Text Indexing

Andrea Esuli

Text indexing

The term *Indexing* usually identifies the process that transforms plain text into a representation that is better manageable by algorithms.

The actual implementation of the indexing process depends on the actual algorithm that will use the indexed output.

E.g., in web search, the main purpose of indexing is to identify the words the text is composed of so as to populate an *inverted index* for efficient retrieval.

Documents:
d1 = a d c g d e
d2 = c b d a c d
d3 = e f c b d
d4 = e c a g e a
d5 = f g c b

Inverted index:
La -> d1 d2 d4
Lb -> d2 d3 d5
Lc -> d1 d2 d3 d4 d5
Ld -> d1 d3
Le -> d1 d3 d4
Lf -> d3 d5

Queries:
a AND d = intersect(La,Ld) = d1
b OR e = union(Lb,Le) = d1 d2 d3 d4 d5

Text indexing

Most machine learning algorithms cannot directly use plain text as input, as they are designed to work with different representations of input information, e.g., probability distribution, vectors, sequences of words (plain text is a sequence of characters, more on this in the following).

The indexing process is in charge to recognize and transform any relevant information contained in text in the format suitable for the specific algorithm that will be used.

Indexing is not just a mechanical transformation, it is a **knowledge engineering** task, in which the expert may use and combine many tools (from IR, NLP, and ML itself) to produce the best possible input data for the successive elaboration steps.

Strings

Text representation

How do computers “see” text?

A piece of text is stored in a **string**, i.e., a **sequence of characters**:

```
In : text = 'piece of text'
In : text
Out: 'piece of text'
In : text [0]
Out: 'p'
In : text [0:3]
Out: 'pie'
In : len(text)
Out: 13
```

Strings

A string is an **immutable** ordered sequence of **Unicode characters**.

A string is defined by either using single quotes or double quotes.

```
a = 'this is a test' # or "this is a test"
```

Triple double quotes define multiline strings.

```
a = """This is a
multiline string"""
Out: 'This is a\nmultiline string'
```

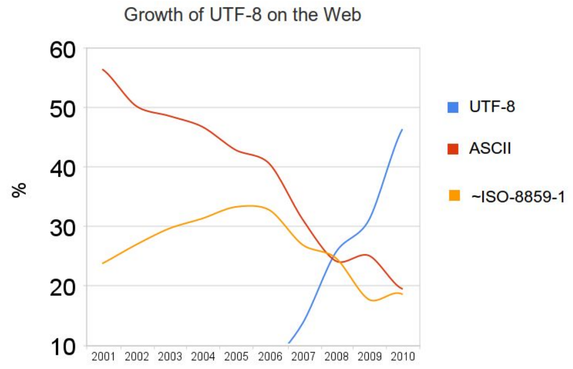
Escape sequences allows to put quotes, newlines, tabs, or other non-trivial chars in strings.

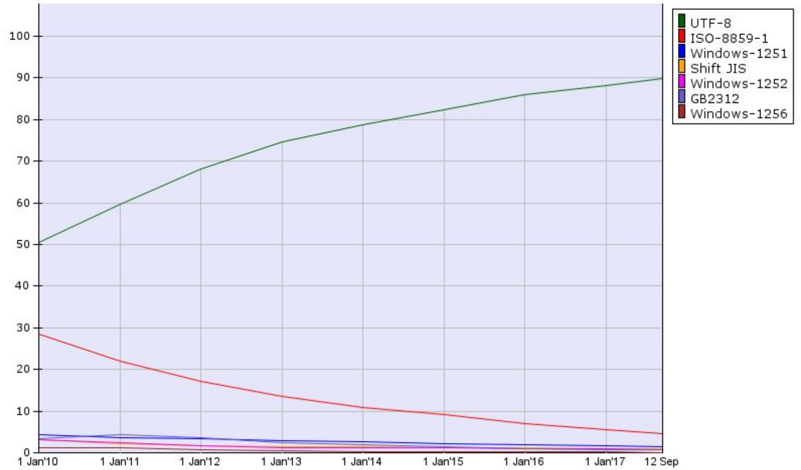
Encodings

Unicode is a standard (dating back to 1988) for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.

UTF-8 is the most common format adopted for the encoding of unicode characters.

UTF-8 is backward compatible with ASCII.





Usage of character encodings for websites, 12 Sep 2017, W3Techs.com

String operations

Strings offer a number of text-oriented operations.

```
capitalize, encode, format, isalpha, islower, istitle, lower,
replace, rpartition, splitlines, title, casefold, endswith,
format_map, isdecimal, isnumeric, isupper, lstrip, rfind, rsplit,
startswith, translate, center, expandtabs, index, isdigit,
isprintable, join, maketrans, rindex, rstrip, strip, upper, count,
find, isalnum, isidentifier, isspace, ljust, partition, rjust,
split , swapcase, zfill
```

From text to **features**

Any **observable information** from text defines a **feature**.

The expert models some heuristics that are deemed to be relevant for the task of turning text into features (*feature engineering* process), e.g.:

- The actual words from text
- Linguistic information
  - Morphology (repeated characters: `good, gooooood,` capitalization: `great, Great, GREAT, GrEaT`)
  - `Part of speech (nouns, verbs, adjectives)`
  - `Parse tree (subjects, objects)`
  - `Entities (from NER or from lexica)`
- `Meta information`
  - `Position in the document, enclosing tag…`

The ML model will determine which are the relevant ones for the task.

Text representation

Strings do not model the concept of word.

```
In : text[2:7]
Out: 'ece o'
```

A simple way to get a list of words (tokens) from text is to split them on spaces:

```
In : words = text.split(' ')
In : words
Out: ['piece', 'of', 'text']
In : words[0]
Out: 'piece'
```

How to deal with punctuation, numbers, links, emails, HTML tags, hashtags, mentions, and the like?

