Development of a Search Engine and Applications in IR

Robert Pinsler
WKWSCI, NTU
N1509281G@e.ntu.edu.sg

Yiding Liu
SCE, NTU
LIUY0130@e.ntu.edu.sg

Yitong Guan
SCE, NTU
GUAN0049@e.ntu.edu.sg

Jenn Bing Ong
SCE, NTU
ONGJ0063@e.ntu.edu.sg

ABSTRACT

1. INTRODUCTION

2. SYSTEM OVERVIEW

The search engine is able to index and search publication records listed in the dblp computer science bibliography (DBLP) [?]. It is built on top of PyLucene, a Python extension of the open-source software library Lucene[1] that provides text indexing and search capabilities.

Publication records from DBLP are provided in a single XML file. The dataset comprises of various kinds of record types, such as articles published in a journal or magazine (*article*), papers published in a conference or workshop proceedings (*inproceedings*), books, and PhD theses. From those, only documents that are classified as *article* or *inproceedings* are considered.

3. INDEXING

Indexing the DBLP dataset is a multi-step process. First, each record has to be extracted from the XML file before it can be further processed. Additional preprocessing is applied to match the expected input format. Finally, the records are processed and indexed through the PyLucene API. Listing 1 gives an overview of the Indexer class that provides above functionality. It takes as input the file to be indexed, the desired index storage location and an analyzer instance that we will introduce later. By calling its own index function, it triggers the indexing process (Listing 1, line 12). In the following, we explain each step in more detail.

```
1  TAGS = ('article', 'inproceedings')
2  VERSION = Version.LUCENE_CURRENT
3  CREATE = IndexWriterConfig.OpenMode.CREATE
4
5  class Indexer():
6    def __init__(self,data_dir,store_dir,analyzer):
7      store = SimpleFSDirectory(File(store_dir))
8      config = IndexWriterConfig(VERSION,analyzer)
9      config.setOpenMode(CREATE)
10     self.writer = IndexWriter(store,config)
11     self.htmlparser = HTMLParser()
12     self.index(data_dir)
13     self.writer.close()
14
15   def index(self,data_dir):
16     context = etree.iterparse(data_dir,tag=TAGS,
17       events=('end',),dtd_validation=True)
18     for event,elem in context:
19       self.index_document(elem)
20
21     def index_document(self,elem):
22       # indexes extracted element
```

Listing 1: Indexer class

Parsing We use an XML parser to extract the DBLP records from the dataset. Due to the large file size, reading in the whole XML tree at once is impractical. Instead, we use a SAX-like parser from the lxml[2] Python library that sequentially reads the document and emits events when it encounters certain XML tags. This allows to only react to <*article*> and <*inproceedings*> tags, thereby ignoring record types we are not interested in. This makes it very fast and memory-efficient. For each emitted event, we call the index_document function. (Listing 1, lines 15-19)

```
FIELDS =
    ['title','author','year','journal','booktitle']
HTML_TAGS = ('i','ref','sub','sup','tt')

def index_document(self,elem):
  etree.strip_tags(elem,HTML_TAGS)
  doc = Document()
  f = StringField('id',elem['key'],Store.YES)
  doc.add(f)
  for ch in elem:
    if ch.tag not in FIELDS:
      continue
    if ch.text is None:
      ch = etree.tostring(ch)
      ch = self.htmlparser.unescape(ch)
      ch = etree.fromstring(ch)

    if ch.tag == 'title':
      f = TextField('title',ch.text,Store.YES)
    elif ch.tag == 'author':
      f = TextField('authors',ch.text,Store.YES)
    elif ch.tag == 'year':
      f = StringField('year',ch.text,Store.YES)
    else:
      f = TextField('venue',ch.text,Store.YES)

    doc.add(f)
    cf = TextField('content',ch.text,Store.NO)
    doc.add(cf)
```

[1] https://lucene.apache.org

[2] http://lxml.de

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

```
29    self.writer.addDocument(doc)
```

Listing 2: index_document function

Preprocessing Given the current element, i.e. an *article* or *inproceedings* record, we can now iterate through its children, i.e. attributes of the record such as title or publication year. This is shown in Listing 2. As some of the records contain basic HTML formatting, irrelevant tags are removed and HTML-encoded characters are replaced (Listing 2, lines 5 and 12-15).

