Outline

1. Vector Space Ranking & TFIDF
2. Lucene

Next Lecture → Assignment 1 marking will be discussed
1. Vector Space Ranking

→ quick recap of last lecture's topic, using David Kauchak's slides
2. Vector Space Ranking

→ Represent the query as a weighted TFIDF vector

→ Represent each document as a weighted TFIDF vector

→ Compute the cosine similarity score between query vector and each document vector

→ Rank documents by their score

→ Return the top K (e.g., K = 10) documents to the user
Term-document count matrices

- Consider the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^v$: a column below

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>157</td>
<td>73</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brutus</td>
<td>4</td>
<td>157</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>232</td>
<td>227</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>57</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

What information is lost with this representation?
Bag of words representation

- Represent a document by the occurrence counts of each word
- **Ordering** of words is lost
- *John is quicker than Mary* and *Mary is quicker than John* have the same vectors
Bag of words

What is the notion of “intersection” for the bag of words model?
Bag of words

Want to take into account term frequency
Some things to be careful of...

Say I take the document and simply append it to itself. What happens to the overlap?
Some things to be careful of...

What is the issue?

Need some notion of the length of a document
Some things to be careful of...

What about a document that contains only frequent words, e.g. *the*?
Some things to be careful of…

Need some notion of the importance of words
Documents as vectors

- We have a $|V|$-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: hundreds of millions of dimensions when you apply this to a web search engine
- This is a very sparse vector - most entries are zero
Use angle instead of distance

- Thought experiment: take a document d and append it to itself. Call this document d’
- “Semantically” d and d’ have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity

- Any other ideas?
- Rank documents according to angle with query
From angles to cosines

- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$.
- The following two notions are equivalent.
  - Rank documents in decreasing order of the angle between query and document.
  - Rank documents in increasing order of $\cos$ine(query,document).
cosine(query, document)

\[
\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{||\vec{q}|| ||\vec{d}||} = \frac{\vec{q}}{||\vec{q}||} \cdot \frac{\vec{d}}{||\vec{d}||} = \frac{\sum_{i=1}^{\mid V \mid} q_i d_i}{\sqrt{\sum_{i=1}^{\mid V \mid} q_i^2} \sqrt{\sum_{i=1}^{\mid V \mid} d_i^2}}
\]

\(\cos(q, d)\) is the cosine similarity of \(q\) and \(d\) ... or, equivalently, the cosine of the angle between \(q\) and \(d\).
Inverse document frequency

- $df_t$ is the **document** frequency of $t$: the number of documents that contain $t$
  - $df$ is a measure of the informativeness of $t$
- We define the idf (inverse document frequency) of $t$ by

\[
idf_t = \log \frac{N}{df_t}
\]

- We use $\log \frac{N}{df_t}$ instead of $\frac{N}{df_t}$ to “dampen” the effect of idf
### idf example, suppose $N = 1$ million

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>sunday</td>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>fly</td>
<td>10,000</td>
<td>2</td>
</tr>
<tr>
<td>under</td>
<td>100,000</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1,000,000</td>
<td>0</td>
</tr>
</tbody>
</table>

There is one idf value for each term $t$ in a collection.
idf example, suppose $N=1$ million

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>sunday</td>
<td>1,000</td>
<td></td>
</tr>
<tr>
<td>fly</td>
<td>10,000</td>
<td></td>
</tr>
<tr>
<td>under</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>1,000,000</td>
<td></td>
</tr>
</tbody>
</table>

What if we didn’t use the log to dampen the weighting?
The table below illustrates the *idf* example, assuming $N = 1$ million.

