Design and use of linguistic tools II. Building a thesaurus

Pablo Gamallo

CiTIUS
Universidade de Santiago de Compostela

Master EMLex

Table of Contents

1. Introduction
2. Distributional Meaning
3. Thesaurus
4. Python Functions

Table of Contents

1 Introduction

2 Distributional Meaning

3 Thesaurus

4 Python Functions

Objectives

- Introduction to distributional semantics.
- Develop and use python scripts.
- Build a probabilistic thesaurus from corpora.

Lexical Semantic Models

- Denotational meaning
- Structural meaning
- Universal meaning
- Relational meaning
- Distributional meaning

Denotational meaning

Lexical meaning

Two types of meaning (according to "The Foundations of Frege's Logic" [Tichý 1988]):

Denotation Relation between a word and its meaning out of context. Two types of denotation:

- **Extension** : Set of entities or individuals associated to a word by denotation. For instance, the set of all dogs is the *extension* of the word dog.
- **Intension** : Set of properties and features shared by an extension or entity set. .

Reference Relation between a word inserted in a discourse and the particular entity the word points out: meaning contextualized and grounded by the discourse:

Example: the reference of dog selects for a specific dog in ("My dog is Trosky")

Lexical meaning and Structuralism

Structural semantics and componential analysis

- **Lexical field**: Set of lexemes sharing some small units of content, but which are opposed by minimal differences or semes (Eugenio Coseriu).
- **Seme**: the smallest unit of meaning used as a distinctive feature.
- **Sememe**: Set of semes constituting the meaning of a lexeme (Bernard Pottier).

Componential analysis: example

Table: Lexical field organized by the seme “to sit on”

	chair	**armchair**	**stool**	**couch**	**pouf**
back	+	+	-	+	-
elevated	+	+	+	+	-
one person	+	+	+	-	+
to sit on	+	+	+	+	+
solid material	+	+	+	+	-
with arms	-	+	-	+	-

Universal Semantics

Universal semantics and semantic primitives

- **Semantic primitives**: They are semantic concepts that are innately understood, but cannot be expressed in simpler terms. They represent words or phrases that are learned through practice, but cannot be defined concretely
- For example, although the meaning of ”touching” is readily understood, a dictionary might define ”touch” as *to make contact* and ”contact” as *touching*.
- **Natural semantic metalanguage:** Any English word can be described (defined) with a text using a primitive lexicon of about 60 words (primitive concepts) in the English natural semantic metalanguage.
- For example: “lie” is defined as *what a person does when he says something not true because he wants someone to think it true*

(Wierzbicka A. Semantics: Primes and Universals. 1996)

Relational model of lexical meaning

Relational meaning of words

- The lexical meaning of a word is the set of lexical relations (synonymy, hypernymy, hyponymy, meronymy...) it holds with other words.

Relational model of meaning

WordNet

- Developed by George Miller and his team at Princeton University, as the implementation of a mental model of the lexicon (ideas based on psycholinguistics).
- Organized around the notion of a *synset*: a set of synonyms in a language that represent a single concept (or word sense)
- Semantic relations between concepts / syntsets

Distributionalism

Distributional meaning

- In 1957, the corpus linguist John R. Firth lays the foundations for the modern distributional theory with the following idea: *"You shall know a word by the company it keeps"* (Firth, 1957).
- Corpora and statistical methods to analyze the word behavior in contexts (e.g. concordances, association measures, etc) are parts and parcels of the lexicographer's toolbox (Lenci 2008).
- Nowdays, This is the most popular semantic model in Natural Language Processing: a lexical unit is defined as a **vector of contexts**.

Distributional meaning: example

	$< N\_subj\_run >$	$< N\_subj\_eat >$	$< red\_mod\_N >$
car	3	0	7
horse	10	15	0
moto	6	0	5
cat	5	21	0

Table: Toy vectors for “car”, “horse”, “motorbike”, and “cat”

Polysemy: meaning - senses

- The **meaning** of a polysemous word consists of a set of related **senses**.
- For instance, “Portugal” is a polysemous word whose meaning consists of, at least, three senses:

  Place: “I’m in Portugal”
  People: “Portugal is losing purchasing power”
  People+Place: “I like Portugal”

Table of Contents

1 Introduction

2 Distributional Meaning

3 Thesaurus

4 Python Functions

Distributional meaning

The distributional hypothesis (Zellig Harris and J.R. Firth)

The meaning of a word is the **set of contexts** in which it occurs in texts.

