VIT
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
VELLORE ▪ CHENNAI
www.vit.ac.in

"Question - Answer Relevancy and Sentiment analysis"

PROJECT REPORT

Submitted for CAL in B. Tech Natural Language Processing (CSE4022)

By

Shubham Barudwale 15BCE1162

Yash Harkhani 15BCE1340

Umang Shah 15BCE1303

Monark Dedakiya 15BCE1275

Tejasvi Malladi 15BCE1208

Name of faculty: Dr. G. Bharadwaja Kumar

(SCHOOL OF COMPUTER SCIENCE ENGINEERING)

<u>CERTIFICATE</u>

This is to certify that the Project work entitled "***Question - Answer Relevancy and Sentiment analysis***" that is being submitted by "***Shubham Barudwale,Yash Harkhani,Umang Shah,Monark Dedakiya,Tejasvi Malladi***" for CAL in B.Tech Natural Language Processing (CSE4022) is a record of bonafide work done under my supervision. The contents of this Project work have not been submitted for any other CAL course.

Place: Chennai

Date: 11 Nov, 2017

ACKNOWLEDGEMENTS

We thank VIT University (**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**) for giving us the opportunity to conduct this project and experiment. We also thank our guide for project ***Dr. Bharadwaja Kumar*** for his constant, good and knowledgeable guidance for the project. Through this project, we learnt many new things about **Question - Answer Relevancy and Sentiment analysis** which will be definitely useful for us.

Shubham Barudwale

Reg. No. 15BCE1162

Yash Harkhani

Reg. No. 15BCE1340

Umang Shah

Reg. No. 15BCE1303

Monark Dedakiya

Reg. No. 15BCE1275

Tejasvi Malladi

Reg. No. 15BCE1208

Abstract

The ability to accurately judge the similarity between natural language sentences is critical to the performance of several applications such as text mining, question answering, and text summarization. Given two sentences, an effective similarity measure should be able to determine whether the sentences are semantically equivalent or not, taking into account the variability of natural language expression.

Determining the similarity between sentences is one of the crucial tasks which have a wide impact in many text applications. In information retrieval, similarity measure is used to assign a ranking score between a query and texts in a corpus. Question answering application requires similarity identification between a question-answer or question-question pair

Two short texts can be very similar while using different words, thus capturing the meaning of the words by word2vec and using this information to compare documents should allow us to find that similar documents using different words are indeed similar. This special case, where almost no overlapping exists between the words of the first document and those of the second, is not correctly handled with commonly used methods based on the Vector Space Model.

WordNet is a lexical database which is available online, and provides a large repository of English lexical items. There is a multilingual WordNet for European languages which is structured in the same way as the English language WordNet. WordNet was designed to establish the connections between four types of Parts of Speech (POS) - noun, verb, adjective, and adverb. The smallest unit in a WordNet is synset, which represents a specific meaning of a word. It includes the word, its explanation, and its synonyms.

An approach for capturing similarity between words that was concerned with the syntactic similarity of two strings. Semantic similarity is a confidence score that reflects the semantic relation between the meanings of two sentences. It is difficult to gain a high accuracy score because the exact semantic meanings are completely understood only in a context. We have done this using two Methods WordNet and Word-to-Vector.

WordNet

WordNet is a lexical database which is available online and provides a large repository of English lexical items. There is a multilingual WordNet for European languages which are structured in the same way as the English language WordNet.

WordNet was designed to establish the connections between four types of Parts of Speech (POS) - noun, verb, adjective, and adverb. The smallest unit in a WordNet is synset, which represents a specific meaning of a word. It includes the word, its explanation, and its synonyms. The specific meaning of one word under one type of POS is called a sense. Each sense of a word is in a different synset. Synsets are equivalent to senses = structures containing sets of terms with synonymous meanings. Each synset has a gloss that defines the concept it represents. For example, the words night, nighttime and dark constitute a single synset that has the following gloss: the time after sunset and before sunrise while it is dark outside. Synsets are connected to one another through the explicit semantic relations. Some of these relations (hypernym, hyponym for nouns and hypernym and troponym for verbs) constitute is-a-kind-of (holonymy) and is-a-part-of (meronymy for nouns) hierarchies.

