COMP3220 — Document Processing and Semantic Technologies

Week 02 Lecture 1: Searching for Information

Diego Mollá

COMP3220 2021H1

Abstract

In this lecture we will survey the key technology that makes it possible to search for relevant documents in a collection of documents.

Update February 23, 2022

Contents

Reading

Essential Reading

- NLTK chapter 6 section 3.3 (precision and recall).
- Manning et al. IR book, chapter 1 (Boolean retrieval), chapter 6 section 2 (Td.idf), chapter 8 section 3 (precision and recall). `http://nlp.stanford.edu/IR-book/`

Additional Reading

- "Information Retrieval", section 23.1 of Jurafsky & Martin's book: `https://web.stanford.edu/~jurafsky/slp3/23.pdf`

1 Information Retrieval

Need for Search

The Problem

- The Web can be seen as a very large, unstructured data store.
- There exist hundreds of millions of Web pages but there is no central index.
- Even worse: It is unknown where all the Web servers are.

The Solution

Search engines

Information Retrieval

Information Retrieval (IR)

- IR is about searching for information.
- IR typically means "document retrieval".
- IR is one of the core components of Web search.

`http://boston.lti.cs.cmu.edu/classes/11-744/treclogo-c.gif`

Stages in an IR System

1: Indexing

- This stage is done off-line, prior to running any searches.
- The goal is to reduce the documents to a description: the indices.
- We want to optimise the representation: for example, ignore the terms that do not contribute.

2: Retrieval

- Use the indices to retrieve the documents (ignore the remaining information in the documents).
- We want retrieval to be fast.

2 Evaluation

Why Evaluate?

- Document processing systems almost never give 100% correct results.
- When you develop a document processing system, you want to know how good it is.
- You want to know if a modification in a system is an improvement.
- Human evaluations are expensive to produce.
- In this lecture we will focus on automatic evaluations.

Of course, *in addition* you have to debug the system.

Training *vs.* Test Data

- For pretty much all evaluations, you want to divide your data into at least two sets: training and test.
- Training data is what you use to develop your models.
  - You only look at the training data.
  - For statistical models (coming later in this course), this is what you use to calculate your statistics.
- Test data is separate, used for evaluation.
- You may also have a third set of data to help develop your system (DevTest).
  - You'll see the use of the DevTest set when we look at statistical models.

Golden Rule
You don't ever, ever, look at the test data (you only look at its evaluation results).

2.1 Precision and Recall

Positives and Negatives

- Whenever a system needs to make a binary choice, it is classifying the text into two classes: a *positive* and a *negative* class.

  Positive: What we want to select.

  Negative: What we do not want to select.
- The choice of what is a positive or a negative class depends on the application.
  - In information retrieval, documents relevant to the query belong to the positive class.
  - In spam filtering, spam documents belong to the positive class.
- We can see that the concept "positive" may not agree with our intuitions!

True, False, Positives, Negatives

We can group results of the system into four categories:

True positive (tp) : The system correctly detects a positive.

True negative (tn) : The system correctly detects a negative.

False positive (fp) : The system wrongly classifies a negative as a positive.

False negative (fn) : The system wrongly classifies a positive as a negative.

system	actual case	
decision	positive	negative
positive	tp	fp
negative	fn	tn

Example: Positives and Negatives in Information Retrieval

In IR, "relevant" documents belong to the positive class.

system	actual case	
decision	relevant	not relevant
retrieved	tp	fp
not retrieved	*fn*	tn

- If our retrieval system fails to retrieve a relevant document, this is a *false negative*.

Example: Spam Filtering

In spam filtering, "spam" emails belong to the positive class.

	actual case	
system	spam	not spam
marked spam	tp	*fp*
not marked spam	fn	tn

- If our spam filter classifies a legitimate email as spam, this is a *false positive*.

Question

False positives in spam filtering are usually more dangerous than false negatives; why?

