿¡/p¿') css_soup.p['class'] # ["body", "strikeout"]

css_soup = BeautifulSoup('¡p class="body"¿¡/p¿') css_soup.p['class'] # ["body"] If an attribute looks like it has more than one value, but its not a multi-valued attribute as defined by any version of the HTML standard, Beautiful Soup will leave the attribute alone:

id_soup = BeautifulSoup('¡p id="my id"¿¡/p¿') id_soup.p['id'] # 'my id' When you turn a tag back into a string, multiple attribute values are consolidated:

rel_soup = BeautifulSoup('¡p¿Back to the ¡a rel="index"¿homepage¡/a¿¡/p¿') rel_soup.a['rel'] # ['index'] rel_soup.a['rel'] = ['index', 'contents'] print(rel_soup.p) # ¡p¿Back to the ¡a rel="index contents"¿homepage¡/a¿¡/p¿ If you parse a document as XML, there are no multi-valued attributes:

xml_soup = BeautifulSoup('¡p class="body strikeout"¿¡/p¿', 'xml') xml_soup.p['class'] # u'body strikeout' NavigableString A string corresponds to a bit of text within a tag. Beautiful Soup uses the NavigableString class to contain these bits of text:

tag.string # u'Extremely bold' type(tag.string) # ¡class 'bs4.element.NavigableString'¿ A NavigableString is just like a Python Unicode string, except that it also supports some of the features described in Navigating the tree and Searching the tree. You can convert a NavigableString to a Unicode string with unicode():

unicode_string = unicode(tag.string) unicode_string # u'Extremely bold' type(unicode_string) # ¡type 'unicode'¿ You cant edit a string in place, but you can replace one string with another, using replace_with():

tag.string.replace_with("No longer bold") tag # ¡blockquote¿No longer bold¡/blockquote¿ NavigableString supports most of the features described in Navigating the tree and Searching the tree, but not all of them. In particular, since a string cant contain anything (the way a tag may contain a string or another tag), strings dont support the .contents or .string attributes, or the find() method.

If you want to use a NavigableString outside of Beautiful Soup, you should call unicode() on it to turn it into a normal Python Unicode string. If you dont, your string will carry around a reference to the entire Beautiful Soup parse tree, even when youre done using Beautiful Soup. This is a big waste of memory.

BeautifulSoup The BeautifulSoup object itself represents the document as a whole. For most purposes, you can treat it as a Tag object. This means it supports most of the methods described in Navigating the tree and Searching the tree.

Since the BeautifulSoup object doesnt correspond to an actual HTML or XML tag, it has no name and no attributes. But sometimes its useful to look at its .name, so its been given the special .name [document]:

soup.name # u'[document]' A bit about html

Beautiful soup;
What is it?
How does it work?

3.3.4 Libextract

What is it
How does it work
Libextract is astatistics-enableddata extraction library that works on HTML and XML documents and written in Python. Originating fromeatiht, the extraction algorithm works by making one simple assumption:data appear as collections of repetitive elements. You can read about the reasoninghere. Overview libextract.api.extract(document, encoding=utf-8, count=5) Given an htmldocument, and optionally theencoding, return a list of nodes likely containing data (5 by default). Installation pip install libextract Usage Due to our simple definition of data, we open up a single interfaceable method. Post-processing is up to you. from requests import get from libextract.api import extract r = get('http://en.wikipedia.org/wiki/Information_extraction') textnodes = list(extract(r.content)) Using lxmls built-in methods for post-processing: ¿¿ print(textnodes[0].text_content()) Information extraction (IE) is the task of automatically extracting structured information... The extraction algo is agnostic to article text as it is with tabular data: height_data = get("http://en.wikipedia.org/wiki/Human_height") tabs = list(extract(height_data.content)) ¿¿ [elem.text_content() for elem in tabs[0].iter('th')] ['Country/Region', 'Average male height', 'Average female height', ...] Dependencies lxml statscounter

json
ON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures. other python libraries

3.4 Matplotlib

is the brainchild of John Hunter (1968-2012), who, along with its many contributors, have put an immeasurable amount of time and effort into producing a piece of software utilized by thousands of scientists worldwide. If matplotlib contributes to a project that leads to a scientific publication, please acknowledge this work by citing the project. You can use this ready-made citation entry.
matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB* or Mathematica), web application servers, and six graphical user interface toolkits.

screenshots

matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory

For simple plotting the pyplot interface provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

3.5 lmxl

lxml is the most feature-rich and easy-to-use library for processing XML and HTML in the Python language. Introduction

The lxml XML toolkit is a Pythonic binding for the C libraries libxml2 and libxslt. It is unique in that it combines the speed and XML feature completeness of these libraries with the simplicity of a native Python API, mostly compatible but superior to the well-known ElementTree API. The latest release works with all CPython versions from 2.6 to 3.5. See the introduction for more information about background and goals of the lxml project. Some common questions are answered in the FAQ. logging lxml is not written in plain Python, because it interfaces with two C libraries: libxml2 and libxslt. Accessing them at the C-level is required for performance reasons. However, to avoid writing plain C-code and caring too much about the details of built-in types and reference counting, lxml is written in Cython, a superset of the Python language that translates to C-code. Chances are that if you know Python, you can write code that Cython accepts. Again, the C-ish style used in the lxml code is just for performance optimisations. If you want to contribute, don't bother with the details, a Python implementation of your contribution is better than none. And keep in mind that lxml's flexible API often favours an implementation of features in pure Python, without bothering with C-code at all. For example, the lxml.html package is written entirely in Python. An Element is the main container object for the ElementTree API. Most of the XML tree functionality is accessed through this class. Elements are easily created through the Element factory: ¿¿¿ root = etree.Element("root") The XML tag name of elements is accessed through the tag property: ¿¿¿ print(root.tag) root Elements are organised in an XML tree structure. To create child elements and add them to a parent element, you can use the append() method: ¿¿¿ root.append(etree.Element("child1")) However, this is so common that there is a shorter and much more efficient way to do this: the SubElement factory. It accepts the same arguments as the Element factory, but additionally requires the parent as first argument: ¿¿¿ child2 = etree.SubElement(root, "child2") ¿¿¿ child3 = etree.SubElement(root, "child3") To see that this is really XML, you can serialise the tree you have created:

3.6 Overview of System

The system was designed to facilitate performing a number of runs, with each run using different configurations of system components. To achieve this a very simple pipeline architecture, as shown in figure 3.2, was used. Each of the components in the system, along with inputs and outputs are configured by reading in a configuration file which contains the settings for each component.

To generate the runs, the system is given a command line argument containing the path to a folder containing one or more configuration files, e.g python run_sifh.py "/home/user/config_folder/" If the path is incorrect or contains no configuration files a warning message displayed detailing the correct usage. When running the system will display a counter indicating the number of documents waiting to be indexed to the user, as the indexing process may take several hours, depending on the size of the document corpus.

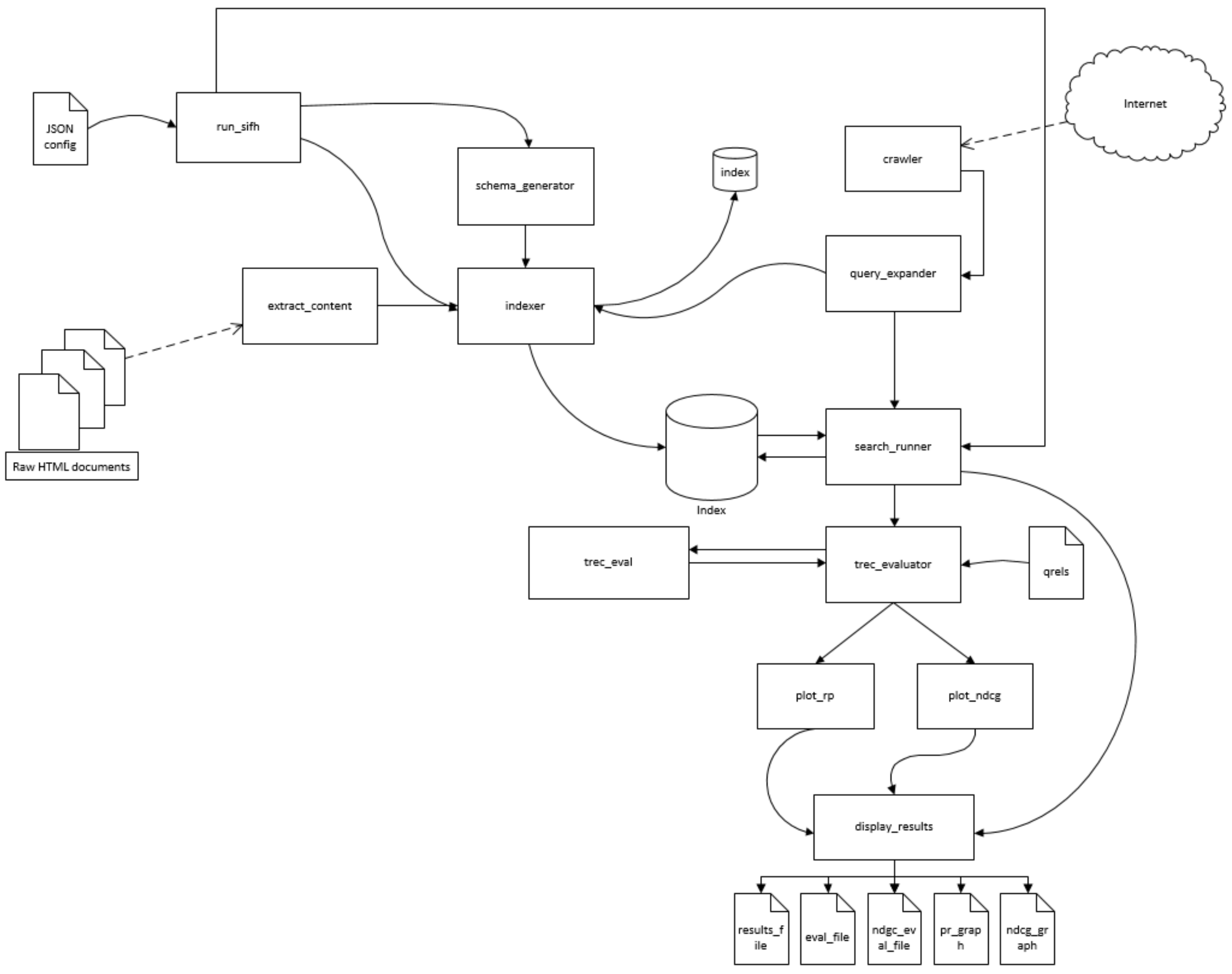


Figure 3.2: An overview of the project components.

