
MarTech

Text analysis in Python

Information retrieval

Ing. Paolo Fantozzi

Perché il testo?

Il linguaggio naturale è un problema interessante perché:

- denso di informazioni
- richiede una conoscenza estesa del mondo
- il suo utilizzo cambia da persona a persona

La conoscenza in numeri

Il **90%** dei dati prodotto dall'uomo è stato creato negli ultimi **2 anni**

La conoscenza in numeri

Attualmente disponibili:

4,4 zettabytes

=

4,4 mila miliardi di GB

La conoscenza in numeri

Stima:

10.574.000 volumi = 12.084 GB

(distruzione della Biblioteca nazionale della Germania nel 1945)

La conoscenza in numeri

Stima:

4,4 zettabytes

\approx

$3,85 \times 10^{15}$ libri

3,85 milioni di miliardi di libri

La conoscenza in numeri

Stima:

4,4 zettabytes

≈

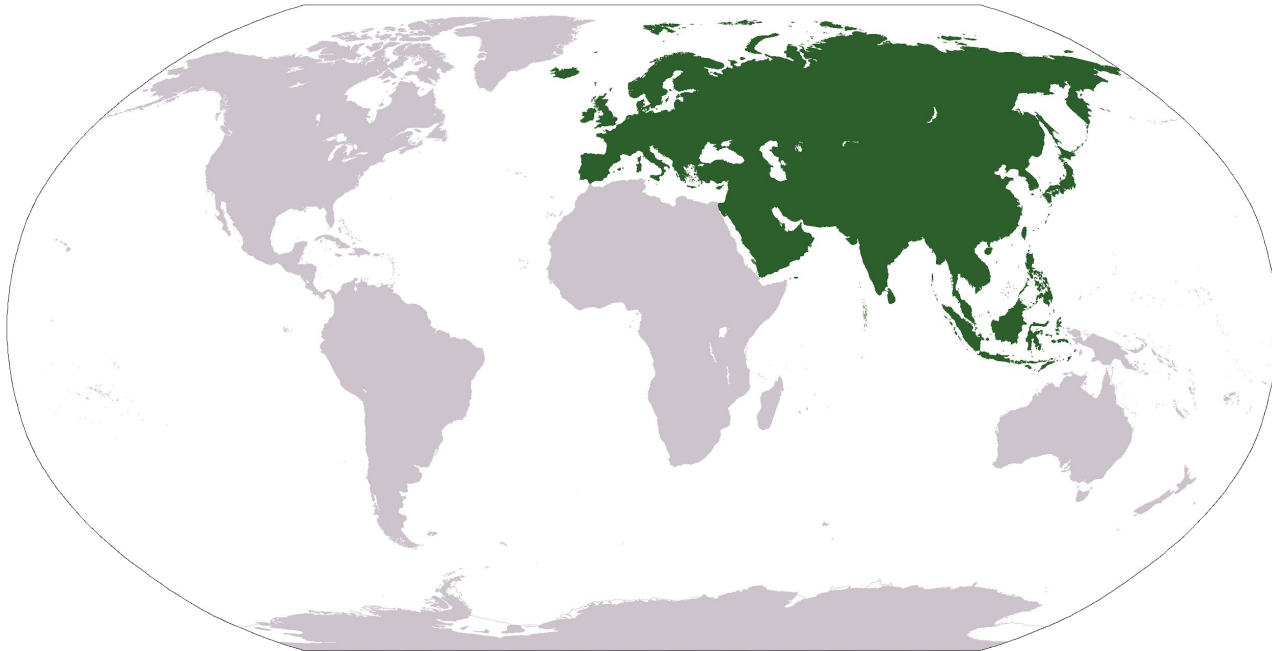
58 milioni di km²

La conoscenza in numeri

4,4 zettabytes

≈

una biblioteca grande **più** dell'**Europa** e dell'**Asia** messe insieme



La conoscenza in numeri

La domanda è:

Come troviamo **un** libro che ci serve?

La conoscenza in numeri

“Se è tutto catalogato bene, allora è possibile!”

La conoscenza in numeri

Ma **catalogare** in una biblioteca vuol dire:
saper trovare il libro conoscendone già il titolo

La conoscenza in numeri

E se noi volessimo solo un libro che
tratta un certo argomento?

La conoscenza in numeri

Oppure il **miglior** libro su **un certo argomento**?

Cos'è l'Information Retrieval

La branca dell'informatica che si occupa di trovare le informazioni

Definizioni

- **Documento:** una singola fonte di informazione (ad es. un libro)
- **Dato strutturato:** tabella
- **Dato non strutturato:** dato grezzo (ad es. testo)
- **Testo semi-strutturato:** dato grezzo organizzato (ad es. sito web)
- **Collezione (o corpus):** insieme di documenti
- **Corpora:** tutte le collezioni di documenti a disposizione

Bag of words

Il metodo classico (e più semplice) per la gestione del testo: **bag of words**.

Ogni documento è *un insieme di parole*

Problema iniziale

Se un documento è *un insieme di parole*, come **dividiamo** un testo in parole?

Dataset

Scaricate il dataset delle opere complete di Shakespeare al seguente
link

<http://bit.ly/shakespeare-txt>

Processamento del dataset

Suddividete il dataset in documenti usando python, in modo che:

- ogni opera deve essere un file separato
- nel file deve essere presente il testo dell'opera e, separatamente, anche eventuali metadati (titolo, anno, etc.)
- le informazioni di copyright devono essere eliminate

(solitamente questo passaggio è ad-hoc per ogni progetto)

Processamento del dataset

```
1 import json
2 from itertools import islice
3 from pathlib import Path
4
5
6 def write_to_json(lines, base_dir):
7     lines_it = iter(lines)
8
9     line = next(lines_it).strip()
10    while not line:
11        line = next(lines_it).strip()
12    year = int(line)
13
14    line = next(lines_it).strip()
15    while not line:
16        line = next(lines_it).strip()
17    title = line
18
19    line = next(lines_it).strip()
20    while not line:
21        line = next(lines_it).strip()
22    author = line.lstrip('by').strip()
23
24    content = ".join(list(lines_it))
25
26    filepath = base_dir / f'{year}-{title}-{author}.json'
27    with open(filepath, 'w') as json_file:
28        json.dump(
29            {'year': year, 'title': title, 'author': author, 'content': content},
30            json_file,
31        )
32
```

```
33 def main():
34     base_dir = Path('works')
35     base_dir.mkdir(parents=True, exist_ok=True)
36
37     shake_txt = Path('shakespeare.txt')
38
39     tmp_dir = base_dir / 'splitted'
40     tmp_dir.mkdir(parents=True, exist_ok=True)
41
42     with open(shake_txt, 'r') as shake_file:
43         go_writing = True
44         doc = []
45         for line in islice(shake_file, 244, None):
46             if '<<' in line:
47                 go_writing = False
48
49             if go_writing:
50                 doc.append(line)
51
52             if 'THE END' in line:
53                 write_to_json(doc, tmp_dir)
54                 doc = []
55
56             if '>>' in line:
57                 go_writing = True
58
59 if __name__ == '__main__':
60     main()
```

Tokenizzazione

La suddivisione in token (parole) è un task che presenta diverse difficoltà.

- Possiamo suddividere con gli spazi?
- Secondo che regole possiamo suddividere?
- Che problemi irrisolti rimangono?
- In questo modo ignoriamo la punteggiatura
- Spazi, a capo, punteggiatura, qualsiasi “non lettera”
- Parole a capo, acronimi

Tokenizzazione dei documenti

Eseguite la tokenizzazione di ogni documento, ignorando i seguenti casi speciali:

- Acronimi (*probabilmente non usati da Shakespeare*)
- Parole a capo (*non sono presenti in questo dataset*)

inoltre:

- Convertire tutto in minuscolo (*gli utenti cercano senza maiuscole*)
- Mantenere solo un insieme di parole per ogni documento

HINT: utilizzare il modulo **re** per tokenizzare

Tokenizzazione dei documenti

```
1 import json
2 import re
3 from pathlib import Path
4
5
6 def load_docs():
7     works_dir = Path('works') / 'splitted'
8
9     docs = []
10    for work in works_dir.iterdir():
11        with open(work, 'r') as work_file:
12            doc = json.load(work_file)
13            doc['words'] = set(re.split(r'\W+', doc['content'].lower()))
14            docs.append(doc)
15
16    return docs
17
18 if __name__ == '__main__':
19     load_docs()
```

Query booleana

*“Tutti i documenti che contengono la parola **Brutus** e la parola **Caesar** ma **non** la parola **Calpurnia**”*

corrisponde alla query

Brutus AND Caesar AND (NOT Calpurnia)

Matrice termini-documenti

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Esempi presi da

Introduction to Information Retrieval - Manning, Raghavan, Schutze

Vettori di incidenza

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Brutus: 110100

Caesar: 110111

Calpurnia: 010000

NOT Calpurnia: 101111

Vettori di incidenza

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

110100 AND 110111 AND 101111 = 100100

100100 = Antony and Cleopatra; Hamlet

Realizzazione

Realizzare un motore di ricerca (solamente da terminale), sul nostro dataset di shakespeare, che comprende query booleane (AND, OR, NOT) utilizzando i concetti appena esposti.

