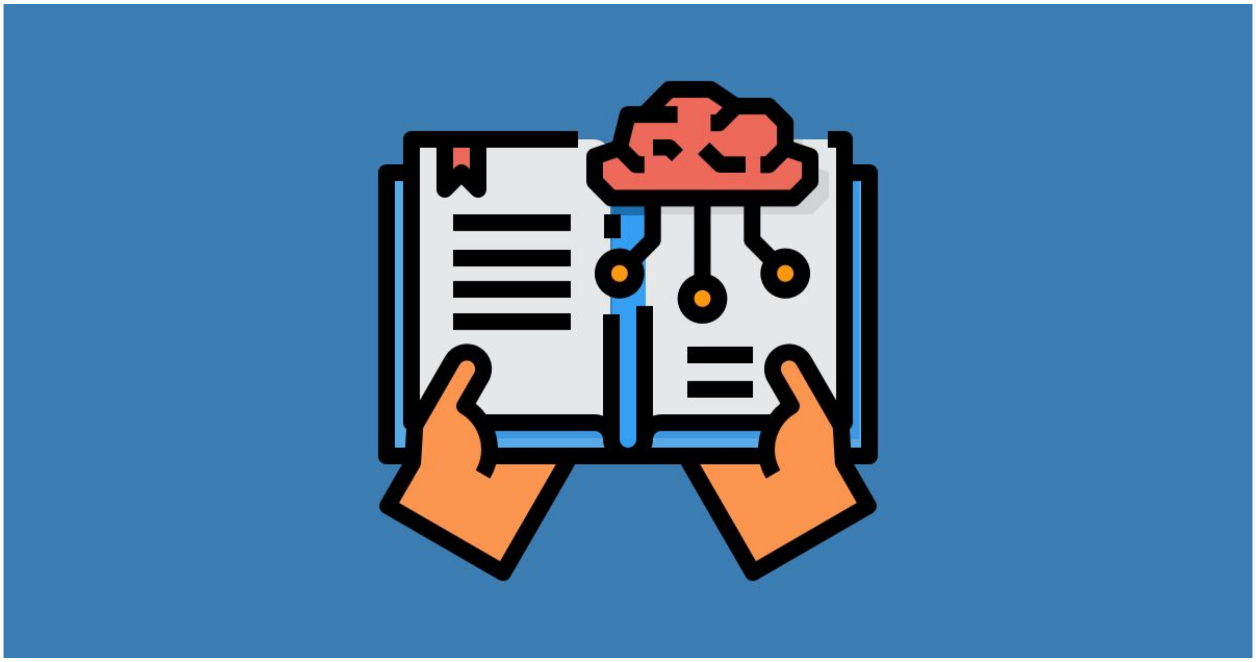

Natural Language Processing Assignment - Word Sense Disambiguation (WSD)

Delhi Technological University - Dr. Seba Susan

Anish Sachdeva (DTU/2K16/MC/13)

[Interactive Project on Jupyter Notebook 📔](#)

[Code on GitHub](#)

10st October 2020

Overview

Introduction

In this assignment we introduce 4 different Metrics through which we can solve the Word Sense Disambiguation (WSD) problem.

Naive Disambiguation (Method 1) [See Notebook]

Naive disambiguation as the name suggests is a very basic and simple method which simply assumes that the first sense offered by the Wordnet corpus is the correct sense. This is computationally very fast, but obviously not the most accurate Method.