<table>
<thead>
<tr>
<th>term</th>
<th>$df_t$</th>
<th>$idf_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>calpurnia</td>
<td>1</td>
<td>1,000,000</td>
</tr>
<tr>
<td>animal</td>
<td>100</td>
<td>10,000</td>
</tr>
<tr>
<td>sunday</td>
<td>1,000</td>
<td>1,000</td>
</tr>
<tr>
<td>fly</td>
<td>10,000</td>
<td>100</td>
</tr>
<tr>
<td>under</td>
<td>100,000</td>
<td>10</td>
</tr>
<tr>
<td>the</td>
<td>1,000,000</td>
<td>1</td>
</tr>
</tbody>
</table>

What if we didn’t use the log to dampen the weighting?
Putting it all together

- We have a notion of term frequency overlap
- We have a notion of term importance
- We have a similarity measure (cosine similarity)

- Can we put all of these together?
  - Define a weighting for each term
  - The tf-idf weight of a term is the product of its tf weight and its idf weight

\[ w_{t,d} = tf_{t,d} \times \log \frac{N}{df_t} \]
tf-idf weighting

\[ w_{t,d} = tf_{t,d} \times \log N / df_t \]

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection
- Works surprisingly well!
- Works in many other application domains
# Binary → count → weight matrix

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>5.25</td>
<td>3.18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.35</td>
</tr>
<tr>
<td>Brutus</td>
<td>1.21</td>
<td>6.1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>8.59</td>
<td>2.54</td>
<td>0</td>
<td>1.51</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1.54</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>2.85</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1.51</td>
<td>0</td>
<td>1.9</td>
<td>0.12</td>
<td>5.25</td>
<td>0.88</td>
</tr>
<tr>
<td>worser</td>
<td>1.37</td>
<td>0</td>
<td>0.11</td>
<td>4.15</td>
<td>0.25</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Each document is now represented by a real-valued vector of tf-idf weights \( \in \mathbb{R}^{|V|} \).

We then calculate the similarity using cosine similarity with these vectors.
There are many variations of TF-IDF weighting

→ $\log( N / DF(T) )$  [as on previous slides]
  gives weight zero, to a term appearing in each document!
→ alternative $\log(1 + N / DF(T))$

→ alternatives to TF:  – divide by largest TF of that term (“normalization”)
  – take $1 + \log(\text{TF})$ (“log-frequency weighting”)
There are many variations of TF-IDF weighting

→ \( \log \left( \frac{N}{DF(T)} \right) \) [as on previous slides]
  gives weight zero, to a term appearing in each document!
→ alternative \( \log \left( 1 + \frac{N}{DF(T)} \right) \)

→ alternatives to TF:
  – divide by largest TF of that term (“normalization”)
  – take \( 1 + \log(\text{TF}) \) (“log-frequency weighting”)

Explanations for taking \( \log \) of \( \frac{N}{DF(T)} \) (“damping”)

→ Probability that random document contains term T:
  \( P(T) = \frac{DF(T)}{N} \)
→ \( IDF(T) = -\log( P(T) ) \)
There are many variations of TF-IDF weighting

→ $\log \left( \frac{N}{DF(T)} \right)$ [as on previous slides] gives weight zero, to a term appearing in each document!
→ alternative $\log \left( 1 + \frac{N}{DF(T)} \right)$

→ alternatives to TF:  
  - divide by largest TF of that term ("normalization")
  - take $1 + \log(TF)$ ("log-frequency weighting")

Explanations for taking $\log$ of $N/DF(T)$ ("damping")

→ **Probability** that *random document* contains term T:  
  $P(T) = DF(T) / N$

→ $IDF(T) = - \log(P(T))$

→ $IDF(T_1 \text{ ‘and’ } T_2) = - \log(P(T_1) * P(T_2)) = IDF(T_1) + IDF(T_2)$

statistically independent
Recall from **Information Theory**: 

Message probabilities \( p(1), p(2), p(3), \ldots, p(N) \)  (sum equals 1) 

Information of Message \( m \): \( I(m) = - \log p(m) \) 

→ see Robertson’s paper, linked on course web page

E.g. \( p(1) = p(2) = 0.5 \)

\[ I(1) = - \log p(1) = 1 \]  1 bit  (because we took log-base 2)

The mathematical theory of information is based on probability theory and statistics, and measures information with several quantities of information. The choice of logarithmic base in the following formulae determines the unit of information entropy that is used. The most common unit of information is the **bit**, based on the binary logarithm. Other units include the nat, based on the natural logarithm, and the **hartley**, based on the base 10 or common logarithm.
Recall from Information Theory:

Message probabilities $p(1), p(2), p(3), \ldots, p(N)$ (sum equals 1)

Information of Message $m$: $I(m) = -\log p(m)$
Recall from Information Theory:

Message probabilities $p(1), p(2), p(3), \ldots, p(N)$ (sum equals 1)

Information of Message $m$: $I(m) = -\log p(m)$

→ faint relationship to Zipf’s law

Mentioned in original article introducing IDF
[Karen Spärck Jones, 1972]
Log-frequency weighting

- Want to reduce the effect of multiple occurrences of a term
- A document about “Clinton” will have “Clinton” occurring many times
- Rather than use the frequency, use the log of the frequency

\[ w_{t,d} = \begin{cases} 
1 + \log \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\
0, & \text{otherwise}
\end{cases} \]

- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.
tf-idf weighting has many variants

<table>
<thead>
<tr>
<th>Term frequency</th>
<th>Document frequency</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>n (natural)</td>
<td>tf$_{t,d}$</td>
<td>n (no) 1</td>
</tr>
<tr>
<td>l (logarithm)</td>
<td>$1 + \log(tf_{t,d})$</td>
<td>t (idf) $\log \frac{N}{df_t}$</td>
</tr>
<tr>
<td>a (augmented)</td>
<td>$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$</td>
<td>p (prob idf) $\max{0, \log \frac{N - df_t}{df_t}}$</td>
</tr>
<tr>
<td>b (boolean)</td>
<td>{1 if $tf_{t,d} &gt; 0$; 0 otherwise}</td>
<td></td>
</tr>
<tr>
<td>L (log ave)</td>
<td>$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}<em>{t \in d}(tf</em>{t,d}))}$</td>
<td></td>
</tr>
</tbody>
</table>

→ how does the base of the logs influence scoring / ranking?
→ take document length into account
   (favour shorter documents)

→ e.g. divide by square root of document length
   (done by Lucene, via the “LengthNorm”)
Lucene’s Scoring Function

\[ \text{score}(q,d) = \sum [\text{tf}(t_d) \times \text{idf}(t) \times \text{boost}(t.f\text{ield}_d) \times \text{lengthNorm}(t.f\text{ield}_d)] \times \text{coord}(q,d) \times \text{qNorm}(q) \]

where \( q \) is the query, \( d \) a document, \( t \) a term, and:

1. \( \text{tf} \) is a function of the term frequency within the document (default: \( \sqrt{\text{freq}} \));

2. \( \text{idf} \): Inverse document frequency of \( t \) within the whole collection (default: \( \log\left(\frac{\text{numDocs}}{\text{docFreq}+1}\right) + 1 \));

3. \( \text{boost} \) is the boosting factor, if required in the query with the ”^“ operator on a given field (if not specified, set to the default field);

4. \( \text{lengthNorm} \): field normalization according to the number of terms. Default: \( \frac{1}{\sqrt{\text{nbTerms}}} \)

5. \( \text{coord} \): overlapping rate of terms of the query in the given document. Default: \( \frac{\text{overlap}}{\text{maxOverlap}} \)

6. \( \text{qNorm} \): query normalization according to its length; it corresponds to the sum of square values of terms’ weight, the global value is multiplied by each term’s weight.
3. Lucene
3. Lucene

→ choose appropriate **Analyzer** for
  – casefolding
  – stemming (wrt a given language)
  – stopping (wrt a given language)

→ insert documents *(per “field”)* into a collection and generate inverted files

→ retrieve top-K ranked documents

→ retrieve score of a document
3. Lucene

→ choose appropriate Analyzer for
  – casefolding
  – stemming (wrt a given language)
  – stopping (wrt a given language)
→ insert documents (per “field”) and generate inverted files
→ retrieve top-K docs and their scores

