Salmon Run

Swimming upstream on the technology tide, one technology at a time. A collection of articles, tips, and random musings on application development and system design.

Thursday, December 04, 2014

Computing Semantic Similarity for Short Sentences

A reader recently recommended a paper for me to read - Sentence Similarity Based on Semantic Nets and Corpus Statistics. I found the algorithm quite interesting and I ended up implementing it. While I was not able to replicate the results exactly, my results did agree with results you would intuitively expect. I describe the algorithm and my implementation in this post.

My implementation is built with Python and Natural Language Tool Kit (NLTK). The Semantic Net referred to in the paper is Wordnet and the Corpus Statistics are from the Brown Corpus, both of which are available using NLTK's corpus API. Here is the complete code, which I explain below.

```
 1  from __future__ import division
 2 import nltk
 3 from nltk.corpus import wordnet as wn
 4 from nltk.corpus import brown
 5 import math
 6 import numpy as np
 7 import sys
 8
 9 # Parameters to the algorithm. Currently set to values that was reported
10 # in the paper to produce "best" results.
11 ALPHA = 0.2
12 BETA = 0.45
13 ETA = 0.4
14 PHI = 0.2
15 DELTA = 0.85
16
17 brown_freqs = dict()
18 N = 0
19
20 ########################## word similarity ##########################
21
22 def get_best_synset_pair(word_1, word_2):
23     """
24     Choose the pair with highest path similarity among all pairs.
25     Mimics pattern-seeking behavior of humans.
26     """
27     max_sim = -1.0
28     synsets_1 = wn.synsets(word_1)
29     synsets_2 = wn.synsets(word_2)
30     if len(synsets_1) == 0 or len(synsets_2) == 0:
31         return None, None
32     else:
33         max_sim = -1.0
```

```
34          best_pair = None, None
35          for synset_1 in synsets_1:
36              for synset_2 in synsets_2:
37                 sim = wn.path_similarity(synset_1, synset_2)
38                 if sim > max_sim:
39                     max_sim = sim
40                     best_pair = synset_1, synset_2
41          return best_pair
42
43  def length_dist(synset_1, synset_2):
44      """
45      Return a measure of the length of the shortest path in the semantic
46      ontology (Wordnet in our case as well as the paper's) between two
47      synsets.
48      """
49      l_dist = sys.maxint
50      if synset_1 is None or synset_2 is None:
51          return 0.0
52      if synset_1 == synset_2:
53          # if synset_1 and synset_2 are the same synset return 0
54          l_dist = 0.0
55      else:
56          wset_1 = set([str(x.name()) for x in synset_1.lemmas()])
57          wset_2 = set([str(x.name()) for x in synset_2.lemmas()])
58          if len(wset_1.intersection(wset_2)) > 0:
59              # if synset_1 != synset_2 but there is word overlap, return 1.0
60              l_dist = 1.0
61          else:
62              # just compute the shortest path between the two
63              l_dist = synset_1.shortest_path_distance(synset_2)
64              if l_dist is None:
65                  l_dist = 0.0
66      # normalize path length to the range [0,1]
67      return math.exp(-ALPHA * l_dist)
68
69  def hierarchy_dist(synset_1, synset_2):
70      """
71      Return a measure of depth in the ontology to model the fact that
72      nodes closer to the root are broader and have less semantic similarity
73      than nodes further away from the root.
74      """
75      h_dist = sys.maxint
76      if synset_1 is None or synset_2 is None:
77          return h_dist
78      if synset_1 == synset_2:
79          # return the depth of one of synset_1 or synset_2
80          h_dist = max([x[1] for x in synset_1.hypernym_distances()])
81      else:
82          # find the max depth of least common subsumer
83          hypernyms_1 = {x[0]:x[1] for x in synset_1.hypernym_distances()}
84          hypernyms_2 = {x[0]:x[1] for x in synset_2.hypernym_distances()}
85          lcs_candidates = set(hypernyms_1.keys()).intersection(
86              set(hypernyms_2.keys()))
87          if len(lcs_candidates) > 0:
88              lcs_dists = []
89              for lcs_candidate in lcs_candidates:
90                  lcs_d1 = 0
91                  if hypernyms_1.has_key(lcs_candidate):
```

```
92                      lcs_d1 = hypernyms_1[lcs_candidate]
93                  lcs_d2 = 0
94                  if hypernyms_2.has_key(lcs_candidate):
95                      lcs_d2 = hypernyms_2[lcs_candidate]
96                  lcs_dists.append(max([lcs_d1, lcs_d2]))
97              h_dist = max(lcs_dists)
98          else:
99              h_dist = 0
100     return ((math.exp(BETA * h_dist) - math.exp(-BETA * h_dist)) /
101         (math.exp(BETA * h_dist) + math.exp(-BETA * h_dist)))
102
103 def word_similarity(word_1, word_2):
104     synset_pair = get_best_synset_pair(word_1, word_2)
105     return (length_dist(synset_pair[0], synset_pair[1]) *
106         hierarchy_dist(synset_pair[0], synset_pair[1]))
107
108 ######################### sentence similarity ##########################
109
110 def most_similar_word(word, word_set):
111     """
112     Find the word in the joint word set that is most similar to the word
113     passed in. We use the algorithm above to compute word similarity between
114     the word and each word in the joint word set, and return the most similar
115     word and the actual similarity value.
116     """
117     max_sim = -1.0
118     sim_word = ""
119     for ref_word in word_set:
120       sim = word_similarity(word, ref_word)
121       if sim > max_sim:
122           max_sim = sim
123           sim_word = ref_word
124     return sim_word, max_sim
125
126 def info_content(lookup_word):
127     """
128     Uses the Brown corpus available in NLTK to calculate a Laplace
129     smoothed frequency distribution of words, then uses this information
130     to compute the information content of the lookup_word.
131     """
132     global N
133     if N == 0:
134         # poor man's lazy evaluation
135         for sent in brown.sents():
136             for word in sent:
137                 word = word.lower()
138                 if not brown_freqs.has_key(word):
139                     brown_freqs[word] = 0
140                 brown_freqs[word] = brown_freqs[word] + 1
141                 N = N + 1
142     lookup_word = lookup_word.lower()
143     n = 0 if not brown_freqs.has_key(lookup_word) else brown_freqs[lookup_word]
144     return 1.0 - (math.log(n + 1) / math.log(N + 1))
145
146 def semantic_vector(words, joint_words, info_content_norm):
147     """