Gli operatori verranno rappresentati come & | ! nelle query, tutto separato da spazi

Ignorate la precedenza degli operatori: applicateli così come vengono letti

Se una parola cercata non esiste nei documenti, allora il suo vettore è pari a tutti zeri

HINT: utilizzare **numpy** per creare le matrici e/o i vettori

brutus | caesar & !calpurnia
brutus OR caesar AND (NOT calpurnia)

Realizzazione

```
1 import numpy as np
2
3 from doc_loader import load_docs
4
5
6 class Term:
7     def __init__(self, array):
8         self.array = array
9
10    def __and__(self, other):
11        return Term(self.array & other.array)
12
13    def __or__(self, other):
14        return Term(self.array | other.array)
15
16    def __invert__(self):
17        return Term(1 - self.array)
18
19
20 class TDMatrix:
21     def __init__(self, docs):
22         self.docs = docs
23
24         all_words_set = set()
25         for doc in docs:
26             all_words_set.update(doc['words'])
27         all_words_list = sorted(all_words_set)
28         self.all_words_list = all_words_list
29         self.word_idx_map = {word: idx for idx, word in enumerate(self.all_words_list)}
30
31         matrix = []
32         for term in self.all_words_list:
33             row = [1 * (term in doc['words']) for doc in self.docs]
34             matrix.append(row)
35         self.matrix = np.array(matrix)
36
37     def term_from_word(self, word):
38         negated = False
39         if word.startswith('!'):
40             word = word[1:]
41             negated = True
42         try:
43             array = self.matrix[self.word_idx_map[word]]
44         except KeyError:
45             array = np.zeros(self.matrix.shape[1], dtype=np.int8)
46         return ~Term(array) if negated else Term(array)
47
48     def query(self, q):
49         if not q:
50             return []
51         words = q.split()
52         word, *words = words
53         term = self.term_from_word(word)
54         while words:
55             op, word, *words = words
56             word_term = self.term_from_word(word)
57             if op == '&':
58                 term = term & word_term
59             elif op == '|':
60                 term = term | word_term
61         indexes = list(np.argwhere(term.array == 1).flatten())
62         return [self.docs[idx] for idx in indexes]
63
64
65 if __name__ == '__main__':
66     docs = load_docs()
67     tdmatrix = TDMatrix(docs)
68     query = input('Cosa vuoi cercare?\n')
69     print('Risultati:')
70     results = tdmatrix.query(query.lower())
71     print(f'{len(results)}/{len(docs)}')
72     for doc in results:
73         print(f'{doc["title"]} - {doc["year"]}")')
```

Grandi collezioni

Consideriamo:

- un corpus di 1 milione di documenti
- ogni documento di 1000 parole (2 o 3 pagine)
- in media ogni parola = 6 byte
- 500 mila differenti parole tra tutti i documenti

Il corpus sarà grande circa 6 GB

Quindi la matrice termini-documenti:
0,5 miliardi di miliardi di 0 e 1

Grandi collezioni

Almeno il 99,8% della matrice sarà uguale a 0

Esiste un modo migliore?

Indice invertito

Ad ogni parola verranno associati solo gli indici delle colonne relative ai documenti in cui quella parola compare, ovvero dove la matrice contiene 1.

parola \rightarrow $\text{doc}_i, \text{doc}_j, \text{doc}_k$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Esempio:

Brutus: 1, 2, 4

Realizzazione

Riscrivere il motore di ricerca per utilizzare gli indici invertiti invece che la matrice termini documenti.

Realizzazione

```
1 from collections import defaultdict
2
3 from doc_loader import load_docs
4
5
6 class InvertedIndex:
7     def __init__(self, all_docs):
8         self.all_words = sorted(set().union(*[d['words'] for d in all_docs]))
9         self.all_docs = all_docs
10
11     class Term:
12         all_docs_idxs = list(range(len(all_docs)))
13
14         def __init__(self, doc_ids=None):
15             self.docs = sorted(doc_ids or self.all_docs_idxs)
16
17         def __and__(self, other):
18             return Term(set(self.docs) & set(other.docs))
19
20         def __or__(self, other):
21             return Term(set(self.docs) | set(other.docs))
22
23         def __invert__(self):
24             return Term(set(self.all_docs_idxs) - set(self.docs))
25
26     self.Term = Term
27     self.index = defaultdict(self.Term)
28
29     for word in self.all_words:
30         word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['words']]
31         self.index[word] = self.Term(word_docs)
32
33     def term_from_word(self, word):
34         negated = False
35         if word.startswith '!':
36             word = word[1:]
37             negated = True
38         term = self.index[word]
39         return ~term if negated else term
40
41     def query(self, q):
42         if not q:
43             return []
44         words = q.split()
45         word, *words = words
46         term = self.term_from_word(word)
47         while words:
48             op, word, *words = words
49             word_term = self.term_from_word(word)
50             if op == '&':
51                 term = term & word_term
52             elif op == '|':
53                 term = term | word_term
54         return [self.all_docs[idx] for idx in set(term.docs)]
55
56
57 if __name__ == '__main__':
58     docs = load_docs()
59     iindex = InvertedIndex(docs)
60     query = input('Cosa vuoi cercare?\n')
61     print('Risultati:')
62     results = iindex.query(query)
63     print(f'{len(results)}/{len(docs)}')
64     for doc in results:
65         print(f'{doc["title"]} - {doc["year"]}')

```

Phrase query

Esiste la possibilità di cercare *frasi* invece di query booleane, focalizzandosi sulla posizione delle singole parole.

Esempio. Se cerco “pasta alla norma” non mi interessa trovare un testo del tipo: “...di **norma**, nella francia del XVIII secolo, si faceva riferimento **alla** convinzione che i nobili non fossero fatti della stessa **pasta**...”

Nella frase della query, inoltre, non possono essere presenti altre parole tra quelle cercate.

Esempio. Se cerco “pasta alla norma” non mi interessa trovare un testo del tipo: “...la **pasta** siciliana denominata **alla norma**...”

Phrase query

Per cercare una frase:

- scorrere contemporaneamente le liste dei documenti, avanzando sempre con l'indice minore
- se viene trovato lo stesso documento in tutte e tre le liste:
 - 1) cercare nell'insieme del documento la prima parola
 - 2) cercare nell'insieme del documento la seconda parola e controllare che sia consecutiva alla prima
 - 3) cercare nell'insieme del documento la terza parola e controllare che sia consecutiva alla seconda
 - 4) se il check ai punti (2) o (3) dà esito negativo, cercare una nuova occorrenza della prima parola nel documento

Per semplificare è stata fatta l'ipotesi che gli indici delle parole nei documenti sono stati inseriti all'interno di insiemi, invece che di liste. (N.B. non è una tecnica allo stato dell'arte!)

Phrase query

Modificare il motore di ricerca in modo che sia in grado di accettare delle frasi come query.

Considerare che la query deve essere • booleana • frase, non entrambe

(cioè se si cerca "search engine", un documento deve contenerle proprio in quell'ordine per essere incluso nei risultati)

Phrase query

```
1 import re
2 from collections import defaultdict
3
4 from doc_loader import load_docs
5
6
7 class InvertedIndex:
8     def __init__(self, all_docs):
9         self.all_words = sorted(set().union(*[d['words'] for d in all_docs]))
10        self.all_docs = all_docs
11
12        class Term:
13            all_docs_idxs = list(range(len(all_docs)))
14
15            def __init__(self, doc_ids=None):
16                actual_doc_ids = doc_ids or []
17                self.docs = sorted(actual_doc_ids)
18                self.pos = {idx: set() for idx in actual_doc_ids}
19
20            def __and__(self, other):
21                return Term(set(self.docs) & set(other.docs))
22
23            def __or__(self, other):
24                return Term(set(self.docs) | set(other.docs))
25
26            def __invert__(self):
27                return Term(set(self.all_docs_idxs) - set(self.docs))
28
29        self.Term = Term
30        self.index = defaultdict(self.Term)
31
32        for word in self.all_words:
33            word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['words']]
34            self.index[word] = self.Term(word_docs)
35
36        for doc_idx, doc in enumerate(self.all_docs):
37            words_list = re.split(r'W+', doc['content']).lower()
38            for word_idx, word in enumerate(words_list):
39                self.index[word].pos[doc_idx].add(word_idx)
40
41        def term_from_word(self, word):
42            negated = False
43            if word.startswith '!'):
44                word = word[1:]
45                negated = True
46            term = self.index[word]
47            return ~term if negated else term
48
49        def bool_query(self, q):
50            if not q:
51                return []
52            words = q.split()
53            word, *words = words
54            term = self.term_from_word(word)
55            while words:
56                op, word, *words = words
57                word_term = self.term_from_word(word)
58                if op == '&':
59                    term = term & word_term
60                elif op == '|':
61                    term = term | word_term
62            return [self.all_docs[idx] for idx in set(term.docs)]
63
64        def phrase_query(self, q):
65            if not q:
66                return []
67            words = q.split()
68            terms = [self.index[word] for word in words]
69            if not terms:
70                return []
71            possible_docs = set(terms[0].docs).intersection(*[set(t.docs) for t in terms])
72            results_ids = set()
73            for doc_idx in possible_docs:
74                first_word_pos = terms[0].pos[doc_idx]
75                for pos in first_word_pos:
76                    if all([(pos + i) in t.pos[doc_idx] for i, t in enumerate(terms)]):
77                        results_ids.add(doc_idx)
78            return [self.all_docs[idx] for idx in results_ids]
79
80
81 if __name__ == '__main__':
82     docs = load_docs()
83     iindex = InvertedIndex(docs)
84     query = input("Cosa vuoi cercare?\n")
85     print("Risultati:")
86     if set(query) & {'!', '&', '|'}:
87         results = iindex.bool_query(query)
88     else:
89         results = iindex.phrase_query(query)
90     print(f"({len(results)}) / ({len(docs)})")
91     for doc in results:
92         print(f"[{doc['title']}] - {doc['year']}")
```

