MEC ITALY

TERM VOCABULARY AND DICTIONARIES

ALFIO FERRARA, STEFANO MONTANELLI

Department of Computer Science

Via Comelico 39, 20135 Milano

{alfio.ferrara,stefano.montanelli}@unimi.it
http://islab.di.unimi.it/

OUTLINE

- Units and granularity
- Token extraction
- Language identification
- Normalization
- Phrase queries
- Biwords
- Tolerant retrieval

Units and Granularity

Before extracting tokens from documents, there are two crucial choices that have to be taken:

What is a document unit?

The document unit will be the object that will be returned as an answer to queries, classified, and analyzed

Which is the indexing granularity?

Within a document, what are we going to index? The unit chosen is what is really searched in a query. Units too small may cause not to retrieve words that are distributed over several mini-docs. Units too large may cause the system to loose precision.

Example: I Promessi Sposi

I Promessi Sposi is composed by 38 chapters (+ the Introduction).

- Document unit: **chapter**, Index granularity: **chapter**
- 38 docs, index size: 8911 x 38, tokens per doc: about 3182
- Document unit: **chapter**, Index granularity: **sentence**
- 38 docs, index size: 8911 x 2639, tokens per doc: about 45

TOKENIZATION

TOKENIZATION

The activity of chopping up a text into pieces, called **tokens**, possibly throwing away certain characters, such as punctuation.

Problems to deal with

- Language affects direction of reading, type of characters, use of punctuation
- Special tokens may include acronyms and special characters (e.g., hashtags)
- Hyphenation and other morphological phenomena should be taken into account
- Character encoding is a problem when the original encoding of text is not known

EXAMPLE: SIMPLE WHITESPACE TOKENIZER

Quel ramo del lago di Como, che volge a mezzogiorno, tra due catene non interrotte di monti, tutto a seni e a golfi, a seconda dello sporgere e del rientrare di quelli, vien, quasi a un tratto, a ristringersi [...]

```
Quel| ramo| del| lago| di| Como,| che| volge| a| mezzogiorno,|
tra| due| catene| non| interrotte| di| monti,| tutto| a| seni|
e| a| golfi,| a| seconda| dello| sporgere| e| del| rientrare|
di| quelli,| vien,| quasi| a| un| tratto,| a| ristringersi|
```

Simple Tokenizer in Python

Usually tokenization is implemented as a pipeline of operations, like:

```
def tokenize(text, pattern=None):
    if pattern is None:
        pattern = r'''(?x)    # set flag to allow verbose regexps
        ([A-Z]\.)+          # abbreviations, e.g. U.S.A.
        | \$(.*?)\$         # latex_formula
        | \w+(-\w+)*        # words with optional internal hyphens
        | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
        | \@?\w+            # names
        | \#?\w+            # hashtag
        | \.\.\.            # ellipsis
        | [][.,;"'?():_`]  # these are separate tokens
        '''
    tokens = nltk.regexp_tokenize(text, pattern)
    tokens = [x.encode('utf-8').decode('utf-8') for x in tokens
                if x not in string.punctuation]
    return tokens
```

Example: Regex Tokenizer

Quel ramo del lago di Como, che volge a mezzogiorno, tra due catene non interrotte di monti, tutto a seni e a golfi, a seconda dello sporgere e del rientrare di quelli, vien, quasi a un tratto, a ristringersi [...]

```
Quel| ramo| del| lago| di| Como| che| volge| a| mezzogiorno|
tra| due| catene| non| interrotte| di| monti| tutto| a| seni| e|
a| golfi| a| seconda| dello| sporgere| e| del| rientrare| di|
quelli| vien| quasi| a| un| tratto| a| ristringersi|
```

TOKEN NORMALIZATION

After tokenization, tokens are normalized in order to find a single representative form for group of tokens that can be processed as the same word.

Moreover, normalization may include pruning of tokens that are very generic in a language and thus less useful.

However, many normalization procedures depend on the document language. Automatic language detection mechanisms are needed.