For example, tree is a kind of plant, tree is a hyponym of plant and plant is a hypernym of tree. Analogously, trunk is a part of a tree and we have that trunk as a meronym of tree and tree is a holonym of trunk. For one word and one type of POS, if there is more than one sense, WordNet organizes them in the order of the most frequently used to the least frequently used (Semcor).

Many natural language processing applications must directly or indirectly assess the semantic similarity of text passages. Modern approaches to information retrieval, summarization, and textual entailment, among others, require robust numeric relevance judgments when a pair of texts is provided as input. Although each task demands its own scoring criteria, a simple lexical overlap measure such as cosine similarity of document vectors can often serve as a surprisingly powerful baseline. We argue that there is room to improve these general-purpose similarity measures, particularly for short text passages.

Most approaches fall under one of two categories. One set of approaches attempts to explicitly account for fine-grained structure of the two passages, e.g. by aligning trees or constructing logical forms for theorem proving. While these approaches have the potential for high precision on many examples, errors in alignment judgments or formula construction are often insurmountable. More broadly, it's not always clear that there is a correct alignment or logical form that is most appropriate for a particular sentence pair. The other approach tends to ignore structure, as canonically represented by the vector space model, where any lexical item in common between the two passages contributes to their similarity score. While these approaches often fail to capture distinctions imposed by, e.g. negation, they do correctly capture a broad notion of similarity or about-ness.

Word to Vector Conversion:-

For word to vector conversation the tree is defined for class and object node for generalize and relevancy checking e.g. if there is a class as a Car then its sub nodes can be sedan, SUV, Hatchback, MPV, Crossover etc. or as similar to that class can be vehicle and its sub node as cycle, bike, car, truck etc. thus tree is defined for word classification.

We calculated the similarity or relevancy between question and answer with both normalized and non-normalized forms of the sentences. After then we had to check for word order similarity which computes the word-order similarity between two sentences as the normalized difference of word order between the two sentences. For that we tokenized the words and converted it into word order vector it computes the semantic vector of a sentence. The sentence is passed in as 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 normalization is True.

This processed set of the word used to look up for info content which Uses the Brown corpus available in NLTK to calculate a Laplace smoothed frequency distribution of words, then uses this information to compute the information content of the lookup word. Also most similar words will be Find the word in the joint word set that is most similar to the word passed in. We use the algorithm above to compute word similarity between the word and each word in the joint word set, and return the most similar word and the actual similarity value. And in parallel we also compute the hierarchy distance and length distance between the set of words of sentences.