148     Computes the semantic vector of a sentence. The sentence is passed in as
149     a collection of words. The size of the semantic vector is the same as the
```

```
    size of the joint word set. The elements are 1 if a word in the sentence
    already exists in the joint word set, or the similarity of the word to the
    most similar word in the joint word set if it doesn't. Both values are
    further normalized by the word's (and similar word's) information content
    if info_content_norm is True.
    """
    sent_set = set(words)
    semvec = np.zeros(len(joint_words))
    i = 0
    for joint_word in joint_words:
        if joint_word in sent_set:
            # if word in union exists in the sentence, s(i) = 1 (unnormalized)
            semvec[i] = 1.0
            if info_content_norm:
                semvec[i] = semvec[i] * math.pow(info_content(joint_word), 2)
        else:
            # find the most similar word in the joint set and set the sim value
            sim_word, max_sim = most_similar_word(joint_word, sent_set)
            semvec[i] = PHI if max_sim > PHI else 0.0
            if info_content_norm:
                semvec[i] = semvec[i] * info_content(joint_word) * info_content(sim_word)
        i = i + 1
    return semvec

def semantic_similarity(sentence_1, sentence_2, info_content_norm):
    """
    Computes the semantic similarity between two sentences as the cosine
    similarity between the semantic vectors computed for each sentence.
    """
    words_1 = nltk.word_tokenize(sentence_1)
    words_2 = nltk.word_tokenize(sentence_2)
    joint_words = set(words_1).union(set(words_2))
    vec_1 = semantic_vector(words_1, joint_words, info_content_norm)
    vec_2 = semantic_vector(words_2, joint_words, info_content_norm)
    return np.dot(vec_1, vec_2.T) / (np.linalg.norm(vec_1) * np.linalg.norm(vec_2))

########################## word order similarity ##########################

def word_order_vector(words, joint_words, windex):
    """
    Computes the word order vector for a sentence. The sentence is passed
    in as a collection of words. The size of the word order vector is the
    same as the size of the joint word set. The elements of the word order
    vector are the position mapping (from the windex dictionary) of the
    word in the joint set if the word exists in the sentence. If the word
    does not exist in the sentence, then the value of the element is the
    position of the most similar word in the sentence as long as the similarity
    is above the threshold ETA.
    """
    wovec = np.zeros(len(joint_words))
    i = 0
    wordset = set(words)
    for joint_word in joint_words:
        if joint_word in wordset:
            # word in joint_words found in sentence, just populate the index
            wovec[i] = windex[joint_word]
        else:
            # word not in joint_words, find most similar word and populate
```

```
            # word_vector with the thresholded similarity
            sim_word, max_sim = most_similar_word(joint_word, wordset)
            if max_sim > ETA:
                wovec[i] = windex[sim_word]
            else:
                wovec[i] = 0
        i = i + 1
    return wovec

def word_order_similarity(sentence_1, sentence_2):
    """
    Computes the word-order similarity between two sentences as the normalized
    difference of word order between the two sentences.
    """
    words_1 = nltk.word_tokenize(sentence_1)
    words_2 = nltk.word_tokenize(sentence_2)
    joint_words = list(set(words_1).union(set(words_2)))
    windex = {x[1]: x[0] for x in enumerate(joint_words)}
    r1 = word_order_vector(words_1, joint_words, windex)
    r2 = word_order_vector(words_2, joint_words, windex)
    return 1.0 - (np.linalg.norm(r1 - r2) / np.linalg.norm(r1 + r2))

######################### overall similarity ##########################

def similarity(sentence_1, sentence_2, info_content_norm):
    """
    Calculate the semantic similarity between two sentences. The last
    parameter is True or False depending on whether information content
    normalization is desired or not.
    """
    return DELTA * semantic_similarity(sentence_1, sentence_2, info_content_norm) + \
        (1.0 - DELTA) * word_order_similarity(sentence_1, sentence_2)

######################### main / test ##########################

# the results of the algorithm are largely dependent on the results of
# the word similarities, so we should test this first...
word_pairs = [
  ["asylum", "fruit", 0.21],
  ["autograph", "shore", 0.29],
  ["autograph", "signature", 0.55],
  ["automobile", "car", 0.64],
  ["bird", "woodland", 0.33],
  ["boy", "rooster", 0.53],
  ["boy", "lad", 0.66],
  ["boy", "sage", 0.51],
  ["cemetery", "graveyard", 0.73],
  ["coast", "forest", 0.36],
  ["coast", "shore", 0.76],
  ["cock", "rooster", 1.00],
  ["cord", "smile", 0.33],
  ["cord", "string", 0.68],
  ["cushion", "pillow", 0.66],
  ["forest", "graveyard", 0.55],
  ["forest", "woodland", 0.70],
  ["furnace", "stove", 0.72],
  ["glass", "tumbler", 0.65],
  ["grin", "smile", 0.49],
```

```
  ["gem", "jewel", 0.83],
  ["hill", "woodland", 0.59],
  ["hill", "mound", 0.74],
  ["implement", "tool", 0.75],
  ["journey", "voyage", 0.52],
  ["magician", "oracle", 0.44],
  ["magician", "wizard", 0.65],
  ["midday", "noon", 1.0],
  ["oracle", "sage", 0.43],
  ["serf", "slave", 0.39]
]
for word_pair in word_pairs:
    print "%s\t%s\t%.2f\t%.2f" % (word_pair[0], word_pair[1], word_pair[2],
                                  word_similarity(word_pair[0], word_pair[1]))

sentence_pairs = [
    ["I like that bachelor.", "I like that unmarried man.", 0.561],
    ["John is very nice.", "Is John very nice?", 0.977],
    ["Red alcoholic drink.", "A bottle of wine.", 0.585],
    ["Red alcoholic drink.", "Fresh orange juice.", 0.611],
    ["Red alcoholic drink.", "An English dictionary.", 0.0],
    ["Red alcoholic drink.", "Fresh apple juice.", 0.420],
    ["A glass of cider.", "A full cup of apple juice.", 0.678],
    ["It is a dog.", "That must be your dog.", 0.739],
    ["It is a dog.", "It is a log.", 0.623],
    ["It is a dog.", "It is a pig.", 0.790],
    ["Dogs are animals.", "They are common pets.", 0.738],
    ["Canis familiaris are animals.", "Dogs are common pets.", 0.362],
    ["I have a pen.", "Where do you live?", 0.0],
    ["I have a pen.", "Where is ink?", 0.129],
    ["I have a hammer.", "Take some nails.", 0.508],
    ["I have a hammer.", "Take some apples.", 0.121]
]
for sent_pair in sentence_pairs:
    print "%s\t%s\t%.3f\t%.3f\t%.3f" % (sent_pair[0], sent_pair[1], sent_pair[2],
        similarity(sent_pair[0], sent_pair[1], False),
        similarity(sent_pair[0], sent_pair[1], True))
```