AUTOMATIC LANGUAGE DETECTION

Example of a naive solution in Python

```python
from nltk.corpus import stopwords

def detect_language(text_or_tokens, tokenize=True):
    languages_ratios = {}
    lang = 'english'
    if tokenize:
        tokens = tokenizer.wordpunct_tokenize(text_or_tokens)
        words = [x.lower() for x in tokens]
    else:
        words = text_or_tokens
    for language in stopwords.fileids():
        stopwords_set = set([x.decode('utf-8') for x in stopwords.words(language)])
        words_set = set(words)
        common_elements = words_set.intersection(stopwords_set)
        languages_ratios[language] = len(common_elements)
    choice = sorted(languages_ratios.items(), key=lambda x:-x[1])
    if choice[0][1] > choice[1][1]:
        lang = choice[0][0]
    return lang
```

STOPWORDS REMOVAL

The simplest normalization procedure is to remove those words that have a very high frequency in the text language.

These words, known as **stopwords**, are less informative in that they are very common (e.g., article, prepositions).

Stopwords are pruned using a dictionary of stopwords for a given language.

EXAMPLE

Italian stopwords (NLTK): ad | al | allo | ai | agli | all | agl | alla | alle | con | ...

```
Quel | ramo | del | lago | di | Como | che | volge | a | mezzogiorno | tra | due | catene |
non | interrotte | di | monti | tutto | a | seni | e | a | golfi | a | seconda | dello |
sporgere | e | del | rientrare | di | quelli | vien | quasi | a | un | tratto | a |
ristringersi |
```

```
it_stopwords = [x.decode('utf-8') for x in stopwords.words('italian')]
tokens = [x for x in tokens if not x.lower() in it_stopwords]
```

```
Quel | ramo | lago | Como | volge | mezzogiorno | due | catene | interrotte | monti | seni |
golfi | seconda | sporgere | rientrare | vien | quasi | tratto | ristringersi |
```

Note: the lowercase option is often the best practical option, but it has also some drawbacks: see "Quel" and "Como"

Note 2: In I promessi Sposi, we have a dictionary of 19175 unique tokens with stopword removal vs 20535 without

Stemming

Stemming aims at reducing inflectional forms of words in documents by exploiting a **purely** syntactic and heuristic process.

Stemming truncates words using language-dependant rules that do not depent on linguistic of grammatical rules. The result is a set of tokens that have nothing in common with the real language words.

- **Pro** independence from external dictionaries or thesauri.
- **Cons** not capturing all the inflections.

Examples

```
from nltk.stem.snowball import SnowballStemmer
tokens = ['play', 'playing', 'is', 'are', 'caresses', 'ponies', 'pony']
stemmer = SnowballStemmer('english')
print [stemmer.stem(x) for x in tokens]
> [u'play', u'play', 'is', u'are', u'caress', u'poni', u'poni']
```

Lemmatization

Lemmatization performs the same activity of stemming but using a vocabulary to find a common entry for words that have one.

The form that is used as the entry in the vocabulary is known as **lemma**.

- **Pro** suitable for all the morphological variety of terms in the language.
- **Cons** requires an external source and sometimes words are not in the vocabulary of may have multiple and ambiguous entries in the vocabulary.

LEMMATIZATION EXAMPLE

Examples

```
from nltk.corpus import wordnet as wn
tokens = ['play', 'playing', 'is', 'are', 'caresses', 'ponies', 'pony']
lemmas = []
for token in tokens:
    try:
        synset = wn.synsets(token)[0] #arbitrary choosing the first synset
        lemma = synset.lemmas[0]      #arbitrary choosing the first lemma
        lemmas.append(lemma.name.replace('_', ' '))
    except IndexError:
        lemmas.append(token)
print lemmas
> ['play', 'play', 'be', 'be', 'caress', 'pony', 'pony']
```

EXERCISE

Use NLTK and WordNet to build a better lemmatizer, dealing with multiple synsets using a reasonable criterion.

