Information Retrieval
INFO/CS 4300

- Instructor: Chris Buckley
  - Office hours Wednesdays 11am Gates 231
- Piazza will be the main communication tool
  - https://piazza.com/cornell/fall2014/info4300/home
  - Lecture notes will appear there.
  - TA office hours and locations appear there.

Course Admin

- Critique 1, Homework 1 – Graded, grades available on-line through CMS, hard copy can be picked up in the homework return room in Gates 216, open Mon-Fri noon-4pm
- Project 1 – Out today via CMS, due October 30. Please form groups within CMS of 2-3 students per group.

Previous Lectures

- Overview
- Evaluation 1
- Indexing
- Retrieval
  - Boolean Model
  - Tf*idf weighting
  - Vector Space Model
  - Retrieval Optimization (DAAT, TAAT, safe vs non-safe)
  - Basic Probabilistic Model
  - Advanced Probabilistic Models
    - BM25
    - Query Likelihood Language Model

Retrieval Models

- Older models
  - Boolean retrieval (still used, special applications)
  - Vector Space model (still used with tf*idf weighting for general retrieval)
  - Basic Probabilistic model
- Newer Probabilistic Models
  - BM25
  - Language models (Query Likelihood now, more later)
- Newer tf*idf variants
  - Pivoted unique normalization
- Combining evidence (later in course)
  - Inference networks
  - Learning to Rank

Improved vector space retrieval

- Standard weighting scheme is tf*idf and cosine similarity
  - We've discussed tf*idf variants (and you'll see more of this next week with project 1) and possible alternatives
  - Can cosine document length normalization be improved?
    - Recall that cosine normalization put all vectors on a unit hypersphere
    - Nice theoretically – is it what we want in practice?
    - The answer turns out to be no, but this was discovered in a very roundabout fashion

Case study: Query expansion digression

- I was looking at query expansion techniques in the mid 1990's (early TREC days where the field was doing experimenting with long documents in test collections for the first time)
- Basic tf*idf weights with cosine normalization
- Discovered a method of expanding (lengthening) queries by adding related terms with appropriate weights
  - Worked very well improving retrieval results, in fact too well!!!
  - Was still getting improvements (small, but significant) by adding 200-300 terms.

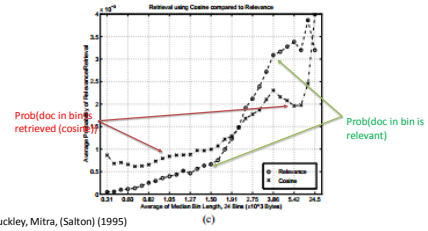
Query expansion => length normalization

- Query expansion results yielded very smooth improvement curves as terms were added. At the end of even 100 added terms, the terms being added were common terms that looked visually to be pretty random. Either
  - A. I had discovered something fundamental like an "atomic" unit or measurement of semantic information content, and how to weight it. Or
  - B. Something else was going on
- I alternated between the two possibilities for a couple of weeks
- The answer turned out to be B.
  - Adding random common terms (with low weights) to the query increases the score of documents randomly (wrt original query), but will tend to increase the score of longer documents more than shorter documents!
  - Cosine length normalization turned out to be biased against long documents

Investigating length normalization

- We had had length normalization on our list of factors to investigate for a while. Steve Robertson and Okapi had just come out with BM25 which approached length normalization differently.
- How do you investigate length normalization issues?
- First step, figure out what is happening in practice now
- We divided the document collection into bins according to length
- Calculated Prob (retrieval of D | D in bin i)
- Calculated Prob (relevance of D | D in bin i)

Prob (cosine retrieval) vs Prob (relevance)

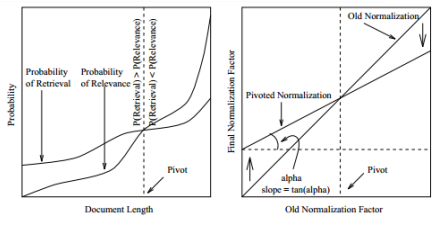


Singhal, Buckley, Mitra, (Salton) (1995)

Cosine favors shorter documents

- In an ideal world, we would hope the two plots would be much closer.
- If we change our retrieval normalization to match Prob(Relevance), will that improve performance?
- Amit Singhal took over finding methods to do this.
- First step was introducing pivoted normalization.