Proceeding from the top-down (wrt the code, or bottom-up wrt the algorithm), the lowest unit of the algorithm is the semantic similarity between a pair of words. The word similarity is a combination of two functions f(l) and f(h), where l is the shortest path between the two words in Wordnet (our Semantic Network) and h the height of their Lowest Common Subsumer (LCS) from the root of the Semantic Network. The intuition behind these is that l is a proxy for how similar the words are, and d is a proxy for the specificity of the LCS, ie, LCS nodes closer to the root indicate broader/more abstract concepts and less similarity. The functions f(l) and f(h) serve to normalize these values to the range [0,1]. In formulas, then:

$$sim(w_1, w_2) = f(l).f(h)$$

where:

$$f(l) = e^{-\alpha l}$$

$$f(h) = \frac{e^{\beta h} - e^{-\beta h}}{e^{\beta h} + e^{-\beta h}}$$

Word similarities between a set of word pairs were reported in the paper. As a test, I computed the similarities between the same word pairs with my code above. The similarity values reported in the paper are shown under Exp.Sim and the ones returned by my code are shown under Act.Sim. As you can see, they are close but not identical - however, note that the computed similarites seem to line up with intuition. For example, sim(autograph, signature) is higher than sim (autograph, shore), sim(magician, wizard) is higher than sim(magician, oracle), etc.

Word #1	Word #2	Exp. Sim	Act. Sim
asylum	fruit	0.21	0.30
autograph	shore	0.29	0.16
autograph	signature	0.55	0.82
automobile	car	0.64	1.00
bird	woodland	0.33	0.20
boy	rooster	0.53	0.11
boy	lad	0.66	0.82
boy	sage	0.51	0.37
cemetery	graveyard	0.73	1.00
coast	forest	0.36	0.36
coast	shore	0.76	0.80
cock	rooster	1.00	1.00
cord	smile	0.33	0.13
cord	string	0.68	0.82
cushion	pillow	0.66	0.82
forest	graveyard	0.55	0.20
forest	woodland	0.70	0.98
furnace	stove	0.72	0.17
glass	tumbler	0.65	0.82
grin	smile	0.49	0.99
gem	jewel	0.83	1.00
hill	woodland	0.59	0.36
hill	mound	0.74	0.99
implement	tool	0.75	0.82
journey	voyage	0.52	0.82
magician	oracle	0.44	0.30
magician	wizard	0.65	1.00
midday	noon	1.00	1.00
oracle	sage	0.43	0.37
serf	slave	0.39	0.55

One thing I did differently from the paper is to select the most similar pair of synsets instead of just picking the first noun synset for each word (see the function get_best_synset_pair). This is because a word can map to multiple synsets, and finding the most similar pair mimics the human tendency to maximize pattern seeking (ie see patterns where there are none).

Sentence similarity is computed as a linear combination of semantic similarity and word order similarity. Semantic Similarity is computed as the Cosine Similarity between the semantic vectors for the two sentences. To build the semantic vector, the union of words in the two sentences is treated as the vocabulary. If the word occurs in the sentence, its value for that position is 1. If it doesn't, the similarity for the word is computed against all the other words in the sentence. If it happens to be above a threshold φ, then the value of the element is φ, else it is 0. This value is further attenuated by the information content for the word as found in the Brown corpus. In equations:

$$S_s = \frac{s_1 \cdot s_2}{||s_1|| * ||s_2||}$$

where:

$$s_i = s * I(w_i) * I(w_j)$$

$$I(w) = 1 - \frac{\log(n + 1)}{\log(N + 1)}$$

where:
  n = number of times word w occurs in corpus
  N = number of words in the corpus

The word order similarity attempts to correct for the fact that sentences with the same words can have radically different meanings. This is done by computing the word order vector for each sentence and computing a normalized similarity measure between them. The word order vector, like the semantic vector is based on the joint word set. If the word occurs in the sentence, its position in the joint word set is recorded. If not, the similarity to the most similar word in the sentence is recorded if it crosses a threshold η else it is 0. In equations:

$$S_r = 1 - \frac{||r_1 - r_2||}{||r_1 + r_2||}$$

where:
  r_1 = word position vector for sentence 1
  r_2 = word position vector for sentence 2

The similarity between two sentences are modeled as a linear combination of their semantic similarity and word order similarity, ie:

$$S = \delta S_s + (1 - \delta) S_r$$

Similar to word similarities, the paper also lists some sentence similarities computed with their algorithm. I tried these same sentences through my code, and as expected, got slightly different results (since the sentence similarity is dependent on word similarities). Here they are. Exp.Sim are the values reported in the paper, Act.Sim (w/o IC) are computed similarities without Information Content normalization and Act.Sim (w/IC) are computed similarities with Information Content normalization. As you can see the Exp.Sim and Act.Sim (w/IC) values are quite consistent.