Precision and Recall

Formulas

- precision $= \frac{\text{tp}}{\text{marked as positive by the system}} = \frac{\text{tp}}{\text{tp+fp}}$
- recall $= \frac{\text{tp}}{\text{should be marked as positive}} = \frac{\text{tp}}{\text{tp+fn}}$

Example

From a total collection of 200 documents, a retrieval system returned 30 documents, but 5 were not relevant. It also missed 12 documents.

Example

	actual case	
system	relevant	not relevant
retrieved	25	5
not retrieved	12	158

Values of measures

- precision $= \frac{25}{25+5} = \frac{25}{30}$
- recall $= \frac{25}{25+12} = \frac{25}{37}$

F-Measure

- The F-measure combines precision and recall into a single measure.

$$F_\beta = (1+\beta^2)\frac{\text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}$$

- The most common combination is when $\beta = 1$, referred to as F_1 :

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

- This is the harmonic mean of precision and recall.
- For our previous example, $F_1 = 0.746$

Accuracy

- Accuracy is the number correctly classified out of the whole set.
  - accuracy $= \frac{\text{correct decisions}}{\text{all data}} = \frac{\text{tp+tn}}{\text{tp+fp+tn+fn}}$
  - For previous example, accuracy is 183/200
- Sometimes used (wrongly) to refer to precision.

Beware of unbalanced data

What happens if you have unbalanced classes, e.g. there are 100 documents, 90 of them belong to the negative class, and the system classifies everything as a negative?

- Recall: $\frac{0}{10} = 0$
- Precision: $\frac{0}{0} = \text{NAN}$
- Accuracy: $\frac{90}{100} = 0.9$

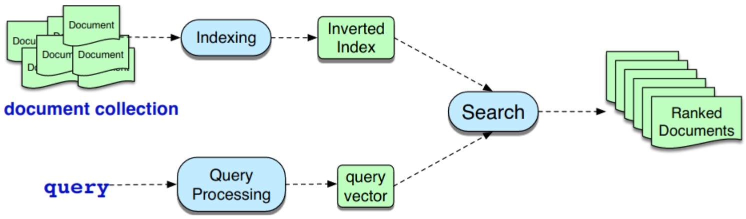
Exercise: Spam Filtering

Exercise

Assume your system processes 1000 emails. It classifies 640 as spam, of which 480 are actually spam. It missed 120 spam emails. What are the precision and recall of the spam detection? What is the accuracy?

3 Indexing and Retrieval

Architecture of an Information Retrieval System



(Jurafsky & Martin, Fig. 23.1)

3.1 Indexing

Bag of Words Representation

Bag of words (BoW)

- At indexing time, a compact representation of the document is built.
- The document is seen as a bag of words.
- Information about word position is (often) discarded.
- Only the important words are kept.

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. Recently, the bag-of-words model has also been used for computer vision. $\Longrightarrow$

{bag, bag-of-words, computer, disregarding, document, grammar, information, IR, keeping, language, model, multiplicity, multiset, natural, order, processing, representation, represented, retrieval, sentence, simplifying, text, vision, word, words}

Stop Words

Stop words

- A simple solution to determine important words is to keep a list of non-important words: the stop words.
- All stop words in a document are ignored.
- Stop words are language-specific.
- Typically, stop words are connecting words.