TECH: NLTK AND WORDNET

WordNet (https://wordnet.princeton.edu/) is a lexical database for English (but available also in other languages)

The main relation among words in WordNet is synonymy. The meaning of each synonym is represented in WordNet are a **Synset** that is words that denote the same concept and are interchangeable in many contexts. One word can belong to several synsets.

Each synset is linked to other synsets by means of a small number of **conceptual relations**. Additionally, a synset contains a brief definition (**gloss**). Word forms with several distinct meanings are represented in as many distinct synsets. Thus, each form-meaning pair in WordNet is unique.

TECH: NLTK AND WORDNET

The majority of the WordNet's relations connect words from the same **part of speech (POS)**. Thus, WordNet really consists of four sub-nets, one each for **nouns**, **verbs**, **adjectives** and **adverbs**.

MAIN CONCEPTUAL RELATIONS

Nouns

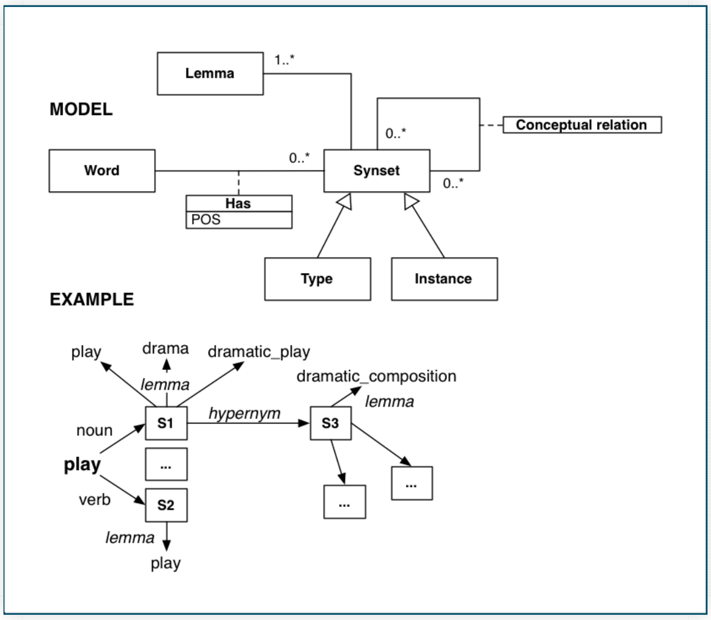
- hyperonymy/hyponymy
- instance hyperonymy/hyponymy
- meronymy

Verbs

- troponyms

Adjectives

- antonymy



TECH: ACCESS WORDNET THROUGH NLTK

```
from nltk.corpus import wordnet as wn

synsets = wn.synsets('play')
s0 = synsets[0]
for s0 in synsets:
    print s0.pos
    print [l.name for l in s0.lemmas]
    for h in s0.hypernyms():
        print [l.name for l in h.lemmas]
    # dir(s0) for other relations
```

```
n
['play', 'drama', 'dramatic_play']
    ['dramatic_composition', 'dramatic_work']
n
['play']
    ['show']
n
['play']
    ['plan_of_action']
...
```

EXAMPLE: PUTTING THINGS TOGETHER (I)

```
def simple_tokenize(text, pattern=None, stemming=False):
    if pattern is None:
        pattern = r'''(?x)    # set flag to allow verbose regexps
            ([A-Z]\.)+          # abbreviations, e.g. U.S.A.
            | \$(.*?)\$         # latex_formula
            | \w+(-\w+)*        # words with optional internal hyphens
            | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
            | \@?\w+            # names
            | \#?\w+            # hashtag
            | \.\.\.            # ellipsis
            | [][.,;"'?():_`]  # these are separate tokens
            '''
    tokens = nltk.regexp_tokenize(text, pattern)
    tokens = [x.lower().encode('utf-8').decode('utf-8') for x in
            tokens if x not in string.punctuation]
    language = detect_language(tokens, tokenize=False)
    stop_list = [x.decode('utf-8') for x in stopwords.words(language)]
    tokens = [x for x in tokens if x not in stop_list]
    if stemming and language in SnowballStemmer.languages:
        stemmer = SnowballStemmer(language)
        tokens = [stemmer.stem(x) for x in tokens]
```