Sentence #1	Sentence #2	Exp.Sim	Act.Sim (w/o IC)	Act.Sim (w/IC)
I like that bachelor.	I like that unmarried man.	0.561	0.801	0.345
John is very nice.	Is John very nice?	0.977	0.592	0.881
Red alcoholic drink.	A bottle of wine.	0.585	0.477	0.307
Red alcoholic drink.	Fresh orange juice.	0.611	0.467	0.274
Red alcoholic drink.	An English dictionary.	0.000	0.237	0.028
Red alcoholic drink.	Fresh apple juice.	0.420	0.389	0.215
A glass of cider.	A full cup of apple juice.	0.678	0.659	0.347
It is a dog.	That must be your dog.	0.739	0.452	0.709
It is a dog.	It is a log.	0.623	0.858	0.497
It is a dog.	It is a pig.	0.790	0.863	0.500
Dogs are animals.	They are common pets.	0.738	0.550	0.377
Canis familiaris are animals.	Dogs are common pets.	0.362	0.458	0.151
I have a pen.	Where do you live?	0.000	0.134	0.158

I have a pen.	Where is ink?	0.129	0.112	0.077
I have a hammer.	Take some nails.	0.508	0.431	0.288
I have a hammer.	Take some apples.	0.121	0.344	0.147

Once more, while the numbers don't match exactly, the results seem intuitively correct. For example, "Red alcoholic drink" is more similar to "A bottle of wine" than "Fresh apple juice", which is more similar than "An English dictionary", etc.

Thats all I have for today. Hope you found this paper (and my implementation) interesting. The (latest) code for this post is available on GitHub here.

Sujit Pal at 11:39 PM

Share G+1 0

69 comments:

Anonymous 12/10/2014 12:47 PM

Interesting...I was trying to deal with it from the other end using text entailment. In all fairness was trying to compare 2 or more entire documents together to see if they are similar. Although entailment was promising at a sentence level, I felt it wasn't great for larger texts. I was also trying to view it from Natural Language Understanding perspective, but in vain. Kindly let me know if you find alternatives :-)

Thanks for the great write up

Ravi Kiran Bhaskar

Reply

Sujit Pal 12/10/2014 10:05 PM

Thanks Ravi, every time we speak I end up learning something new :-). I didn't know about text entailment; from the little I understand now (from this paper), it seems to me that entailment at a reasonably high probability may be a stronger guarantee than what this is computing.

One possibility to compute semantic similarity between documents could be something similar to what we do at work - we reduce both documents to a bag of concepts by annotating phrases in it to an ontology, then computing the similarity between their concept vectors. There is some upfront effort to create an ontology but once you have it, the process is reasonably accurate and has good performance.

Reply

Anonymous 12/12/2014 9:46 AM

Sujit,
I love your blog man, I can only dream to understand math as clearly as you do and apply them as algorithms :-)

I do understand and know that we can calculate document similarity based on bag of words/term vectors/features. However, I am still not convinced its a true representation of similarity. For example, consider the overly simplistic view of 2 docs

1. "Ravi went to USA. He loves NLP and is working in a company using the awesome technologies."

2. "Ravi is a probably a good boy. The jobless rate depicted in USA news is misleading, it depends from comapany to company and tech involved".

Both have the same set terms "Ravi", "USA", "Company", although the context/meaning of both these docs are totally different.

From the limited knowledge I have, BOW/Term Vectors can only decipher "relatedness" NOT "similarity" ...there seems to be a fine distinction which eludes most NLPers...that's the real head scratcher I am after :-)

Thanks,

Ravi Kiran Bhaskar

Reply

Anonymous 12/12/2014 1:46 PM

BTW if you want to look at Entailment look at European Union Funded Project http://hltfbk.github.io/Excitement-Open-Platform/

I felt this was better than the rest and others felt rudimentary

Ravi Kiran Bhaskar

Reply

Sujit Pal 12/13/2014 2:03 PM

Thanks Ravi, both for the kind words and the link. Actually I mean Bag of Concepts rather than Bag of Words, it gives you a little more in terms of synonymy and relationships. Perhaps if in the example, we recognized more entities such as NLP and jobless, we could not only use the matched terms/concepts but also the mismatched ones to form a more well rounded version of similarity that would be closer to what you are looking for?

Reply

Mahesh 6/29/2015 3:47 AM

Hi Sujit Pal,

Really, you did a great job. From last few days I am searching for sentence matching. The code which you have written is very very useful to me. Simply you did an awesome job.

But I have one problem, when I integrate your code and checking with 1000 sentences its getting too slower and taking at most 8 mins time.

Is there any way to reduce the processing time. Could you please help me to out of this issue.

Thanks in advance.

Reply

Sujit Pal 6/29/2015 7:21 AM

Thanks for the kind words Mahesh, glad you found it useful. Regarding your question about processing time, the algorithm is doing quite a lot of work so I am not surprised that you found it slow. I didn't investigate the slowness, but you may want to do some profiling to see which part of the algorithm is slow, then focus on that. So for example, Wordnet lookups can perhaps be cached so lookup of the same word across different sentences can be speeded up.

Reply

Mahesh 6/30/2015 4:56 AM

Thank you very much Sujit Pal. As you said I try with caching the lookups and the performance was improved and execution time reduced half of the time of previous.

Once again thank you very much Sujit.

Reply

Sujit Pal 6/30/2015 8:51 PM

Cool! Glad my suggestion helped.

Reply

Mahesh 7/02/2015 10:15 AM

I am very new to NLTK and a little bit of experience in python. The code which you have posted make me more confident on NLTK and python.

Really, I am so happy with your suggestions, especially with your code.

I need your valuable suggestions or code snippets for some critical scenario's.

Is it possible to add own synonyms to nltk - wordnet? If yes, please suggest me how?
Is it possible to use dictionary in nltk? If yes, please suggest me how?

Your suggestions are valuable and great guidance to me.

Thanks in advance.

Reply

Sujit Pal 7/02/2015 10:53 AM

Hi Mahesh, thanks for the kind words, happy to help. With regard to your question, no, it is not possible to add your own synonyms to Wordnet as far as I know. However, you can have a dictionary (Map in Java) for {word => synonym, synonym => word} lookups that you consult first and only go to Wordnet if lookup fails. Regarding dictionaries, I am guessing you mean a data structure that supports O(1) lookups by key, right? In that case, yes, and the data structure is called a dictionary in Python :-).

Reply

Mahesh 7/07/2015 11:31 PM

Thanks for giving your valuable time and suggestions.