Stemming

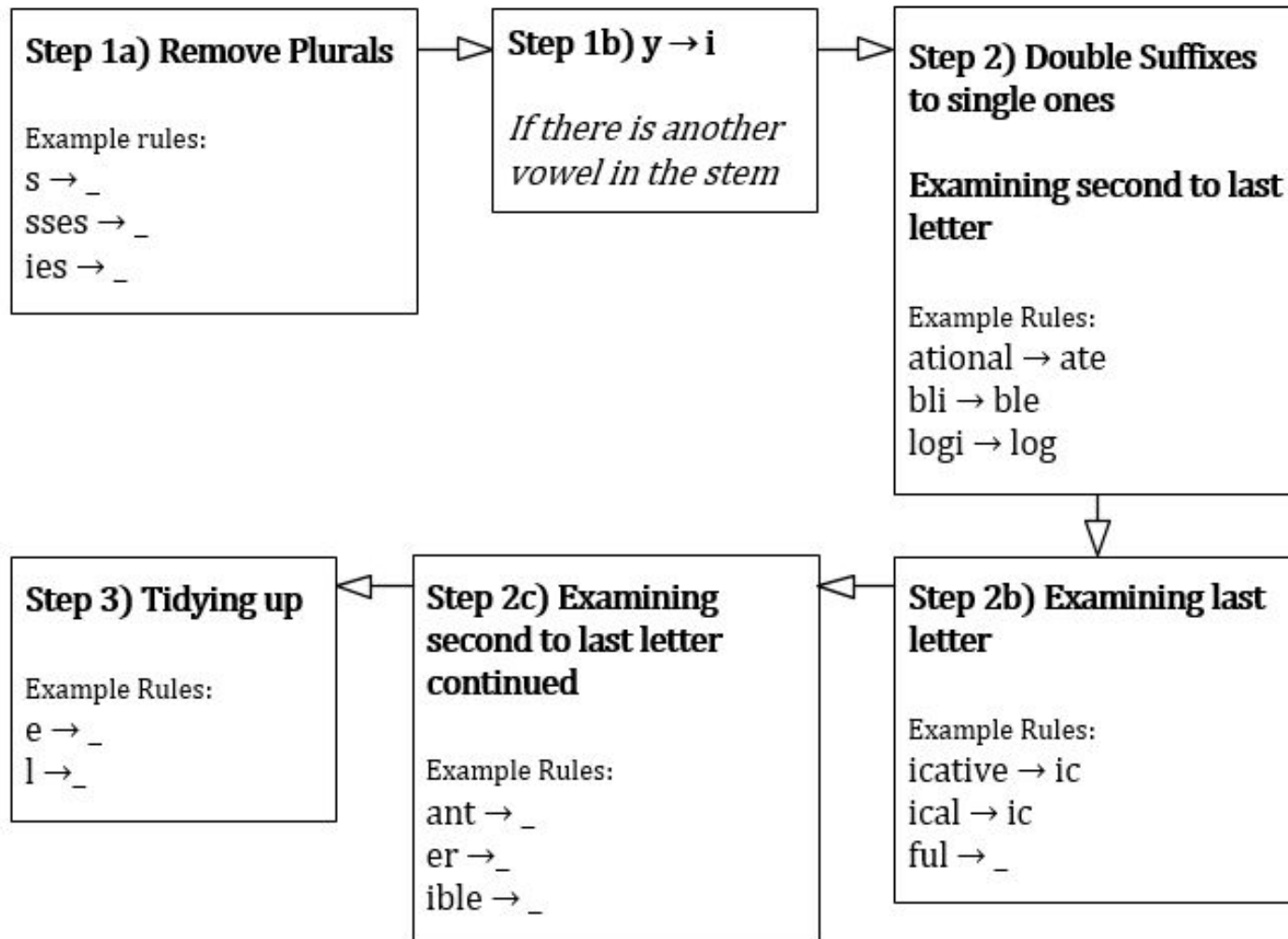
Lo **stemming** è un metodo che identifica la **radice** delle parole per generalizzare le operazioni di interrogazione e selezione dei documenti in un archivio.

Trasforma le parole nella loro forma flessa e più generale.



Stemming

ALGORITMO PORTER STEMMER



Stemming

```
1 #Definizione dello stemmer
2
3 from nltk.stem import PorterStemmer
4 from nltk.tokenize import sent_tokenize, word_tokenize
5 ps = PorterStemmer()
6
7 #Scelta di parole con stem simili:
8
9 example_words = ["python","pythoner","pythoning","pythoned","pythonly"]
10
11 #Processo di stemming:
12
13 for w in example_words:
14     print(ps.stem(w))
```

Stemming

ALGORITMO SNOWBALL STEMMER

Alcune regole sui suffissi:

ILY -----> ILI

LY -----> *Null*

SS -----> SS

S -----> *Null*

ED -----> E,*Null*

Risulta essere più aggressivo del Porter Stemmer, le cui regole derivano proprio da questo algoritmo

Stemming

```
1 #Definizione dello stemmer
2 import nltk
3 from nltk.stem.snowball import SnowballStemmer
4
5 # Parametri di linguaggio
6 snow_stemmer = SnowballStemmer(language='english')
7
8 # Lista di parole
9 words = ['cared','university','fairly','easily','singing',
10         'sings','sung','singer','sportingly']
11
12 # Stemming
13 stem_words = []
14 for w in words:
15     x = snow_stemmer.stem(w)
16     stem_words.append(x)
17
18 # Print
19 for e1,e2 in zip(words,stem_words):
20     print(e1+' ----> '+e2)
```

Stemming

Inserire la funzione di Stemming nel motore di ricerca.

Stemming

```
1 import re
2 from collections import defaultdict
3 from nltk.stem.snowball import SnowballStemmer
4
5 from doc_loader import load_docs
6
7 stemmer = SnowballStemmer(language='english')
8
9
10 class InvertedIndex:
11     def __init__(self, all_docs):
12         for doc in all_docs:
13             doc['stemmed_words'] = set(map(stemmer.stem, doc['words']))
14         self.all_words = sorted(set().union(*[doc['stemmed_words'] for doc in all_docs]))
15         self.all_docs = all_docs
16
17     class Term:
18         all_docs_idxs = list(range(len(all_docs)))
19
20         def __init__(self, doc_ids=None):
21             actual_doc_ids = doc_ids or []
22             self.docs = sorted(actual_doc_ids)
23             self.pos = {idx: set() for idx in actual_doc_ids}
24
25         def __and__(self, other):
26             return Term(set(self.docs) & set(other.docs))
27
28         def __or__(self, other):
29             return Term(set(self.docs) | set(other.docs))
30
31         def __invert__(self):
32             return Term(set(self.all_docs_idxs) - set(self.docs))
33
34     self.Term = Term
35     self.index = defaultdict(self.Term)
36
37     for word in self.all_words:
38         word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['stemmed_words']]
39         self.index[word] = self.Term(word_docs)
40
41     for doc_idx, doc in enumerate(self.all_docs):
42         words_list = re.split(r'\W+', doc['content']).lower()
43         for word_idx, word in enumerate(words_list):
44             word = stemmer.stem(word)
45             self.index[word].pos[doc_idx].add(word_idx)
46
47     def term_from_word(self, word):
48         negated = False
49         if word.startswith '!'):
50             word = word[1:]
51             negated = True
52         word = stemmer.stem(word)
53         term = self.index[word]
54         return ~term if negated else term
55
56     def bool_query(self, q):
57         if not q:
58             return []
59         words = q.split()
60         word, *words = words
61         term = self.term_from_word(word)
62         while words:
63             op, word, *words = words
64             word_term = self.term_from_word(word)
65             if op == '&':
66                 term = term & word_term
67             elif op == '|':
68                 term = term | word_term
69         return [self.all_docs[idx] for idx in set(term.docs)]
70
71     def phrase_query(self, q):
72         if not q:
73             return []
74         words = q.split()
75         terms = [self.index[stemmer.stem(word)] for word in words]
76         if not terms:
77             return []
78         possible_docs = set(terms[0].docs).intersection(*[set(t.docs) for t in terms])
79         results_ids = set()
80         for doc_idx in possible_docs:
81             first_word_pos = terms[0].pos[doc_idx]
82             for pos in first_word_pos:
83                 if all([(pos + 1) in t.pos[doc_idx] for i, t in enumerate(terms)]):
84                     results_ids.add(doc_idx)
85         return [self.all_docs[idx] for idx in results_ids]
86
87
88 if __name__ == '__main__':
89     docs = load_docs()
90     iindex = InvertedIndex(docs)
91     query = input("Cosa vuoi cercare?\n")
92     print("Risultati:")
93     if set(query) & {'!', '&', '|'}:
94         results = iindex.bool_query(query)
95     else:
96         results = iindex.phrase_query(query)
97     print(f'({len(results)}/{len(docs)})')
98     for doc in results:
99         print(f'[{doc["title"]} - {doc["year"]}')
```

Lemmatization

Il **Lemmatization** è l'operazione di raggruppamento di varie forme inflesse di una stessa parola, estrapolate dal contesto in cui si trovano, in modo da poterle analizzare come un unico elemento in termini di radice e significato (*lemma*).

Richiede un dizionario per operare

Stemming

Form	Suffix	Stem
studies	-es	studi
study	-ing	study
niñas	-as	niñ
niñez	-ez	niñ

Lemmatization

Form	Morphological information	Lemma
studies	Third person, singular number, present tense of the verb study	study
studying	Gerund of the verb study	study
niñas	Feminine gender, plural number of the noun niño	niño
niñez	Singular number of the noun niñez	niñez

Lemmatization

Installare spacy e solo dopo scaricare il modello che andremo ad usare, usando:

```
python -m spacy download en_core_web_sm
```

```
1 import spacy
2
3 # Inizializzazione
4 nlp = spacy.load('en_core_web_sm')
5
6 sentence = "The striped bats are hanging on their feet for best"
7
8 # Analisi della frase
9 doc = nlp(sentence)
10
11 # Estrazione del lemma per ogni parola e unificazione
12 print(" ".join([token.lemma_ for token in doc]))
```

Lemmatization

Utilizzare Tokenizzazione e Lemmatizzazione di spacy nel motore di ricerca.