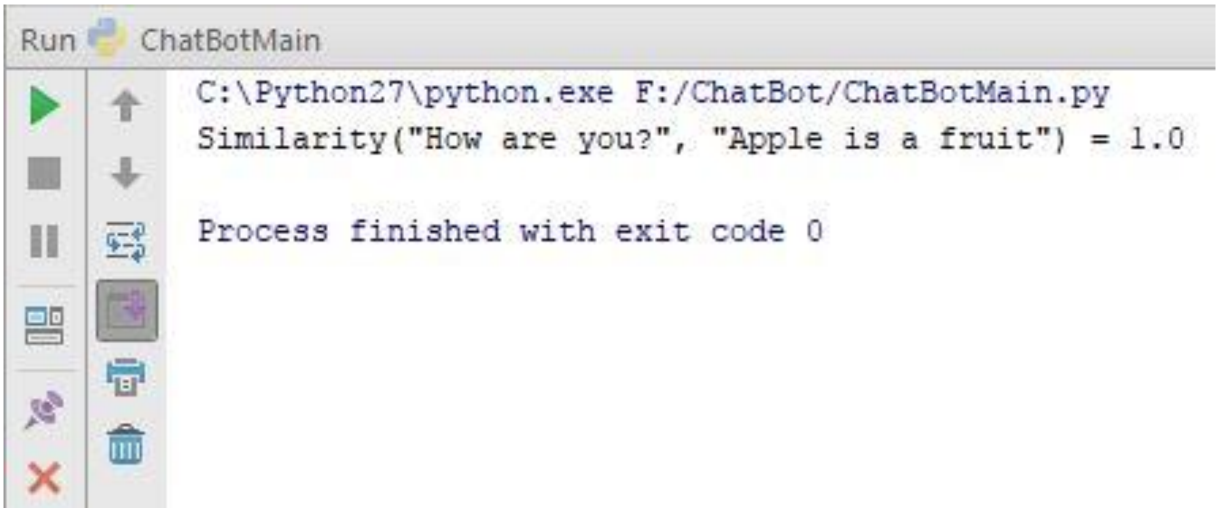
Code: chatbot main interface: -

```
# from gensim import corpora,models,similarities
#
# doc1 = "How old are you?"
#
# doc2 = "I am eighteen years old."

from nltk import word_tokenize, pos_tag
from nltk.corpus import wordnet as wn
#from nltk.corpus import nps_chat as npsc

def penn_to_wn(tag):
    """ Convert between a Penn Treebank tag to a simplified Wordnet
tag """
    if tag.startswith('N'):
        return 'n'

    if tag.startswith('V'):
        return 'v'

    if tag.startswith('J'):
        return 'a'

    if tag.startswith('R'):
        return 'r'

    return None


def tagged_to_synset(word, tag):
    wn_tag = penn_to_wn(tag)
    if wn_tag is None:
        return None

    try:
        return wn.synsets(word, wn_tag)[0]
    except:
        return None


def sentence_similarity(sentence1, sentence2):
    """ compute the sentence similarity using Wordnet """
    # Tokenize and tag
    sentence1 = pos_tag(word_tokenize(sentence1))
    sentence2 = pos_tag(word_tokenize(sentence2))

    # Get the synsets for the tagged words
    synsets1 = [tagged_to_synset(*tagged_word) for tagged_word in
sentence1]
```

```
    synsets2 = [tagged_to_synset(*tagged_word) for tagged_word in
sentence2]

    # Filter out the Nones
    synsets1 = [ss for ss in synsets1 if ss]
    synsets2 = [ss for ss in synsets2 if ss]

    score, count = 0.0, 0

    # For each word in the first sentence
    for synset in synsets1:
        # Get the similarity value of the most similar word in the
other sentence

        ScoreList = []

        for ss in synsets2:
            appendscore = synset.path_similarity(ss)
            if(appendscore != None):
                ScoreList.append(appendscore)


        if(len(ScoreList)==0):
            ScoreList.append(0)

        best_score = max(ScoreList)


        # Check that the similarity could have been computed
        if best_score is not None:
            score += best_score
            count += 1

    # Average the values
    score /= count
    return score


sentences = [
    "Clock rotates?"
]

from SpeechToText import text

focus_sentence = text

#focus_sentence = "Apple is a fruit"

for sentence in sentences:
    #print("Similarity(\"%s\", \"%s\") = %s" % (focus_sentence,
sentence, sentence_similarity(focus_sentence, sentence)))
```

```
    print("Similarity(\"%s\", \"%s\") = %s" % (sentence,
focus_sentence, sentence_similarity(sentence, focus_sentence)))
```