I have two problems,
1.) How to handle digits (See Ex.1)
2.) Can we check the negative scenarios with this code. (See Ex.2)
Please check the expected and actual ratios

Ex1:-
Sent1 = "score 4/10"
Sent2 = "score 5/10"

Output put from console:
==========
Sent1 Sent2 Expected Act True Act False
----
score 4/10 score 5/10 0.650 0.425 0.164

Ex2:-
Sent1 = "Headache gradual in onset"
Sent2 = "Headache sudden onset"

Output put from console:
==========
Sent1 Sent2 Expected Act True Act False

Headache gradual in onset Headache sudden onset 0.335 0.577 0.692

Reply

Sujit Pal 7/09/2015 8:16 AM

Hi Mahesh, good call on the numbers. If the numbers are equal, then obviously they are similar, so you could have a check in both the word_similarity and hierarchical_similarity methods to check if the input words are numbers and return 1 if they are equal. Also you could modify the tokenization so it splits numbers on punctuation so 4/10 becomes ["4", "/", "10"]. For the "gradual" vs "sudden" case, they are both found in Wordnet and path_similarity will/may give some indication that they have opposite meanings (coverage is better for nouns than other parts of speech and these are adjectives). BTW I wasn't sure what the numbers are in your output - I am guessing that the "Expected" is a number that human judges have come up with, but not sure what "Act True" and "Act False" are.

Reply

Mahesh 7/17/2015 2:44 AM

Thank you for giving reply with max clarity and most patience.

Reply

Sujit Pal 7/17/2015 7:09 AM

You are welcome Mahesh, glad it helped.

Reply

Ahmed 8/19/2015 3:10 AM

Dear sir,
I want to implement Computing Semantic Similarity between two documents.Can you give me some brief detail on it.Any tutorial to help or any help will be appreciated.Thanks sir for such a good tutorial.

Reply

Sujit Pal 8/20/2015 7:19 AM

You're welcome, Ahmed. Regarding semantic similarity between two documents, this approach probably won't scale very well. One way to compute semantic similarity between two documents may be to use word2vec word vectors to produce document vectors by summing up the word vectors and comparing their similarity using standard measures like cosine similarity. This is on my to-do list, not sure how effective this would be though. Other approaches could use an ontology, and group related words into a coarser entity and use that for similarity calculations. With word2vec you could probably simulate this by clustering where you choose a number of clusters so there are approximately N (you control N) elements in each cluster, and then replace all the words in the cluster with the synthetic word represented by the cluster center, then use that for your similarity calculations. Anyway, the idea is to come up with some way to replace groups of words into a single representative word, then do similarity calculations against the representative words.

Reply

Ahmed 8/23/2015 2:23 AM

Dear sir,
I am using your code to get better understanding.but "wn.synsets(word_1)" is not working.Is there any change of API or what.I don't that .try to reply as soons as possible.

Reply

Sujit Pal 8/23/2015 9:14 AM

Hi Ahmed, not sure why its not working. For reference, my nltk version (using nltk.__version__) is 3.0.2 running against Python 2.7.10. It is possible it may have changed, although I doubt it if you are on a 3.0 version as well. in any case it returns the synsets associated with the word, so you might want to google for something equivalent. The output of wn.synsets("cat") on my machine looks like this:

```
>>> wn.synsets("cat")
[Synset('cat.n.01'), Synset('guy.n.01'), Synset('cat.n.03'), Synset('kat.n.01'), Synset('cat-o'-nine-tails.n.01'),
Synset('caterpillar.n.02'), Synset('big_cat.n.01'), Synset('computerized_tomography.n.01'), Synset('cat.v.01'),
Synset('vomit.v.01')]
```

The other option is that perhaps you have loaded wordnet? To verify check to see if some of the other wordnet commands work. If they don't then you should do nltk.download as explained here.

Reply

Ahmed 9/04/2015 1:58 PM

Dear Sir
Can you tell me for example Synset('kat.n.01').....what is n and 01 ?

Reply

Sujit Pal 9/04/2015 4:41 PM

Hi Ahmed, "n" is noun and 01 is a index into the sequence of noun synsets found for "kat" - run wn.synsets("kat", "n") or better wn.synsets("cat", "n") and you will see what I mean.

Reply

Anonymous 9/08/2015 9:35 PM

Hi, I have a question about negation. Consider: "This is good" and "This is not good".
Semantically, these two are opposite. Will the above algorithm handle such a pair? In general, what is the most effective way to handle negation? For example, "It is not the case that it is OK" vs. "It is acceptable and it is not quite unreasonable". In the first case, the whole clause is modified in meaning; in the second case, the negation is contained only in one of the clauses of the compound sentence. I have an impression that this issue is taken for granted, but I am looking for an effective way to address negation in deriving the meaning in a universally applicable way in all situations.

Reply

Sujit Pal 9/09/2015 8:38 AM

This algorithm will not handle this case, although (at least in theory, depending on Wordnet coverage) it will handle antonyms such as "gradual" vs "sudden" (from an example in an earlier comment). In the past I have handled negation using a rule based algorithm called negex (I have a Scala implementation here if you are interested). However, a nuance of negex is that it computes negation status of specific phrases in a sentence rather than the sentence itself, so you will need some way to find the phrases (maybe using an NER to find noun phrases and compute negation status for each). A simpler (but cruder) way is to only consider words in the sentence before a negator word (from a list) is seen. Recently, I also read (in the context of keyword based sentiment analysis) about changing the sign of the sentence vector for the sentence upon encountering negator words, so that could be another option.

Reply

Anonymous 9/09/2015 11:45 AM

Hi, Thank you very much for your quick and detailed reply. Also, I truly appreciate your sharing of the above code with excellent comments; it is indeed very nice of you. I am learning about Python as well as WordNet. One doubt: The cited paper seems to look for the distance from the root of the LCS, where as the code seems to calculate the distance between the LCS and the given synsets. I could be wrong, but could it be reason for the differences in the two sets of answers?

Reply

Sujit Pal 9/09/2015 6:30 PM

I believe so. Its been a while since I looked at the paper, but from what I remember I couldn't figure out a way to calculate the hierarchical distance the way it was computed in the paper so I decided to do it in an equivalent way.