Stop words in NLTK

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> stop[:5]
['i', 'me', 'my', 'myself', 'we']
```

Note that NLTK's list of stop words has words in lowercase.

Term Frequency

Term Frequency

- Usually, words that are not frequent are not important.
- Words that are too frequent may occur in most documents and therefore can't be used to discriminate among documents.
- *Usually, important words are in the middle.*

Zipf's Law for term frequency

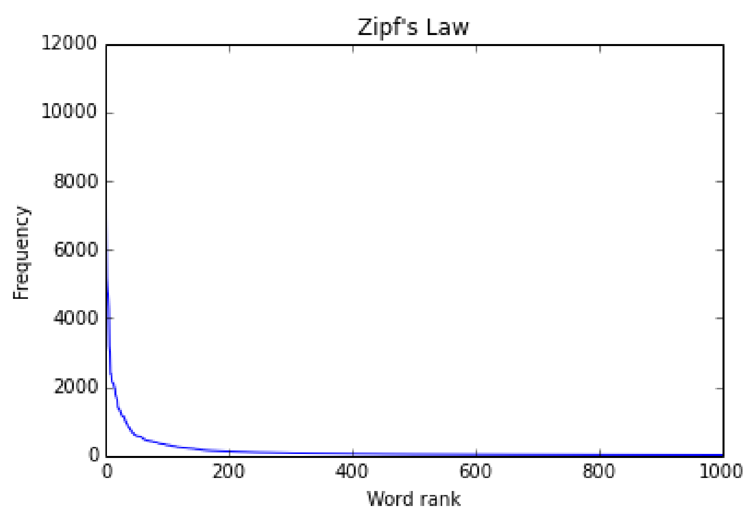
- A small percentage of words are very frequent.
- A large percentage of words have very little frequency.
- The relation approximates a Zipfian distribution.
- This is also referred as a "long-tailed" distribution.

Zipf's Law in Action

Python code