EXAMPLE: PUTTING THINGS TOGETHER (II)

Quel ramo del lago di Como, che volge a mezzogiorno, tra due catene non interrotte di monti, tutto a seni e a golfi, a seconda dello sporgere e del rientrare di quelli, vien, quasi a un tratto, a ristringersi [...]

```
quel| ram| lag| com| volg| mezzogiorn| due| caten| interrott|
mont| sen| golf| second| sporg| rientr| vien| quas| tratt|
ristring|
```

Example (English)

It ain't no use to sit and wonder why, babe
It don't matter, anyhow
An'it ain't no use to sit and wonder why, babe
If you don't know by now
When your rooster crows at the break of dawn
Look out your window and I'll be gone
You're the reason I'm trav'lin' on
Don't think twice, it's all right

```
ain| use| sit| wonder| babe| matter| anyhow| ain| use| sit|
wonder| babe| know| rooster| crow| break| dawn| look| window|
ll| gone| re| reason| m| trav| lin| think| twice| right|
```

Comment

Normalization **loses** information, but...

Reduces remarkably the number of unique tokens to process (i.e., from 19175 to about 9000 in I Promessi Sposi)

Makes texts easier to compare.

Whether to normalize the text is a choice that should be made depending on the goals of the system and the size of the corpus that has to be processed.

TOKENIZATION AND NORMALIZATION SCHEME

Since there are several choices in the configuration of the tokenization and normalization processes, it may be useful to configure the system with a tokenization and normalization scheme

Example

```
{
    'tokenization': [pattern],
    'language_identification': True|False,
    'lowercase': True|False,
    'remove_punctuation': True|False,
    'stopword_removal': True|False,
    'normalization': none|stemming|lemmatization
}
```

Given a tokenization and normalization scheme, each query must be processed using the same scheme before searching tokens in the index

PHRASE QUERIES

Phrase queries are queries where the search components can be compound terms or phrases like "la sventurata rispose"

In phrase queries the user is interested in finding exactly the expression she is looking for, not just the conjunction of the word components

Two approaches

- **Biword (or phrase) indexes**
- **Positional indexes**

Biword indexes

In the index construction process each pair of consecutive tokens (after normalization) are indexed as a unique token

Then more complex queries than biword queries are processed as a conjunctive query as follows:

Query: "lei ha intenzione di maritar domani"
Normalization: ['intenzione', 'maritar', 'domani']
Processing: intenzione maritar AND maritar domani

The biword approach may be extended to built indexes for a number of words higher than 2, or even phrases

Relevance of n-words

Given a n-words index (e.g., 2-word, 3-word), we can limit the number of entries in the index by calculating a measure of relevance for each n-word

Mutual information as relevance of a 2-word

$$\mu(x, y) = p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right)$$

where:

- x and y are the tokens in the 2-word "x y"
- $p(x, y)$ is calculated as the relative frequency of xy over all the 2-words in the corpus
- $p(x)$ (and $p(y)$) is calculated as the relative frequency of x in the corpus

Example

Some results from mutual information on the first 2 chapters of 'I Promessi Sposi'

Note that we should filter out tokens with very few occurrences...why?

don abbondio
p(don abbondio) = 0.0074
p(don) = 0.0095
p(abbondio) = 0.0076

poi disse
p(poi disse) = 0.00016
p(poi) = 0.0064
p(disse) = 0.0043

Biword	$\mu(x, y)$
don abbondio	0.034756
illustrissimo eccellentissimo	0.00765
amor cielo	0.007372
don rodrigo	0.006237
signor curato	0.005815
volete tacere	0.005411
eccellentissimo signore	0.005017
qua là	0.004372
aver fallato	0.003988
pover uomo	0.003948
...	...