3.7 Reading the configuration files

Each configuration file is written in JSON format (JavaScript Object Notation) which is an open-standard format that uses human-readable text to transmit data objects consisting of attributevalue pairs. The JSON objects in the configuration file are transposed to python dictionaries. Each component of the system has a dictionary to which contains the values to be assigned upon initialisation. The system will iterate through each configuration file in the folder in turn, completing each run before starting on the next. For each run the system will produce a results folder containing the a file detailing the individual query results, a trec_eval file which details the trec performance measures of that run, a graph of the interpolated recall precision curve and a graph of the NDCG The system will also generate graphs of each run plotted against each other for the interpolated recall precision curve and the NDCG.

3.8 Building the index

Files created ¡revision_number¿.toc The master file containing information about the index and its segments. The index directory will contain a set of files for each segment. A segment is like a mini-index when you add documents to the index, whoosh creates a new segment and then searches the old segment(s) and the new segment to avoid having to do a big merge every time you add a document. When you get enough small segments whoosh will merge them into larger segments or a single segment.

¡segment_number¿.dci Contains per-document information (e.g. field lengths). This will grow linearly with the number of documents. ¡segment_number¿.dcz Contains the stored fields for each document. ¡segment_number¿.tiz Contains per-term information. The size of file will vary based on the number of unique terms. ¡segment_number¿.pst Contains per-term postings. The size of this file depends on the size of the collection and the formats used for each field (e.g. storing term positions takes more space than storing frequency only). ¡segment_number¿.fvz contains term vectors (forward indexes) for each document. This file is only created if at least one field in the schema stores term vectors. The size will vary based on the number of documents, field length, the formats used for each vector (e.g. storing term positions takes more space than storing frequency only), etc.
whoosh api modules indexer
-schema
Designing a schema About schemas and fields The schema specifies the fields of documents in an index.

Each document can have multiple fields, such as title, content, url, date, etc.

Some fields can be indexed, and some fields can be stored with the document so the field value is available in search results. Some fields will be both indexed and stored.

The schema is the set of all possible fields in a document. Each individual document might only use a subset of the available fields in the schema.

For example, a simple schema for indexing emails might have fields like from_addr, to_addr, subject, body, and attachments, where the attachments field lists the names of attachments to the email. For emails without attachments, you would omit the attachments field.

Built-in field types Whoosh provides some useful predefined field types:

whoosh.fields.TEXT This type is for body text. It indexes (and optionally stores) the text and stores term positions to allow phrase searching.

TEXT fields use StandardAnalyzer by default. To specify a different analyzer, use the analyzer keyword argument to the constructor, e.g. TEXT(analyzer=analysis.StemmingAnalyzer()). See About analyzers.

By default, TEXT fields store position information for each indexed term, to allow you to search for phrases. If you dont need to be able to search for phrases in a text field, you can turn off storing term positions to save space. Use TEXT(phrase=False).

By default, TEXT fields are not stored. Usually you will not want to store the body text in the search index. Usually you have the indexed documents themselves available to read or link to based on the search results, so you dont need to store their text in the search index. However, in some circumstances it can be useful (see How to create highlighted search result excerpts). Use TEXT(stored=True) to specify that the text should be stored in the index.

whoosh.fields.KEYWORD This field type is designed for space- or comma-separated keywords. This type is indexed and searchable (and optionally stored). To save space, it does not support phrase searching.

To store the value of the field in the index, use stored=True in the constructor. To automatically lowercase the keywords before indexing them, use lowercase=True.

By default, the keywords are space separated. To separate the keywords by commas instead (to allow keywords containing spaces), use commas=True.

If your users will use the keyword field for searching, use scorable=True.

whoosh.fields.ID The ID field type simply indexes (and optionally stores) the entire value of the field as a single unit (that is, it doesnt break it up into individual terms). This type of field does not store frequency information, so its quite compact, but not very useful for scoring.

Use ID for fields like url or path (the URL or file path of a document), date, category fields where the value must be treated as a whole, and each document only has one value for the field.

By default, ID fields are not stored. Use ID(stored=True) to specify that the value of the field should be stored with the document for use in the search results. For example, you would want to store the value of a url field so you could provide links to the original in your search results.

whoosh.fields.STORED This field is stored with the document, but not indexed and not searchable. This is useful for document information you want to display to the user in the search results, but dont need to be able to search for. Advanced schema setup Field boosts You can specify a field boost for a field. This is a multiplier applied to the score of any term found in the field. For example, to make terms found in the title field score twice as high as terms in the body field:

schema = Schema(title=TEXT(field_boost=2.0), body=TEXT) Field types The predefined field types listed above are subclasses of fields.FieldType. FieldType is a pretty simple class. Its attributes contain information that define the behavior of a field.

Attribute Type Description format fields.Format Defines what kind of information a field records about each term, and how the information is stored on disk. vector fields.Format Optional: if defined, the format in which to store per-document forward-index information for this field. scorable bool If True, the length of (number of terms in) the field in each document is stored in the index. Slightly misnamed, since field lengths are not required for all scoring. However, field lengths are required to get proper results from BM25F. stored bool If True, the value of this field is stored in the index. unique bool If True, the value of this field may be used to replace documents with the same value when the user calls document_update() on an IndexWriter. The constructors for most of the predefined field types have parameters that let you customize these parts. For example:

Most of the predefined field types take a stored keyword argument that sets FieldType.stored. The TEXT() constructor takes an analyzer keyword argument that is passed on to the format object.

You dont have to fill in a value for every field. Whoosh doesnt care if you leave out a field from a document.

Indexed fields must be passed a unicode value. Fields that are stored but not indexed (i.e. the STORED field type) can be passed any pickle-able object.

Whoosh will happily allow you to add documents with identical values, which can be useful or annoying depending on what youre using the library for: Schema desgin
Whats a schema
-analysis
searching
-parsing -fields
-scoring
–Weighting models
—bm25f
—Tf-idf
—Pl2
Once youve created an index and added documents to it, you can search for those documents.