Code [naive_method.py]

The following code takes in a word from the user and returns the naive disambiguation (first sense) of the word and displays definition as the output. We will also randomly select one word from each document and print the disambiguation of that.

```
# Method 1

# Implementing a Naive method that simply returns the first sense found in the wordnet synsets as the disambiguated
# sense

import pprint
import pickle
import random
import nltk
from nltk.corpus import wordnet
# nltk.download('wordnet')
```

```
def naive_disambiguation(word: str):

   synsets = wordnet.synsets(word)

   try:

       return synsets[0]

   except:

       return 'no sense found'




word = input('Enter word for disambiguation:\t')

sense = naive_disambiguation(word)

if isinstance(sense, str):

   print('No sense found')

else:

   print('Definition:', sense.definition())

   print('Examples:')

   pprint.pprint(sense.examples())

   print('\n\n')



# We now load in the keywords we extracted from resume that have been
divided into 6 documents and we will randomly

# disambiguate one keyword from each document
```

```python
print('\n\nWe now disambiguate a few keywords from the resume')

documents = pickle.load(open('../assets/documents.p', 'rb'))

for document in documents:

   document = list(document)

   word = document[random.randint(0, len(document) - 1)]

   sense = naive_disambiguation(word)

   if isinstance(sense, str):

       print('No sense found')

   else:

       print(word.capitalize() + ':', sense.definition().capitalize())
```

Output

Enter word for disambiguation: car

Definition: a motor vehicle with four wheels; usually propelled by an internal combustion engine

Examples:

['he needs a car to get to work']

We now disambiguate a few keywords from the resume

Delhi: A city in north central india

Auckland: The largest city and principal port of new zealand

Group: Any number of entities (members) considered as a unit

Requests: A formal message requesting something that is submitted to an authority

Structures: A thing constructed; a complex entity constructed of many parts

London: The capital and largest city of england; located on the thames in southeastern england; financial and industrial and cultural center

Simple LESK Algorithm Disambiguation (Method 2) [See Notebook]

In the Simple LESK Algorithm we use the words present in the gloss surrounding the main token to disambiguate it's meaning and we assign Inverse Document Frequency (IDF) values and assign weights to all possible senses of the given token.

Code [simple_lesk_algorithm.py]

```
# Method 2 (Simple LESK Algorithm)
# The simple LESK Algorithm Computes the disambiguation of a word from it's gloss by computing the count of the gloss
# in the examples and definition of each sense of the word and mainting a weight vector

# importing required packages
import pprint

import numpy as np
from nltk.corpus import wordnet
# nltk.download('wordnet')


from src import tokenize


def simple_lesk(gloss: str, word: str):
    """":returns the sense most suited to the given word as per the Simple LESK Algorithm"""

    # converting everything to lowercase
    gloss = gloss.lower()
    word = word.lower()

    # obtaining tokens from the gloss
```

```
gloss_tokens = tokenize(gloss, word)

# calculating the word sense disambiguation using simple LESK
synsets = wordnet.synsets(word)
weights = [0] * len(synsets)
N_t = len(synsets)
N_w = {}

# Creating the IDF Frequency column using Laplacian Scaling
for gloss_token in gloss_tokens:
    N_w[gloss_token] = 1

    for sense in synsets:
        if gloss_token in sense.definition():
            N_w[gloss_token] += N_t
            continue

        for example in sense.examples():
            if gloss_token in example:
                N_w[gloss_token] += N_t
                break

for index, sense in enumerate(synsets):
    # adding tokens from examples into the comparison set
    comparison = set()
    for example in sense.examples():
        for token in tokenize(example, word):
            comparison.add(token)

    # adding tokens from definition into the comparison set
    for token in tokenize(sense.definition(), word):
        comparison.add(token)
```

```
        # comparing the gloss tokens with comparison set
        for token in gloss_tokens:
            if token in comparison:
                weights[index] += np.log(N_w[token] / N_t)

    max_weight = max(weights)
    index = weights.index(max_weight)
    return synsets[index], weights



gloss = input('Enter the Gloss (document):\t')
word = input('Enter word for disambiguation:\t')
sense, weights = simple_lesk(gloss, word)
print('The disambiguated meaning is:', sense.definition())
print('The weight vector is:', weights)
```

Output

Enter the Gloss (document): i love me a hot cup of java in the morning

Enter word for disambiguation:	java

The disambiguated meaning is: a beverage consisting of an infusion of ground coffee beans

The weight vector is: [0, 0.28768207245178085, 0]

Path Length Similarity Disambiguation (Method 3) [See Notebook]

The Path Length Similarity computes the minimum hop path between any 2 words in the wordnet corpus using the Hypernym Paths available and then computes the similarity score as -log (pathlen(w1, w2)).