```
import nltk
import collections
import matplotlib.pyplot as plt
words = nltk.corpus.gutenberg.words('austen-emma.txt')
fd = collections.Counter(words)
data = [f for w, f in fd.most_common()]
plt.plot(data[:1000])
plt.show()
```

500 most frequent words



tf.idf

tf.idf

- Term frequency: If a word is very frequent in a document, it is important for the document.

$$tf(t,d) = \text{frequency of word } t \text{ in document } d$$

- Inverse document frequency: If a word appears in many documents, it is not important for any of the documents.

$$idf(t) = \log \frac{\text{number of documents}}{\text{number of documents that contain } t}$$

- $tf.idf$ combines these two characteristics.

$$tf.idf(t,d) = tf(t,d) \times idf(t)$$

Note that tf is a function of the term and the document, whereas idf is a function of the term, irrelevant of the document. To compute $tf.idf$ we need to have a collection of documents, otherwise idf is irrelevant.

Problems with Bag of Word Representations

BoW representations ignore important information such as:

Word position: "Australia beat New Zealand" is not the same as "New Zealand beat Australia"

Morphology: If you search for "table", a webpage that uses the word "tables" might be relevant.

Words with similar meanings: If you search for "truck", a webpage that uses the word "lorry" might be relevant.

Ambiguity: If you search for "Apple" you might be interested in the company and not in the fruit.

Still, BoW representations are very simple, fast, and often surprisingly good.

Beyond BoW Representations

- A simple way to account for (some) information about word positions is to use n-grams:
  - Bigrams, trigrams, 4-grams (usually there is no need for longer n-grams).
- Thus, instead of representing a text as a bag of words, it can be represented as a bag of n-grams.
- Using n-grams instead of words may introduce other kinds of problems.
  - We will see some of these problems in a future lecture.

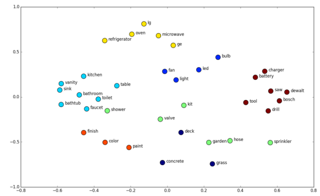
Accounting for Word Meaning

Ambiguity

- Word disambiguation attempts to determine the sense of a word.
- A word like "Apple" could be disambiguated as "apple1" or "apple2" to account for its several meanings.
- Word disambiguation systems usually look at the "context" of the word:
  - Yesterday I ate an apple$_1$.
  - Apple$_2$ reported a benefit last fiscal year.

Synonymy

- There are lexical resources such as thesauri (singular: thesaurus) that list words with related meanings.
  - WordNet is a popular resource
- Recent innovations include the use of distributional semantics to map words to vectors of numbers called word embeddings.
  - Word2Vec and Glove are two popular early systems.



`https://neptune.ai/blog/word-embeddings-guide`

Inverted Indices

Index

D1
computer
language
software
information

D2
computer
retrieval
document
library

D3
computer
filtering
information
retrieval

Inverted Index

computer
D1 D2 D3

software
D1

information
D1 D3

language
D1

document
D2

retrieval
D2 D3

library
D2

filtering
D3

Inverted indices are called like this because, in a sense, they do the inverse of a normal index. An index would indicate all the words listed in a document. In contrast, an inverted index indicates all the documents that contain a particular word. In the example above:

- index(D1) = { computer, language, software, information }.
- inverted_index(computer) = { D1, D2, D3 }.

3.2 Boolean Retrieval

Retrieval

- In the retrieval stage, the index is searched.
- This enables fast retrieval.
- Note that the index does not contain the full information from the documents.
- For example, searching a stop word will be useless.

Boolean Retrieval

- Use Boolean operations among the search terms.

  x AND y Documents that contain both terms.

  x OR y Documents that contain at least one term.

  NOT x Documents that do not contain the term.

- The use of inverted indices simplifies this method.

  x AND y Set intersection.

  x OR y Set union.

  NOT x Set complement.

Example of Boolean Retrieval

Keywords
D1:{computer, software, information, language}
D2:{computer, document, retrieval, library}
D3:{computer, information, filtering, retrieval}

Inverted Index
computer → {D1, D2, D3}, software → {D1}, information → {D1,D3},
language → {D1}, document → {D2}, retrieval → {D2, D3},
library → {D2}, filtering → {D3}

Boolean Query
(information OR document) AND retrieval

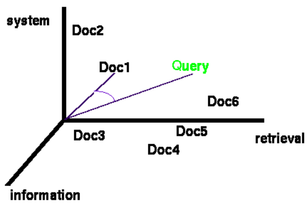
Result
$(\{D1,D3\} \cup \{D2\}) \cap \{D2,D3\} = \{D2,D3\}$

3.3 Vector Retrieval

Vector Retrieval

Boolean retrieval and ranking

- There are no obvious methods to rank the results of Boolean retrieval.
  - A very successful attempt to handle this is Google's PageRank but we will not cover it here.
- An easy method to rank documents is to represent them as vectors and use well-established methods for vector comparison.



From Documents to Vectors

- We can define a document as a vector where each word in the entire vocabulary represents an element in a vector.
- The final document matrix will typically be **sparse** since a document will typically contain only a small fraction of all the possible words.
- Possible information to store in the vector:
  - The occurrence of a word/stem/n-gram (1) or not (0) ← see example below.
  - The word frequency.
  - $tf.idf$ ← a popular choice.
  - Distributional semantics ← a hot research topic.

Example of Vector Space Model

Template:

{*computer,software,information,document,retrieval,language,library,filtering*}

Initial documents
D1:{computer, software, information, language}
D2:{computer, document, retrieval, library}
D3:{computer, information, filtering, retrieval}

Document vectors
D1: (1,1,1,0,0,1,0,0)
D2: (1,0,0,1,1,0,1,0)
D3: (1,0,1,0,1,0,0,1)

Document matrix
(typically a **sparse matrix**)

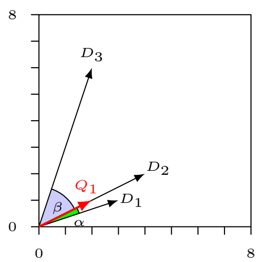
$$D = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The document matrix is typically a sparse matrix because often a document will contain only a small fragment of the words listed in the template. Consequently, most elements in the matrix are zero. Our example is contrived, it does not reflect real documents, so the resulting document matrix is not sparse.

Cosine Similarity

Cosine Method

- This is a popular approach to compare vectors.
- We calculate the cosine of the angle between vectors.
- If the angle is zero, then the cosine is 1.



$$\begin{aligned} \cos(D_1, Q_1) &= \cos(\alpha) \\ \cos(D_2, Q_1) &= \cos(0) = 1 \\ \cos(D_3, Q_1) &= \cos(\beta) \end{aligned}$$

Cosine Similarity: Formulas

General Formula

$$\cos(D_j, Q_k) = \frac{\Sigma_{i=1}^{N} D_{j,i} Q_{k,i}}{\sqrt{\Sigma_{i=1}^{N} D_{j,i}^2}\sqrt{\Sigma_{i=1}^{N} Q_{k,i}^2}} = \frac{D_j \cdot Q_k}{||D_j||_2 \, ||Q_k||_2}$$

If the vectors are normalised

$$\cos(D_j, Q_k) = \Sigma_{i=1}^{N} D_{j,i} Q_{k,i} = D_j \cdot Q_k$$

The terms at the rightmost side of the equations use vector operations: $A \cdot B$ is the scalar ("dot") product of vectors A and B, and $||A||$ is the Euclidean norm of A (also called l^2 norm).

Below is an example with two vectors of dimension 3. If D_j is the vector $(d_1, d_2, d3)$ and Q_k is the vector (q_1, q_2, q_3), then the cosine similarity is:

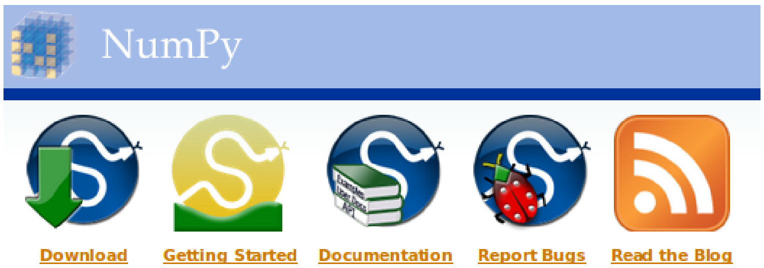
$$\frac{d_1 \times q_1 + d_2 \times q_2 + d_3 \times q_3}{\sqrt{d_1^2 + d_2^2 + d_3^2} \times \sqrt{q_1^2 + q_2^2 + q_3^2}}$$

3.4 Vector Retrieval in Python

Vectors and Matrices in Python

numpy

- Python's numpy is a collection of libraries that include manipulation of vectors and matrices.
- `http://www.numpy.org/`
- It's pre-loaded in the Anaconda distribution.



Manipulating Vectors