The Searcher object To get a whoosh.searching.Searcher object, call searcher() on your Index object:

searcher = myindex.searcher() Youll usually want to open the searcher using a with statement so the searcher is automatically closed when youre done with it (searcher objects represent a number of open files, so if you dont explicitly close them and the system is slow to collect them, you can run out of file handles):

with ix.searcher() as searcher: ... This is of course equivalent to:

try: searcher = ix.searcher() ... finally: searcher.close() The Searcher object is the main high-level interface for reading the index. It has lots of useful methods for getting information about the index, such as lexicon(fieldname).

¿¿¿ list(searcher.lexicon("content")) [u"document", u"index", u"whoosh"] However, the most important method on the Searcher object is search(), which takes a whoosh.query.Query object and returns a Results object:

from whoosh.qparser import QueryParser

qp = QueryParser("content", schema=myindex.schema) q = qp.parse(u"hello world")

with myindex.searcher() as s: results = s.search(q) By default the results contains at most the first 10 matching documents. To get more results, use the limit keyword:

results = s.search(q, limit=20) If you want all results, use limit=None. However, setting the limit whenever possible makes searches faster because Whoosh doesnt need to examine and score every document.

Since displaying a page of results at a time is a common pattern, the search_page method lets you conveniently retrieve only the results on a given page:

results = s.search_page(q, 1) The default page length is 10 hits. You can use the pagelen keyword argument to set a different page length:

results = s.search_page(q, 5, pagelen=20) Results object The Results object acts like a list of the matched documents. You can use it to access the stored fields of each hit document, to display to the user.

¿¿¿ # Show the best hit's stored fields ¿¿¿ results[0] "title": u"Hello World in Python", "path": u"/a/b/c" ¿¿¿ results[0:2] ["title": u"Hello World in Python", "path": u"/a/b/c", "title": u"Foo", "path": u"/bar"] By default, Searcher.search(myquery) limits the number of hits to 20, So the number of scored hits in the Results object may be less than the number of matching documents in the index.

¿¿¿ # How many documents in the entire index would have matched? ¿¿¿ len(results) 27 ¿¿¿ # How many scored and sorted documents in this Results object? ¿¿¿ # This will often be less than len() if the number of hits was limited ¿¿¿ # (the default). ¿¿¿ results.scored_length() 10 Calling len(Results) runs a fast (unscored) version of the query again to figure out the total number of matching documents. This is usually very fast but for large indexes it can cause a noticeable delay. If you want to avoid this delay on very large indexes, you can use the has_exact_length(), estimated_length(), and estimated_min_length() methods to estimate the number of matching documents without calling len():

found = results.scored_length() if results.has_exact_length(): print("Scored", found, "of exactly", len(results), "documents") else: low = results.estimated_min_length() high = results.estimated_length()

print("Scored", found, "of between", low, "and", high, "documents") Scoring and sorting Scoring Normally the list of result documents is sorted by score. The whoosh.scoring module contains implementations of various scoring algorithms. The default is BM25F.

You can set the scoring object to use when you create the searcher using the weighting keyword argument:

from whoosh import scoring

with myindex.searcher(weighting=scoring.TF_IDF()) as s: ... A weighting model is a WeightingModel subclass with a scorer() method that produces a scorer instance. This instance has a method that takes the current matcher and returns a floating point score.

Sorting Parsing user queries Overview The job of a query parser is to convert a query string submitted by a user into query objects (objects from the whoosh.query module).

For example, the user query:

rendering shading might be parsed into query objects like this:

And([Term("content", u"rendering"), Term("content", u"shading")]) Whoosh includes a powerful, modular parser for user queries in the whoosh.qparser module. The default parser implements a query language similar to the one that ships with Lucene. However, by changing plugins or using functions such as whoosh.qparser.MultifieldParser(), whoosh.qparser.SimpleParser() or whoosh.qparser.DisMaxParser(), you can change how the parser works, get a simpler parser or change the query language syntax.

(In previous versions of Whoosh, the query parser was based on pyparsing. The new hand-written parser is less brittle and more flexible.)

Note

Remember that you can directly create query objects programmatically using the objects in the whoosh.query module. If you are not processing actual user queries, this is preferable to building a query string just to parse it. Using the default parser To create a whoosh.qparser.QueryParser object, pass it the name of the default field to search and the schema of the index youll be searching.

from whoosh.qparser import QueryParser

parser = QueryParser("content", schema=myindex.schema) Tip

You can instantiate a QueryParser object without specifying a schema, however the parser will not process the text of the user query. This is useful for debugging, when you want to see how QueryParser will build a query, but dont want to make up a schema just for testing. Once you have a QueryParser object, you can call parse() on it to parse a query string into a query object:

¿¿¿ parser.parse(u"alpha OR beta gamma") And([Or([Term('content', u'alpha'), Term('content', u'beta')]), Term('content', u'gamma')]) See the query language reference for the features and syntax of the default parsers query language.