Lemmatization

```
1 import json
2 import re
3 from collections import defaultdict
4 from pathlib import Path
5
6 import spacy
7
8 nlp = spacy.load("en_core_web_sm")
9
10 def clean_token(token):
11     return re.sub('[^0-9a-zA-Z]+', '', token.lower())
12
13
14 def load_docs():
15     works_dir = Path('works') / 'splitted'
16     docs = []
17     for work in works_dir.iterdir():
18         with open(work, 'r') as work_file:
19             doc = json.load(work_file)
20             doc['tokens'] = [clean_token(token.lemma_) for token in nlp(doc['content']) if not token.is_punct]
21             doc['lemmas'] = set(doc['tokens'])
22             docs.append(doc)
23     return docs
24
25
26 class InvertedIndex:
27     def __init__(self, all_docs):
28         self.all_words = sorted(set().union(*[d['lemmas'] for d in all_docs]))
29         self.all_docs = all_docs
30
31         class Term:
32             all_docs_idxs = list(range(len(all_docs)))
33
34             def __init__(self, doc_ids=None):
35                 actual_doc_ids = doc_ids or []
36                 self.docs = sorted(actual_doc_ids)
37                 self.pos = {(idx: set) for idx in actual_doc_ids}
38
39             def __and__(self, other):
40                 return Term(set(self.docs) & set(other.docs))
41
42             def __or__(self, other):
43                 return Term(set(self.docs) | set(other.docs))
44
45             def __invert__(self):
46                 return Term(set(self.all_docs_idxs) - set(self.docs))
47
48         self.Term = Term
49         self.index = defaultdict(self.Term)
50
51     for word in self.all_words:
52         word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['lemmas']]
53         self.index[word] = self.Term(word_docs)
54
55     for doc_idx, doc in enumerate(self.all_docs):
56         for word_idx, word in enumerate(doc['tokens']):
57             self.index[word].pos[doc_idx].add(word_idx)
58
```

```
59 def term_from_word(self, word):
60     negated = False
61     if word.startswith('!'):
62         word = word[1:]
63         negated = True
64     term = self.index[word]
65     return ~term if negated else term
66
67 def bool_query(self, q):
68     if not q:
69         return []
70     words = q.split()
71     word, *words = words
72     term = self.term_from_word(word)
73     while words:
74         op, word, *words = words
75         word_term = self.term_from_word(word)
76         if op == '&':
77             term = term & word_term
78         elif op == '|':
79             term = term | word_term
80     return [self.all_docs[idx] for idx in set(term.docs)]
81
82 def phrase_query(self, q):
83     if not q:
84         return []
85     terms = [self.index[token.lemma_] for token in nlp(q)]
86     if not terms:
87         return []
88     possible_docs = set(terms[0].docs).intersection(*[set(t.docs) for t in terms])
89     results_ids = set()
90     for doc_idx in possible_docs:
91         first_word_pos = terms[0].pos[doc_idx]
92         for pos in first_word_pos:
93             if all([(pos + i) in t.pos[doc_idx] for i, t in enumerate(terms)]):
94                 results_ids.add(doc_idx)
95     return [self.all_docs[idx] for idx in results_ids]
96
97
98 if __name__ == '__main__':
99     docs = load_docs()
100     iindex = InvertedIndex(docs)
101     query = input("Cosa vuoi cercare?\n")
102     print(Risultati:)
103     if set(query) & {'!', '&', '|'}:
104         results = iindex.bool_query(query)
105     else:
106         results = iindex.phrase_query(query)
107     print(f"({len(results)})/({len(docs)})")
108     for doc in results:
109         print(f"({doc['title']}) - ({doc['year']})")

```

Spelling correction

Statisticamente, tra il 10% ed il 15% delle ricerche effettuate sui motori di ricerca contengono errori di battitura.

Lo **Spelling Correction** consente di comprendere cosa si sta cercando nonostante tali errori.

Spelling correction

Solitamente ci si basa sulla “distanza” tra una parola ed un’altra.

Questo tipo di distanza è detta

EDIT DISTANCE

Si basa sul conteggio del numero minimo di operazioni da effettuare per trasformare una stringa in un’altra.

Tra le tipologie di edit distance, la più diffusa è la

LEVENSHTEIN DISTANCE

Si basa sul conteggio del numero minimo di modifiche su singoli da caratteri da effettuare (inserimento, cancellazione e sostituzione) da effettuare per trasformare una stringa in un’altra.

Spelling correction

Da notare che due query potrebbero essere uguali anche se il loro contenuto non ha alcun senso.

Nello Spelling Correction ci si limita ad una **correzione formale** del testo.

Spelling correction

Implementare lo Spelling Correction nel motore di ricerca:

- **solamente per le phrase queries**
- se già esiste la parola -> ok
- altrimenti calcolo la Levenshtein distance con tutte le altre
- la correggo con quella con distanza più piccola
- a parità: una vale l'altra (per il momento)

Mostrare all'utente la query corretta che si sta utilizzando

Hint: utilizzare il pacchetto `python-Levenshtein`

Spelling correction

```
1 import json
2 import re
3 from collections import defaultdict
4 from pathlib import Path
5
6 import spacy
7 from Levenshtein import distance
8
9 nlp = spacy.load("en_core_web_sm")
10
11 def clean_token(token):
12     return re.sub('[^0-9a-zA-Z]+', '', token.lower())
13
14
15 def load_docs():
16     works_dir = Path('works') / 'splitted'
17     docs = []
18     for work in works_dir.iterdir():
19         with open(work, 'r') as work_file:
20             doc = json.load(work_file)
21             doc['tokens'] = []
22             doc['lemmas'] = set()
23             doc['words'] = set()
24             for token in nlp(doc['content']):
25                 if token.is_punct:
26                     continue
27                 word = clean_token(token.text)
28                 lemma = clean_token(token.lemma_)
29                 doc['tokens'].append(lemma)
30                 doc['lemmas'].add(lemma)
31                 doc['words'].add(word)
32             docs.append(doc)
33     return docs
34
35
36 class InvertedIndex:
37     def __init__(self, all_docs):
38         self.all_words = sorted(set().union(*[d['lemmas'] for d in all_docs]))
39         self.all_docs = all_docs
40
41         class Term:
42             all_docs_idxs = list(range(len(all_docs)))
43
44             def __init__(self, doc_ids=None):
45                 actual_doc_ids = doc_ids or self.all_docs_idxs
46                 self.docs = sorted(actual_doc_ids)
47                 self.pos = {idx: set() for idx in actual_doc_ids}
48
49             def __and__(self, other):
50                 return Term(set(self.docs) & set(other.docs))
51
52             def __or__(self, other):
53                 return Term(set(self.docs) | set(other.docs))
54
55             def __invert__(self):
56                 return Term(set(self.all_docs_idxs) - set(self.docs))
57
58         self.Term = Term
59         self.index = defaultdict(self.Term)
60
61         for word in self.all_words:
62             word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['lemmas']]
63             self.index[word] = self.Term(word_docs)
64
65         for doc_idx, doc in enumerate(self.all_docs):
66             for word_idx, word in enumerate(doc['tokens']):
67                 self.index[word].pos[doc_idx].add(word_idx)
68
69     def term_from_word(self, word):
70         negated = False
71         if word.startswith('!'):
72             word = word[1:]
73             negated = True
74         term = self.index[word]
75         return ~term if negated else term
76
```

```
77 def bool_query(self, q):
78     if not q:
79         return []
80     words = q.split()
81     word_term = self.term_from_word(word)
82     term = self.term_from_word(word)
83     while words:
84         op, word, *words = words
85         word_term = self.term_from_word(word)
86         if op == '&':
87             term = term & word_term
88         elif op == '|':
89             term = term | word_term
90     return [self.all_docs[idx] for idx in set(term.docs)]
91
92 def phrase_query(self, q):
93     if not q:
94         return []
95     terms = [self.index[token.lemma_] for token in nlp(q)]
96     if not terms:
97         return []
98     possible_docs = set(terms[0].docs).intersection(*[set(t.docs) for t in terms])
99     results_ids = set()
100     for doc_idx in possible_docs:
101         first_word_pos = terms[0].pos[doc_idx]
102         for pos in first_word_pos:
103             if all([(pos + i) in t.pos[doc_idx] for i, t in enumerate(terms)]):
104                 results_ids.add(doc_idx)
105     return [self.all_docs[idx] for idx in results_ids]
106
107
108 class SpellingCorrection:
109     def __init__(self, all_docs):
110         self.all_words = set().union(*[doc['words'] for doc in all_docs])
111
112     def nearest(self, word):
113         distances = [(distance(word, voc_word), voc_word) for voc_word in self.all_words]
114         distances = sorted(distances, key=lambda e: e[0])
115         return distances[0][1]
116
117     def correct(self, query):
118         words = query.split()
119         correct_words = []
120         for word in words:
121             if word in self.all_words:
122                 correct_words.append(word)
123             else:
124                 correct_words.append(self.nearest(word))
125         return ' '.join(correct_words)
126
127
128 if __name__ == '__main__':
129     docs = load_docs()
130     corrector = SpellingCorrection(docs)
131     iindex = InvertedIndex(docs)
132     query = input('Cosa vuoi cercare?\n')
133     if set(query) & {'!', '&', '|'}:
134         results = iindex.bool_query(query)
135     else:
136         correct_query = corrector.correct(query)
137         if query != correct_query:
138             print('Risultati per:')
139             print(correct_query)
140             results = iindex.phrase_query(correct_query)
141     print('Risultati:')
142     print(f'{len(results)} / {len(docs)}')
143     for doc in results:
144         print(f'({doc["title"]} - {doc["year"]})')
```