OUTPUT: -



Code for Word to vector conversion: -

```
from __future__ import division
import nltk
from nltk.corpus import wordnet as wn
from nltk.corpus import brown
import math
import numpy as np
import sys

# Parameters to the algorithm. Currently set to values that was
reported
# in the paper to produce "best" results.
ALPHA = 0.2
BETA = 0.45
ETA = 0.4
PHI = 0.2
DELTA = 0.85

brown_freqs = dict()
N = 0
```

```
######################### word similarity ##########################

def get_best_synset_pair(word_1, word_2):
    """
    Choose the pair with highest path similarity among all pairs.
    Mimics pattern-seeking behavior of humans.
    """
    max_sim = -1.0
    synsets_1 = wn.synsets(word_1)
    synsets_2 = wn.synsets(word_2)
    if len(synsets_1) == 0 or len(synsets_2) == 0:
        return None, None
    else:
        max_sim = -1.0
        best_pair = None, None
        for synset_1 in synsets_1:
            for synset_2 in synsets_2:
                sim = wn.path_similarity(synset_1, synset_2)
                if sim > max_sim:
                    max_sim = sim
                    best_pair = synset_1, synset_2
        return best_pair


def length_dist(synset_1, synset_2):
    """
    Return a measure of the length of the shortest path in the
semantic
    ontology (Wordnet in our case as well as the paper's) between two
    synsets.
    """
    l_dist = sys.maxint
    if synset_1 is None or synset_2 is None:
        return 0.0
    if synset_1 == synset_2:
        # if synset_1 and synset_2 are the same synset return 0
        l_dist = 0.0
    else:
        wset_1 = set([str(x.name()) for x in synset_1.lemmas()])
        wset_2 = set([str(x.name()) for x in synset_2.lemmas()])
        if len(wset_1.intersection(wset_2)) > 0:
            # if synset_1 != synset_2 but there is word overlap,
return 1.0
            l_dist = 1.0
        else:
            # just compute the shortest path between the two
            l_dist = synset_1.shortest_path_distance(synset_2)
            if l_dist is None:
                l_dist = 0.0
    # normalize path length to the range [0,1]
```

```
    return math.exp(-ALPHA * l_dist)


def hierarchy_dist(synset_1, synset_2):
    """
    Return a measure of depth in the ontology to model the fact that
    nodes closer to the root are broader and have less semantic
similarity
    than nodes further away from the root.
    """
    h_dist = sys.maxint
    if synset_1 is None or synset_2 is None:
        return h_dist
    if synset_1 == synset_2:
        # return the depth of one of synset_1 or synset_2
        h_dist = max([x[1] for x in synset_1.hypernym_distances()])
    else:
        # find the max depth of least common subsumer
        hypernyms_1 = {x[0]: x[1] for x in
synset_1.hypernym_distances()}
        hypernyms_2 = {x[0]: x[1] for x in
synset_2.hypernym_distances()}
        lcs_candidates = set(hypernyms_1.keys()).intersection(
            set(hypernyms_2.keys()))
        if len(lcs_candidates) > 0:
            lcs_dists = []
            for lcs_candidate in lcs_candidates:
                lcs_d1 = 0
                if hypernyms_1.has_key(lcs_candidate):
                    lcs_d1 = hypernyms_1[lcs_candidate]
                lcs_d2 = 0
                if hypernyms_2.has_key(lcs_candidate):
                    lcs_d2 = hypernyms_2[lcs_candidate]
                lcs_dists.append(max([lcs_d1, lcs_d2]))
            h_dist = max(lcs_dists)
        else:
            h_dist = 0
    return ((math.exp(BETA * h_dist) - math.exp(-BETA * h_dist)) /
            (math.exp(BETA * h_dist) + math.exp(-BETA * h_dist)))


def word_similarity(word_1, word_2):
    synset_pair = get_best_synset_pair(word_1, word_2)
    return (length_dist(synset_pair[0], synset_pair[1]) *
            hierarchy_dist(synset_pair[0], synset_pair[1]))


######################### sentence similarity
##########################

def most_similar_word(word, word_set):
```

```
    """
    Find the word in the joint word set that is most similar to the
word
    passed in. We use the algorithm above to compute word similarity
between
    the word and each word in the joint word set, and return the most
similar
    word and the actual similarity value.
    """
    max_sim = -1.0
    sim_word = ""
    for ref_word in word_set:
        sim = word_similarity(word, ref_word)
        if sim > max_sim:
            max_sim = sim
            sim_word = ref_word
    return sim_word, max_sim


def info_content(lookup_word):
    """
    Uses the Brown corpus available in NLTK to calculate a Laplace
    smoothed frequency distribution of words, then uses this
information
    to compute the information content of the lookup_word.
    """
    global N
    if N == 0:
        # poor man's lazy evaluation
        for sent in brown.sents():
            for word in sent:
                word = word.lower()
                if not brown_freqs.has_key(word):
                    brown_freqs[word] = 0
                brown_freqs[word] = brown_freqs[word] + 1
                N = N + 1
    lookup_word = lookup_word.lower()
    n = 0 if not brown_freqs.has_key(lookup_word) else
brown_freqs[lookup_word]
    return 1.0 - (math.log(n + 1) / math.log(N + 1))


def semantic_vector(words, joint_words, info_content_norm):
    """
    Computes the semantic vector of a sentence. The sentence is
passed in as
    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
##########################
```