Common customizations Searching for any terms instead of all terms by default If the user doesnt explicitly specify AND or OR clauses:

physically based rendering ...by default, the parser treats the words as if they were connected by AND, meaning all the terms must be present for a document to match:

physically AND based AND rendering To change the parser to use OR instead, so that any of the terms may be present for a document to match, i.e.:

physically OR based OR rendering ...configure the QueryParser using the group keyword argument like this:

from whoosh import qparser

parser = qparser.QueryParser(fieldname, schema=myindex.schema, group=qparser.OrGroup) The Or query lets you specify that documents that contain more of the query terms score higher. For example, if the user searches for foo bar, a document with four occurances of foo would normally outscore a document that contained one occurance each of foo and bar. However, users usually expect documents that contain more of the words they searched for to score higher. To configure the parser to produce Or groups with this behavior, use the factory() class method of OrGroup:

og = qparser.OrGroup.factory(0.9) parser = qparser.QueryParser(fieldname, schema, group=og) where the argument to factory() is a scaling factor on the bonus (between 0 and 1).

Letting the user search multiple fields by default The default QueryParser configuration takes terms without explicit fields and assigns them to the default field you specified when you created the object, so for example if you created the object with:

parser = QueryParser(”content”, schema=myschema) And the user entered the query:

three blind mice The parser would treat it as:

content:three content:blind content:mice However, you might want to let the user search multiple fields by default. For example, you might want unfielded terms to search both the title and content fields.

In that case, you can use a whoosh.qparser.MultifieldParser. This is just like the normal QueryParser, but instead of a default field name string, it takes a sequence of field names:

from whoosh.qparser import MultifieldParser

mparser = MultifieldParser([”title”, ”content”], schema=myschema) When this MultifieldParser instance parses three blind mice, it treats it as:

(title:three OR content:three) (title:blind OR content:blind) (title:mice OR content:mice) Choosing which fields to index
Extract the content from html

3.9 Content extraction from HTML

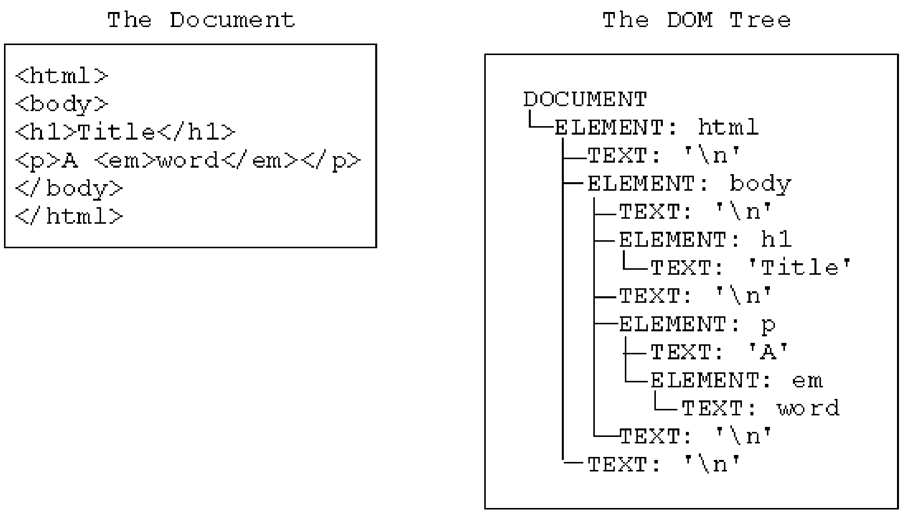
HTML and the DOM

Out[3]: Most modern browsers have aparserthat reads in the HTMLdocument, parses it into a DOM (Document Object Model) structure, and then renders theDOMstructure. Much like HTTP, the DOM is anagreed-upon standard. The DOM ismuch more than what I’ve described, but for our purposes, what is most important to understand is that the text is only one part of an HTML element, and we need to select it explicitly.

3.10 Token analysis

3.11 Problems Encountered

Badly formatted html
Unicode and python



```
The Document

<html>
<body>
<h1>Title</h1>
<p>A <em>word</em></p>
</body>
</html>
```

```
The DOM Tree

DOCUMENT
└─ELEMENT: html
  ├─TEXT: '\n'
  ├─ELEMENT: body
  │ ├─TEXT: '\n'
  │ ├─ELEMENT: h1
  │ │ └─TEXT: 'Title'
  │ ├─TEXT: '\n'
  │ ├─ELEMENT: p
  │ │ ├─TEXT: 'A'
  │ │ └─ELEMENT: em
  │ │   └─TEXT: word
  │ └─TEXT: '\n'
  └─TEXT: '\n'
```

Figure 3.3: The crowdsourcing task for obtaining training data. The symptom shown is edema.

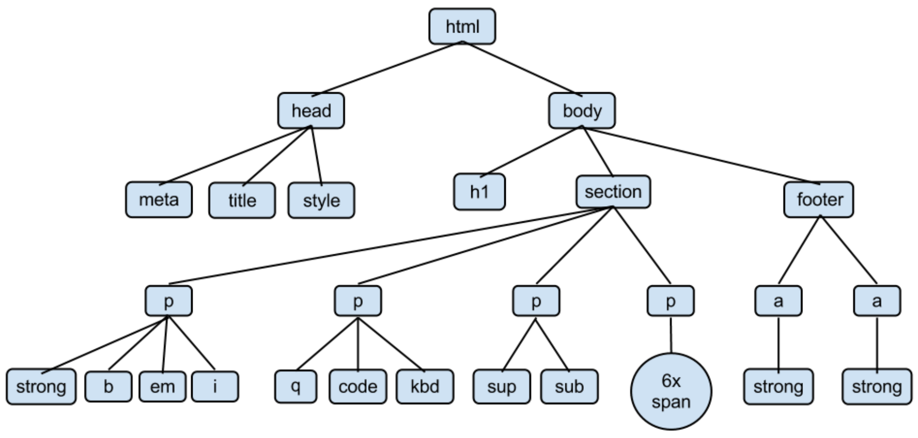


Figure 3.4: The crowdsourcing task for obtaining training data. The symptom shown is edema.

airlocking

slow performance at scale
- threading not worth effort
- whoosh or python?