Snippet dei risultati

Potrebbe essere utile per l'utente avere degli **Snippet dei risultati** contenenti la query

The image shows a Google search interface with the query "best rank" entered in the search bar. The search bar is annotated with a red arrow pointing to it from the word "keyword". Below the search bar, the text "About 375,000,000 results (0.30 seconds)" is visible. On the left side, there is a sidebar with links to "Everything", "Images", "Videos", "News", and "More". A red arrow points from the word "URL" to the first search result's URL. The first search result is titled "Internet Marketing & Search Engine Marketing Company - Best Rank ..." and is annotated with a red arrow pointing to it from the word "title". Below the title is a description: "Best Rank is a San Diego based Internet Marketing & Search Engine Marketing Company. We provide SEO, PPC, Web Design & Social Media services." This description is annotated with a red arrow pointing to it from the word "description". The URL "www.bestrank.com/" is also visible. Below the first result, there are links to "Web Design Portfolio", "Search Engine Optimization", "Blog", and "Office Address". Further down, there are links to "Blog - Best Rank" and "Meet Best Rank's Team".

Google

best rank **keyword** Search

About 375,000,000 results (0.30 seconds) Advanced search

Everything
Images
Videos
News
More

URL

title

description

Internet Marketing & Search Engine Marketing Company - **Best Rank ...**
Best Rank is a San Diego based Internet Marketing & Search Engine Marketing Company. We provide SEO, PPC, Web Design & Social Media services.
www.bestrank.com/ - Cached - Similar

Web Design Portfolio Search Engine Optimization
Blog Office Address

More results from bestrank.com »

Blog - **Best Rank**
Best Rank's search engine marketing blog offers free internet marketing ...
www.bestrank.com/blog - Cached - Similar

Meet **Best Rank's Team**
Best Rank Senior Management Matt Walker, Co-Founder and Chief Executive Officer.
www.bestrank.com/team - Cached - Similar

Show more results from bestrank.com

Snippet dei risultati

Aggiungere gli snippet ai risultati del motore di ricerca, solamente per le phrase queries. Prendendo un numero fisso di caratteri prima e dopo.

Restituire uno qualsiasi degli snippet che contengono la frase.

Hint: spacy fornisce la posizione di inizio e fine per ogni token. guardate la proprietà `sent`

Snippet dei risultati

```
1 import json
2 import re
3 from collections import defaultdict
4 from pathlib import Path
5
6 import spacy
7 from Levenshtein import distance
8
9 nlp = spacy.load("en_core_web_sm")
10
11 def clean_token(token):
12     return re.sub('[^0-9a-zA-Z]+', '', token.lower())
13
14
15 def load_docs():
16     works_dir = Path('works') / 'splitted'
17     docs = []
18     for work in works_dir.iterdir():
19         with open(work, 'r') as work_file:
20             doc = json.load(work_file)
21             doc['tokens'] = []
22             doc['lemmas'] = set()
23             doc['words'] = set()
24             doc['positions'] = []
25             for token in nlp(doc['content']):
26                 if token.is_punct:
27                     continue
28                 word = clean_token(token.text)
29                 lemma = clean_token(token.lemma_)
30                 doc['tokens'].append(lemma)
31                 doc['lemmas'].add(lemma)
32                 doc['words'].add(word)
33                 doc['positions'].append((token.sent.start_char, token.sent.end_char))
34             docs.append(doc)
35     return docs
36
37
38 class InvertedIndex:
39     def __init__(self, all_docs):
40         self.all_words = sorted(set().union(*[d['lemmas'] for d in all_docs]))
41         self.all_docs = all_docs
42
43     class Term:
44         all_docs_idxxs = list(range(len(all_docs)))
45
46         def __init__(self, doc_idx=None):
47             actual_doc_ids = doc_idx or self.all_docs_idxxs
48             self.docs = sorted(actual_doc_ids)
49             self.pos = {idx: set() for idx in actual_doc_ids}
50
51         def __and__(self, other):
52             return Term(set(self.docs) & set(other.docs))
53
54         def __or__(self, other):
55             return Term(set(self.docs) | set(other.docs))
56
57         def __invert__(self):
58             return Term(set(self.all_docs_idxxs) - set(self.docs))
59
60     self.Term = Term
61     self.index = defaultdict(self.Term)
62
63     for word in self.all_words:
64         word_docs = [idx for idx, doc in enumerate(self.all_docs) if word in doc['lemmas']]
65         self.index[word] = self.Term(word_docs)
66
67     for doc_idx, doc in enumerate(self.all_docs):
68         for word_idx, word in enumerate(doc['tokens']):
69             self.index[word].pos[doc_idx].add(word_idx)
70
71     def term_from_word(self, word):
72         negated = False
73         if word.startswith("!"):
74             word = word[1:]
75             negated = True
76         term = self.index[word]
77         return ~term if negated else term
78
79     def bool_query(self, q):
80         if not q:
81             return []
82         words = q.split()
83         word, *words = words
84         term = self.term_from_word(word)
85         while words:
86             op, word, *words = words
87             word_term = self.term_from_word(word)
88             if op == '&':
89                 term = term & word_term
90             elif op == '|':
91                 term = term | word_term
92             return [(self.all_docs[idx], (0, 0)) for idx in set(term.docs)]
93
94     def phrase_query(self, q):
95         if not q:
96             return []
97         terms = [self.index[token.lemma_] for token in nlp(q)]
98         if not terms:
99             return []
100         possible_docs = set(terms[0].docs).intersection(*[set(t.docs) for t in terms])
101         results_ids = {}
102         for doc_idx in possible_docs:
103             first_word_pos = terms[0].pos[doc_idx]
104             for pos in first_word_pos:
105                 if all([(pos + i) in t.pos[doc_idx] for i, t in enumerate(terms)]):
106                     the_doc = self.all_docs[doc_idx]
107                     content = the_doc['content']
108                     first_pos_start = the_doc['positions'][pos][0]
109                     last_pos_end = the_doc['positions'][pos+len(terms)-1][1]
110                     results_ids[doc_idx] = (max(0, first_pos_start - 50), last_pos_end + 50)
111
112         return [(self.all_docs[idx], snippet_pos) for idx, snippet_pos in results_ids.items()]
113
114
115 class SpellingCorrection:
116     def __init__(self, all_docs):
117         self.all_words = set().union(*[doc['words'] for doc in all_docs])
118
119     def nearest(self, word):
120         distances = [(distance(word, voc_word), voc_word) for voc_word in self.all_words]
121         distances = sorted(distances, key=lambda e: e[0])
122         return distances[0][1]
123
124     def correct(self, query):
125         words = query.split()
126         correct_words = []
127         for word in words:
128             if word in self.all_words:
129                 correct_words.append(word)
130             continue
131             correct_words.append(self.nearest(word))
132         return ' '.join(correct_words)
133
134
135 if __name__ == '__main__':
136     docs = load_docs()
137     corrector = SpellingCorrection(docs)
138     iindex = InvertedIndex(docs)
139     query = input("Cosa vuoi cercare?\n")
140     if set(query) & {'!', '&', '|'}:
141         results = iindex.bool_query(query)
142     else:
143         correct_query = corrector.correct(query)
144         if query != correct_query:
145             print("Risultati per:")
146             print(correct_query)
147         results = iindex.phrase_query(correct_query)
148     print("Risultati:")
149     print(f'{len(results)} / {len(docs)}')
150     for doc, (snippet_start, snippet_end) in results:
151         print(f'{doc["title"]} - {doc["year"]}')
152         print(doc["content"][snippet_start:snippet_end])
153     print()
```

Scoring dei risultati

Trovati i documenti giusti, qual è il più rilevante?

Quali parametri vengono utilizzati?

Term frequency

La **Term Frequency** indica la frequenza con cui occorre una singola parola in un documento.

Importanza = frequenza?

La parola più frequente in un documento è anche la più importante?

Mostrare le dieci parole più frequenti di ogni doc

Mostrare le dieci parole (lemmi) che hanno frequenza maggiore in ogni documento

Ed anche le dieci più frequenti in tutto il dataset

Mostrare le dieci parole più frequenti di ogni doc

```
1 import json
2 import re
3 from collections import Counter
4 from pathlib import Path
5
6 import spacy
7
8 spacy.require_gpu()
9 nlp = spacy.load("en_core_web_sm")
10
11 def clean_token(token):
12     return re.sub('[^0-9a-zA-Z]+', '', token.lower())
13
14 def load_docs():
15     works_dir = Path('works') / 'splitted'
16     meta_counter = Counter()
17     for work in works_dir.iterdir():
18         with open(work, 'r') as work_file:
19             doc = json.load(work_file)
20             print(f'{{doc[\'title\']}} - {{doc[\'year\']}}')
21             lemmas = [clean_token(token.lemma_) for token in nlp(doc[\'content\']) if not token.is_punct]
22             counter = Counter([l for l in lemmas if l])
23             print(counter.most_common(10))
24             meta_counter.update(dict(counter.most_common()))
25     print('Tutti i docs:')
26     print(meta_counter.most_common(10))
27
28 if __name__ == '__main__':
29     load_docs()
```

Stop words

Le **Stop Words** sono tutte quelle parole che hanno un'elevata frequenza di utilizzo in una particolare lingua, ma che sono ritenute poco significative.