Regular expressions

Regular expressions

A regular expression is a **search pattern**.

Regular expressions are used to find matching patterns in text and to extract relevant substrings from text.

The re module defines objects and methods to apply regular expressions to strings.

Regular expressions are defined as strings that follow a specific syntax.

'[A-Z][a-z]{3}' = *match a sequence of any capital letter followed by exactly three lower-case letters, e.g., 'Pisa'*

Regular expressions

Basic matching

'a'	= character a
'abc'	= string abc
'a|b'	= match a or b
'a*'	= zero or more a
'a+'	= one or more a
'a?'	= zero or one a
'a{3}'	= exactly 3 a
'a{2,5}'	= from 2 to 5 a (the more the better)
'a{2,5}?'	= from 2 to 5 a (the less the better)
'a{4,}'	= at least 4 a
'a{,3}'	= at most 3 a

Regular expressions

Other common matches

^	= start of string
$	= end of string
\n	= newline
\r	= carriage return
\t	= tab

Regular expressions

Characters classes

[abc]	= one in set a,b,c
[a-z0-9]	= one in set of character from a to z and from 0 to 9
[^a-z]	= one character but not those from a to z
\d	= one digit character
\D	= one non-digit character
\s	= one white space character
\S	= one non white space character
\w	= one word character (e.g. a-z A-Z 0-9 _)
\W	= one non-word character
.	= any character

Regular expressions

Match a specific string : 'word'

Match any four characters word: '[a−z]{4}'

Match any sequence of digits : '[0−9]+'

Match any email address: '([ˆ@\s]+@[ˆ@\s]+\.[ˆ@\s]+)'

Regular expressions

Groups allow to identify parts of a regular expression that are of interest.

'(abc)'	= group, sequence of characters abc
'(abc)+'	= one or more time the sequence
'(?P<name>...)'	= group named "name"
'(?P=name)'	= match the content of the group with that name
'(?:...)'	= non capturing (just to define a sequence)
'(?!abc)'	= 'abc' must NOT be present in the string (in the position where this pattern appears).

Start exploring regular expressions here, here, and here.

Regular expressions

Compilation allows efficient reuse of regular expressions, and a clean separation between their definition and their use.

```
tagre = re.compile('<(?P<tag>.+)>(?P<text>.*?)</(?P=tag)>')

tagged = tagre.match('<pre>Ciao</pre>')

tagged['tag']
Out: 'pre'

tagged['text']
Out: 'Ciao'
```

Recognition of special entities

Different entities can be recognized using different regular expressions:

```
mentionre = re.compile(r'@[\w]+')

hashtagre = re.compile(r'#[\w]+')

urlre = re.compile(r'https?://[^\s"]+')

emoticonre = re.compile(r'[:;=8]-?[)(/O*\P]')
```

When to use/not use regular expressions?

Regular expressions are a powerful tool, yet they should be used only for low-level matching tasks (e.g. specifically formatted labels and codes), and not be used when the desired match is defined by a high-level concept and when dedicated tools are available, e.g.:

- Parse HTML/XML with a proper parser (e.g., beautiful soup)
- Tokenize text with a proper tokenization model
- Recognize entities with a trained NER model

Text from the Web

Getting text from the Web

The urllib package implements methods that enable low level interaction with Web servers.

Getting any Web page content is typically simple:

```
from urllib import request
url = "http://www.esuli.it/"
response = request.urlopen(url)
html = response.read().decode('utf8')
html
```

```
Out: '<!doctype html>\n\n<head>\n    <meta charset="utf-8">\n
<title>esuli.it</title>\n    <meta name="description"… '
```

Extracting (clean) text from the Web

The BeautifulSoup package implements methods to navigate, modify, and extract data from HTML and XML data.

It can be used on Web pages to extract information of interest.

```
from urllib import request
url = "http://www.esuli.it"
response = request.urlopen(url)
html = response.read().decode('utf8')
```

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, "html5lib")
```

`soup.get_text()` ← *simple way or* ↓ *more control*

```
[''.join(s.findAll(text=True))for s in soup.findAll('p')]
```

NLTK

NLTK

NLTK is an open source library that support quick development of NLP applications.

NLTK comes with an online book that has guided examples for many tasks.

Typical installation steps:

```
>conda install nltk
>python
```

```
In : import nltk
In : nltk.download()
```

```
(pick option to download popular packages)
```

Tokens & Sentences

Tokenization

One of the basic operation on text is to identify the words (tokens) composing it.

NLTK provides a basic function to perform a *language aware* tokenization.

```
> from nltk import word_tokenize
> tokens = word_tokenize(text)
> tokens[1000:1050]
```

```
Out: ['with', 'you', 'and', 'you', 'are', 'no', 'longer', 'my',
'friend', ',', 'no', 'longer', 'my', '‘', 'faithful', 'slave',
',', '’', 'as', 'you', 'call', 'yourself',… ]
```

Sentence splitting

Splitting a text into its sentences is not always a trivial task.

Use of punctuation is the most relevant clue, yet its use to report other information (acronyms, numbers, initials...) can produce erroneous splits.

I'll take the 8.30 train to St. Louis... or maybe I'll stay in LA. Time will tell!

Which of the above dots separate sentences?

The `sent_tokenize` function implements a language-aware sentence splitter.

Tokenization

The NLTK tokenize package implements tokenizers and splitters for a number of different cases:

- twitter text (very informal text, and platform specific style)
- multi-word expressions (providing a list of MWE in input)
- based on regular expressions
- based on Stanford CoreNLP tool
- other dataset-specific tokenizers

Some basic explorations

Given a tokenized text, we may want to extract some information out of it:

- build a vocabulary of the terms used in it
- count frequency of use of each term
- plot the distribution of frequencies across terms.
- see where terms appear across the text
- see in which contexts terms are used

The Text and FreqDist objects from NLTK are two simple tools to perform this tasks.

POS tagging

POS tagging

POS tagging is the process of assigning each term in text to its Part Of Speech.

POS identify large classes of terms with similar syntactic roles in language: noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, numeral, determiner.

POS tagging can also identify more precise syntactic classes, such as common names, proper names, the mode of verbs.

NLTK's pos_tag function implements a POS tagger for English and Russian.

RDRPOSTagger provides pretrained POS tagging model for 40+ languages.

POS tagging

Marking tokens with their part of speech (PoS) can reduce ambiguity.

'I $\texttt{saw}_1$ a bird'

'Can you lend me a $\texttt{saw}_2$?'

semantic($\texttt{saw}_1$) != semantic($\texttt{saw}_2$)

POS tagging

Marking tokens with their part of speech (PoS) can reduce ambiguity.