Indexing Each record is indexed as a Lucene *document*, comprising of the following *fields*: id, title, authors, year and venue. The venue is derived from the journal or booktitle attribute, depending on the record type. To facilitate the search, an additional *content* field is added that concatenates the values of the above fields except the id. This can be easily achieved by subsequently adding all values to the same field. This method is also used to index multiple authors. (Listing 2, lines 19-20 and 27-28).

All fields are treated as strings[3]. Contents of the original fields (i.e. without the *content* field) are stored in the index, which allows to retrieve them later during search. This is especially useful to enhance search results with additional information when presented to the user. The values of the fields *id* and *year* are stored as-is, which is a property of the StringField class. All other fields are tokenized (f_{token}) and further processed. The processing applied to each of those TextFields is determined by the analyzer that is used, which is known by the index writer (Listing 1, lines 6-10 and Listing 2, line 29). By default, all TextFields are processed in the same way, which may include lowercase conversion (f_{low}), removal of stopwords (f_{stop}) and stemming (f_{stem}). In particular, we use the Porter stemmer and the default stopword list of Lucene, which consists of 33 common English words.

Table 1 summarizes the analysis applied to each field.

Table 1: Analysis applied to document fields

Field	f_{token}	f_{low}	f_{stop}	f_{stem}
id				
year				
authors	X	X	X	
venue	X	X	X	
content	X	X	X	
title	X	X	X	X

Evaluation In order to evaluate the impact of additional token processing onto the index, different configurations of lowercase conversion, removal of stopwords and/or stemming are applied to the *title* field. For this, we define a custom analyzer that gives us full control over the configurations by extending the base analyzer of PyLucene (Listing 3).

```
VERSION = Version.LUCENE_CURRENT
class CustomAnalyzer(PythonAnalyzer):
  def __init__(self,config):
    self.lowercase = config['lowercase']
    self.stemming = config['stemming']
    self.stopwords = config['stopwords']
    PythonAnalyzer.__init__(self)

  def createComponents(self,fieldName,reader):
    src = StandardTokenizer(VERSION,reader)
    fltr = StandardFilter(VERSION,src)
    if self.lowercase:
      fltr = LowerCaseFilter(VERSION,fltr)
    if self.stemming:
      fltr = PorterStemFilter(fltr)
    if self.stopwords:
      sw = StopAnalyzer.ENGLISH_STOP_WORDS_SET
      fltr = StopFilter(VERSION,fltr ,sw)
    return self.TokenStreamComponents(src , fltr)
```

Listing 3: CustomAnalyzer class

We measure the speed of indexing (including parsing and analyzing the documents) in seconds, t_{ind}, as well as the number of terms in the vocabulary, $|V_{\text{title}}|$. The results are shown in Table 2. Note that indexing speed slightly varies over different runs.

Table 2: Evaluation of token processing on *title* field

f_{low}	f_{stop}	f_{stem}	t_{ind}	$\|V_{\text{title}}\|$
			417.97 (100%)	436 354 (100%)
X			430.57 (+3%)	346 286 (−21%)
	X		414.89 (−1%)	436 321 (±0%)
		X	439.75 (+5%)	365 534 (−16%)
X	X	X	417.94 (±0%)	288 710 (−34%)

Applying lowercase conversion or stemming reduces the vocabulary significantly, whereas the removal of stopwords has nearly no impact. This is expected, since the first two techniques are able to map multiple tokens onto the same term whereas the exclusion of stopwords decreases the vocabulary size only by the number of stopwords. By combining multiple processing steps, the vocabulary can be further reduced, leading to a reduction of to up a third of the original size when no further processing is applied. In terms of indexing speed, we found that the additional processing steps are negligible. In particular, there seems to be no positive correlation between indexing speed and the number of processing steps that are applied. One reason for this is that while processing each document is more time-consuming, the cost for indexing a processed document is actually reduced. Therefore, we conclude that, under the given evaluation criteria, it is recommended to apply all three processing techniques due to the reduction in vocabulary size. However, it should be noted that this also has an effect on other properties of the system, e.g. the quality of search results, which might or might not be desired.

4. SEARCH

Keyword search The search module makes use of the document index to retrieve the most relevant documents to a given query. It supports free text keyword queries on any combination of the attributes title, authors, year and venue. By default, all attributes are considered. This is achieved by utilizing the *content* field of the index. We refer to this as a standard search. Additionally, it is possible to query a combination of specific fields through an advanced search. Those keywords are directly matched against the respective fields in the index. In this case, documents must contain the provided keywords to be returned in the result set. Standard and advanced search can be freely combined during a single

[3] The *year* attribute may also be stored as integer, allowing for features like range searches. However, this would require additional effort during search that we try to avoid.

search request.