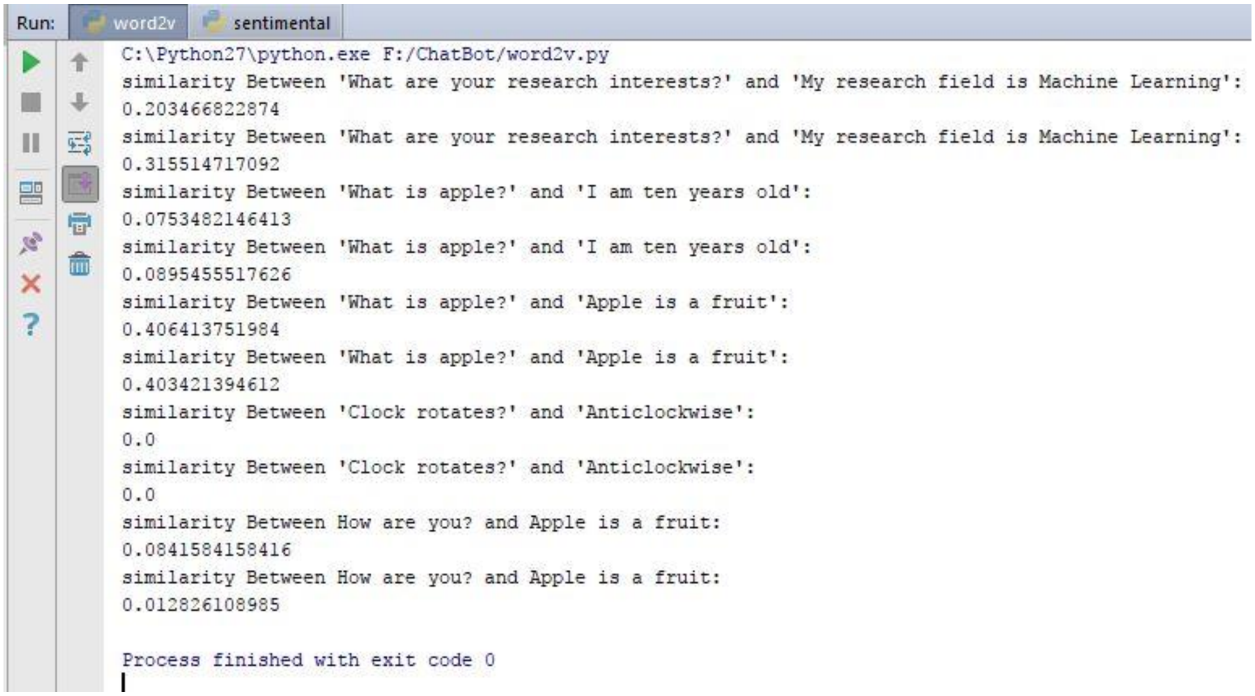
```
def word_order_vector(words, joint_words, windex):
    """
    Computes the word order vector for a sentence. The sentence is 
passed
    in as a collection of words. The size of the word order vector is 
the
    same as the size of the joint word set. The elements of the word 
order
    vector are 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))

print (similarity('What are your research interests?', 'My research
field is Machine Learning', False))
print (similarity('What are your research interests?', 'My research
field is Machine Learning', True))

print (similarity('What is apple?', 'I am ten years old', False))
print (similarity('What is apple?', 'I am ten years old', True))

print (similarity('What is apple?', 'Apple is a fruit', False))
print (similarity('What is apple?', 'Apple is a fruit', True))

print (similarity('Clock rotates?', 'Anticlockwise', False))
```

```
print (similarity('Clock rotates?', 'Anticlockwise', True))
```