We now create a file that takes in 2 words from the user and computes the Path Length similarity metric between the 2 words.

Code [path_length_similarity.py]

```
# Method 3 (Path Length Similarity)

# The Path length similarity computes the similarity between 2 synsets based on the hop length between the nodes

# in the Hypernym Path in the wordnet corpus


# importing required packages

import numpy as np

from nltk.corpus import wordnet

# nltk.download('wordnet')



# define the path length similarity metric

def path_similarity(hypernym_path1: list, hypernym_path2: list) -> float:

    """":returns the shortest path similarity metric between 2 hypernym paths"""

    count = 0

    for index, synset in enumerate(hypernym_path1):

        if len(hypernym_path2) <= index or synset != hypernym_path2[index]:

            break

        count += 1

    return -np.log(len(hypernym_path1) + len(hypernym_path2) - 2 * count)
```

```
# Define a method return maximum path similarity score given 2 synsets in
wordnet
def max_similarity_path(synset_1, synset_2) -> float:
    """:returns the highest path similarity metric score between 2 synsets"""
    max_similarity = -float('inf')
    for hypernym_path_1 in synset_1.hypernym_paths():
        for hypernym_path_2 in synset_2.hypernym_paths():
            max_similarity = max(max_similarity,
path_similarity(hypernym_path_1, hypernym_path_2))
    return max_similarity


# Defining a method which returns the closest synsets given 2 string words,
the resulting synsets may be nouns,
# verbs etc.
def closest_synsets(word_1: str, word_2: str):
    """:returns the closest synsets for 2 given words based on path similarity
metric"""
    word_1 = wordnet.synsets(word_1.lower())
    word_2 = wordnet.synsets(word_2.lower())
    max_similarity = -float('inf')
    synset_1_optimal = word_1[0]
    synset_2_optimal = word_2[0]

    for synset_1 in word_1:
        for synset_2 in word_2:
            similarity = max_similarity_path(synset_1, synset_2)
            if max_similarity < similarity:
                max_similarity = similarity
                synset_1_optimal = synset_1
                synset_2_optimal = synset_2

    return synset_1_optimal, synset_2_optimal, max_similarity
```

```
word_1 = input('Enter first word:\t')
word_2 = input('Enter second word:\t')
word_1_synset, word_2_synset, similarity = closest_synsets(word_1, word_2)

print(word_1.capitalize() + ' Definition:', word_1_synset.definition())
print(word_2.capitalize() + ' Definition:', word_2_synset.definition())
print('similarity:', similarity)
```

Output

Enter first word: car

Enter second word: dog

Car Definition: a wheeled vehicle adapted to the rails of railroad

Dog Definition: metal supports for logs in a fireplace

similarity: -1.791759469228055

We now write a program that will take in the 6 documents that we created by dividing our resume and will compute the similarity between the 6th document and the other 5 documents.