```
In : import nltk
In : text1 = 'I saw a bird.'
In : text2 = 'Can you lend me a saw?'

In : token1 = nltk.word_tokenize(text1)
In : token2 = nltk.word_tokenize(text2)

In : token1, token2
Out: (['I', 'saw', 'a', 'bird', '.'], ['Can', 'you', 'lend', 'me', 'a', 'saw', '?'])

In : nltk.pos_tag(token1), nltk.pos_tag(token2)
Out: [('I', 'PRP'), ('saw', 'VBD'), ('a', 'DT'), ('bird', 'NN'), ('.', '.')]
Out: [('Can', 'MD'), ('you', 'PRP'), ('lend', 'VB'), ('me', 'PRP'), ('a', 'DT'),
('saw', 'NN'), ('?', '.') ]
```

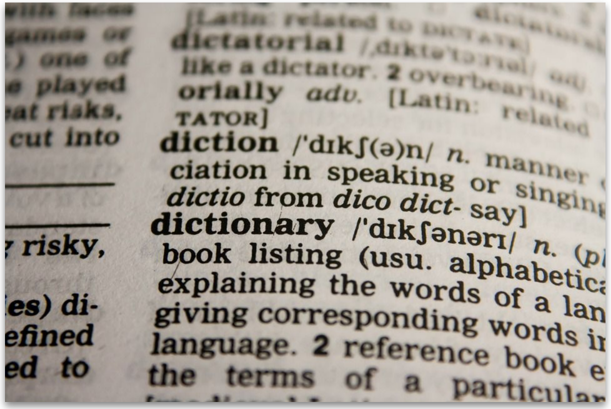
Stems & Lemmas

Stemming and lemmatization

Stemming and lemmatization aim at reducing the different inflections of a words to its root.

Stemming does it by applying a set of language-dependent word transformation rules, which can result in a lexically incorrect or non-meaningful word.

Lemmatization use a deeper, and costlier, NLP analysis to reduce a word to its dictionary form.



Stemming and lemmatization

Stemming:

```
In : from nltk.stem.porter import PorterStemmer

In : stemmer = PorterStemmer()

In : stemmer.stem('cars')

Out: 'car'

In : stemmer.stem('was')

Out: 'wa'
```

Stemming and lemmatization

Lemmatization:

```
In : from nltk.stem.wordnet import WordNetLemmatizer
In : lmtzr = WordNetLemmatizer()

In : lmtzr.lemmatize('cars')
Out: 'car'

In : lmtzr.lemmatize('was') # needs POS
Out: 'wa'

In : lmtzr.lemmatize('was', pos='v')
Out: 'be'
```

Bag of Words

From list of words...

Features are extracted from text in a list, possibly with repetitions.

```
In : text = 'the president of the united states of america'
In : feats = word_tokenize(text)
In : len( feats )
Out: 8

In : feats
Out: ['the', 'president', 'of', 'the', 'united', 'states', 'of',
'america']
```

Representing text with that **list of variable length** can be a problem for many ML algorithms.

...to Bag of Words

The most frequently adopted model is the Bag of Words (BOW) model, i.e., a document is represented by the set of words it is composed of.

```
In : bow = set(feats)

In : len(bow)
Out: 6

In : bow
Out: ['america', 'of', 'president', 'states', 'the', ' united']
```

...to Bag of Words

BOW loses information on word frequency.

> Counts of word occurrences in every document can be stored in additional data structures.

Similarly to unigram models, BOW loses word order information.

> N-grams can be used also in this case.

BOW representation has a fixed length, easier to manage by ML algorithms.

> The set of all the distinct extracted features may be called 'vocabulary', 'dictionary', 'feature set, 'feature space'.

Bag of Words

BOW loses the frequency and word order information.

```
In : t1 = set(word_tokenize('I won, and thus you lose.') )

In : t2 = set(word_tokenize('I lose, and thus you won.'))

In : t1
Out: ['and', 'lose', 'thus', 'won', 'you']

In : t2
Out: ['and', 'lose', 'thus', 'won', 'you']

In : t1==t2
Out: True
```

N-grams

Word n-grams

Information on positional relations can be extracted with *Word n-grams* features, which capture local word order.

```
In : t1 = set(nltk.ngrams(nltk.word_tokenize('I won, and thus, you lose.'),2))
In : t2 = set(nltk.ngrams(nltk.word_tokenize('I lose, and thus, you won.'),2))
In : t1, t2, t1==t2

Out: {'W2G_and_thus', 'W2G_thus_you', 'W2G_won_and', 'W2G_you_lose', 'and', 'lose',
'thus', 'won', 'you'},
Out: {'W2G_and_thus', 'W2G_lose_and', 'W2G_thus_you', 'W2G_you_won', 'and', 'lose',
'thus', 'won', 'you'}
Out: False
```

Feature format ('W2G_...') doesn't matter as long as there is no ambiguity between features extracted by different methods.

Character n-grams

Character n-grams mitigate the effect of typos.