Phrase queries Phrase queries are supported using double quotation marks (e.g. “information retrieval”) in both standard and advanced search. Documents containing a particular phrase receive a higher weight during scoring. This requires an exact match up to the processing on the respective field. If a document does not contain the phrase, it may still be considered when there is a match for at least one of the keywords within the phrase. This considerably increases recall while risking an increase of false positives. Note that there is no restriction for the number of phrases within a query.

Query results The search returns the N most relevant documents along with their ranks, scores, ids and snippets, where N is a configurable parameter. Relevance is determined based on Lucene's internal scoring function. It also measures the time needed to retrieve the results. The complete procedure for handling queries is outlined in Listing 4.

```
VERSION = Version.LUCENE_CURRENT
def search(query,adv_query,N,analyzer,searcher):
  bq = BooleanQuery()
  if query != '':
    # handle phrases in standard search
    pq,query = get_pq(query,'content')
    if pq is not None:
        bq.add(pq,BooleanClause.Occur.MUST)

  if query != '':
    # handle remaining keywords
    qparser = QueryParser(VERSION,
                          'content',
                          analyzer)
    q = qparser.parse(query)
    bq.add(q,BooleanClause.Occur.SHOULD)

  if adv_query is not None:
    for field,query in adv_query.iteritems():
      # handle phrases in advanced search
      pq,query = get_pq(query,field)
      if pq is not None:
        bq.add(pq,BooleanClause.Occur.MUST)
      if query != '':
        # handle remaining keywords
        qparser = QueryParser(VERSION,
                              field,
                              analyzer)
        q = qparser.parse(query)
        bq.add(q,BooleanClause.Occur.MUST)

  start = time.clock()
  docs = searcher.search(bq,N).scoreDocs
  end = time.clock()
  duration = end-start
  return docs,duration

def get_pq(q,field,analyzer,slop=0,boost=5):
  phrases = re.findall(r'"([^"]*)"',q)
  if len(phrases) == 0:
    return None,q

  q = re.sub(r'"([^"]*)"',"",q).strip()
  bq = BooleanQuery()
  for phrase in phrases:
    qparser = QueryParser(VERSION,
                          field,
                          analyzer)
    # handle phrase and single keywords
    pq = qparser.parse('%s "%s"~%d^%.1f' %
                       (phrase,phrase,
                        slop,boost))
    bq.add(pq,BooleanClause.Occur.MUST)
  return bq,q
```

Listing 4: Query handling

5. EVALUATION

6. USER INTERFACE

There are two ways to interact with the search engine: via a text-based command line interface or via a web UI. The latter is built with Flask[4], a leightweight Python web framework that leverages Jinja2[5] as a templating engine. This allows to easily create HTML documents from within Python. Additionally, we incorporate the web framework Bootstrap[6] to achieve responsive web design. Figure 1 depicts the search results for an example query. As the UI is not our focus, we omit further implementation details; the interested reader is referred to the source code.

7. APPLICATIONS IN IR

We implement two applications based on dblp data. The first one is to find the top-10 most popular research topics of a specific year. The second one is finding top-10 similar publication venues and years with a given publication venue and year. Only paper titles are used in both application.

In this section, the related techniques and the detailed implementations of the applications are demonstrated, followed by the experiment results.

7.1 Popular research topics

Before mining popular research topics from paper titles, we need to firstly extract topics from such data. In our application, we identify meaningful topics from titles based on the patterns of words. More precisely, given a query (e.g. 2014), we have following process to extract topics from it:

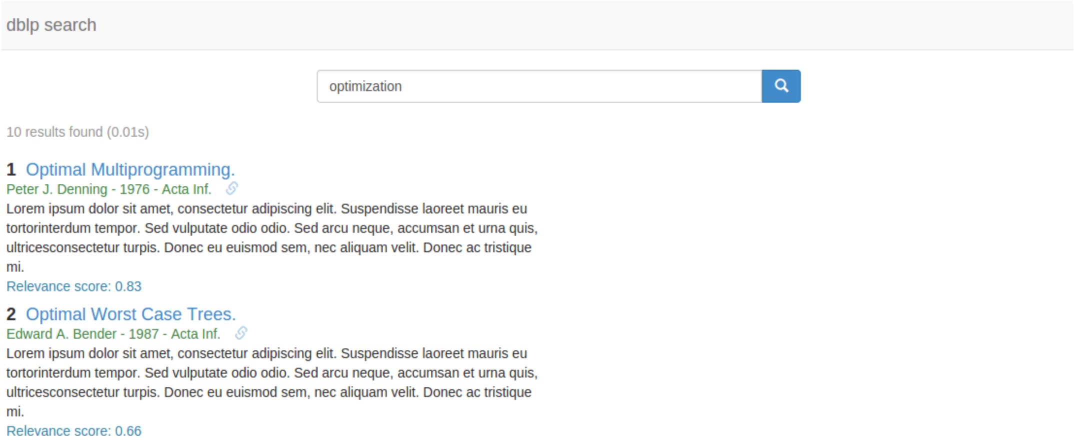
- Retrieve the titles of all the papers published in 2014, using the index.
- Tokenize each title and remove punctuations and stop words.
- Use Part-Of-Speech Tagger (POS Tagger)[7] to assign parts of speech to each word, such as noun, verb, adjective, etc.
- Extract topics from the tagged words based on the given pattern, using regular expression tools.

The key point of topic extraction is the above steps is to define the word pattern of topic. From the observation of real topic names, we can conclude that

- A topic name usually consist of adjectives and nouns.
- The adjectives always come before nouns for a topic name.
- Normally the number of words in a topic name is less than four.

Therefore, in the third step, we define the pattern as adj+adj+...+noun+...+noun. Moreover, only unigram, bigram and trigram are considered in this application. In addition, we ignore those unigram terms with very high (top 1%) frequency because they may be some widely-used terminology

[4] http://flask.pocoo.org
[5] http://jinja.pocoo.org
[6] http://getbootstrap.com
[7] http://nlp.stanford.edu/software/tagger.shtml



dblp search

optimization

10 results found (0.01s)

1 Optimal Multiprogramming.
Peter J. Denning - 1976 - Acta Inf.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse laoreet mauris eu tortorinterdum tempor. Sed vulputate odio odio. Sed arcu neque, accumsan et urna quis, ultricesconsectetur turpis. Donec eu euismod sem, nec aliquam velit. Donec ac tristique mi.
Relevance score: 0.83

2 Optimal Worst Case Trees.
Edward A. Bender - 1987 - Acta Inf.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse laoreet mauris eu tortorinterdum tempor. Sed vulputate odio odio. Sed arcu neque, accumsan et urna quis, ultricesconsectetur turpis. Donec eu euismod sem, nec aliquam velit. Donec ac tristique mi.
Relevance score: 0.66

Figure 1: Web-based search user interface

in many different topics. After the topic identification, we count the frequency of each topic appeared in the titles. Finally, the top-10 most frequent topics will be return as the result.

```
def get_popular_topics(self, q_year, top_k):
    titles = self.searcher.search_year(q_year)
    unigram_dist = {}
    bigram_dist = {}
    trigram_dist = {}
    ngram_dist = {}

    tagset = None
    tagger = PerceptronTagger()
    grammar = ”NP: {<JJ>*(<NN>|<NNS>)*<NN>(<NN>|<NNS>)*}”
    cp = nltk.RegexpParser(grammar)
    for title in titles:
        title = title.lower()
        text = word_tokenize(title)
        sentence = nltk.tag._pos_tag(text, tagset, tagger)
        result = cp.parse(sentence)
        for node in list(result):
            if isinstance(node, nltk.tree.Tree):
                entity = zip(*list(node))[0]
                if len(entity) == 1:
                    self.dict_append(entity, unigram_dist)
                elif len(entity) == 2:
                    self.dict_append(entity, bigram_dist)
                elif len(entity) == 3:
                    self.dict_append(entity, trigram_dist)
                else:
                    self.dict_append(entity, ngram_dist)

    unigram_result = Counter(unigram_dist).most_common(int(len(unigram_dist) * 0.01) + top_k)[int(len(unigram_dist) * 0.01):]
    bigram_result = Counter(bigram_dist).most_common(top_k)
    trigram_result = Counter(trigram_dist).most_common(top_k)

    result = unigram_result + bigram_result + trigram_result
    result = sorted(result, key=lambda k: k[1], reverse=True)[:top_k]
    return result
```