Code [path_similarity_resume.py]

```
# We will compare the 6th document from our resume with the first 5 and see which document it matches most closely to
# in the resume

# importing required packages
import nltk
import pickle
import pprint
```

```python
from nltk.corpus import wordnet
# nltk.download('wordnet')
import pandas as pd
import numpy as np
from scipy import stats


infinity = float('inf')


# define the path length similarity metric
def path_similarity(hypernym_path1: list, hypernym_path2: list) -> float:
    """":returns the shortest path similarity metric between 2 hypernym
paths"""
    count = 0
    for index, synset in enumerate(hypernym_path1):
        if len(hypernym_path2) <= index or synset != hypernym_path2[index]:
            break
        count += 1
    return -np.log(len(hypernym_path1) + len(hypernym_path2) - 2 * count)


# Define a method return maximum path similarity score given 2 synsets in
wordnet
def max_similarity_path(synset_1, synset_2) -> float:
    """:returns the highest path similarity metric score between 2 synsets"""
    max_similarity = -float('inf')
    for hypernym_path_1 in synset_1.hypernym_paths():
        for hypernym_path_2 in synset_2.hypernym_paths():
            max_similarity = max(max_similarity,
path_similarity(hypernym_path_1, hypernym_path_2))
    return max_similarity
```

```
# Defining a method which returns the closest synsets given 2 string words,
the resulting synsets may be nouns,
# verbs etc.
def closest_synsets(word_1: str, word_2: str):
   """:returns the closest synsets for 2 given words based on path similarity
metric"""
   word_1 = wordnet.synsets(word_1.lower())
   word_2 = wordnet.synsets(word_2.lower())
   max_similarity = -float('inf')
   try:
       synset_1_optimal = word_1[0]
       synset_2_optimal = word_2[0]
   except:
       return None, None, -infinity

   for synset_1 in word_1:
       for synset_2 in word_2:
           similarity = max_similarity_path(synset_1, synset_2)
           if max_similarity < similarity:
               max_similarity = similarity
               synset_1_optimal = synset_1
               synset_2_optimal = synset_2

   return synset_1_optimal, synset_2_optimal, max_similarity


# loading in the 6 documents from the resume
documents = pickle.load(open('../assets/documents.p', 'rb'))
print('The documents are:')
pprint.pprint(documents)
```

```
# We will now find the similarity between the 6th document and every other
document
similarity_mat = np.zeros((len(documents) - 1, len(documents[0])))

for column, keyword in enumerate(documents[len(documents) - 1]):
   for row in range(len(documents) - 1):
       similarity_mat[row][column] = closest_synsets(keyword,
documents[row][column])[2]

print('\nThe similarity coefficients are:\n')
similarity = pd.DataFrame(similarity_mat, columns=documents[5])
print(similarity.to_string())

# saving the similarity coefficient matrix in text file
results = open('../assets/path_similarity_matrix.txt', 'w')
results.write(similarity.to_string())
results.close()

# We now select the highest and lowest similarity document for each word in
the 6th document
min = similarity_mat.argmin(axis=0)
max = similarity_mat.argmax(axis=0)

# document with least/maximum similarity
document_min_similarity = stats.mode(min).mode[0]
document_max_similarity = stats.mode(max).mode[0]

print('\nDocument with Minimum Similarity to 6th document:',
documents[document_min_similarity])
print('Document with Maximum Similarity to 6th document:',
documents[document_max_similarity])
```

Output [see path_similarity_matrix.txt]

The documents are:

```
[['python', 'data', 'structures', 'students', 'com', 'delhi'],
 ['java', 'auckland', 'geometry', 'mathematics', 'theory', 'batch'],
 ['cern', 'applications', 'worked', 'research', 'group', 'core'],
 ['worked', 'also', 'requests', 'participated', 'many', 'teaching'],
 ['structures', 'computer', 'algorithms', 'java', 'university', 'mathematics'],
 ['trinity', 'college', 'london', 'plectrum', 'guitar', 'grade']]
```

The similarity coefficients are:

	trinity	college	london	plectrum	guitar	grade
0	-1.609438	-1.609438	-2.079442	-2.197225	-inf	-2.564949
1	-2.302585	-2.302585	-2.772589	-2.708050	-2.708050	-1.098612
2	-inf	-2.079442	-2.079442	-2.397895	-2.397895	-1.609438
3	-1.945910	-1.945910	-2.302585	-2.197225	-2.397895	-1.791759
4	-1.609438	-1.945910	-2.639057	-2.079442	-2.079442	-2.302585

Document with Minimum Similarity to 6th document: ['java', 'auckland', 'geometry', 'mathematics', 'theory', 'batch']

Document with Maximum Similarity to 6th document: ['python', 'data', 'structures', 'students', 'com', 'delhi']

Resnik Similarity Disambiguation (Method 4) [see Notebook]

In the Resnik Similarity metric we compute the lowest Common subsumer **LCS** of the given words w1 and w2. We then compute the probability of the subsumer being given a corpus and we then compute the similarity score as −log LCS(w1,w2). weshow below how to compute the closest possible synsets for 2 given words using the Resnik Similarity and we also then use this metric on our resume to see which document matches most closely with the 6th document.