Reply

Unknown 9/11/2015 9:36 AM

Thanks so much for this. Very useful!

I believe that the word_order_vector function should be as follows:

```
def word_order_vector(words, joint_words):
"""
Computes the word order vector for a sentence. The sentence is passed
in as a collection of words. The size of the word order vector is the
same as the size of the joint word set. The elements of the word order
vector are the position mapping (from the windex dictionary) of the
word in the joint set if the word exists in the sentence. If the word
does not exist in the sentence, then the value of the element is the
position of the most similar word in the sentence as long as the similarity
is above the threshold ETA.
"""
wovec = np.zeros(len(joint_words))
i = 0
# wordset = set(words) in original but set changes element order
wordDict = {x[1]: x[0] for x in enumerate(words)}
for joint_word in joint_words:
if joint_word in wordDict:
# word in joint_words found in sentence, just populate the index
wovec[i] = wordDict[joint_word]
else:
# word not in joint_words, find most similar word and populate
# word_vector with the thresholded similarity
wordSet = set(words)
sim_word, max_sim = most_similar_word(joint_word, wordSet)
if max_sim > ETA:
wovec[i] = wordDict[sim_word]
else:
wovec[i] = 0
i = i + 1
return wovec
```

Reply

Sujit Pal 9/12/2015 9:57 AM

You are welcome and thanks for the code change. Regarding the code change, I think my original code was passing in the word order via the windex argument, whereas your modification figures it out by enumerating words and creating a dict. Did the original code not work or is this an improvement? In any case, since Blogger comments tend to destroy formatting and formatting is so important for Python code, I am going to repost it with ".." replacing one indent, that way someone can copy-

paste your version into the code easily.

```
def word_order_vector(words, joint_words):
..""""
..Computes the word order vector for a sentence. The sentence is passed
..in as a collection of words. The size of the word order vector is the
..same as the size of the joint word set. The elements of the word order
..vector are the position mapping (from the windex dictionary) of the
..word in the joint set if the word exists in the sentence. If the word
..does not exist in the sentence, then the value of the element is the
..position of the most similar word in the sentence as long as the similarity
..is above the threshold ETA.
..""""
..wovec = np.zeros(len(joint_words))
..i = 0
..# wordset = set(words) in original but set changes element order
..wordDict = {x[1]: x[0] for x in enumerate(words)}
..for joint_word in joint_words:
....if joint_word in wordDict:
......# word in joint_words found in sentence, just populate the index
......wovec[i] = wordDict[joint_word]
....else:
......# word not in joint_words, find most similar word and populate
......# word_vector with the thresholded similarity
......wordSet = set(words)
......sim_word, max_sim = most_similar_word(joint_word, wordSet)
......if max_sim > ETA:
........wovec[i] = wordDict[sim_word]
......else:
........wovec[i] = 0
....i = i + 1
..return wovec
```

Reply

Ekta Singh 10/15/2015 6:01 AM

Hi Sujit,

Excellent paper.

Just wanted to ask why aren't we using wup-similarity for calculating sentence similarity. How is it different from the function that you have written.

Reply

Sujit Pal 10/15/2015 8:59 AM

Thanks Ekta, although I didn't write the paper, just implemented it at best as I could. You are right, Wu-Palmer Similarity could be used also to find the difference between individual words since it is so similar (and uses similar metrics to derive the similarity as the one described). In my case I was trying to follow the paper as closely as possible. Once you have the inter-word similarities you could use the algorithm in the paper to find the distance between the sentences.

Reply

colla 10/30/2015 3:53 AM

Dear Sujit, thanks for your codes. Really helpful. Can I please have your email address to discuss off here?
thanks

Reply

Sujit Pal 10/30/2015 7:58 AM

Hi Colla, I prefer to not share my email on a public page. If you send me your email address as a comment, I can contact you on it and delete your comment so your email address is not public either.

Reply

Anonymous 11/08/2015 3:53 PM

Hello, I like your job in implement of algorithm, I can use it in my dissertation?

Reply

Sujit Pal 11/08/2015 7:17 PM

The implementation is based on an algorithm put forward in an existing paper (referenced in the first sentence of this blog post). I am not sure what you mean by "using", so I guess the answer would depend on that.

Reply

Naveed Afzal 12/17/2015 8:03 AM

Hi Sujit,

Great post. My question regarding this implementation is that this code will never assign a score of 0 (zero) to completely dissimilar sentences. Do you have any suggestion that how to modify this code that it gives a score of zero to completely dissimilar sentences.
As an example from your sentence pairs:
Red alcoholic drink. An English dictionary. 0.000 0.237
Here two sentences are totally dissimilar and actual score is 0 but the code results in a score of 0.237
Really appreciate your guidance to address this shortcoming.
Thanks

Reply

Sujit Pal 12/17/2015 2:37 PM

Thanks Naveed. I guess one simple way could be to consider everything below a certain cutoff as dissimilar, as long as it was universally applicable, which it doesn't seem to be (at least w/o IC). I think its a good way to compare two pairs of strings rather than an absolute indicator of similarity/dissimilarity.

Reply

zahra 12/20/2015 9:55 AM

Hi Sujit,thanks for your codes. do you have java implemention of this code?!
Thanks in advance.

Reply

Sujit Pal 12/21/2015 6:20 AM

Hi Zahra, no I don't have a Java implementation. But you should be able to build it fairly easily from the Scala version.

Reply

Anonymous 1/17/2016 9:22 AM

hi sujit,
thanks alot sujit for great article , it helped alot though, could you please give reference of how to extract corpus reuters.

i am trying to cluster Reuters, NEWS20 and OSHUMD based on content semantic using Wordnet ontology, could you please help me out with this problem, i will be agglomatitve hierachal based clustring.

Reply

Sujit Pal 1/17/2016 10:39 AM

You are welcome, glad it helped you. For Reuters, NLTK has a reader as described here. If you want to extract the text from the source, there are various parsers available such as this one and this one.