Words with similar contexts have similar meanings.

How do we grasp word meaning?

I found a cute, hairy wampimuk sleeping behind the tree

(Example by McDonald & Ramscar 2001)

Distributional hypothesis

Lenci (2008)

Weak assumption

A **quantitative method** for lexical similarity and semantic analysis.

Strong assumptions

A **cognitive/logical hypothesis** about semantic representations:

- Mental objects linked to *intensions* of logical expressions (Katrin Erk, 2013).
- Ideal distributions linked to *extensions* of logical expressions (Copestake and Herbelot, 2012).

Vector space model

A co-occurrence matrix collected from large text corpora, with distributional vectors representing words as rows, and contextual elements of some kind as columns/dimensions.

	$< N\_subj\_run >$	$< N\_subj\_eat >$	$< red\_mod\_N >$
car	3	0	10
horse	7	15	0
moto	6	0	7
cat	12	9	0

Table: Toy vectors for “car”, “horse”, “motorbike”, and “cat”

Dimensionality reduction

Corpus-based matrices have many dimensions and are sparse. How to fix it?

- Dense matrices:
  - Singular Value Decomposition: Latent Semantics Analysis (Landauer, 1998)
  - Neural-based learning: Word Embeddings, word2vec (Mikolov, 2013)

But contexts in dense matrices **are not transparent**. How to solve it?

- Explicit matrices filtering out non-relevant contexts.

Gamallo Pablo and Stefan Bordag (2011) "Is Singular Value Decomposition Useful for Word Similarity Extraction?" *Language Resources and Evaluation*, 45(2).

Gamallo, Pablo (2016) "Comparing explicit and predictive distributional semantic models endowed with syntactic contexts", *Language Resources and Evaluation*.

Dimensionality reduction

Corpus-based matrices have many dimensions and are sparse. How to fix it?

- Dense matrices:
  - Singular Value Decomposition: Latent Semantics Analysis (Landauer, 1998)
  - Neural-based learning: Word Embeddings, word2vec (Mikolov, 2013)

But contexts in dense matrices **are not transparent**. How to solve it?

- Explicit matrices filtering out non-relevant contexts.

Gamallo Pablo and Stefan Bordag (2011) "Is Singular Value Decomposition Useful for Word Similarity Extraction?" *Language Resources and Evaluation*, 45(2).

Gamallo, Pablo (2016) "Comparing explicit and predictive distributional semantic models endowed with syntactic contexts", *Language Resources and Evaluation*.

Dimensionality reduction

Corpus-based matrices have many dimensions and are sparse. How to fix it?

- Dense matrices:
  - Singular Value Decomposition: Latent Semantics Analysis (Landauer, 1998)
  - Neural-based learning: Word Embeddings, word2vec (Mikolov, 2013)

But contexts in dense matrices **are not transparent**. How to solve it?

- Explicit matrices filtering out non-relevant contexts.

Gamallo Pablo and Stefan Bordag (2011) "Is Singular Value Decomposition Useful for Word Similarity Extraction?" *Language Resources and Evaluation*, 45(2).

Gamallo, Pablo (2016) "Comparing explicit and predictive distributional semantic models endowed with syntactic contexts", *Language Resources and Evaluation*.

Dimensionality reduction

Corpus-based matrices have many dimensions and are sparse. How to fix it?

- Dense matrices:
  - Singular Value Decomposition: Latent Semantics Analysis (Landauer, 1998)
  - Neural-based learning: Word Embeddings, word2vec (Mikolov, 2013)

But contexts in dense matrices **are not transparent**. How to solve it?

- Explicit matrices filtering out non-relevant contexts.

Gamallo Pablo and Stefan Bordag (2011) “Is Singular Value Decomposition Useful for Word Similarity Extraction?” *Language Resources and Evaluation*, 45(2).

Gamallo, Pablo (2016) “Comparing explicit and predictive distributional semantic models endowed with syntactic contexts”, *Language Resources and Evaluation*.

Dimensionality reduction

Corpus-based matrices have many dimensions and are sparse. How to fix it?

- Dense matrices:
  - Singular Value Decomposition: Latent Semantics Analysis (Landauer, 1998)
  - Neural-based learning: Word Embeddings, word2vec (Mikolov, 2013)

But contexts in dense matrices **are not transparent**. How to solve it?