Listing 5: Finding popular topics for a given year

7.2 Similar publication venues

For retrieving similar publication venues and years, firstly we need to measure the similarity between two publication venues and years (e.g How similar are ”TKDE’14” and ”CIKM’15”) using the paper titles. One simple way is to firstly tokenize every title of the venues and years, and use bag-of-words model, computing the cosine similarity of the word frequency vectors between them, which is defined as:

$$Sim_{i,j} = \frac{\sum_{k \in W} freq_{i,k} \cdot freq_{j,k}}{\sqrt{\sum_{k \in W} freq_{i,k}^2}\sqrt{\sum_{k \in W} freq_{j,k}^2}},$$

where i, j are two (venue, year) pairs. W refers to the collection contains all the words appeared in the corpus. $freq_{i,k}$ is the frequency of word k appears in the ith venue and year.

However, based on our statistical analysis, the dimensionality is too high. It is very inefficient to compute similarity. Besides, the data may also be very noisy, containing many words that only appears once in the whole dataset.

To address the aforementioned problems, we adopted Latent Dirichlet allocation (LDA)[?] to learn the latent topics from the data. Instead of using word frequency vectors, the similarity is defined based on the cosine similarity between the topic distribution of two venues and years. Thus, for a query venue and year, we compute the topic similarities between it and every other venues and years, which is given by

$$Sim_{i,j} = \frac{\sum_{z \in Z} \theta_{i,z} \cdot \theta_{j,z}}{\sqrt{\sum_{z \in Z} \theta_{i,z}^2}\sqrt{\sum_{z \in Z} \theta_{j,z}^2}},$$

where Z is the topic set; $\theta_{i,z}$ is the probability of the ith venue and year belongs to topic z.Then, based on the similarity, top-10 similar venues and years are returned as the result.

7.3 Application Experiments

For retrieving popular research topics, we have the results as follows:

For finding similar publication venues and years, we firstly learn the topics as follows:

The results of the queries are given as:

Table 3: Top-10 most popular research topics of the papers published in 2013

Rank	Topic	Frequency
1	wireless sensor networks	1412
2	case study	1182
3	performance analysis	568
4	cognitive radio networks	468
5	special issue	452
6	performance evaluation	404
7	cloud computing	373
8	genetic algorithm	341
9	empirical study	340
10	neural network	340

Table 4: Representive words for the topics leant by LDA

Topic	Words
Geographic Information System	data using based sar spatial sensing analysis remote images radar model land image satellite surface detection gis classification resolution hyperspectral
Robotics	robot control based using robots mobile motion robotic planning human autonomous design navigation vision dynamic sensor multi force tracking visual
Wireless Sensor Networks	networks wireless based network mobile routing ad hoc sensor protocol performance control using traffic efficient scheme qos algorithm 802 multi
Software Technology	software testing analysis based using test engineering development study code case model systems java quality source empirical tool approach program
Social Media	social online media networks community network behavior use internet effects study analysis communities understanding games self role game impact influence

Table 5: Top-10 most similar venues & years to "IEEE Trans. Knowl. Data Eng. 2014"

Rank	Venue	Year	Similarity
1	IEEE Trans. Knowl. Data Eng.	2015	0.9898
2	IEEE Trans. Knowl. Data Eng.	2012	0.9888
3	DaWaK	2015	0.9887
4	IEEE Trans. Knowl. Data Eng.	2013	0.9856
5	LD4IE@ISWC	2014	0.9850
6	CIKM	2006	0.9845
7	WAIM	2013	0.9830
8	CIKM	2007	0.9823
9	FQAS	2009	0.9815
10	CIKM	2004	0.9803

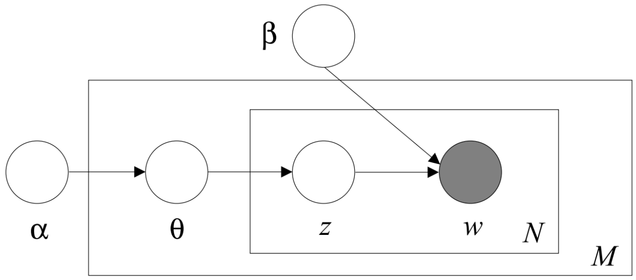


Figure 2: Graph representation of Latent Dirichlet allocation (LDA)

8. CONCLUSIONS

9. REFERENCES