Reply

Anonymous 1/18/2016 12:01 AM

Thanks alot Sujit, again, but could u please tell me how to apply clustring decision based on semantics of corpus using wordNet or anything u want to suggest where i will be making clustere decision based on semantics

Reply

Anonymous 3/03/2016 2:20 AM

Hi Sujit,

This is really a great post. Enjoying it totally. Wanted to discuss with you on importance of word order vector.

Taking word order into account is surely a better approach than bag of words. But, a lot of times, two sentences may have same meaning but completely different word orders. Is word order actually a measure of similarity between sentences?

Also, how easy/difficult it would be to take care of n-grams with this algorithm?

Thanks a lot again !

Reply

Anonymous 3/03/2016 2:23 AM

Hi Sujit,

What are your observations about coverage of WordNet.

Lot of researchers seem to mention that the coverage of WordNet is low and can be an issue but still its one of the most used resource. Have you come across any better alternatives?

Thanks

Reply

Sujit Pal 3/03/2016 9:43 AM

Thank you anonymous, and you bring up good points. Regarding word order, it seems to work well to predict similarity for the dataset. I did not actually think too much about the negative cases, but I am guessing the author of the paper on which this is based (referenced at the top of the post) must have, and concluded that the word order helps more than it harms, ie, maybe there are fewer cases where word order is different for two sentences with similar meaning, so overall its a win. I think it should be possible to use n-grams for the length similarity but probably less so for the others (hierarchical and semantic). Regarding wordnet coverage, it /is/ quite low, but if you have enough cases, you can sort of "smooth" over that, so it works out. I guess you could consider things like Yago similar, but it is not manually curated like Wordnet is and its coverage is broader.

Reply

Sujit Pal 3/08/2016 6:11 PM

Replying to this comment dated Jan 18 2016, must have missed it, sorry about that, found it in my queue when I looked today.

>> Thanks alot Sujit, again, but could u please tell me how to apply clustring decision based on semantics of corpus using wordNet or anything u want to suggest where i will be making clustere decision based on semantics

You are welcome, but wouldn't the choice of the ontology (in this case I am thinking of Wordnet as an ontology of words by grammar sense) to lookup be dependent on the type of content you are looking to cluster? Also I think we might have already discussed this question in more detail later (if you are the same Anonymous).

Reply

Vikram Gupta 3/09/2016 3:55 AM

Hi Sujit,

Thanks for the excellent blog !

I want to understand the intuition behind using "path_similarity" to find the best matching synsets and then using a different measure "shortest_path_distance" to find the actual distance. Why are we using two different algorithm to find similarity between two synsets ?

Reply

Nadhiya Nadhi 3/15/2016 2:09 AM

i need the algorithm of wu plamer similarity algorithm

Reply

Sujit Pal 3/15/2016 7:33 AM

@Vikram: you are welcome, glad you found it useful. I am using the two because there can be multiple synset pairs to compare because the word can correspond to multiple synsets. So I am using the pair that is the closest. I could also have done the shortest path distance across all the synset pairs and chosen the minimum distance.

@Nadhiya: From the Wordnet Similarity page, here is the definition: Return a score denoting how similar two word senses are, based on the depth of the two senses in the taxonomy and that of their Least Common Subsumer (most specific ancestor node).

Reply

Ramanujan 3/25/2016 3:13 PM

Hi Sujit,

I am working in a part of project for sentence similarity, we are using unsupervised learning. I need your help in adding extra feature(or papers) to add as part of our model. It would be very helpful for me, Can you please provide if you have it.

Thanks & Regards

Reply

Sujit Pal 3/25/2016 4:27 PM

Hi Ramanujan, this is the only paper on sentence similarity for short sentences that I know about. But I looked up some results on Scopus, maybe these are helpful?

Reply

Anonymous 4/01/2016 1:58 AM

hello sujit,
i m using reuters-21578 for clustering based on semantics, which i will take from wordNET, approach i m trying to follow is to extract topics using LDA then i want to map those topics to wordNET hypernym, as GRAIN topic will be concept food,instead of making term frequency vector i will make concept vector , which will reduce high demensionality of large data set because it will be based on concept vector instead of term vector. Now the problem is i am unable to map wordNET hypernym with Topics i extracted using LDA. i m using Pythoin for that
.
regards

Reply

Sujit Pal 4/01/2016 10:12 PM

Are you unable to map because the correct hypernym does not exist? I tried with your example, I see that the 2nd synset maps approximately to what you want.