```
In : t1 = set(nltk.ngrams('rainbow',3))
In : t2 = set(nltk.ngrams('rainbaw',3))
In : t1, t2

Out:{'C3G_a_i_n', 'C3G_b_o_w', 'C3G_i_n_b', 'C3G_n_b_o',
'C3G_r_a_i', 'rainbow'}
Out:{'C3G_a_i_n', 'C3G_b_a_w', 'C3G_i_n_b', 'C3G_n_b_a',
'C3G_r_a_i', 'rainbaw'}

In : t1.intersection(t2)

Out: {'C3G_a_i_n', 'C3G_i_n_b', 'C3G_r_a_i'}
```

Collocations

Collocations & Terminology Extraction

Some words are used together to form expressions that have a semantic that's different from the simple composition of the original words.

```
"strong tea", *"powerful tea",

"break a leg", "star wars", "formula 1"
```

A simple automatic recognition of collocations and multi-word terms can be done by observing statistical anomalies in the co-occurrence of words with respect to an assumption of independence.

Collocations & Terminology Extraction

Collocation extraction methods are implemented in both NLTK and GenSim.

GenSim implementation uses the following formula to rank token pairs by their relevance as collocations.

$$\mathrm{score}(w_i, w_j) = \frac{\mathrm{count}(w_i w_j) - \delta}{\mathrm{count}(w_i) \times \mathrm{count}(w_j)}$$

$\mathrm{count}(w_i)$ (or w_j) counts how many time w_i (or w_j) appears.
$\mathrm{count}(w_i w_j)$ counts how many time w_i and w_j appear together.

The higher the ratio, the more probable the use of the expression $w_i w_j$ has a different meaning than the simple juxtaposition of w_i and w_j

Collocations & Terminology Extraction

GenSim's collocation extraction method can be applied iteratively, to extract collocations composed of more than two words.

Functional words (determiners, conjunctions, prepositions, pronouns, auxiliary verbs…) may be ignored when considering adjacency of words, so as to be able to capture longer collocations, e.g.,

"President *of the* United States."

Without considering the high frequency of terms "of" and "the" in the computation.

NLP tools: Chunking, NER, & Parsing

Chunking

Chunking (also called *shallow parsing*) consists in the grouping of words in a sentence into short phrases that form a syntactic/semantic unit, e.g., a noun phrase.

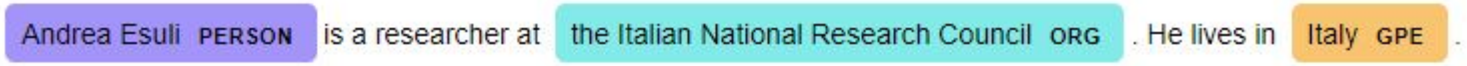
It is a in intermediate complexity task between *POS tagging* and *parsing*.

She sells sea shells by the sea shore

NER

Named entity recognition is an information extraction process that aims at recognizing expressions denoting entities in text and classifying them by the type of entity:

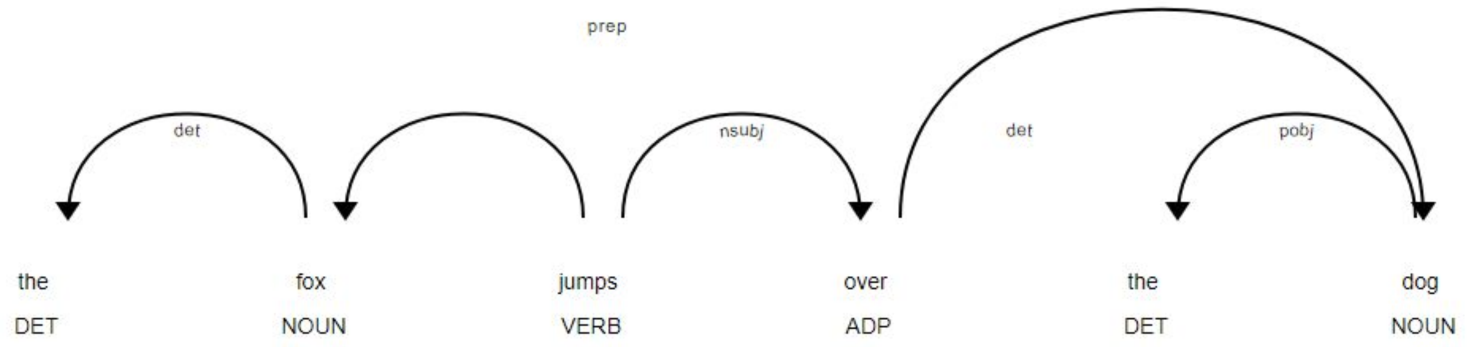
- Person: Andrea Esuli, Elon Musk, Italian President,
- Geo-political entity: Italy, Spain, United States, Pisa, Paris
- Organization: Italian National Research Council, University of Pisa
- Other (events, nationalities…)


Andrea Esuli PERSON is a researcher at the Italian National Research Council ORG . He lives in Italy GPE .

Parsing

Parsing consists in the recognition in a sequence words of their syntactic/semantic relations according to the grammar of the language.

The result of dependency parsing is typically a parse tree, rooted in the main verb of the sentence.



spaCy

spaCy

spaCy implements a rich range of NLP tools for the many languages:

English, German, Spanish, Portuguese, French, Italian, Dutch...

and also a multi-language model that is able to handle multiple languages at the same time, but only to perform Named Entity Extraction.

A piece of text is passed to a spaCy model, that returns an object which has method to access many different NLP outputs.