Positional index

The alternative solution for phrase queries (and the most commonly employed) is to store in the index the position in a document of each token

Example of positional index

```
{
    "token_i": {
            "doc_j": (x_i_j, [p_1, p_2, ..., p_n]),
            ...
            }
}
```

The token "token_i" occurs x_i_j times in "doc_j" at positions p_2, p_2, ..., p_n

QUERY PROCESSING

For processing phrase queries, we first search for documents containing all the tokens of interest, then we check that the position of the tokens is compatible with the position of tokens in the query

Given the query: "la sventurata rispose", we need documents containing both "sventurata" and "rispose" and where the position of "rispose" is higher than the position of "sventurata"

Advantages of positional index: it is easily adaptable to n-words queries for every n and can support queries where the user can specify the required distance, like: "find documents where 'rispose' appears within 5 words from 'sventurata'"

TOLERANT RETRIEVAL

Tolerant retrieval aims at supporting wildcard queries of the form

pla*ed

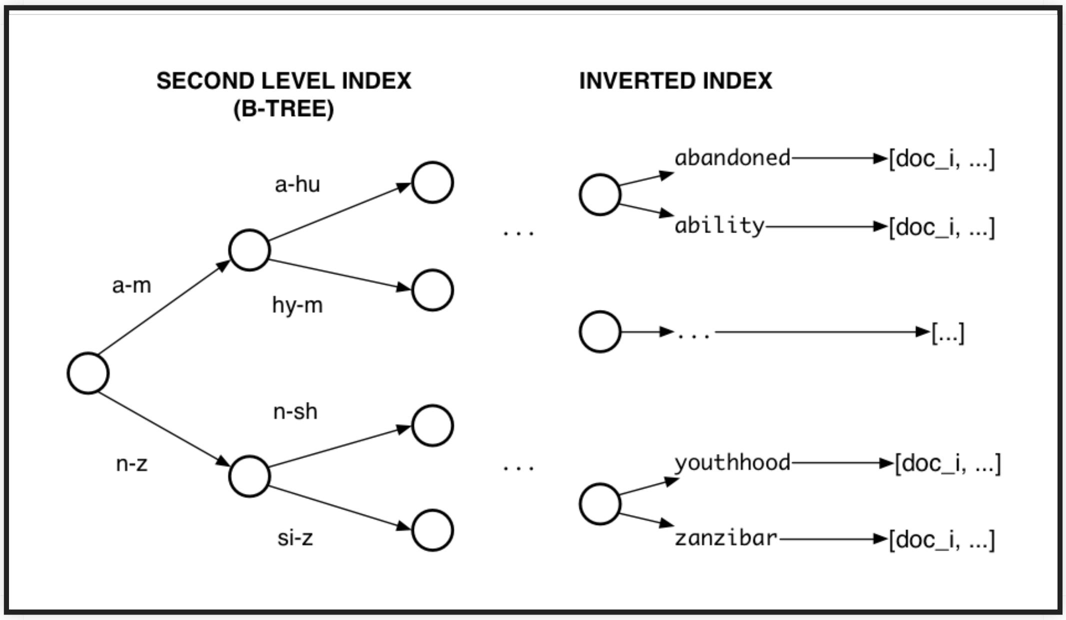
that searches for any token starting with the string **pla** followed by any sequence of characters (i.e., *****) and ending with **ed**

Examples of matching strings

- **pla**y**ed**
- **pla**nn**ed**
- **pla**ying plann**ed**

SECOND LEVEL INDEXES

A strategy for answering wildcard queries is to build a second level index on top of the keys of the inverted index, using a **b-tree**



QUERY PROCESSING

- Given a query **pla*ed**, build the b-tree B and the **reverse b-tree** B^{-} by indexing the entries of the inverted index in reverse order (i.e., played → deyalp)
- Search the entries W starting with **pla** using B
- Search the entries R ending with **ed** using B^{-} (by searching **de**)
- Get the entries in $W \cap R$