3.12 running the system

Overview
A pipeline

3.13 inputs

Reading a json config file
Config index
Config searcher
Config result output

3.14 searching

Searching the index
Parsing the queries
Query expansion //TODO

3.15 outputs

Formatting for trec

Graphing the eval files
Rp graph
Ndcg graph //TODO

Chapter 4

Experimentation and Results

4.1 Baseline Run

a. Comparing whoosh inbuilt weighting models

    i. (graphs etc)

b. Query expansion?
c. Field boosts?
d. Entity extraction?
e. Readability modules?

Chapter 5

Conclusion

1. Summary of learning /reflection
2. Conclusion

5.1 Future work

a. Front end Dragon drop style interface?

Appendices

Full topic set with narratives

Listing 1: Full topic set with narratives

```
<topics>
        <top>
                <num>clef2015.test.1</num>
                <query>many red marks on legs after traveling from us</query>
                <disease>rocky mountain spotted fever (rmsf)</disease>
                <type>most</disease>
                <query_index>14</query_index>
        </top>
        <top>
                <num>clef2015.test.2</num>
                <query>lump with blood spots on nose</query>
                <disease>basal cell carcinoma</disease>
                <type>pivot</disease>
                <query_index>17</query_index>
        </top>
        <top>
                <num>clef2015.test.3</num>
                <query>dry red and scaly feet in children</query>
                <disease>ringworm</disease>
                <type>pivot</disease>
                <query_index>5</query_index>
        </top>
        <top>
                <num>clef2015.test.4</num>
                <query>itchy lumps skin</query>
                <disease>sarcoptes scabiei, a.k.a. scabies</disease>
                <type>pivot</disease>
                <query_index>10</query_index>
        </top>
        <top>
                <num>clef2015.test.5</num>
                <query>whistling noise and cough during sleeping + children</query>
                <disease>croup stridor barking cough</disease>
                <type>least</disease>
                <query_index>1</query_index>
        </top>
        <top>
                <num>clef2015.test.6</num>
                <query>child make hissing sound when breathing</query>
                <disease>asthma wheezing</disease>
                <type>most</disease>
                <query_index>18</query_index>
        </top>
        <top>
                <num>clef2015.test.7</num>
                <query>rosacea symptoms</query>
                <disease>basal cell carcinoma</disease>
                <type>least</disease>
                <query_index>17</query_index>
        </top>
        <top>
                <num>clef2015.test.8</num>
                <query>cloudy cornea and vision problem</query>
                <disease>catarat</disease>
                <type>pivot</disease>
                <query_index>6</query_index>
        </top>
        <top>
                <num>clef2015.test.9</num>
                <query>red itchy eyes</query>
                <disease>conjunctivitis</disease>
                <type>least</disease>
                <query_index>20</query_index>
        </top>
        <top>
                <num>clef2015.test.10</num>
                <query>crater type bite mark</query>
                <disease>skin cancer: basal cell carcinoma, squamous cell carcinoma,
                   melanoma</disease>
                <type>pivot</disease>
                <query_index>13</query_index>
        </top>
        <top>
                <num>clef2015.test.11</num>
                <query>white patchiness in mouth</query>
                <disease>strep throat</disease>
                <type>most</disease>
                <query_index>3</query_index>
        </top>
        <top>
                <num>clef2015.test.12</num>
                <query>baby has dry cough and has problem to swallow saliva</query>
                <disease>whooping cough (pertussis)</disease>
                <type>least</disease>
                <query_index>15</query_index>
        </top>
        <top>
                <num>clef2015.test.13</num>
                <query>cloudy pupil affecting vision</query>
                <disease>catarat</disease>
                <type>most</disease>
                <query_index>6</query_index>
        </top>
        <top>
                <num>clef2015.test.14</num>
                <query>infant difficulty breathing</query>
                <disease>croup stridor barking cough</disease>
                <type>pivot</disease>
                <query_index>1</query_index>
        </top>
        <top>
                <num>clef2015.test.15</num>
                <query>asthma attack</query>
                <disease>asthma wheezing</disease>
                <type>least</disease>
                <query_index>18</query_index>
        </top>
        <top>
                <num>clef2015.test.16</num>
                <query>red patchy bruising over legs</query>
                <disease>polyarteritis nodosa</disease>
                <type>most</disease>
                <query_index>12</query_index>
        </top>
        <top>
                <num>clef2015.test.17</num>
                <query>scaly rash</query>
                <disease>squamous cell carcinoma</disease>
                <type>least</disease>
                <query_index>4</query_index>
        </top>
        <top>
                <num>clef2015.test.18</num>
                <query>poor gait and balance with shaking</query>
                <disease>parkinson’s disease</disease>
                <type>pivot</disease>
                <query_index>8</query_index>
        </top>
        <top>
                <num>clef2015.test.19</num>
                <query>bruised thumb nail</query>
                <disease>onycholysis</disease>
                <type>most</disease>
                <query_index>19</query_index>
        </top>
        <top>
        <num>clef2015.test.20</num>
                <query>movement difficulty with involuntary hand trembling</query>
                <disease>parkinson’s disease</disease>
                <type>most</disease>
                <query_index>8</query_index>
        </top>
        <top>
                <num>clef2015.test.21</num>
                <query>common itchy skin rashes</query>
                <disease>sarcoptes scabiei, a.k.a. scabies</disease>
                <type>least</disease>
                <query_index>10</query_index>
        </top>
        <top>
                <num>clef2015.test.22</num>
                <query>yellow thick eye leakage</query>
                <disease>conjunctivitis</disease>