```
nlp = spacy.load('en')
processed_text = nlp(text)
```

spaCy

Installation:

```
>conda install spacy
```

Select the languages you need and download relative the models:

```
    import spacy.cli
    spacy.cli.download("en")
    spacy.cli.download("it")
    spacy.cli.download(...)
```

If the above fails try using command line (if the following command fails too try running it as administrator):

```
>python -m spacy download en
>python -m spacy download it
>python -m spacy download ...
```

Too much features

Too much features

It is up to the ML expert to define the set of features (*feature engineering*)

Features can easily grow to an unmanageable size.

A trade off between information richness and computational cost must be met.

The 9,000 training documents of the Reuters 21578 collection (a small standard dataset for ML experiments) contains 20,123 distinct words (numbers excluded).

- Adding word bi-grams pushes the number of features over 100k.
- The distribution of words in text follows a Zipf law.
  313 words generate half of the 500k occurrences in Reuters 21578.

Zipf law

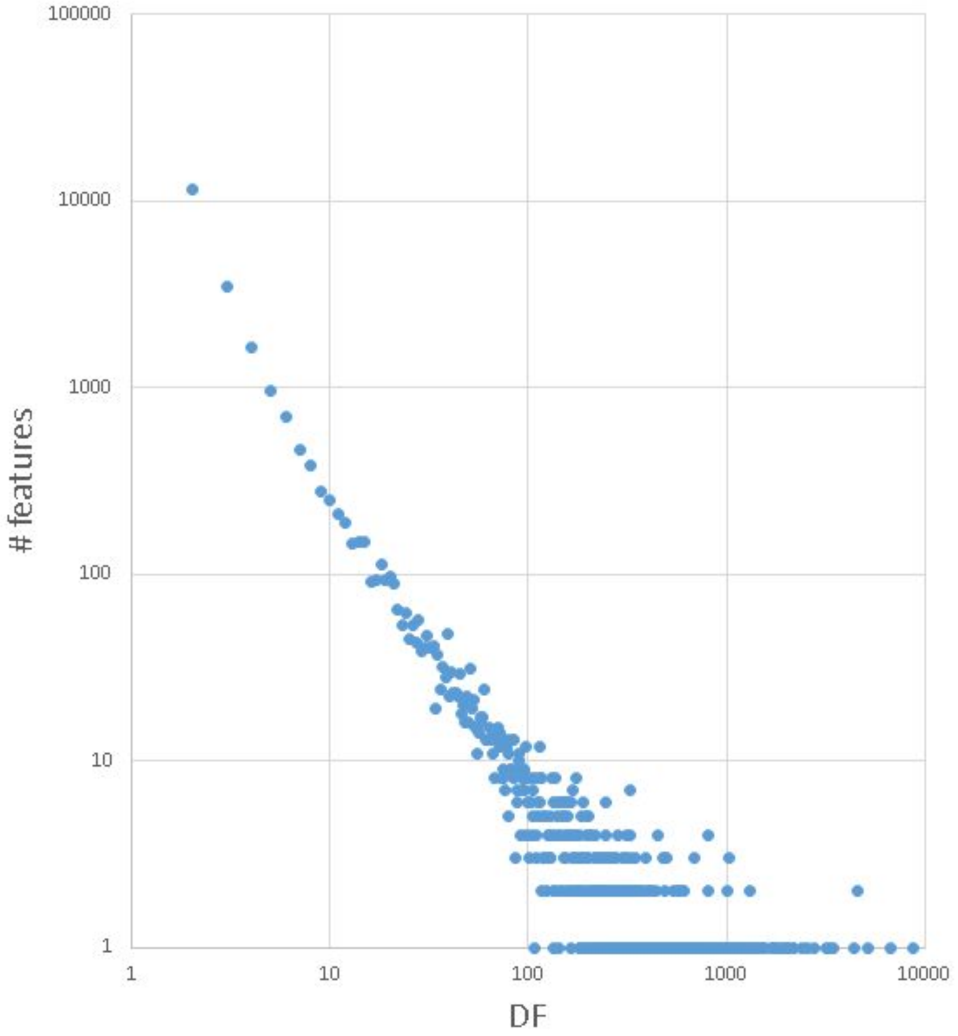
Frequency of use is inversely proportional to rank by frequency: $f \propto 1/r$

Principle of least effort: both the speaker and the hearer in communication try to minimize effort:

- Speakers tend to use a small vocabulary of common (shorter) words
- Hearers prefer a large vocabulary of rarer less ambiguous words
- Zipf's law is the result of this compromise

Related laws …

- Number of meanings m of a word obeys the law: $m \propto 1/f$
- Inverse relationship between frequency and length


100000
10000
1000
100
10
1
# features
1
10
100
1000
10000
DF

Stopwords

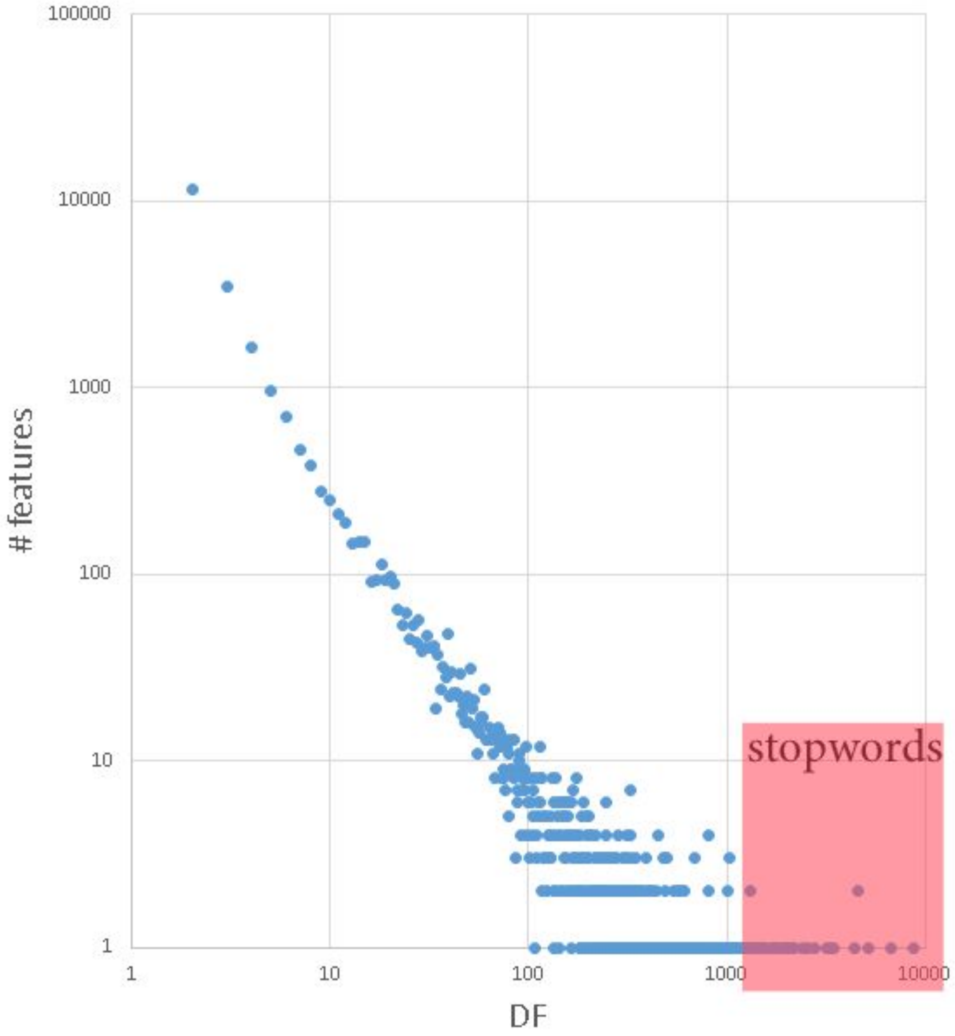
How much words like *of, the, a, an, to*, contribute to the semantic of a document?