Lucene is a huge library

→ we use Version 5.4.0

→ most books use older Versions, e.g. Versions 4 or 3

→ the Versions are not downward compatible :-(

1999 on SourceForge
Lucene Indexing

```java
public static void insertDoc(IndexWriter i, String doc_id, String line){
    Document doc = new Document();
    doc.add(new TextField("doc_id", doc_id, Field.Store.YES));
    doc.add(new TextField("line", line, Field.Store.YES));
    try { i.addDocument(doc); } catch (Exception e) { e.printStackTrace(); }
}

public static void rebuildIndexes(String indexPath) {
    try {
        IndexWriterConfig config = new IndexWriterConfig(new SimpleAnalyzer());
        IndexWriter i = new IndexWriter(directory, config);
        i.deleteAll();
        insertDoc(i, "1", "The old night keeper keeps the keep in the town");
        insertDoc(i, "2", "In the big old house in the big old gown.");
    ... }
```
public static void insertDoc(IndexWriter i, String doc_id, String line) {
    Document doc = new Document();
    doc.add(new TextField("doc_id", doc_id, Field.Store.YES));
    doc.add(new TextField("line", line, Field.Store.YES));
    try { i.addDocument(doc); } catch (Exception e) { e.printStackTrace(); }
}

public static void rebuildIndexes(String indexPath) {
    try {
        Path path = Paths.get(indexPath);
        System.out.println("Indexing to directory " + indexPath);
        Directory directory = FSDirectory.open(path);
        IndexWriterConfig config = new IndexWriterConfig(new SimpleAnalyzer());
        IndexWriter i = new IndexWriter(directory, config);
        i.deleteAll();
        insertDoc(i, "1", "The old night keeper keeps the keep in the town");
        insertDoc(i, "2", "In the big old house in the big old gown.");
        insertDoc(i, "3", "The house in the town had the big old keep");
        insertDoc(i, "4", "Where the old night keeper never did sleep.");
        insertDoc(i, "5", "The night keeper keeps the keep in the night");
        insertDoc(i, "6", "And keeps in the dark and sleeps in the light.");
        i.close();
        directory.close();
    } catch (Exception e) { e.printStackTrace(); }
}
3. Lucene

```java
public static void rebuildIndexes(String indexPath) {
    try {
        IndexWriterConfig config = new IndexWriterConfig(new SimpleAnalyzer());
        IndexWriter i = new IndexWriter(directory, config);
    }
}
```

**SimpleAnalyzer**

→ Analyzer that filters **LetterTokenizer** with **LowerCaseFilter**

**LetterTokenizer**

→ divides text at non-letters.
→ tokens as maximal strings of adjacent letters,
  as defined by java.lang(Character.isLetter()) predicate.

Note: this does a decent job for most European languages, but does a terrible job for some Asian languages, where words are not separated by spaces.

**LowerCaseFilter**

→ Normalizes token text to lower case.
Private static TopDocs search(String searchText);

IndexReader indexReader = DirectoryReader.open(directory);
IndexSearcher indexSearcher = new IndexSearcher(indexReader);
QueryParser queryParser = new QueryParser(searchField, new SimpleAnalyzer());

Query query = queryParser.parse(searchText);
TopDocs topDocs = indexSearcher.search(query, 10000);
System.out.println("Number of Hits: " + topDocs.totalHits);
for (ScoreDoc scoreDoc : topDocs.scoreDocs) {
  Document doc = indexSearcher.doc(scoreDoc.doc); 
  System.out.println("doc_id: " + doc.get("doc_id") + ", score: " + scoreDoc.score + " [" + doc.get("line") + "]");
}

Keyword Search

Output
→ doc_id
→ score
→ content (line)
$ java Searcher "old"

Running search(old, line)
Number of Hits: 4
doc_id: 2, score: 0.5225172 [In the big old house in the big old gown.]
doc_id: 1, score: 0.36947548 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.36947548 [The house in the town had the big old keep]
doc_id: 4, score: 0.36947548 [Where the old night keeper never did sleep.]