```
>>> import numpy as np
>>> a = np.array([1,2,3,4])
>>> a[0]
1
>>> a[1:3]        # slicing
array([2, 3])
>>> a+1           # add a constant to a vector
array([2, 3, 4, 5])
>>> b=np.array([2,3,4,5])
>>> a+b           # add two vectors
array([3, 5, 7, 9])
>>> a*b           # pairwise multiplication
array([ 2,  6, 12, 20])
>>> np.dot(a,b) # dot product between vectors, a . b
40
```

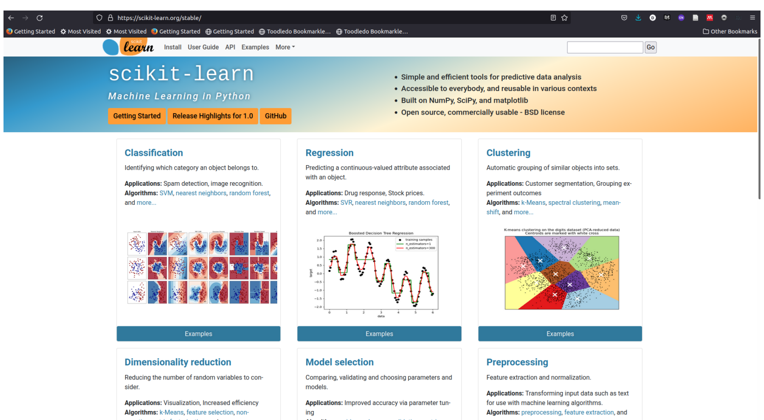
Manipulating Matrices

```
>>> x = np.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> y = np.array([[1,1,1],[2,2,2]])
>>> x+y              # add two matrices
array([[2, 3, 4],
       [6, 7, 8]])
>>> x*y              # pairwise multiplication
array([[ 1,  2,  3],
       [ 8, 10, 12]])
```