Pivot old normalization around a point

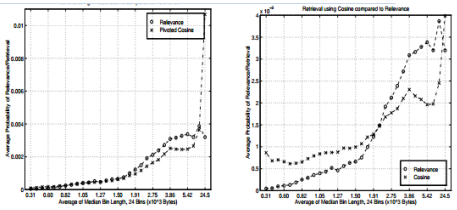


Pivoted_normalization = (1.0 – slope) * pivot + slope * old_normalization

Pivoted Normalization

- *Pivoted_normalization = (1.0 – slope) * pivot + slope * old_normalization*
- If we take the pivot point as being the average old normalization document, and divide by it (doesn't affect ranking)
- *Pivoted_normalization = (1.0 – slope) + slope * * $\frac{old\ normalization}{average\ old\ normalization}$
- A document of length average old normalization will be unchanged
- *Slope* can be interpreted as our "belief in length"
- Derived very differently, but similar to Okapi's BM25 length normalization factor (done first) $K = k_1((1-b) + b \cdot \frac{dl}{avdl})$

Pivoted cosine normalization

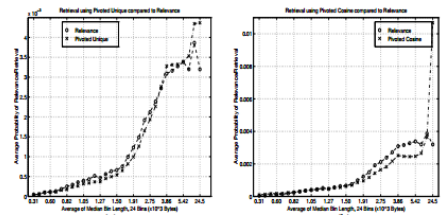


Plot on left matches relevance much better than old plot on right
MAP performance improves by 11.7%

Pivoted unique terms

- Still a problem with very long documents.
- Analyzing further, issue is the tf contribution in very long documents
  - Even standard taking logs is too much (for the length part)
- Instead of cosine of *log (1.0+tf)* weighting or something similar, use the number of unique terms.
- *Pivoted_unique = (1-slope) * pivot + slope * # of unique terms*
- Again can take the pivot as the average number of unique terms in a doc

Pivoted unique vs pivoted cosine



Matches relevance better in the longer documents
MAP performance is another 6% better – total improvement of 18.3% over cosine

Pivoted unique: Final tf*idf weighting

- Lnu.ltc where Lnu weighting in documents is

$$\frac{(1+\log(tf))}{1+\log(average\ tf)}$$

$$(1.0 - slope) + slope * \frac{(\#\ of\ unique\ terms)}{average\ \#\ of\ unique\ terms}$$

Where *slope = 0.20* works well across collections

Retrieval Models

- Older models
  - Boolean retrieval (still used, special applications)
  - Vector Space model (still used with tf*idf weighting for general retrieval)
  - Basic Probabilistic model
- Newer Probabilistic Models
  - BM25
  - Language models
- Newer tf*idf variants
  - Pivoted unique normalization
- Combining evidence (later in course)
  - Inference networks
  - Learning to Rank

Features of these newer models

- All work about the same as far as test collection evaluation goes
- All require estimating parameter(s)
  - The estimations are not completely motivated by the models
  - But most parameters are insensitive to small changes (should do reasonably on other collections)
- Precursors, in some ways, to "Learning to Rank" models, covered later

Open source IR systems

- Widely used academic systems
  - Terrier (Java, U. Glasgow) http://terrier.org
  - Indri/Galago/Lemur (C++ (& Java), U. Mass & CMU)
  - Tail of others (Zettair, …)
  - SMART no longer used (got tied up due to a bad licensing agreement)
- Widely used non-academic open source systems
  - **Lucene**
    - Things built on it: Solr, ElasticSearch
  - A few others (Xapian, …)

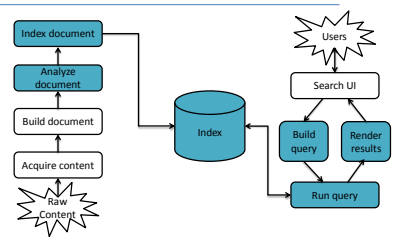
Lucene (adapted from tutorial at Stanford)

- Open source Java library for indexing and searching
  - Lets you add search to your application
  - Not a complete search system by itself
  - Written by Doug Cutting
  - Used by: Twitter, LinkedIn; Reddit, Zappos; CiteSeer, Eclipse, …
  - … and many more (see http://wiki.apache.org/lucene-java/PoweredBy)
- Ports/integrations to other languages
  - C/C++, C#, Ruby, Perl, Python, PHP, …

Lucene

- Quite flexible in certain ways
  - Witnessed by variety of folks using it
  - Many indexing and similarity options
- Easy flexibility limited by information available
  - A feature shared by all operational systems
  - Examples
    - Use of CollectionFrequency rather than DocumentFrequency (idf)
    - Document Normalization techniques limited (cosine, idf cause issues)
      - Less directly usable for Project 1 than hoped for!