```
the president of the united states of america
president united states america
```

The most common words of a language are usually referred as *stopwords*, i.e., words that can be removed from text without losing much information.

Stopwords lists are commonly provided by many NLP tools, yet they may not fit all the applications.

For example, MySQL's stopword list includes words like *appreciate*, *serious* and *unfortunately*, which are relevant for sentiment classification.


100000
10000
1000
100
10
1
# features
1
10
100
1000
10000
DF
stopwords

Stopwords

```
In : import nltk
In : from nltk.corpus import stopwords
In : stopwords.words('english')

Out: ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', ' ourselves',
...]

In : features = set(nltk.word_tokenize('the president of the united
states of america'))

In : less_features = features.difference(stopwords.words('english'))
In : less_features

Out: {'america', 'president', 'states', 'united'}
```

Rare features

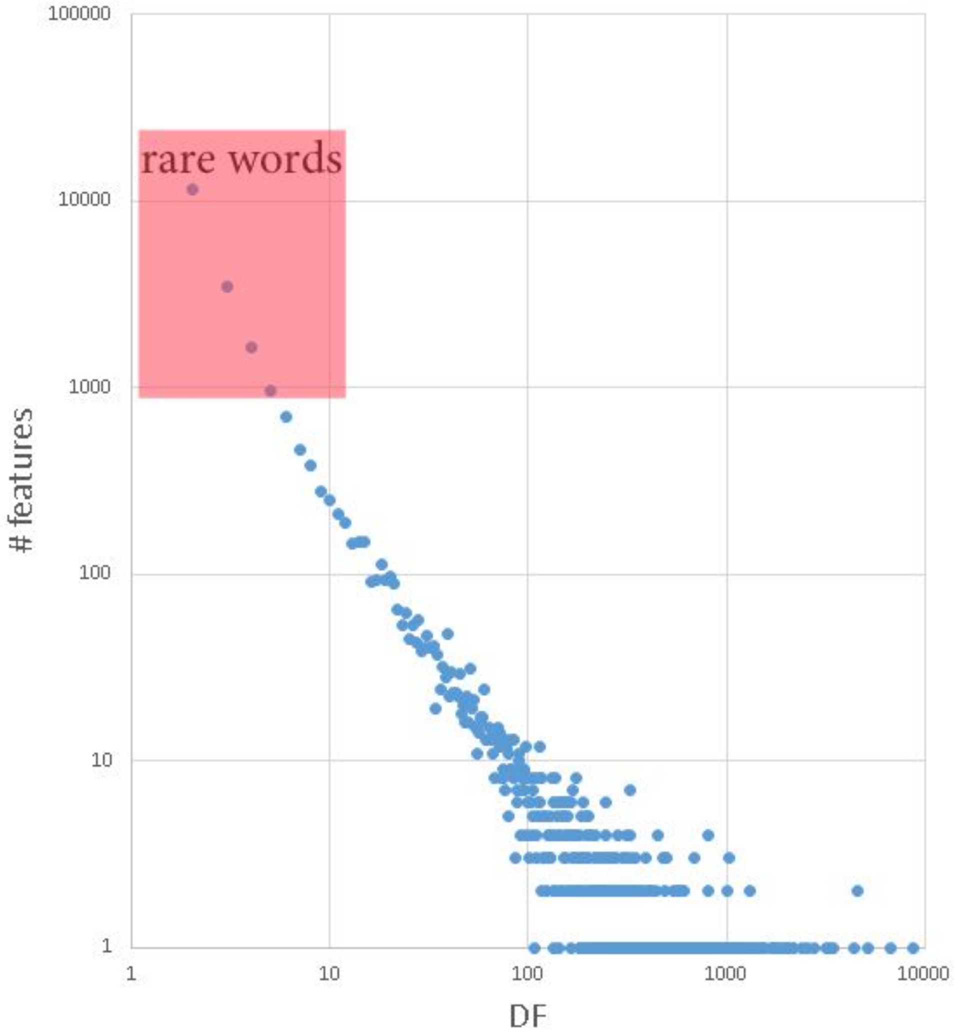
Features that appear in very few documents do not bring useful information to generalize the TM problem we are facing.

If a words appears rarely in text observed in the past, it will likely be rare in future text. Making it of little help to process most of the future input text.

A word can be rare because it is a random typo, or some kind of artificially built identifier that is bound to the document, e.g., a slug.

Hapax legomena (words that appear only once) usually account for a large portion of the distinct words appearing in a text collection.

Removing rare words makes it faster to process the indexed text, also requiring less memory space.


rare words
100000
10000
1000
100
10
1
# features
1
10
100
1000
10000
DF

Feature selection

What about the features with an in-between frequency?
Are they all of similar usefulness?

`'My` **`wife Sandra`** `bought this` **`awesome`** `TV.'`

For sentiment analysis ***`wife`*** and ***`Sandra`*** are less informative than ***`awesome`***.

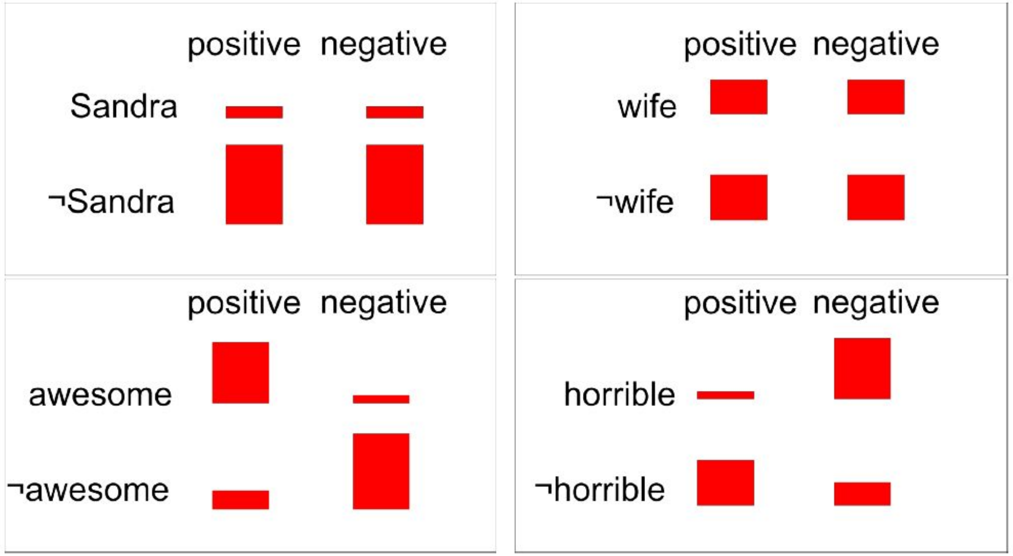
Can we estimate the information contribute of a feature?

We can use information theory functions to estimate the relevance of features and remove the less informative ones:

- it makes less computationally expensive to build the model.
- it removes noise that could decrease the model accuracy.

Feature selection

Exploiting feature-category correlation (either positive or negative) to select informative features:



Feature selection is a supervised process, i.e., it requires labeled data.

Feature selection

Typical feature selection functions:

- *mutual information* measures the feature-class correlation

$$MI(f,c) = \sum_{x \in \{f,\overline{f}\}} \sum_{y \in \{c,\overline{c}\}} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)}$$

- *chi-square* tests the feature-class independence

$$\chi^2(f,c) = \frac{D(D_{fc}D_{\overline{f}\overline{c}} - D_{\overline{f}c}D_{f\overline{c}})^2}{(D_{fc}+D_{\overline{f}c})(D_{fc}+D_{f\overline{c}})(D_{f\overline{c}}+D_{\overline{f}\overline{c}})(D_{\overline{f}c}+D_{\overline{f}\overline{c}})}$$

Keeping the 20%-10% of the original features is a common choice in text classification*.

*Y. Yang, J.O. Pedersen. A comparative study on feature selection in text categorization. ICML, 1997.

Feature selection

$$\chi^2(f,c) = \frac{D(D_{fc}D_{\overline{f}\overline{c}} - D_{\overline{f}c}D_{f\overline{c}})^2}{(D_{fc}+D_{\overline{f}c})(D_{fc}+D_{f\overline{c}})(D_{f\overline{c}}+D_{\overline{f}\overline{c}})(D_{\overline{f}c}+D_{\overline{f}\overline{c}})}$$

$D = 1000$

$D_c = 250$ $\quad D_{\neg c} = 750$

$D_{f1} = 100$ $\quad D_{\neg f1} = 900$

$D_{f1,c} = 50$ $\quad D_{\neg f1,c} = 200$ $\quad D_{f1,\neg c} = 50$ $\quad D_{\neg f1,\neg c} = 700$

$X^2(f1,c) = 1000(50*700-200*50)^2/(250*100*750*900) =$ **37.04**

$D_{f2} = 100$ $\quad D_{\neg f2} = 900$

$D_{f2,c} = 90$ $\quad D_{\neg f2,c} = 110$ $\quad D_{f2,\neg c} = 10$ $\quad D_{\neg f2,\neg c} = 740$

$X^2(f2,c) = 1000(90*740-110*10)^2/(250*100*750*900) =$ **254.24**

f2 is more informative than f1

Feature selection

$$MI(f,c) = \sum_{x \in \{f,\overline{f}\}} \sum_{y \in \{c,\overline{c}\}} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)}$$

From counts to probabilities:

$D = 1000$

$D_c = 250$ $D_{\neg c} = 750$

$p(c) = D_c/D = 250/1000 = 0.25$ **$p(\neg c) = D_{\neg c}/D = 750/1000 = 0.75$**

$D_{f1} = 100$ $D_{\neg f1} = 900$

$p(f1) = D_{f1}/D = 100/1000 = 0.1$ **$p(\neg f1) = D_{\neg f1}/D = 900/1000 = 0.9$**

$D_{f1,c} = 50$ $D_{\neg f1,c} = 200$ $D_{f1,\neg c} = 50$ $D_{\neg f1,\neg c} = 700$

$p(f1,c) = D_{f1,c}/D = 50/1000 = 0.05$ **$p(\neg f1,c) = D_{\neg f1,c}/D = 200/1000 = 0.2$**

$p(f1,\neg c) = D_{f1,\neg c}/D = 50/1000 = 0.05$ **$p(\neg f1,\neg c) = D_{\neg f1,\neg c}/D = 700/1000 = 0.7$**

Feature selection

$$MI(f,c) = \sum_{x \in \{f, \overline{f}\}} \sum_{y \in \{c, \overline{c}\}} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)}$$

D	= 1000						
p(c)	= 0.25	p(¬c)	= 0.75				
p(f1)	= 0.1	p(¬f1)	= 0.9				
p(f1,c)	= 0.05	p(¬f1,c)	= 0.2	p(f1,¬c)	= 0.05	p(¬f1,¬c)	= 0.7

MI(f1,c) = 0.05*log(0.05/(0.1*0.25))+0.2*log(0.2/(0.9*0.25))
+0.05*log(0.05/(0.1*0.75))+0.7*log(0.7/(0.9*0.75)) = **0.007**

p(f2)	= 0.1	p(¬f2)	= 0.9				