```
>>> x.T              # transpose
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.dot(x.T,y)  # dot product
array([[ 9,  9,  9],
       [12, 12, 12],
       [15, 15, 15]])
>>>
```

Scikit-learn

- `http://scikit-learn.org/`
- Incorporates an extensive set of machine learning algorithms into Python.
- It has a consistent and intuitive interface.
- The documentation is very complete.
- Includes generic tutorials on the main machine learning algorithms.
- To install Scikit-learn:
  - `https://scikit-learn.org/stable/install.html`



tf.idf with scikit-learn

```
>>> import glob
>>> files = glob.glob('enron1/ham/*.txt')
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(input='filename',stop_words='english')
>>> tfidf.fit(files)
>>> tfidf_values = tfidf.transform(files)
>>> len(tfidf.get_feature_names_out())
19891
>>> tfidf.get_feature_names_out()[10000:10005]
array(['grandpa', 'grandsn', 'grandsons', 'grant', 'granted'],
dtype=object)
```

```
>>> type(tfidf_values)
scipy.sparse.csr.csr_matrix
>>> type(tfidf_values.toarray())
numpy.ndarray
>>> tfidf_values.shape
(3672, 19891)
```

In this code, "tfidf" is an instance of the "TfidfVectorizer" class, which will accept a list of text files and will ignore stop words.

The "fit" method gathers important information from the data, such as the vocabulary, and the inverse document frequencies.

The "transform" method computes tf.idf of the files. When we do fit and transform using the same files, as is the case here, you can combine "fit" and "transform" using "fit_transform".

The output of "transform" (and "fit_transform") is a sparse matrix. Scikit-learn provides many functions that operate with sparse matrices, so often we will not need to convert sparse matrices to regular arrays. If we need to convert a sparse matrix to an array we can use the "toarray" method.

The method "shape" returns the dimensions of the array or sparse matrix. In our case, the output of the function says that the variable "tfidf_values" has 3,672 rows (one row per file) and 19891 columns (one column per distinct word).

Normalised tf.idf and cosine similarity in Python

```
>>> tfidf_norm = TfidfVectorizer(input='filename',
                                 stop_words='english',
                                 norm='l2')
>>> tfidf_norm_values = tfidf_norm.fit_transform(files).toarray()
>>> import numpy as np
>>> def cosine_similarity(X,Y):
        return np.dot(X,Y)
>>> cosine_similarity(tfidf_norm_values[0,:],
                      tfidf_norm_values[1,:])
0.017317648885111028
```

In the code above, note that the first character of 'l2' is lowercase L (not the number 1).

This code produces normalised tf.idf so that the computation of the cosine similarity is faster. To normalise tf.idf we use the l^2 norm, which is the standard Euclidean norm.

Scikit-learn provides an implementation of cosine similarity. For details, look at the documentation page at `http://scikit-learn.org/stable/modules/metrics.html`.

Some Open Source Search Engines

If you don't want to implement your search engine from scratch, try these (`http://www.mytechlogy.com/IT-blogs/8685/top-5-open-source-search-engines/`):

1. Elasticsearch: `https://www.elastic.co`
2. Solr: `https://solr.apache.org`
3. Lucene: `https://lucene.apache.org`
4. Sphinx: `http://sphinxsearch.com`
5. Xapian: `https://xapian.org`
6. Indri: `https://www.lemurproject.org`

7. Zettair: `http://www.seg.rmit.edu.au/zettair`

The following is a Python library that can be used for indexing and retrieving documents (among many other things):

1. Gensim: `https://radimrehurek.com/gensim/`

Take-home Messages

1. What is indexing? what is retrieval?
2. What is an inverted index?
3. Perform Boolean retrieval by hand.
4. Implement Boolean retrieval in Python.
5. Use sklearn to build a vector model with tf.idf.
6. Use sklearn to implement cosine similarity.

What's Next

Week 3

- Introduction to Statistical Classification.
- *Submit Assignment 1 by Friday week 3.*

Reading

- NLTK Chapter 6 "Learning to Classify Text".