Lucene in a search system



Lucene demos

- Command line **`Indexer`**
  - `org.apache.lucene.demo.IndexFiles`
- Command line **`Searcher`**
  - `org.apache.lucene.demo.SearchFiles`

Core indexing classes

- `IndexWriter`
  - Central component that allows you to create a new index, open an existing one, and add, remove, or update documents in an index
  - Built on an `IndexWriterConfig` and a `Directory`
- `Directory`
  - Abstract class that represents the location of an index
- `Analyzer`
  - Extracts tokens from a text stream

Creating an `IndexWriter`

```
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
...

IndexWriter getIndexWriter(String dir) {
    Directory indexDir = FSDirectory.open(new File(dir));
    IndexWriterConfig luceneConfig = new IndexWriterConfig(
         luceneVersion, new StandardAnalyzer(luceneVersion));

    return(new IndexWriter(indexDir, luceneConfig));
}
```

Core indexing classes (contd.)

- `Document`
  - Represents a collection of named `Field`s. Text in these `Field`s are indexed.
- `Field`
  - Note: Lucene `Field`s can represent both "fields" and "zones"
  - Or even other things like numbers.

A `Document` contains `Field`s

```
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
...
protected Document getDocument(File f) throws Exception {
      Document doc = new Document();
      doc.add(new TextField("contents", new FileReader(f)))
   doc.add(new StringField("filename", f.getName()));
   doc.add(new StringField("fullpath",

f.getCanonicalPath()));
   return doc;
}
```

CACM Fields (Document 3139)

New Methods to Color the Vertices of a Graph
This paper describes efficient new heuristic
methods to color the vertices of a graph which rely
upon the comparison of the degrees and structure of a graph. A method
is developed which is exact for bipartite graphs and is an
important part of heuristic procedures to find maximal cliques in general
graphs. Finally an exact method is given which performs better
than the Randall-Brown algorithm and is able to color larger
graphs, and the new heuristic methods, the classical methods, and
the exact method are compared.
CACM April, 1979
Brelaz, D.
NP-complete, graph structure, balancing, graph
coloring, scheduling, comparison of the methods
5.25 5.32
CA790405 DH June 5, 1979 2:05 PM

- Fields in original CACM collection were tagged to distinguish them

Index a `Document` with `IndexWriter`

```
private IndexWriter writer;
...
private void indexFile(File f) throws
          Exception {
     Document doc = getDocument(f);
     writer.addDocument(doc);
}
```

Indexing a directory

```
private IndexWriter writer;
...
public int index(String dataDir,
                 FileFilter filter)
          throws Exception {
     File[] files = new File(dataDir).listFiles();
     for (File f: files) {
          if (... &&
               (filter == null ||
                filter.accept(f))) {
               indexFile(f);
          }
     }
     return writer.numDocs();
}
```

Closing the `IndexWriter`

```
private IndexWriter writer;
...
public void close() throws IOException {
     writer.close();
}
```

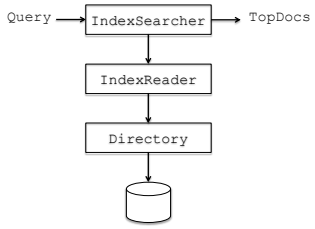
The Index

- The Index is the kind of inverted index we know and love
- The default Lucene40 codec is:
  - variable-byte coding of delta values
  - multi-level skip lists
  - natural ordering of docIDs
  - interleaved docIDs and position data
  - Very short postings lists are inlined into the term dictionary
- Other codecs are available: PFOR-delta, Simple9, …

Core searching classes

- `IndexSearcher`
  - Central class that exposes several search methods on an index
  - Accessed via an `IndexReader`
- `Query`
  - Abstract query class. Concrete subclasses represent specific types of queries, e.g., matching terms in fields, boolean queries, phrase queries, …
- `QueryParser`
  - Parses a textual representation of a query into a `Query` instance