OUTPUT: -



Sentiment analysis: -

The human language is complex. Teaching a machine to analyse the various grammatical nuances, cultural variations, slang and misspellings is a difficult process. Teaching a machine to understand how context can affect tone is even more difficult. This is what is done in sentiment analysis.

It is the process of determining the emotional tone behind a series of words, used to gain an understanding of the attitudes, opinions and emotions expressed within an online mention. Sentiment analysis ideally is about subjective impressions, not facts.

Humans are fairly intuitive when it comes to interpreting the tone of a piece of writing.

Consider the following sentence: “My flight’s been delayed. Brilliant!”

Most humans would be able to quickly interpret that the person was being sarcastic. We know that for most people having a delayed flight is not a good experience (unless there’s a free bar as recompense involved). By applying this contextual understanding to the sentence, we can easily identify the sentiment as negative.

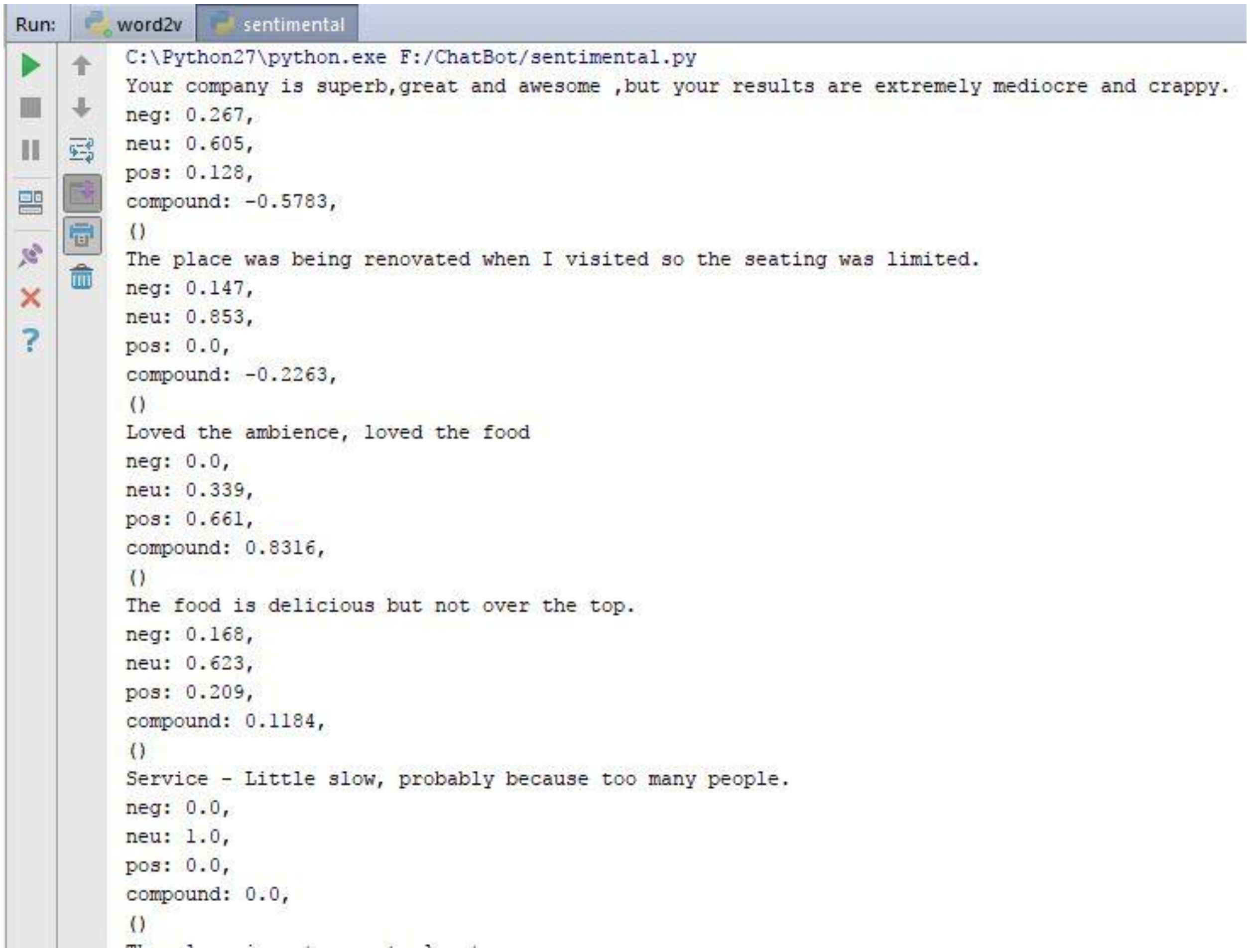
Without contextual understanding, a machine looking at the sentence above might see the word "brilliant" and categorise it as positive. Considering the tone, attitude, intention of a person is a crucial task that all the interviewers are supposed to do, irrespective of whether the interviewer is a human or a bot (piece of intelligent and interactive code). The concept of sentiment analysis is a primitive but promising solution for considering the sentiment of the participant. A positive sentiment reveals that the person has a positive approach towards the question put across.