PERMUTERM INDEX

Another solution for wildcard queries is to use a **permuterm index** build as follows

- Introduce a special character (like #) to mark the end of a word, such as play → play#
- Construct an index for all the forms of a token obtained by rotating the characters (including the special character)
  play#, lay#p, ay#pl, y#pla → play#
- Given a query **p*y**, rotate it in a way that * appears at the end of the string → y#p*
- Now search in the permuterm index those entries starting with y#p (using a B-tree)

K-GRAM INDEXES

A further solution for wildcard queries is to exploit a n-gram index

- Build an index for each n-gram (say 3-gram) in tokens, including the special characters for starting and ending:
  planned → #pl, pla, lan, ann, nne, ned, ed#
- Given a query like **pl*ed**, search for **#pl AND ed#**
- However, queries like ***ned** force the system to search for **ed# AND ned**, which leads to the retrieval of words like i**ned**i**ted** that do not match the query
- Thus, a further post-processing and filtering phase is required on the results

Exercise

Study and develop a solution for tolerant retrieval

- Discuss the kind of wildcard queries is supported
- Discuss the indexing structure used for query processing
- Evaluate the efficiency of the solution

Spelling correction

Spelling correction aims at making the system tolerant with respect to misspelled queries, like **pleyed** instead of **played**

Spelling correction is based on the notion of **proximity** among tokens

Then, among possible corrections for a query, we choose the most relevant according to the number of occurrences in the corpus and/or on the frequency in the queries run by users

EDIT DISTANCE

EDIT DISTANCE

Given two strings s_i and s_j, their edit distance $edit(s_i, s_j)$ is the minimum number of editing operations (i.e., insert, delete, replace characters) required to transform s_i in s_j. Edit distance is also known as **Levenshtein distance**

```
def edit_distance(a, b):
    x, y, m = len(a)+1, len(b)+1, m
    for i in range(x):
        m[i,0] = i
    for j in range(y):
        m[0,j] = j
    for i in range(1, x):
        for j in range(1, y):
            cost = 0 if a[i-1] == b[j-1] else 1
            m[i,j] = min(m[i, j-1]+1, m[i-1, j]+1, m[i-1, j-1]+cost)
    return m[len(a), len(b)], m
```

USAGE OF EDIT DISTANCE

Of course, we must avoid to calculate the edit distance between a query and each entry in the inverted index

To this end several heuristic strategies can be used, including n-grams indexes

EXERCISE

Find a suitable solution for finding just a subset of tokens among the entries of the inverted index that should be compared for the query for spelling correction

N-gram indexes

N-gram indexes may be used for spelling correction as follows:

- We linearly scan the entries of the inverted index by looking at the number of n-grams (usually 2-grams) in common between an entry and the query
- For minimizing the number of tokens to take into account, we can measure the degree of overlapping between a token and the query
- Given A as the set of n-grams of the token and B as the set of n-grams in the query, their degree of overlapping is measured by the **Jaccard** coefficient as:

$$jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Phonetic correction

Phonetic correction is based on the idea that a word can be transformed (and normalized) according to some phonetic properties of the language (depending on the specific language), such as i) vowels similarity, ii) some consonants have similar sounds

Example: **soundex algorithm**

- Retain the first letter of a token
- Change all occurrences of A, E, I, O, U, H, W, Y into a 0 (zero)
- Change letters to digits according to a pre-defined map (such as D, T → 3, M, N → 4, ...), in order to put sound-similar letters into a class of equivalence
- Remove one out of each pair of consecutive identical digits
- Remove zeros and return the first four positions
  Hermann → H655

Then we use these transformed forms for searching

RECAP

Topic	Chapters
Tokenization	2
Normalization: stopwords, stemming, lemmatization	2
Biword indexes and compound terms	2
Tolerant retrieval	3
Phonetic correction	2