`IndexSearcher`



Creating an `IndexSearcher`

```
import org.apache.lucene.search.IndexSearcher;
...
public static void search(String indexDir,
                          String q)
            throws IOException, ParseException {
     IndexReader rdr =
       DirectoryReader.open(
           FSDirectory.open(new File(indexDir)));
      IndexSearcher is = new IndexSearcher(rdr);
      ...
}
```

`Query` and `QueryParser`

```
import org.apache.lucene.search.Query;
import org.apache.lucene.queryParser.QueryParser;
...
public static void search(String indexDir, String q)
            throws IOException, ParseException
      ...
      QueryParser parser =
        new QueryParser(Version.LUCENE_40,
                        "contents",
                         new StandardAnalyzer(
                                        Version.LUCENE_40));
      Query query = parser.parse(q);
      ...
}
```

Core searching classes (contd.)

- TopDocs
  - Contains references to the top documents returned by a search
- ScoreDoc
  - Represents a single search result

search() returns TopDocs

```
import org.apache.lucene.search.TopDocs;
...
public static void search(String indexDir,String q)
        throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    Query query = ...;
    ...
    TopDocs hits = is.search(query, 10);
}
```

TopDocs contain ScoreDocs

```
import org.apache.lucene.search.ScoreDoc;
...
public static void search(String indexDir, String q)
        throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    TopDocs hits = ...;
    ...
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = is.doc(scoreDoc.doc);
        System.out.println(doc.get("fullpath"));
    }
}
```

Closing IndexSearcher

```
public static void search(String indexDir,

        String q)
      throws IOException, ParseException
    ...
    IndexSearcher is = ...;
    ...
    is.close();
}
```

How Lucene models content

- A Document is the atomic unit of indexing and searching
  - A Document contains Fields
- Fields have a name and a value
  - You have to translate raw content into Fields
  - Examples: Title, author, date, abstract, body, URL, keywords, ...
  - Different documents can have different fields
  - Search a field using name:term, e.g., title:lucene

Fields

- Fields may
  - Be indexed or not
    - Indexed fields may or may not be analyzed (i.e., tokenized with an Analyzer)
      - Non-analyzed fields view the entire value as a single token (useful for URLs, paths, dates, social security numbers, ...)
  - Be stored or not
    - Useful for fields that you'd like to display to users
  - Optionally store term vectors
    - Like a positional index on the Field's terms
    - Useful for highlighting, finding similar documents, categorization

Analyzer

- Tokenizes the input text
- Common Analyzers
  - WhitespaceAnalyzer
    Splits tokens on whitespace
  - SimpleAnalyzer
    Splits tokens on non-letters, and then lowercases
  - StopAnalyzer
    Same as SimpleAnalyzer, but also removes stop words
  - StandardAnalyzer
    Most sophisticated analyzer that knows about certain token types, lowercases, removes stop words, ...

Analysis example

- "The quick brown fox jumped over the lazy dog"
- WhitespaceAnalyzer
  - [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- SimpleAnalyzer
  - [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- StopAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
- StandardAnalyzer
  - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]

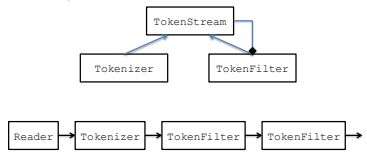
Another analysis example

- "XY&Z Corporation – xyz@example.com"
- WhitespaceAnalyzer
  - [XY&Z] [Corporation] [-] [xyz@example.com]
- SimpleAnalyzer
  - [xy] [z] [corporation] [xyz] [example] [com]
- StopAnalyzer
  - [xy] [z] [corporation] [xyz] [example] [com]
- StandardAnalyzer
  - [xy&z] [corporation] [xyz@example.com]

What's inside an Analyzer?

- Analyzers need to return a TokenStream

```
public TokenStream tokenStream(String fieldName,
    Reader reader)
```



Tokenizers and TokenFilters

- Tokenizer
  - WhitespaceTokenizer
  - KeywordTokenizer
  - LetterTokenizer
  - StandardTokenizer
  - ...
- TokenFilter
  - LowerCaseFilter
  - StopFilter
  - PorterStemFilter
  - ASCIIFoldingFilter
  - StandardFilter
  - ...