Mostrare le dieci parole non-stop più frequenti

Mostrare le dieci parole (lemmi) che hanno frequenza maggiore in ogni documento, **ma che non sono stopwords**

Ed anche le dieci più frequenti in tutto il dataset

HINT: in spacy, il token ha la proprietà `is_stop`

Stop words - le eliminiamo?

Non sempre vanno eliminate!

Esempio: *"To be or not to be"*

...perciò lasciamole

Inverse document frequency

Solitamente le parole meno utilizzate sono le più caratterizzanti

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

La presenza del logaritmo rende maggiormente confrontabili due situazioni molto differenti in termini di numero di documenti

N : numero totale di documenti

df_t : Document Frequency, numero di documenti che contengono il termine t

Peso di un termine

Il prodotto tra il *Document Frequency* e l'*Inverse Document Frequency* restituisce una misura di importanza del termine t per il documento d .

$$w_{t,d} = tf_{t,d} \cdot idf_t$$

Tf-idf query

Il punteggio di un documento sarà, in definitiva, dato dalla somma dei pesi tf-idf di ogni termine della query, relativi a quel documento.

Tf-idf query

Modificare il motore di ricerca per ordinare i risultati in base ai punteggi tf-idf.

Per le query booleane il tfidf di un termine per un documento è lo stesso a prescindere se utilizziamo and oppure or.

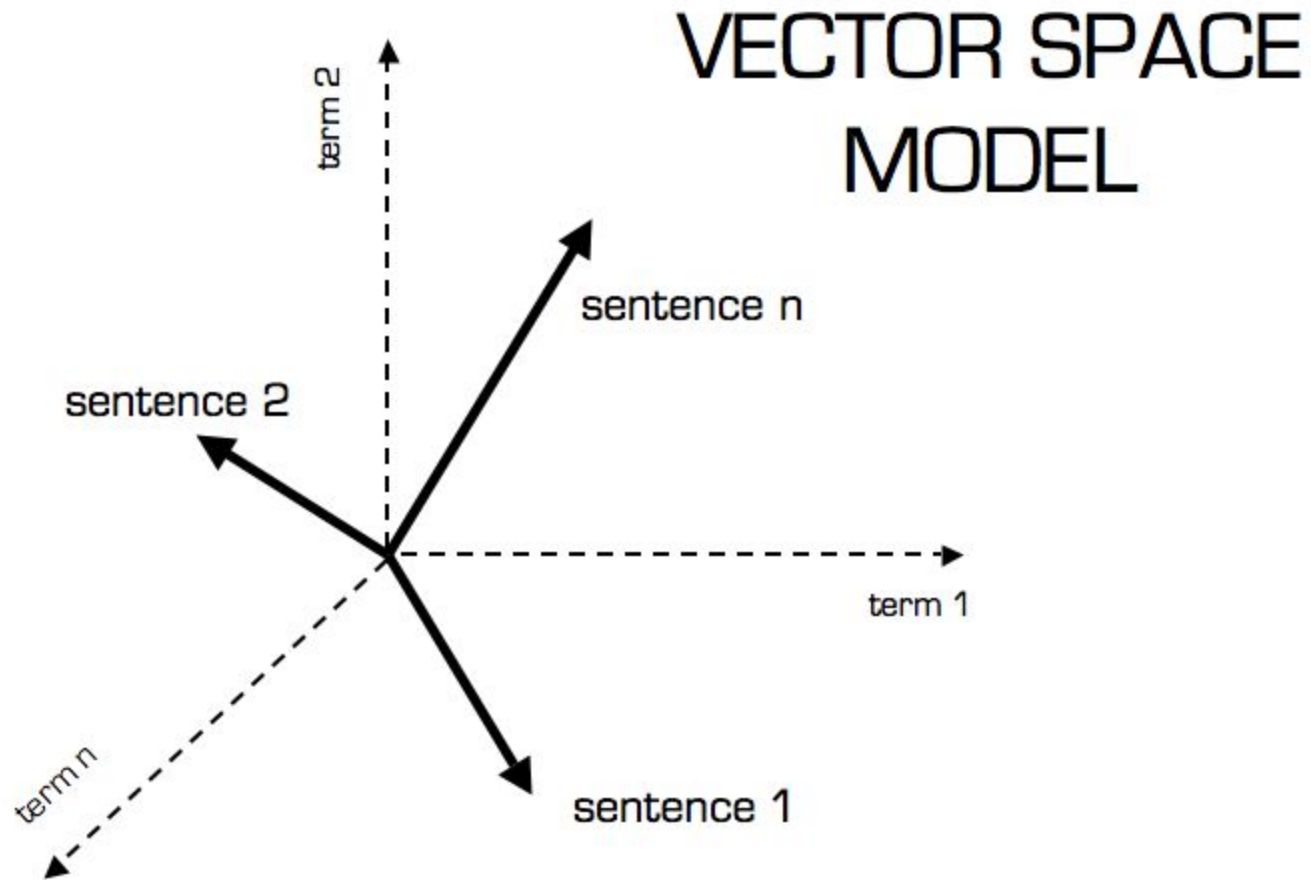
Tf-idf query

file: 11_search_engine_tfidf.py

Problemi con tf-idf

Cosa succede, però, se un documento è molto più lungo di un altro?

Vector space model



Vector space model - vettore

Un vettore è composto dal punteggio tf-idf di ogni termine.

Quindi come è fatta una ricerca?

Vector space model - cosine similarity

Dati due vettori ***a*** e ***b***, il loro grado di similitudine viene espresso attraverso il coseno dell'angolo compreso tra di essi, calcolato come il rapporto tra il loro prodotto scalare ed i loro moduli.

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

Possono essere distinti tre casi particolari:

$$\cos(\theta) = \begin{cases} -1 & \theta = 180^\circ \\ 0 & \theta = 90^\circ \\ +1 & \theta = 0^\circ \end{cases}$$

Corrispondenza opposta
(non di interesse per noi)

Corrispondenza nulla

Corrispondenza esatta

Vector space model - query

Ogni documento è un vettore

\mathbf{d}_i

La query è un vettore

\mathbf{q}_j

$$w_d = \frac{\mathbf{d}_i \cdot \mathbf{q}_j}{\|\mathbf{d}_i\| \cdot \|\mathbf{q}_j\|} = \frac{\sum_{k=1}^n d_{i,k} \cdot q_{j,k}}{\sqrt{\sum_{k=1}^n d_{i,k}^2} \sqrt{\sum_{k=1}^n q_{j,k}^2}}$$

Vector space model

Implementare lo scoring tramite Vector space model nel motore di ricerca.

Per la cosine similarity utilizzare
`sklearn.metrics.pairwise.cosine_similarity`

Vector space model

file: 12_search_engine_vsm.py

Web Information Retrieval

Nell'ambito del Web, i dati sono nel formato *html*.

Ciò comporta che si dovrà tener conto dei title, degli h1, degli url, degli alt delle immagini, etc.

Web Information Retrieval

Scaricare il dataset da

<http://www.dia.uniroma3.it/db/weir/>

Web Information Retrieval

Per fare il parsing di un html in python si consiglia la libreria
BeautifulSoup

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

Web Information Retrieval

Realizzare un parser dei file html per salvare le seguenti informazioni in un singolo file json

- dominio della pagina
- url della pagina
- titolo della pagina
- tokens nel titolo della pagina (text e lemma)
- titoli di paragrafi
- tokens nei titoli dei paragrafi (text e lemma)
- link con il loro anchor
- tokens negli anchor (text e lemma)
- testo nella pagina
- tokens nel testo della pagina (text, lemma e posizioni)