We introduce a program that takes in 2 words and returns resnik similarity metric score along with the closest synsets.

Code [see resnik_similarity.py]

```
# Resnik Similarity (Method 4)

# In the Resnik similarity method we compute the negative log of the
probability of the lowest common subsumer of the

#   2 given

# words. In this assignment we introduce Resnik and compute the closest
synsets of 2 given words.

# NOTE: We can only compute the Resnik similarity of 2 words when they have
the same Part of Speech (PoS) tag

# NOTE 2: We need a corpus to refer to the probabilities in computing the
Resnik Similarity and in this case we will be

#       using the Brown IC Corpus


# Importing packages

import nltk

from nltk.corpus import wordnet, wordnet_ic

# nltk.download('wordnet')

# nltk.download('wordnet_ic')

import numpy as np


# Defining Infinity

infinity = float('inf')
```

```python
# Importing the Brown Corpus
brown_ic = wordnet_ic.ic('ic-brown.dat')


# Defining the Closest Synsets Function Based on Resnik Similarity Score
def closest_synsets(word_1: str, word_2: str):
    word_1 = wordnet.synsets(word_1)
    word_2 = wordnet.synsets(word_2)
    max_similarity = -infinity
    try:
        synset_1_shortest = word_1[0]
        synset_2_shortest = word_2[0]
    except:
        return None, None, -infinity

    for synset_1 in word_1:
        for synset_2 in word_2:
            if synset_1.pos() != synset_2.pos():
                continue
            similarity = synset_1.res_similarity(synset_2, ic=brown_ic)
            if similarity > max_similarity:
                max_similarity = similarity
                synset_1_shortest = synset_1
                synset_2_shortest = synset_2

    return synset_1_shortest, synset_2_shortest, max_similarity


# Taking User Input
word_1 = input('Enter the first word:\t')
word_2 = input('Enter the second word:\t')
word_1_synset, word_2_synset, similarity = closest_synsets(word_1, word_2)
```

```
print(word_1.capitalize() + ' Definition:', word_1_synset.definition())
print(word_2.capitalize() + ' Definition:', word_2_synset.definition())
print('similarity:', similarity)
```

Output

Enter the first word: java

Enter the second word: language

Java Definition: a platform-independent object-oriented programming language

Language Definition: a systematic means of communicating by the use of sounds or conventional symbols

similarity: 5.792086967391197

We will now write a program that will take the 6 documents of our divided resume as the input and then output the closest and minimum similarity between the 6th and other documents.