p(f2,c)	= 0.09	p(¬f2,c)	= 0.11	p(f2,¬c)	= 0.01	p(¬f2,¬c)	= 0.74

MI(f2,c) = 0.09*log(0.09/(0.1*0.25))+0.11*log(0.11/(0.9*0.25))
+0.01*log(0.01/(0.1*0.75))+0.74*log(0.74/(0.9*0.75)) = **0.037**

Vector Space Model

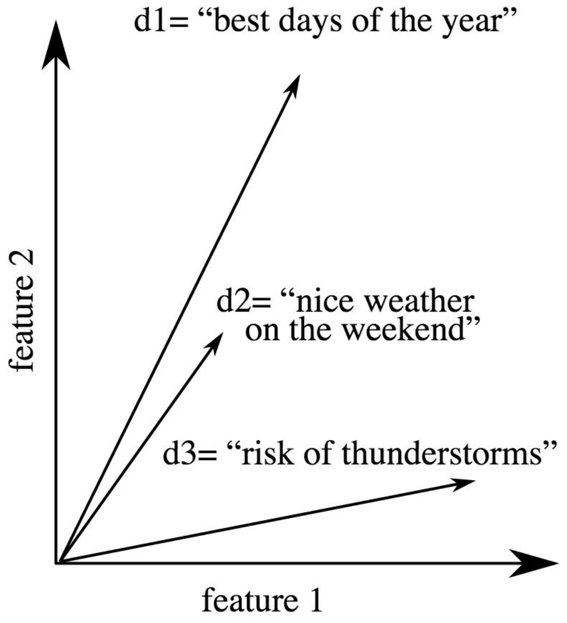
Vector Space Model

We now have each document represented as a $|F|$-dimensional vector w, where $|F|$ is the number of distinct features extracted from text.

Documents whose vector are close one to each other are likely to be similar.

w_{df} indicates the relevance of the feature f in the document d.

Vectors are usually **sparse**, i.e., most of their values are zero.


d1= “best days of the year”
d2= “nice weather on the weekend”
d3= “risk of thunderstorms”
feature 2
feature 1

Vector Space Model

Each feature is mapped to a distinct dimension in $R^{|F|}$ using a *one-hot* vector:

$$
\begin{aligned}
v('I') &= [1, 0, 0, \ldots, 0, 0, \ldots, 0, 0, 0] \\
v('you') &= [0, 1, 0, \ldots, 0, 0, \ldots, 0, 0, 0] \\
v('won') &= [0, 0, 1, \ldots, 0, 0, \ldots, 0, 0, 0] \\
&\vdots \\
v('a\_i\_n') &= [0, 0, 0, \ldots, 0, 1, \ldots, 0, 0, 0] \\
&\vdots \\
v('NN') &= [0, 0, 0, \ldots, 0, 0, \ldots, 0, 0, 1]
\end{aligned}
$$

Vector Space Model

A document is represented as the weighted sum of its features vectors:

$$v(d) = \sum_{f \in d} w_{fd} v(f)$$

For example:

d = 'you played a good game'

$v(d)$ = $[0, w_{played,d}, w_{game,d}, 0, \ldots \ldots 0, w_{good,d}, 0 \ldots \ldots 0, 0]$

The resulting document vectors are sparse:

$$|\{i | v_i(d) \neq 0\}| \ll n$$

How do set the weights w_{fd}?

Weighting

Typical weighting methods:

- binary (bag of words): $bin(d, f) = \begin{cases} 1 & : f \in d \\ 0 & : f \notin d \end{cases}$
- tf (frequency), rewarding features that appears many times in the document:

$$\mathrm{tf}(d, f) = \begin{cases} 1 + \log(f(d, f)) & : f \in d \\ 0 & : f \notin d \end{cases}$$

- tf-idf, rewarding rare features with high tf :

$$\text{tf-idf}(d, f) = \begin{cases} \mathrm{tf}(d, f) \log \frac{N}{|\{d \in D : f \in d\}|} & : f \in d \\ 0 & : f \notin d \end{cases}$$

Vectors can be normalized to unit length, to factor out differences deriving by the difference of length between documents.

Summary

- Text indexing enables building machine-processable representations of text.
- Features identify in text the observable sources of information.
- Not all features are relevant, an accurate selection can be beneficial to successive processing steps, both in terms of efficiency and in terms of efficacy.
- The vector space is a high dimensional space in which documents are distributed according to their content.