Fig. 1. The Keeper database. It consists of six one-line documents.
$ java Searcher "old"

Running search(old, line)
Number of Hits: 4
doc_id: 2, score: 0.5225172 [In the big old house in the big old gown.]
doc_id: 1, score: 0.36947548 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.36947548 [The house in the town had the big old keep]
doc_id: 4, score: 0.36947548 [Where the old night keeper never did sleep.]

Explanation: 0.5225172 = weight(line:old in 1) [DefaultSimilarity], result of:
  0.5225172 = fieldWeight in 1, product of:
    1.4142135 = tf(freq=2.0), with freq of:
      2.0 = termFreq=2.0
    1.1823215 = idf(docFreq=4, maxDocs=6)
  0.3125 = fieldNorm(doc=1)
$ java Searcher "big old house"

Running search(big old house, line)
Number of Hits: 4
doc_id: 2, score: 1.0412337 [In the big old house in the big old gown.]
doc_id: 3, score: 0.83452004 [The house in the town had the big old keep]
doc_id: 1, score: 0.054527204 [The old night keeper keeps the keep in the town]
doc_id: 4, score: 0.054527204 [Where the old night keeper never did sleep.]

Fig. 1. The Keeper database. It consists of six one-line documents.
$ java Searcher "the"

Running search("the", line)
Number of Hits: 6

doc_id: 1, score: 0.4578294  [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.4578294  [The house in the town had the big old keep]
doc_id: 5, score: 0.4578294  [The night keeper keeps the keep in the night]
doc_id: 2, score: 0.37381613  [In the big old house in the big old gown.]
doc_id: 6, score: 0.37381613  [And keeps in the dark and sleeps in the light.]
doc_id: 4, score: 0.2643279  [Where the old night keeper never did sleep.]
$ java Searcher "the"

Running search("the", line)
Number of Hits: 8
doc_id: 8, score: 0.55138564 [The house.]
doc_id: 7, score: 0.5458439 [The house is the house.]
doc_id: 1, score: 0.47751394 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.47751394 [The house in the town had the big old keep]
doc_id: 5, score: 0.47751394 [The night keeper keeps the keep in the night]
doc_id: 2, score: 0.38988853 [In the big old house in the big old gown.]
doc_id: 6, score: 0.38988853 [And keeps in the dark and sleeps in the light.]
doc_id: 4, score: 0.27569282 [Where the old night keeper never did sleep.]

1 The old night keeper keeps the keep in the town
2 In the big old house in the big old gown.
3 The house in the town had the big old keep
4 Where the old night keeper never did sleep.
5 The night keeper keeps the keep in the night
6 And keeps in the dark and sleeps in the light.

7 The house is the house.
8 The house.

Even shorter

shorter
<table>
<thead>
<tr>
<th></th>
<th>The old night keeper keeps the keep in the town</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>In the big old house in the big old gown.</td>
</tr>
<tr>
<td>3</td>
<td>The house in the town had the big old keep</td>
</tr>
<tr>
<td>4</td>
<td>Where the old night keeper never did sleep.</td>
</tr>
<tr>
<td>5</td>
<td>The night keeper keeps the keep in the night</td>
</tr>
<tr>
<td>6</td>
<td>And keeps in the dark and sleeps in the light</td>
</tr>
</tbody>
</table>

$\text{java Searcher "the"}$

<table>
<thead>
<tr>
<th>doc_id</th>
<th>score</th>
<th>doc_id</th>
<th>score</th>
<th>doc_id</th>
<th>score</th>
<th>doc_id</th>
<th>score</th>
<th>doc_id</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>0.939</td>
<td>12</td>
<td>0.939</td>
<td>13</td>
<td>0.830</td>
<td>10</td>
<td>0.813</td>
<td>11</td>
<td>0.664</td>
</tr>
<tr>
<td>8</td>
<td>0.587</td>
<td>16</td>
<td>0.587</td>
<td>7</td>
<td>0.581</td>
<td>14</td>
<td>0.469</td>
<td>15</td>
<td>0.469</td>
</tr>
<tr>
<td>1</td>
<td>0.508</td>
<td>2</td>
<td>0.415</td>
<td>6</td>
<td>0.415</td>
<td>4</td>
<td>0.294</td>
<td>3</td>
<td>0.508</td>
</tr>
</tbody>
</table>