- Explicit matrices filtering out non-relevant contexts.

Gamallo Pablo and Stefan Bordag (2011) “Is Singular Value Decomposition Useful for Word Similarity Extraction?” *Language Resources and Evaluation*, 45(2).

Gamallo, Pablo (2016) “Comparing explicit and predictive distributional semantic models endowed with syntactic contexts”, *Language Resources and Evaluation*.

New trends

Contextualized Word Embeddings and Compositional Distributional Semantics

Contextualized Word Embeddings

- Generated with Transformers and Masked Language Models: BERT, ELMo, OpenAI GPT-2, ULMFit, ...
- Deep Learning libraries: PyTorch (FaceBook), TensorFlow (Google), ...

Compositional Distributional Semantics

Gamallo, Pablo, Susana Sotelo, José Ramom Pichel, Mikel Artetxe (2019). "Contextualized Translations of Phrasal Verbs with Distributional Compositional Semantics and Monolingual Corpora", *Computational Linguistics*

Gamallo, Pablo (2019). "A dependency-based approach to word contextualization using compositional distributional semantics", *Journal of Language Modelling*, 7(1), pp. 53-92.

Applications

Word Similarity / Association

- Thesaurus construction from monolingual corpus.
- Lexical inference and analogies:
  wood $\leftrightarrow$ *carpenter* / *stone* $\leftrightarrow$ *X* X = mason
- Bilingual dictionaries from comporable corpora.

  ...
- (Contextualized) word embeddings are being used in almost all NLP applications.
- Historical linguistics

Applications

Word Similarity / Association

- Thesaurus construction from monolingual corpus.
- Lexical inference and analogies:
  wood $\leftrightarrow$ *carpenter* / *stone* $\leftrightarrow$ *X* **X = mason**
- Bilingual dictionaries from comporable corpora.

  ...
- (Contextualized) word embeddings are being used in almost all NLP applications.
- Historical linguistics

Table of Contents

1 Introduction

2 Distributional Meaning

3 Thesaurus

4 Python Functions

Thesaurus and Ontologies

Thesaurus

Terms (and their senses) organized by means of semantic relations (main relation: synonymy)

Ontologies

Concepts organized by conceptual relations (main relation: hyperonymy)

Thesaurus and Relations

Synonymy bank $\rightarrow$ depository_financial_institution

Hyperonymy {bank,...} $\rightarrow$ {institution,...}

Co-hyponymy {bank,...} $\rightarrow$ {foundation,...}

Meronymy {bank,...} $\rightarrow$ {banker,...}

Automatic generation of thesaurus

- **Distributional Hypothesis**: words sharing similar contexts are semantically related
- **Types of contexts**:
  - co-occurrences within a window of words (n-grams and bag-of-words)
  - co-occurrences in lexico-syntactic contexts
- **Output**: Words semantically related to other words by unknown relations and similarity weights (probabilistic lexicon)

Steps to generate a thesaurus

- **Distributional matrix**: Collect a co-occurrence matrix from a corpus with distributional vectors representing words as rows, and contextual words as columns/dimensions
- **Transform the matrix**: Re-weighting raw frequencies by computing the degree of relevance of each context given a word (PMI, Loglikehood,...)
- **Similarity**: Compute the similarity score between words based on their contexts (Cosine, Dice, Jaccard,...)
- **Ranking**: For each word, identify and select its most similar words (ranking by similarity)

Non-zero word-context matrix from corpus

Corpus: *Some mammals eat meat and plants.*

- Tokenization:

  Some mammals eat meat and plants .

- Generate n-grams (trigrams):

some	mammals	eat
mammals	eat	meat
eat	meat	and
meat	and	plants
and	plants	.

Non-zero word-context matrix from corpus

- trigrams without stopwords:

STOP	mammals	eat
mammals	eat	meat
eat	meat	STOP
meat	STOP	plants
STOP	plants	STOP

- “word context frequency” triples:

mammals	eat	1
eat	mammals	1
mammals	meat	1
meat	mammals	1
eat	meat	1
meat	eat	1
meat	plants	1
plants	meat	1

Transforming the matrix by context weighting

- The less frequent the context word is, the higher the weight given to the word-context co-occurrence count should be. For instance, co-occurrence with frequent context word *time* is less informative than co-occurrence with rarer *motorbike*.
- Point-wise Mutual Information (PMI) widely used and pretty robust.