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets("grain")
[Synset('grain.n.01'), Synset('grain.n.02'), Synset('grain.n.03'), Synset('grain.n.04'), Synset('grain.n.05'), Synset('grain.n.06'), Synset('grain.n.07'), Synset('grain.n.08'), Synset('grain.n.09'), Synset('grain.n.10'), Synset('texture.n.05'), Synset('ingrain.v.01'), Synset('grain.v.02'), Synset('granulate.v.01'), Synset('granulate.v.02')]
>>> wn.synset("grain.n.01").hypernyms()
[Synset('atom.n.02')]
>>> wn.synset("grain.n.02").hypernyms()
[Synset('foodstuff.n.02')]
```

Cant think of a simple way to do this automatically though... there is no indication in the word about its hypernym. One possibility could be to look at different words in the topic and see if they map to locations that are closer to one hypernym than another, and choose the closest hypernym to all words in the topic, but then thats kind of circular...

Reply

Anonymous 4/05/2016 3:05 AM

hello Sujit,thanks for reply,
i would simply do it prototype level, whatever will be availble i will only use those concepts if any topics concept is not avaible thn wouldnot take it ..
so could you refer in this scenario now.

Reply

Sujit Pal 4/05/2016 6:59 AM

Then it might be a workable idea, although there is still the problem of choosing the right synset to collect hypernyms from. Since LDA yields a distribution of words for each topic, you could choose maybe the top N or some threshold to consider a subset of most probable words for each topic, and use the synset which is closest to the subset of these words.

Reply

Anonymous 6/02/2016 7:27 PM

Hello Sujit, I would like to ask permission to use their implements as a reference for testing of Sentence Similarity Based on Semantic Nets and Corpus Statistics, this was the only implementation I found that article.

Reply

Sujit Pal 6/03/2016 12:24 AM

Hi, if you are looking for my permission to use the code in this blog post, you have my permission to use it. If you are looking for permission to use the code in the paper (I don't think there was any, but from your question it appears that there might be), then you have to ask the authors of the paper.

Reply

Anonymous 6/16/2016 6:38 PM

Hi Sujit,
Great blog post. I was wondering if you were aware of any open source projects which do Sentence Semantic Similarity Analysis. I am making a QnA system and I need to match User Input with the most similar question of my Database. I have tried your above algorithm plus a few of my own, but didnt reach too much accuracy since the whole idea is built on matching words rather than matching sentences.

Also, should we not give higher weightages e.g when nouns match instead of adjectives? Are there any libraries that do that?

Reply

Sujit Pal 6/19/2016 4:20 PM

Thank you, and sorry, I don't know of a project that does sentence semantic similarity analysis like you described. Perhaps you could do something with word2vec? You could look up words in your query and get their word vectors, then sum them up to form your query vector. On the database side, you could do the same thing for words in the candidate questions, then sum them up to form question vectors, then find the one closest by some metric such as cosine or euclidean distance. You could also include weights to prefer nouns over adjectives in such a pipeline - there are libraries such as NLTK (Python) and OpenNLP (Java/Scala) to get POS tags for words in sentences.

Reply

Ganesh 9/12/2016 11:42 PM

Hi,

Your post is really useful who wants to learn NLP things and implement them.
I want to achieve similar thing i.e. I want to check whether two statements are similar or opposite to each other.
Can your implementation achieve this? I know your implementation is for similarity but is h there any way to check opposite-ness?

Thanks,
Ganesh

Reply

Sujit Pal 10/05/2016 8:03 PM

Thanks for the kind words Ganesh. Unfortunately I can't think of a way to measure out oppositeness like you are looking for. One obvious way, especially if the range of the metric value is 0-1 would be to think of it as a probability and compute oppositeness as 1 - similarity, but I don't remember if the similarity metric has that range or not.

Reply

sherlockatszx 10/10/2016 3:22 AM

I tested "Apples eat me" and "I eat apples". Sadly, this sentence pair socres 0.59,which is not reasonable. It seems that need to add semantci dependency parsing into.Do you got any improved method to deal with this kind semantic tricks

Reply

Sujit Pal 10/10/2016 9:50 AM

Hi sherlockatszx, Here is something more recent that uses intermediate nodes in the parse tree for the sentences, although you might not find much difference with that particular sentence.

Reply

Anonymous 10/20/2016 4:47 PM

Hi Sujit,

First of all, excellent code.

However, I wanted to reproduce the original values obtained in the paper by modifying your code, which I am unable to. Can you tell me how to modify the get_best_pair function so that the code produces the same values as in the paper?

Thanks.

Shaown S.

Reply

Sujit Pal 10/21/2016 7:55 AM

Hi Shaown, thanks for the kind words. As I mention at the beginning of the post, I wasn't able to reproduce the values reported in the paper either, however, the rankings seem to be correct and largely match the rankings reported in the paper. Its been a while since I wrote the post, but IIRC there were some places where I improvised on the original paper because (a) the paper was underspecified or (b) I thought I could do better. One such case is mentioned (search for "did differently") - you can try computing the difference of synsets_1[0] and synsets_2[0] instead.

Reply

Abebawu Eshetu 11/20/2016 2:51 AM

Dear Salmon Run

I am one of your followers. I always think that you are lucky guy, because I believe that the most interesting task in this world is sharing what you have to others. Thank you really sir.
I was looking for your early post on "Computing Semantic Similarity for Short Sentences" to compare two essay text semantically. While I was looking for it I had some confusion
1. Does it work for two essay text?
2. When computing order similarity of two sentence what if they are semantically same while their word order in active and passive sentence form.
3. What is necessity of large text corpus used "brown"

Hopefully you will help me because this time the answer is compulsory for me. I am graduate student and this task is part of my work. I now you are too busy, but I appreciate any time give for me.

Reply

Sujit Pal 12/01/2016 9:31 AM

You are welcome and hank you for the kind words, Abebawu. To answer your questions:
1) This is for short sentences, it is likely to become computationally too expensive for long texts. For long text, it might be better to reduce the document to a single meaning vector. Easiest is adding up the sparse one-hot vectors for the words for a bag-of-words model, more complex if you want to use word embeddings and combine them to form meaning vectors.
2) The semantic similarity component of the score should pick that up.
3) Its for computing the information content of a word in context of some "standard" text. This is used to discount the similarity component.

Reply

Sravanthi Pantul 12/15/2016 2:04 AM

hi sir , thanks for the implementation which is very useful for my work but i have a query regarding the word_pairs u have mentioned the values along with word-pairs , may i know on what basis we are getting those values.
sir plz clarify my doubt i stopped implementing for work.

Reply

Sujit Pal 12/15/2016 7:49 PM

Hi Sravanthi, glad you found the post helpful to your work. The expected similarities came from the paper referenced towards the beginning of the post, and the actual similarities come from my code shown in the post. The code attempts to follow the algorithm laid down in the paper, but is not identical, I made some changes based on what was convenient and available to me. I have done another post with a different algorithm on the same data, maybe you might find that interesting also.

Reply

frolickbbc 1/04/2017 9:47 PM

Hi Sujit,
I found this post really helpful and thanks for making out this post. I have close to 1 million support ticket data and its a short text. I need to cluster the sentence based on semantic and word order similarity. How I can go about this..

Thanks
Bharath

Reply

Sujit Pal 1/30/2017 9:35 AM

Hi Bharath, one possibility might be to package up the code in the blog into a custom distance function and then call it in a K nearest neighbors classifier. Look at the accepted answer in this thread on Stack Overflow, it has links to the relevant documentation for scikit-learn components.

Reply

Enter your comment...

Comment as: Google Accour

Publish Preview

Links to this post

Create a Link

Rs999	Rs599	Rs1,295
Rs899	Rs499	Rs699

About me

Sujit Pal

I am a programmer interested in Semantic Search, Ontology, Natural Language Processing and Machine Learning. My programming languages of choice are Java, Scala, and Python. I love solving problems and exploring different possibilities with open source tools and frameworks.

View my complete profile

Powered by Blogger.