$\text{The house is the house.}$

$\text{In the big old house in the big old gown.}$

$\text{The old night keeper keeps the keep in the town}$

$\text{Where the old night keeper never did sleep.}$

$\text{The night keeper keeps the keep in the night}$

$\text{And keeps in the dark and sleeps in the light.}$

$\text{Where the old night keeper never did sleep.}$
SimpleAnalyzer
→ filters LetterTokenizer with LowerCaseFilter

StandardAnalyzer
→ filters StandardTokenizer with StandardFilter, LowerCaseFilter
    and StopFilter, using a list of English stop words.

StandardTokenizer
→ grammar-based tokenizer (done in JFlex), implements the Word Break rules
    from the Unicode Text Segmentation algorithm, as specified in
    Unicode Standard Annex #29.

Standard Filter
→ normalizes tokens extracted with StandardTokenizer.

StopFilter
→ removes stop words from a token stream.
The old night keeper keeps the keep in the town
In the big old house in the big old gown.
The house in the town had the big old keep
Where the old night keeper never did sleep.
The night keeper keeps the keep in the night
And keeps in the dark and sleeps in the light.

$ java Searcher "the"

Running search(the, line)
Number of Hits: 0
StandardAnalyzer – Search

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The old night keeper keeps the keep in the town</td>
</tr>
<tr>
<td>2</td>
<td>In the big old house in the big old gown.</td>
</tr>
<tr>
<td>3</td>
<td>The house in the town had the big old keep</td>
</tr>
<tr>
<td>4</td>
<td>Where the old night keeper never did sleep.</td>
</tr>
<tr>
<td>5</td>
<td>The night keeper keeps the keep in the night</td>
</tr>
<tr>
<td>6</td>
<td>And keeps in the dark and sleeps in the light.</td>
</tr>
</tbody>
</table>

$ java Searcher "the"

Running search(the, line)
Number of Hits: 0

$ java Searcher "and"

Running search(and, line)
Number of Hits: 0
StandardAnalyzer – Search

<table>
<thead>
<tr>
<th></th>
<th>The old night keeper keeps the keep in the town</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>In the big old house in the big old gown.</td>
</tr>
<tr>
<td>3</td>
<td>The house in the town had the big old keep</td>
</tr>
<tr>
<td>4</td>
<td>Where the old night keeper never did sleep.</td>
</tr>
<tr>
<td>5</td>
<td>The night keeper keeps the keep in the night</td>
</tr>
<tr>
<td>6</td>
<td>And keeps in the dark and sleeps in the light.</td>
</tr>
</tbody>
</table>

$ java Searcher “the”
Running search(the, line)
Number of Hits: 0

$ java Searcher “and”
Running search(and, line)
Number of Hits: 0

$ java Searcher “in”
Running search(in, line)
Number of Hits: 0
$ java Searcher "keeper"

Running search(keeper, line)
Number of Hits: 3

<table>
<thead>
<tr>
<th>doc_id</th>
<th>score</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.6149</td>
<td>[The night keeper keeps the keep in the night]</td>
</tr>
<tr>
<td>1</td>
<td>0.5270</td>
<td>[The old night keeper keeps the keep in the town]</td>
</tr>
<tr>
<td>4</td>
<td>0.5270</td>
<td>[Where the old night keeper never did sleep.]</td>
</tr>
</tbody>
</table>

And keeps in the dark and sleeps in the light.
$ java Searcher "keeping"

Running search(keeping, line)
Number of Hits: 0

→ stemming?
$ java Searcher "keeping"

Running search(keeping, line)
Number of Hits: 3
doc_id: 5, score: 0.614891 [The night keeper keeps the keep in the night]
doc_id: 1, score: 0.5270494 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.5270494 [The house in the town had the big old keep]

Stemming

→ EnglishAnalyzer (in the Query part)
A reconstruction of York Castle in the 14th century, showing the castle's stone keep (top) overlooking the castle bailey (below)
END
Lecture 11