Transforming the matrix by context weighting

- Point-wise Mutual Information (PMI):

$$PMI(meat, eat) = log \frac{prob(meat, eat)}{prob(meat)prob(eat)}$$

mammals	eat	0.69
eat	mammals	0.69
mammals	meat	0.28
meat	mammals	0.28
eat	meat	0.28
meat	eat	0.28
meat	plants	0.98
plants	meat	0.98

Transforming the matrix by context weighting: matrix reduction

The size of a matrix can be reduced by selecting the most relevant contexts of each word:

- Rank contexts by relevance (PMI values)
- Select the N most relevant (N=1).

mammals	eat	0.69
eat	mammals	0.69
meat	plants	0.98
plants	meat	0.98

Word similarity

- Two words are similar if they share many (relevant) contexts
- Different similarity measures can be used: Cosine, Jaccard, Dice, etc.

Word similarity

- Dice coefficient:

$$dice(eat, meat) = \frac{2 * \sum_i min(pmi(eat, cntx_i), pmi(meat, cntx_i))}{PMI(eat) + PMI(meat)}$$

Word similarity: a toy corpus

```
 Pedro read books and Maria read books too,
Pedro read novels and Maria read novels and
books, Pedro and Maria read many things, but
Pedro loves Maria, Maria loves books, in fact
Maria loves many things.
 Maria is eating an apple and Pedro is eating
an apple too, Pedro is eating eggs now, Pedro
and Maria are eating many things, Maria is
eating eggs, Maria and Pedro loves eggs a lot.
```

Word similarity

- Dice similarity between some word pairs of the toy corpus:

books	novels	0.547388
apple	eggs	0.279988
Pedro	Maria	0.095035
things	books	0.438387
books	apple	0.027512
Maria	novels	0.000000

Ranking by similarity

- Dice similarity between novels and their N most similar words:

novels	books	0.547388
novels	things	0.372190
novels	apple	0.254601
novels	reads	0.202930
novels	eating	0.189784
novels	eggs	0.072307
novels	Maria	0.000000

Table of Contents

1 Introduction

2 Distributional Meaning

3 Thesaurus

4 Python Functions

Functions of the tokenizer_spaces.py

- **strip()** returns a copy of the string in which all chars defined as argument have been stripped from the beginning and the end of the string (default whitespace characters). In the following example, all 0 are removed from the input:

```
str = "0000000this is string example....wow!!!0000000";
print str.strip( '0' )
```

- **re.sub(pattern, repl, string)** replaces all occurrences of the pattern in string with repl, substituting all occurrences. In the following example, all "-" symbols of the phone number are replaced by whitespaces:

```
phone = "2004-959-559
num = re.sub(r"-", " ", phone)
print num
```

Running tokenizer_spaces.py

```
cat corpus.txt |./tokenizer_spaces.py
```

Functions of tokenizer.py (one token per line)

- **split()** returns a list of all the words in the string (splits on all whitespace if left unspecified). The following example returns the list ['this', 'is', 'Bob'].

```
string = "this is Bob";
print string.split()
```

- **for iterating_var in sequence:** iterates over the items of any sequence, such as a list.

```
str = "this is Bob";
str = str.split()
for token in str:
    print token
```

Functions of ngrams.py

- **Building bigrams:** given a string of words, all possible bigrams are generated from the string as follows:

```
str = "this is Bob";
output = []
for i in range(len(str)-2+1):
    output.append(str[i:i+2])
print output
```

- **Defining a function of n-grams with def:** given a string of words (str) and a number of grams (n), a generic function (ngrams) is defined as follows:

```
def ngrams(str, n):
    str = str.split()
    output = []
    for i in range(len(str)-n+1):
        output.append(str[i:i+n])
    return output
```

Functions of ngrams.py

- **printing bigrams returned by the function *ngrams*:** given two specific arguments, namely the string “this is Bob” and number “2”, the list of bigrams returned by the function is: [[’this’, ’is’], [’is’, ’Bob’]], which is stored in variable *result*. Then the tokens of each bigram are grouped in variable *juntar*, which is finally printed. This is done as follows:

```
result = ngrams("this is Bob",2)
for ngram in result:
    juntar = ""

    for token in ngram:
         juntar += token + " "
    print juntar
```