Tokenizer setup example (see MyAnalyzer.java)

```
final StandardTokenizer src = new StandardTokenizer(reader);
TokenStream tok = new StandardFilter(src);
tok = new LowerCaseFilter(tok);

// Add additional filters here
//tok= new PorterStemFilter(tok);
//tok = new StopFilter(tok, indri_stopwords);
```

Index format

- Each Lucene index consists of one or more segments
  - A segment is a standalone index for a subset of documents
  - All segments are searched
  - A segment is created whenever `IndexWriter` flushes adds/deletes
- Periodically, `IndexWriter` will merge a set of segments into a single segment
  - Policy specified by a `MergePolicy`
- You can explicitly invoke `optimize()` to merge segments

Basic merge policy

- Segments are grouped into levels
- Segments within a group are roughly equal size (in log space)
- Once a level has enough segments, they are merged into a segment at the next level up
  - E.g. Logarithmic Merge from our earlier class

Searching a changing index

```
Directory dir = FSDirectory.open(...);
IndexReader reader = IndexReader.open(dir);
IndexSearcher searcher = new IndexSearcher(reader);
```

Above `reader` does not reflect changes to the index unless you `reopen` it. `Reopen`ing is more resource efficient than `open`ing a new `IndexReader`.

```
IndexReader newReader = reader.reopen();
If (reader != newReader) {
      reader.close();
      reader = newReader;
      searcher = new IndexSearcher(reader);
}
```

Near-real-time search

```
IndexWriter writer = ...;
IndexReader reader = writer.getReader();
IndexSearcher searcher = new IndexSearcher(reader);

// Now let us say there's a change to the index using writer
writer.addDocument(newDoc);

// reopen() and getReader() force writer to flush
IndexReader newReader = reader.reopen();
if (reader != newReader) {
      reader.close();
      reader = newReader;
      searcher = new IndexSearcher(reader);
}
```

`IndexSearcher`

- Methods
  - `TopDocs search(Query q, int n);`
  - `Document doc(int docID);`

`QueryParser`

- Constructor
  - `QueryParser(Version matchVersion, String defaultField, Analyzer analyzer);`
  - Important: Need to ensure that `Analyzer`s used at indexing time are consistent with `Analyzer`s used at searching time
- Parsing methods
  - `Query parse(String query) throws ParseException;`
  - … and many more

`QueryParser` syntax examples

Query expression	Document matches if…
java	Contains the term *java* in the default field
java junit java OR junit	Contains the term *java* or *junit* or both in the default field (*the default operator can be changed to* AND)
+java +junit java AND junit	Contains both *java* and *junit* in the default field
title:ant	Contains the term *ant* in the title field
title:extreme –subject:sports	Contains *extreme* in the title and not *sports* in subject
(agile OR extreme) AND java	Boolean expression matches
title:"junit in action"	Phrase matches in title
title:"junit action"~5	Proximity matches (within 5) in title
java*	Wildcard matches
java~	Fuzzy matches
lastmodified:[1/1/09 TO 12/31/09]	Range matches

`TopDocs` and `ScoreDoc`

- `TopDocs` methods
  - Number of documents that matched the search
    `totalHits`
  - Array of `ScoreDoc` instances containing results
    `scoreDocs`
  - Returns best score of all matches
    `getMaxScore()`
- `ScoreDoc` methods
  - Document id
    `doc`
  - Document score
    `score`

Scoring

- Original scoring function uses basic tf-idf scoring with
  - Programmable boost values for certain fields in documents
  - Length normalization
  - Boosts for documents containing more of the query terms
- `IndexSearcher` provides an `explain()` method that explains the scoring of a document
  - Sample debugging output

```
if (queryId == 32) {
    System.out.print( searcher.explain(query, results.scoreDocs[0].doc));
}
```

Lucene 4.0 Scoring

- As well as traditional tf.idf vector space model, Lucene 4.0+ adds:
  - BM25
  - drf (divergence from randomness)
  - ib (information (theory)-based similarity)

```
indexSearcher.setSimilarity(
     new BM25Similarity());
BM25Similarity custom =
          new BM25Similarity(1.2, 0.75); // k1, b
indexSearcher.setSimilarity(custom);
```

Default Lucene Similarity

- $score(q,d) = coord(q,d) * queryNorm(q) * \sum_{t\ in\ q} (tf(t\ in\ d) * idf(t)^2 * t.getboost(\ \ ) * norm(t,d)$
- Where
  - *Coord(q,d)* is fraction of query terms in *q* that *d* contains
  - *queryNorm (q)* is our familiar cosine length normalization
  - *t.getboost()* could be a user supplied weight in advanced queries
  - *Norm(t,d)* is set at indexing time and considers field boosts and lengths