                <type>pivot</disease>
                <query_index>20</query_index>
        </top>
        <top>
                <num>clef2015.test.23</num>
                <query>red bloodshot eyes</query>
                <disease>non-ulcerative sterile keratitis</disease>
                <type>most</disease>
                <query_index>22</query_index>
        </top>
        <top>
                <num>clef2015.test.24</num>
                <query>yellow gunk coming from one eye itchy</query>
                <disease>conjunctivitis</disease>
                <type>most</disease>
                <query_index>20</query_index>
        </top>
        <top>
                <num>clef2015.test.25</num>
                <query>red rash baby face</query>
                <disease>port wine stain</disease>
                <type>most</disease>
                <query_index>9</query_index>
        </top>
        <top>
                <num>clef2015.test.26</num>
                <query>raised red lumps skin</query>
                <disease>hemangioma</disease>
                <type>least</disease>
                <query_index>2</query_index>
        </top>
        <top>
                <num>clef2015.test.27</num>
                <query>return from overseas with mean spots on legs</query>
                <disease>rocky mountain spotted fever (rmsf)</disease>
                <type>pivot</disease>
                <query_index>14</query_index>
        </top>
        <top>
                <num>clef2015.test.28</num>
                <query>baby skin fungal</query>
                <disease>port wine stain</disease>
                <type>least</disease>
                <query_index>9</query_index>
        </top>
        <top>
                <num>clef2015.test.29</num>
                <query>red patch on skin with blister and dry pus</query>
                <disease>squamous cell carcinoma</disease>
                <type>pivot</disease>
                <query_index>4</query_index>
        </top>
        <top>
                <num>clef2015.test.30</num>
                <query>weird sounds when breathing</query>
                <disease>asthma wheezing</disease>
                <type>pivot</disease>
                <query_index>18</query_index>
        </top>
        <top>
                <num>clef2015.test.31</num>
                <query>toddler having squeaky breath</query>
                <disease>bronchiolitis (caused by rsv)</disease>
                <type>most</disease>
                <query_index>11</query_index>
        </top>
        <top>
                <num>clef2015.test.32</num>
                <query>red irritated skin on baby’s face</query>
                <disease>port wine stain</disease>
                <type>pivot</disease>
                <query_index>9</query_index>
        </top>
        <top>
                <num>clef2015.test.33</num>
                <query>white infection in pharynx</query>
                <disease>strep throat</disease>
                <type>least</disease>
                <query_index>3</query_index>
        </top>
        <top>
                <num>clef2015.test.34</num>
                <query>cavity problem</query>
                <disease>caries</disease>
                <type>pivot</disease>
                <query_index>7</query_index>
        </top>
        <top>
                <num>clef2015.test.35</num>
                <query>lot of irritation with contact lenses</query>
                <disease>non-ulcerative sterile keratitis</disease>
                <type>least</disease>
                <query_index>22</query_index>
        </top>
        <top>
                <num>clef2015.test.36</num>
                <query>eye are shaking</query>
                <disease>nystagmus</disease>
                <type>least</disease>
                <query_index>21</query_index>
        </top>
        <top>
                <num>clef2015.test.37</num>
                <query>scaly skin</query>
                <disease>nummular eczema</disease>
                <type>pivot</disease>
                <query_index>16</query_index>
        </top>
        <top>
                <num>clef2015.test.38</num>
                <query>scaly red itchy feet in children</query>
                <disease>ringworm</disease>
                <type>most</disease>
                <query_index>5</query_index>
        </top>
        <top>
                <num>clef2015.test.39</num>
                <query>skin eczema with blister and white pus</query>
                <disease>squamous cell carcinoma</disease>
                <type>most</disease>
                <query_index>4</query_index>
        </top>
        <top>
                <num>clef2015.test.40</num>
                <query>baby red blotch on face</query>
                <disease>hemangioma</disease>
                <type>most</disease>
                <query_index>2</query_index>
        </top>
        <top>
                <num>clef2015.test.41</num>
                <query>eye iris large</query>
                <disease>catarat</disease>
                <type>least</disease>
                <query_index>6</query_index>
        </top>
        <top>
                <num>clef2015.test.42</num>
                <query>patchy bleeding under skin</query>
                <disease>polyarteritis nodosa</disease>
                <type>pivot</disease>
                <query_index>12</query_index>
        </top>
        <top>
                <num>clef2015.test.43</num>
                <query>itchy raised bumps skin</query>
                <disease>sarcoptes scabiei, a.k.a. scabies</disease>
                <type>most</disease>
                <query_index>10</query_index>
        </top>
        <top>
                <num>clef2015.test.44</num>
                <query>nail getting dark</query>
                <disease>onycholysis</disease>
                <type>least</disease>
                <query_index>19</query_index>
        </top>
        <top>
                <num>clef2015.test.45</num>
                <query>dry feel with irritation</query>
                <disease>ringworm</disease>
                <type>least</disease>
                <query_index>5</query_index>
        </top>
        <top>
                <num>clef2015.test.46</num>
                <query>baby cough</query>
                <disease>whooping cough (pertussis)</disease>
                <type>most</disease>
                <query_index>15</query_index>
        </top>
        <top>
                <num>clef2015.test.47</num>
                <query>cavities</query>
                <disease>caries</disease>
                <type>most</disease>
                <query_index>7</query_index>