Sentiment analysis is done using NLP, statistics, or machine learning methods to extract, identify, or otherwise characterize the sentiment content of a text unit.

In the work carried out, python's NLTK sentiment analyser was used to do sentiment analysis of the participant's response the following is the code used in sentiment analysis.

```
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

hotel_rev = ["Your company is superb,great and awesome ,but your results are extremely mediocre and crappy.",
             "The place was being renovated when I visited so the seating was limited.",
             "Loved the ambience, loved the food",
             "The food is delicious but not over the top.",
             "Service - Little slow, probably because too many people.",
             "The place is not easy to locate",
             ""]

sid = SentimentIntensityAnalyzer()
for sentence in hotel_rev:
    print(sentence)
    ss = sid.polarity_scores(sentence)
    for k in ss:
        print('{0}: {1}, '.format(k, ss[k]))
    print()
```

<u>OUTPUT</u>: -



<u>CONCLUSION</u>: -

When the input question was “How are you?” and given answer was “Apple is a fruit” the non-word-to-vec algorithm implemented in ChatbotMain.py gave an accuracy of 1.0 because it found strong relation between are in question and ‘is’ in answer. Thus, without deleting stopping words the algorithm gives wrong output. Whereas for same sentence the word-to-vec algorithm gave an accuracy of 0.08 which is less than the threshold value 0.3 thus Word-to-vec is better approach here.

<u>REFRENCES</u>: -

- https://www.codeproject.com/Articles/11835/WordNet-based-semantic-similarity-measurement
- http://www.jsoftware.us/vol8/jsw0806-22.pdf
- https://www.kernix.com/blog/similarity-measure-of-textual-documents_p12
- https://demos.explosion.ai/similarity/?text1=This%20is%20a%20sentence&text2=This%20is%20another%20sentencehttp://stp.lingfil.uu.se/~santinim/sais/Ass1_Essays_FinalVersion/Segeblad_Jesper_essay.pdf
- http://www.nltk.org/
- https://wordnet.princeton.edu/
- http://textminingonline.com/getting-started-with-word2vec-and-glove-in-python
- https://github.com/stanfordnlp/GloVe
- https://nlp.stanford.edu/projects/glove/
- https://stackoverflow.com/
- word2vec documentation