Web Information Retrieval

file: 13_split_html.py

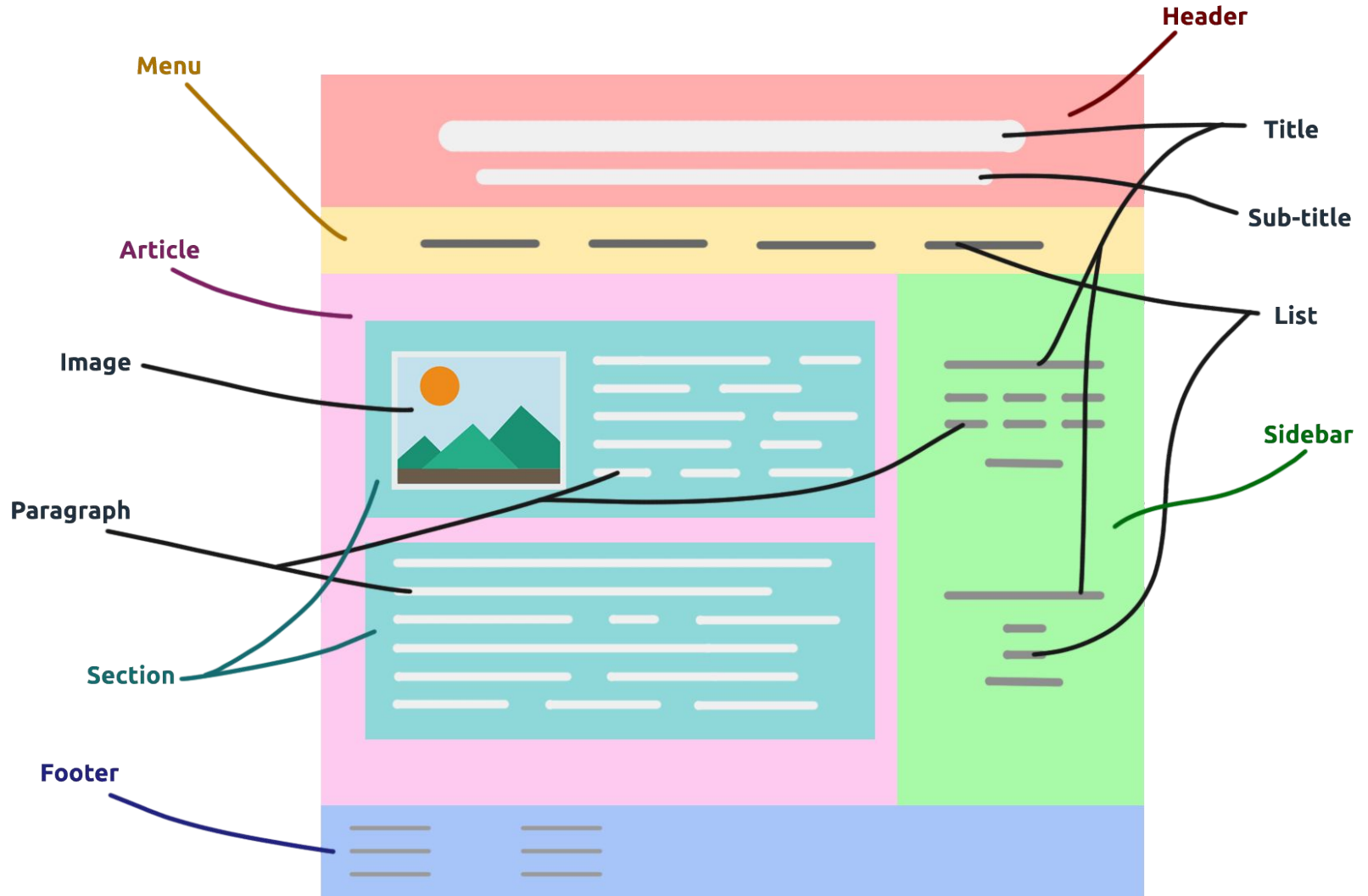
Web Information Retrieval

Modificare il motore di ricerca per caricare i dati elaborati dalle pagine
html

Web Information Retrieval

file: 14_search_engine_html.py

Zone scoring



Zone scoring

Il sistema di attribuzione dei punteggi può essere ulteriormente approfondito andando a vedere la “zona” della pagina in cui compare la query (**Zone Scoring**).

Ogni zona (titolo, paragrafi, contenuto, etc.) viene trattato come un corpus differente. Quindi viene creato un indice per ogni zona e vengono fatte le query per ogni zona.

I coefficienti da attribuire alle varie zone sono complessi da definire. Solitamente sono appresi tramite machine learning.

Zone scoring

Modificare il motore di ricerca per considerare le zone nel punteggio di scoring.

Considerate la “zona titolo” che vale tripla e la “zona paragrafo” che vale doppia rispetto alla semplice “zona contenuto”

Trattate la “zona paragrafo” come concatenazione di tutti i titoli dei paragrafi

Ignorate le posizioni su titoli e paragrafo. Mettetele sempre a 0.

Zone scoring

file: 15_search_engine_zones.py

Importanza di un documento

Non tutti i documenti sono ugualmente importanti.

Ad esempio:

- un blog con opinioni personali
- il sito del sole 24ore
- queste fantastiche slide

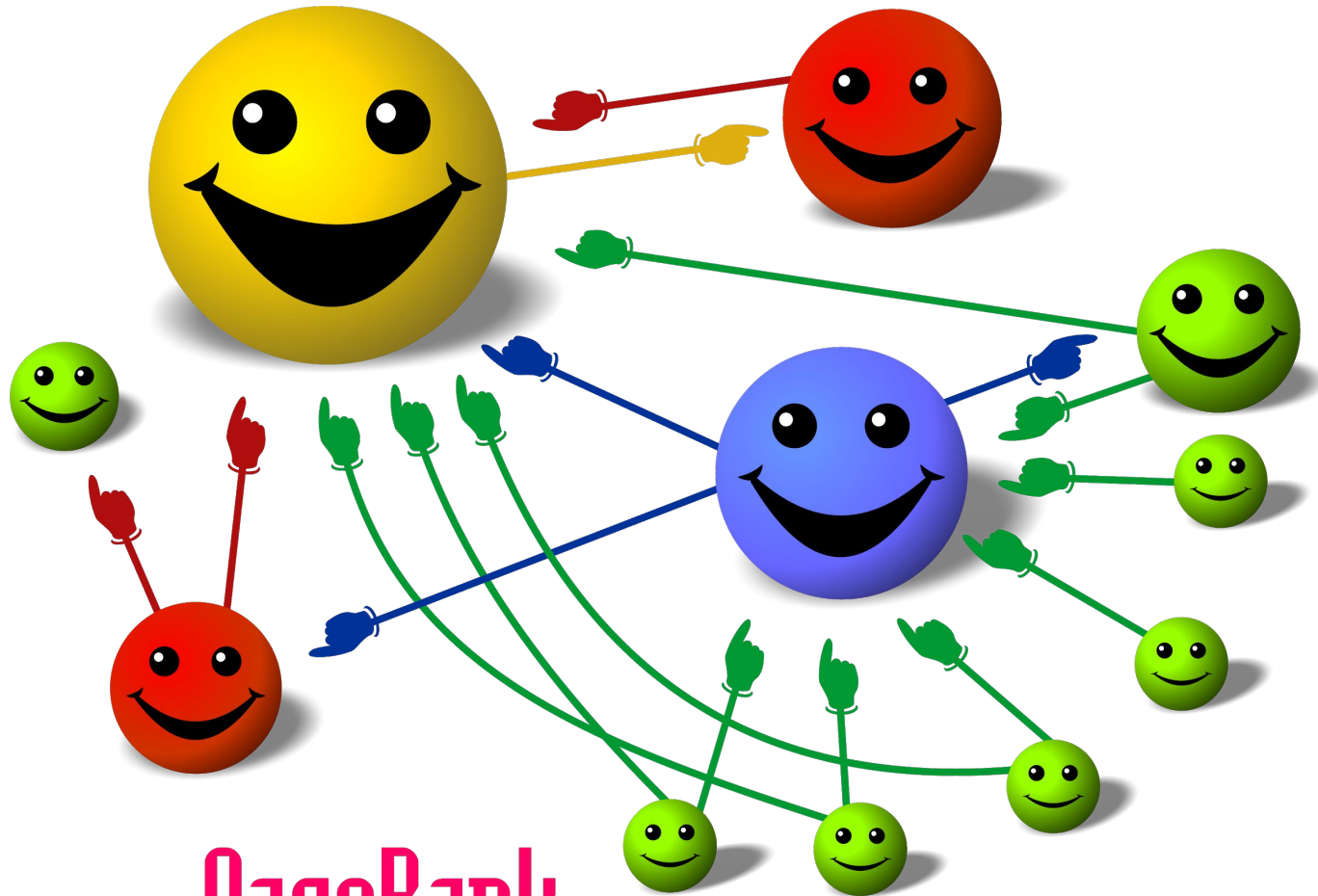
Come giudicare *“l’importanza”*?

Importanza di un documento

L'importanza di un testo viene solitamente giudicata in base alle **interconnessioni** con altri testi:

- citazioni
- link
- etc.

PageRank



PageRank

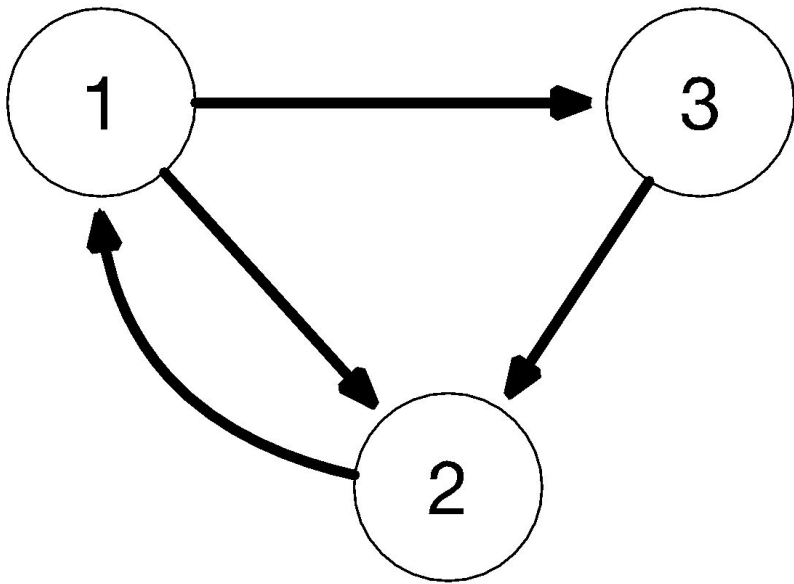
PageRank

Un sito web autorevole è quello che è puntato da siti web autorevoli

Algoritmo di funzionamento

- Inizialmente ogni pagina ha un valore di **PageRank**, detto “peso” della pagina, w_i pari a 1.
- Si ripete la seguente procedura (finchè i valori convergono, ovvero non cambiano più tra una iterazione e l'altra):
 - Ogni pagina divide il suo peso tra le pagine che punta
 - “Normalizzazione”: divido tutti i pesi per il peso massimo tra tutte le pagine

PageRank - Esempio



$$w_1 = w_2$$

$$w_2 = \frac{w_1}{2} + w_3$$

$$w_3 = \frac{w_1}{2}$$

PageRank

Calcolare il PageRank di ogni sito web del dataset (non le pagine) ed utilizzarlo per lo scoring dei risultati sul motore di ricerca
(es: $\text{Score} * \text{PageRank}$)

Per calcolare il pagerank potete utilizzare:

- `from urllib.parse import urlparse` per estrarre il dominio dall'url
- un dizionario del tipo
`{<dominio_src>: {<dominio_tgt>: {'weight': <n_links>}}}`
per creare il grafo
- il metodo `pagerank` di `networkx` restituisce un dizionario

PageRank

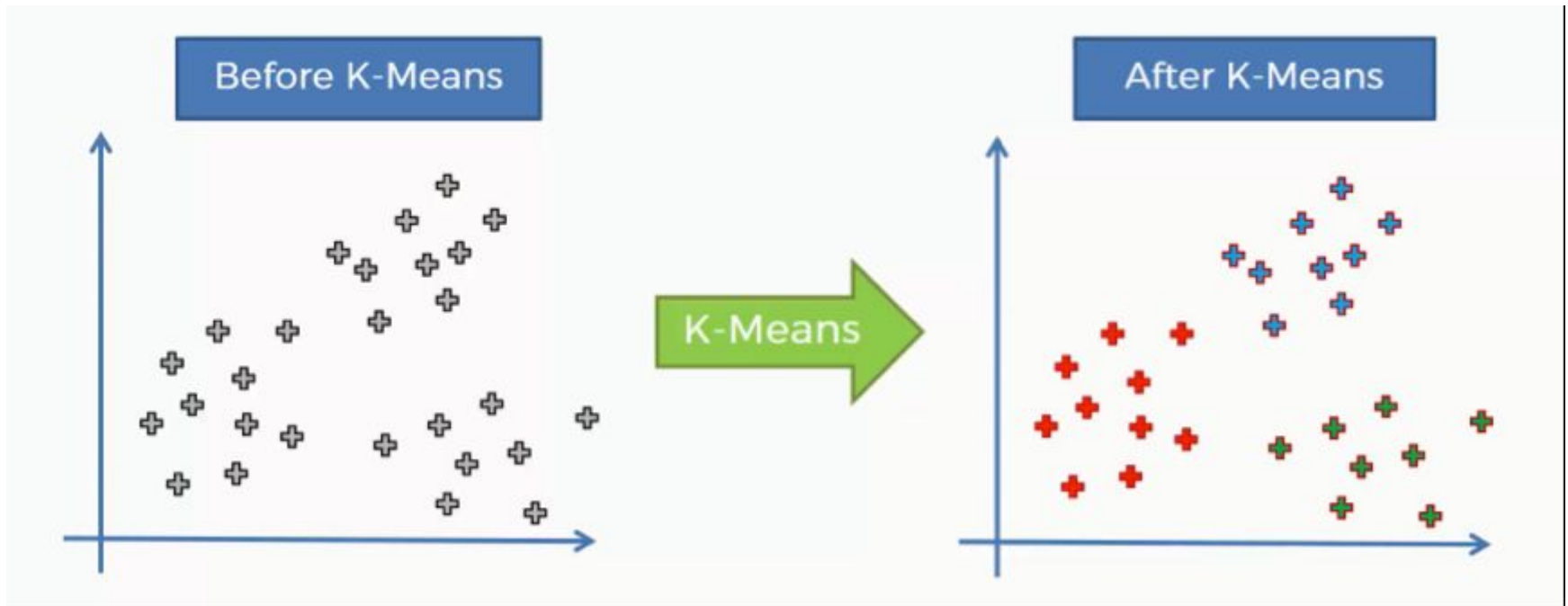
file: 16_search_engine_pagerank.py

Topic clustering

Il processo di **Topic Clustering** suddivide i documenti di cui si è a disposizione in un certo numero di argomenti, non noti a priori, mediante opportuni algoritmi di clustering.

Topic clustering

Uno degli algoritmi maggiormente utilizzati è il *K-Means*, che verrà applicato sullo spazio vettoriale in esame.



Topic clustering

In molte librerie software, non è possibile utilizzare la cosine similarity come distanza nel k-means. Ma se i vettori sono normalizzati (cioè con lunghezza 1), allora la cosine distance e la euclidian distance sono linearmente dipendenti.

Inoltre, per calcolare il topic per un intero dominio (formato da molte pagine ovviamente), si dovrebbe stabilire come assegnare il topic al dominio:
majority, proporzione, etc.

Topic clustering

Applicare la suddivisione in topic sul dataset con l'algoritmo K-Means sui vettori tfidf normalizzati e salvare la classificazione, per ogni dominio in un file a parte.

Visto lo scopo didattico, prendere **solamente 1 pagina per ogni dominio**.

Utilizzare il content, e la stessa funzione cosine_similarity già vista.

Per normalizzare utilizzare

```
from sklearn.preprocessing import normalize
```

E per classificare

```
from sklearn.cluster import KMeans
```

```
classifications = KMeans(n_clusters=10).fit_predict(X_norm)
```

Topic clustering

file: 17_topic_clustering.py

Query classification

Una volta stabiliti i topic presenti nell'insieme di documenti, sarà possibile classificare ogni query (**Query Classification**) ad un topic usando l'algoritmo KNN (*K-Nearest Neighbor*).

In questo modo verrà stabilito uno scoring più preciso sui documenti, sulla base degli argomenti trattati.

Personalised PageRank

È possibile dare importanza differente alle pagine in base all'utente che sta facendo la ricerca.

Si categorizzano le pagine per argomenti e poi si calcolano i gusti delle persone come frazioni di argomento.

Il risultato è utilizzato come pesi iniziali del PageRank.

Personalised PageRank

Qual è il GRANDE problema di questo approccio?

Si dovrebbe calcolare il PageRank per tutto il web una volta per ogni utente del web!

E inoltre, cosa accade se cambiassero gli interessi?

Si possono calcolare i PageRank di ogni argomento singolarmente.

I risultati per un singolo utente possono essere creati come combinazione lineare dei risultati per argomento.

Personalised PageRank

Modificare il motore di ricerca **facendo scegliere manualmente il numero del topic all'utente**. Utilizzare lo scoring del Personalised PageRank in base al topic scelto.

Caricate i topic dal json prodotto prima.

Vi basta passare un dizionario di valori da assegnare ad ogni dominio come argomento `personalization` del pagerank.

Personalised PageRank

file: 18_search_engine_pers_pagerank.py

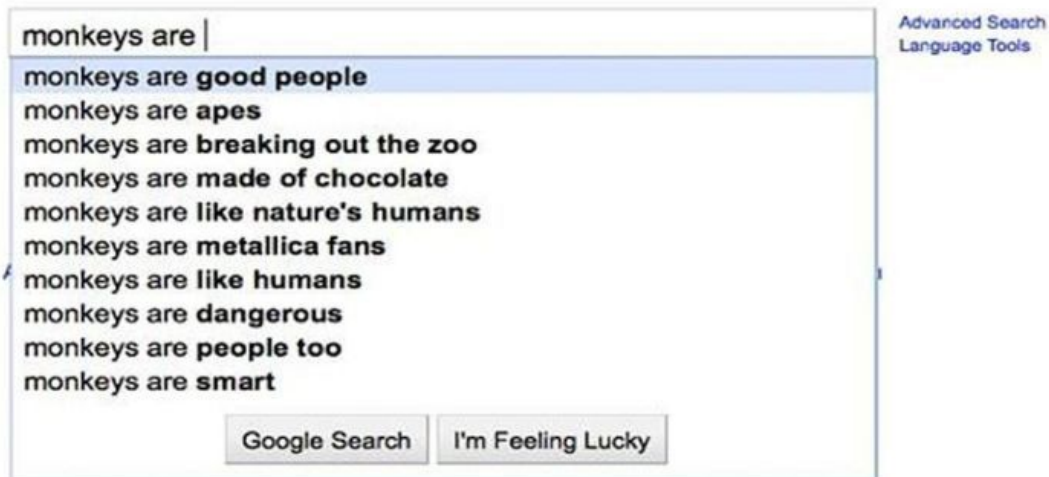
Interessi degli utenti

Sarà possibile costruire la “distribuzione degli interessi di un utente” andando a prendere tutte le sue ricerche o altri dati ad esso pertinenti.

Andando ad affinare questa distribuzione nel tempo, sarà possibile utilizzarla per calcolare il Personalised PageRank in maniera più precisa.

Autocompletamento

La funzione di **autocompletamento** utilizza le precedenti query di un utente per fornirgli dei suggerimenti riguardanti le nuove query in corso di digitazione.

A screenshot of a Google search interface. The search bar contains the text "monkeys are |". Below the search bar, a dropdown menu displays a list of autocomplete suggestions: "monkeys are good people", "monkeys are apes", "monkeys are breaking out the zoo", "monkeys are made of chocolate", "monkeys are like nature's humans", "monkeys are metallica fans", "monkeys are like humans", "monkeys are dangerous", "monkeys are people too", and "monkeys are smart". The first suggestion, "monkeys are good people", is highlighted with a blue background. To the right of the search bar, there are links for "Advanced Search" and "Language Tools". At the bottom of the search bar, there are two buttons: "Google Search" and "I'm Feeling Lucky".

monkeys are |

monkeys are **good people**

monkeys are **apes**

monkeys are **breaking out the zoo**

monkeys are **made of chocolate**

monkeys are **like nature's humans**

monkeys are **metallica fans**

monkeys are **like humans**

monkeys are **dangerous**

monkeys are **people too**

monkeys are **smart**

Advanced Search
Language Tools

Google Search I'm Feeling Lucky

Autocompletamento

Implementare una funzione di autocompletamento:
salvare ogni query fatta e, se si scrive ">" prima della query, invece dei risultati mostrare le dieci query più frequenti che iniziano con la query scritta

(salvare solamente le phrase queries già corrette da spelling, e suggerire dopo aver corretto lo spelling)

Autocompletamento

file: 19_search_engine_suggestions.py