Code [see resnik_similarity_resume.py]

```
# Resnik Similarity (Method 4)
# In the Resnik similarity method we compute the negative log of the probability of the lowest common subsumer of the
#   2 given
# words. In this assignment we introduce Resnik and compute the closest synsets of 2 given words.
# NOTE: We can only compute the Resnik similarity of 2 words when they have the same Part of Speech (PoS) tag
# NOTE 2: We need a corpus to refer to the probabilities in computing the Resnik Similarity and in this case we will be
#       using the Brown IC Corpus

# Importing packages
import nltk
```

```python
from nltk.corpus import wordnet, wordnet_ic
# nltk.download('wordnet')
# nltk.download('wordnet_ic')
from nltk.stem import WordNetLemmatizer
import numpy as np
import pickle
import pprint
import pandas as pd
from scipy import stats

# Defining Infinity
infinity = float('inf')

# Importing the Brown Corpus
brown_ic = wordnet_ic.ic('ic-brown.dat')


# Defining the Closest Synsets Function Based on Resnik Similarity Score
def closest_synsets(word_1: str, word_2: str):
    word_1 = wordnet.synsets(word_1)
    word_2 = wordnet.synsets(word_2)
    max_similarity = -infinity
    try:
        synset_1_shortest = word_1[0]
        synset_2_shortest = word_2[0]
    except:
        return None, None, -infinity

    for synset_1 in word_1:
        for synset_2 in word_2:
            if synset_1.pos() != synset_2.pos():
                continue
```

```
            similarity = synset_1.res_similarity(synset_2, ic=brown_ic)
            if similarity > max_similarity:
                max_similarity = similarity
                synset_1_shortest = synset_1
                synset_2_shortest = synset_2

    return synset_1_shortest, synset_2_shortest, max_similarity

# loading in the 6 documents from the resume
documents = pickle.load(open('../assets/documents.p', 'rb'))
print('The documents are:')
pprint.pprint(documents)

# Viewing the document as a Table
documents_table = pd.DataFrame(documents)
print('\nDocuments:')
print(documents_table)

# We will now find the similarity between the 6th document and every other
document
similarity_mat = np.zeros((len(documents) - 1, len(documents[0])))

for column, keyword in enumerate(documents[len(documents) - 1]):
    for row in range(len(documents) - 1):
        similarity_mat[row][column] = closest_synsets(keyword,
documents[row][column])[2]

print('\nThe similarity coefficients are:\n')
similarity = pd.DataFrame(similarity_mat, columns=documents[5])
print(similarity.to_string())

# saving the similarity coefficient matrix in text file
```

```
results = open('../assets/resnik_similarity_matrix.txt', 'w')
results.write(similarity.to_string())
results.close()

# We now select the highest and lowest similarity document for each word in the 6th document
max = [0, 0, 0, 0, 4, 1]
min = [3, 3, 2, 3, 4, 0]

# document with least/maximum similarity
document_min_similarity = stats.mode(min).mode[0]
document_max_similarity = stats.mode(max).mode[0]

print('\nDocument with Minimum Similarity to 6th document:', documents[document_min_similarity])
print('Document with Maximum Similarity to 6th document:', documents[document_max_similarity])
```

Output [see resnik_similarity_matrix.txt]

The documents are:

[['python', 'data', 'structures', 'students', 'com', 'delhi'],
 ['java', 'auckland', 'geometry', 'mathematics', 'theory', 'batch'],
 ['cern', 'applications', 'worked', 'research', 'group', 'core'],
 ['worked', 'also', 'requests', 'participated', 'many', 'teaching'],
 ['structures', 'computer', 'algorithms', 'java', 'university', 'mathematics'],
 ['trinity', 'college', 'london', 'plectrum', 'guitar', 'grade']]

Documents:

	0	1	2	3	4	5
0	python	data	structures	students	com	delhi
1	java	auckland	geometry	mathematics	theory	batch

2	cern	applications	worked	research	group	core
3	worked	also	requests	participated	many	teaching
4	structures	computer	algorithms	java	university	mathematics
5	trinity	college	london	plectrum	guitar	grade

The similarity coefficients are:

	trinity	college	london	plectrum	guitar	grade
0	5.738632	2.855294	1.531834	1.531834	-inf	1.290026
1	0.596229	1.290026	-0.000000	-0.000000	-0.000000	7.054047
2	-inf	0.801759	-inf	-0.000000	0.801759	3.335576
3	-inf	-inf	-0.000000	-inf	-inf	2.644521
4	2.855294	2.305849	-0.000000	1.290026	2.305849	2.644521

Document with Minimum Similarity to 6th document: ['worked', 'also', 'requests', 'participated', 'many', 'teaching']

Document with Maximum Similarity to 6th document: ['python', 'data', 'structures', 'students', 'com', 'delhi']

Bibliography

1. Speech & Language Processing ~Jurafsky
2. nltk
3. pickle
4. Porter Stemmer Algorithm
5. Porter Stemmer Implementation ~anishLearnsToCode
6. NLTK WordNetInterface
7. NLTK Stemming Submodule
8. pandas.DataFrames
9. Indexing and Slicing in Pandas DataFrame