Running ngrams.py (trigrams)

```
cat corpus.txt |./tokenizer_spaces.py |./ngrams.py 3
```

Functions of stopwords.py

- **building a list from a file:** given a file containing a word per line, we stored all words in a list as follows:

```
file = open(file_name, 'rU')
for token in file:
    token = token.strip()
    stop.append(token)
```

- **checking if an element is in a list:** given word *w1* and the list of words *stop*, if that word is in the list, then we print "YES":

```
if w1 in stop:
   print "YES"
```

Running stopwords.py

```
cat corpus.txt |./tokenizer_spaces.py
               |./ngrams.py 3
               |./stopwords.py resources/stopwords-en.txt
```

Functions of trigrams2matrix.py

- **building a dictionary of words with their frequency:** given a list of words, each word and its frequency in the list is stored in a dictionary (dico), which can be used to print the frequency of each word. This is an example that returns the frequency of "apple" (=2)

```
from collections import defaultdict

words = "apple banana apple strawberry banana lemon"

dico = defaultdict(int)
for word in words.split():
    dico[word] += 1
if dico["apple"]:
    print dico["apple"]
```

- **printing elements of a dictionary:** given dictionary *dico*, their attributes and values are printed as follow:

```
for w in dico:
    print '%s\t%.3f' % (w, dico[w])
```

Running trigrams2matrix.py

```
cat corpus.txt |./tokenizer_spaces.py
               |./ngrams.py 3
               |./stopwords.py resources/stopwords-en.txt
               |./trigrams2matrix.py
```

Functions of filterning_words_cntx.py

- **sorting a dictionary by keys:**

```
mydict = {'carl':40,
          'alan':2,
          'bob':1,
          'danny':3}

for key in sorted(mydict):
    print "%s: %s" % (key, mydict[key])
```

- **sorting a dictionary by values and reverse order:**

```
mydict = {'carl':40,
          'alan':2,
          'bob':1,
          'danny':3}

for key in sorted(mydict, key=mydict.get, reverse=True):
    print "%s: %s" % (key, mydict[key])
```

Functions of filterning_words_cntx.py

- **building a dictionary containing another dictionary:** given a list of triples (name, property, value), we can create a dictionary of names whose values is another dictionary of properties having specific values for the pair name-property. This is an example that returns the value assigned to “Bob” for the property “age” (=60)

```
from collections import defaultdict

data = "Bob_age_60 Mary_age_80 Bob_weight_80 Mary_weight_70"

dico = defaultdict(dict)
for triples in data.split():
   (name, prop, value) = triples.split('_')
   dico[name][prop] = value
if dico["Bob"]["age"]:
   print dico["name"]["age"]
```

Functions of filterning_words_cntx.py

- **sorting a dictionary of dictionaries by values and reverse order:**

```
from collections import defaultdict

data = "Bob_age_60 Mary_age_80 Bob_weight_80 Mary_weight_70"

dico = defaultdict(dict)
for triples in data.split():
    (name, prop, value) = triples.split('_')
    dico[name][prop] = float(value)

for name, properties in sorted(dico.items() ):
  for prop in sorted(properties,key=properties.get,reverse=True):
      print '%s\t%s\t%.d' % (word, prop, dico[word][prop])
```

Running filtering_words_cntx.py

```
cat corpus.txt |./tokenizer_spaces.py
               |./ngrams.py 3
               |./stopwords.py resources/stopwords-en.txt
               |./trigrams2matrix.py
               | ./filtering_words_cntx.py 1 3 > matrix.txt
```

Algorithm of dice.py

```
Read the filtered matrix and build two dictionaries:
   WordContext[word][context] = weight
   Word[word] = all weights

Read each word pair w1, w2:
   For each context c of w1 in WordContext dictionary:
       If c is also shared by w2:
           Select the minimum weight between
           WordContext[w1][c] and WordContext[w2][c] and
           add that minimum value to variable Common.
   Compute the dice similarity for w1 and w2 as follows:
       Common divided by Word[w1] and Word[w2]
   Print the dice value for w1 and w2
```

Running dice.py and other scripts

```
##select all pairs with word "novels":
cat matrix.txt |./pairs_selection.py novels > pairs_novels.txt

##compute dice similarity:
cat matrix.txt |./dice.py pairs_novels.txt |./ranking_simil.py 5

##look up contexts shared by "novels" and "books":
cat matrix.txt |./compara.py  books  novels
```