        </top>
        <top>
                <num>clef2015.test.48</num>
                <query>cannot stop moving my eyes medical condition</query>
                <disease>nystagmus</disease>
                <type>most</disease>
                <query_index>21</query_index>
        </top>
        <top>
                <num>clef2015.test.49</num>
                <query>baby always breathing with mouth closed</query>
                <disease>bronchiolitis (caused by rsv)</disease>
                <type>least</disease>
                <query_index>11</query_index>
        </top>
        <top>
                <num>clef2015.test.50</num>
                <query>red spot baby face</query>
                <disease>hemangioma</disease>
                <type>pivot</disease>
                <query_index>2</query_index>
        </top>
        <top>
                <num>clef2015.test.51</num>
                <query>eyes red blood vessels</query>
                <disease>non-ulcerative sterile keratitis</disease>
                <type>pivot</disease>
                <query_index>22</query_index>
        </top>
        <top>
                <num>clef2015.test.52</num>
                <query>white spotting on gums</query>
                <disease>strep throat</disease>
                <type>pivot</disease>
                <query_index>3</query_index>
        </top>
        <top>
                <num>clef2015.test.53</num>
                <query>swollen legs</query>
                <disease>polyarteritis nodosa</disease>
                <type>least</disease>
                <query_index>12</query_index>
        </top>
        <top>
                <num>clef2015.test.54</num>
                <query>red lumps on nose</query>
                <disease>basal cell carcinoma</disease>
                <type>most</disease>
                <query_index>17</query_index>
        </top>
        <top>
                <num>clef2015.test.55</num>
                <query>crate type mark in skin</query>
                <disease>skin cancer: basal cell carcinoma, squamous cell carcinoma,
                   melanoma</disease>
                <type>most</disease>
                <query_index>13</query_index>
        </top>
        <top>
                <num>clef2015.test.56</num>
                <query>slouching when walking</query>
                <disease>parkinson's disease</disease>
                <type>least</disease>
                <query_index>8</query_index>
        </top>
        <top>
                <num>clef2015.test.57</num>
                <query>infant labored breathing and tight wheezing cough</query>
                <disease>croup stridor barking cough</disease>
                <type>most</disease>
                <query_index>1</query_index>
        </top>
        <top>
                <num>clef2015.test.58</num>
                <query>39 degree and chicken pox</query>
                <disease>rocky mountain spotted fever (rmsf)</disease>
                <type>least</disease>
                <query_index>14</query_index>
        </top>
        <top>
                <num>clef2015.test.59</num>
                <query>heavy and squeaky breath</query>
                <disease>bronchiolitis (caused by rsv)</disease>
                <type>pivot</disease>
                <query_index>11</query_index>
        </top>
        <top>
                <num>clef2015.test.60</num>
                <query>baby white dot in iris</query>
                <disease>bilateral cataracts in an infant due to congenital rubella
                   syndrome</disease>
                <type>pivot</disease>
                <query_index>23</query_index>
        </top>
        <top>
                <num>clef2015.test.61</num>
                <query>fingernail bruises</query>
                <disease>onycholysis</disease>
                <type>pivot</disease>
                <query_index>19</query_index>
        </top>
        <top>
                <num>clef2015.test.62</num>
                <query>ring womb below wrinkled eyelid</query>
                <disease>skin cancer: basal cell carcinoma, squamous cell carcinoma,
                   melanoma</disease>
                <type>least</disease>
                <query_index>13</query_index>
        </top>
```

```
        <top>
                <num>clef2015.test.63</num>
                <query>crusty skin patches</query>
                <disease>nummular eczema</disease>
                <type>most</disease>
                <query_index>16</query_index>
        </top>
        <top>
                <num>clef2015.test.64</num>
                <query>involuntary rapid left-right eye motion</query>
                <disease>nystagmus</disease>
                <type>pivot</disease>
                <query_index>21</query_index>
        </top>
        <top>
                <num>clef2015.test.65</num>
                <query>weird brown patches on skin</query>
                <disease>nummular eczema</disease>
                <type>least</disease>
                <query_index>16</query_index>
        </top>
        <top>
                <num>clef2015.test.66</num>
                <query>treatment of coughs in babies</query>
                <disease>whooping cough (pertussis)</disease>
                <type>pivot</disease>
                <query_index>15</query_index>
        </top>
        <top>
                <num>clef2015.test.67</num>
                <query>black tooth</query>
                <disease>caries</disease>
                <type>least</disease>
                <query_index>7</query_index>
        </top>
</topics>
```

Bibliography

[1] Task 2: User-centred health information retrieval.

[2] The CLEF Initiative. http://www.clef-initiative.eu/.

[3] Trechome.

[4] Whoosh 2.7.4 documentation.

[5] *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[6] S Fox.

[7] Joao Palotti, Guido Zuccon, Lorraine Goeuriot, Liadh Kelly, Allan Hanbury, Gareth JF Jones, Mihai Lupu, and Pavel Pecina. Clef ehealth evaluation lab 2015, task 2: Retrieving information about medical symptoms.

[8] Isabelle Stanton, Samuel Ieong, and Nina Mishra. Circumlocution in diagnostic medical queries. In *Proceedings of the 37th International ACM SIGIR Conference on Research &#38; Development in Information Retrieval*, SIGIR '14, pages 133–142, New York, NY, USA, 2014. ACM.

[9] Horvitz White, R.W.

[10] Guido Zuccon, Bevan Koopman, and Joao Palotti. Diagnose this if you can: On the effectiveness of search engines in finding medical self-diagnosis information. In *37th European Conference on Information Retrieval (ECIR 2015)*, Vienna University of Technology, Gusshausstrasse, Vienna, March 2015. Springer.