CS-5340/6340, Programming Assignment #3
Due: Friday, October 28, 2022 by 11:59pm

In this assignment you will need to implement algorithms for word sense disambiguation, which assigns a word sense to an occurrence of a target word. You will need to implement: 1) the Lesk algorithm, and 2) a nearest neighbor method using distributional vectors.

PART 1: Lesk Algorithm

You should implement the **simplified Lesk algorithm** (see page 2) to determine the best sense for each instance of a target word. As input, your program should accept (1) a Test Sentences file, (2) a Sense Definitions file, and (3) a Stopwords file. Your program should be named "**lesk**" and should accept the files as command-line arguments in the order above. For example, we should be able to run your program like so:

```
python3 lesk.py test.txt definitions.txt stopwords.txt
```

If you use Java, the arguments should similarly be accepted on the command-line in the same order.

Test Sentences File

The test sentences file will consist of sentences, where each sentence contains an instance of the target word that needs to be disambiguated. The sentences are separated by a line-break "\n".

Each occurrence of the target word will be a single word and enclosed in brackets like this: "<occurrence>WORD</>". You may find different morphological variants of the target word, but all occurrences will have the same root. For example, if the target word is "*line*", you may find both "*line*" and "*lines*" labeled as occurrences of "*line*".

For example, the test sentences file for the target word "*line*" might look like this:

> *Just say the <occurrence>lines</> flat without much emphasis .*
> *Canadians use the North American qwerty keyboard , named for the sequence of keys on the top <occurrence>line</> of letters .*
> *In Houston , <occurrence>lines</> of families wrapped around the shopping mall that houses the legalization center .*

Sense Definitions File

The sense definitions file will contain all possible senses for the target word. Each sense will be accompanied by a definition (its "gloss") and an example sentence that are each separated by a tab "\t". Each line of the file will have the following format:

<sense>\t<one definition sentence>\t<one example sentence>

For example, an entry for one sense of the target word "line" could be:

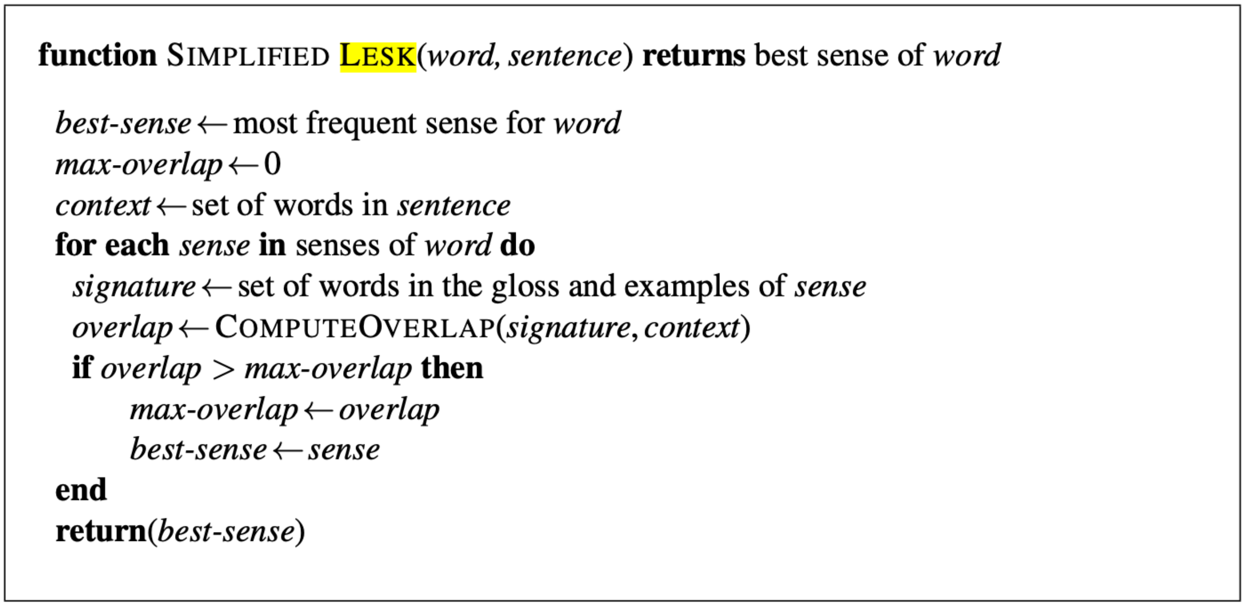
phone *a telephone connection* *the line went dead*

where "*phone*" is the sense name, "*a telephone connection*" is the sense's definition (gloss) and "*the line went dead*" is the example sentence for the phone sense.

Stopwords File

The stopwords file will contain a list of stopwords, each separated by a line break "\n".

Simplified Lesk Algorithm Details



```
function SIMPLIFIED LESK(word, sentence) returns best sense of word

  best-sense ← most frequent sense for word
  max-overlap ← 0
  context ← set of words in sentence
  for each sense in senses of word do
    signature ← set of words in the gloss and examples of sense
    overlap ← COMPUTEOVERLAP(signature, context)
    if overlap > max-overlap then
        max-overlap ← overlap
        best-sense ← sense
  end
  return(best-sense)
```

- In the simplified Lesk algorithm, stop words and words without any alphabetical letter should **not** be considered as context words or signature words. For example, "*a*" should be ignored since it is a stop word, and "." should be ignored since it does not contain any alphabetical letters.

- The algorithm pseudo-code above (originally from the textbook) identifies the sense with the maximum overlap score. For this assignment, you should compute a score for *every* sense and print a ranking of the senses based on their scores. So there is no need to know the most frequent sense for a word, as in the first line of the pseudo-code.

- The word overlap score should count the number of distinct words that overlap. So if a word occurs twice in a context, it only counts as one word. The word matching should be case-insensitive, so for example "dog" should match "Dog".

- If there is a tie in the score, you should break the tie using alphabetical order so that the sense with the smaller string value is ranked higher. For example, if the sense "*division*" receives the same overlap score as the sense "*formation*", then rank "*division*" higher than "*formation*".

Output: Your program should produce an output file with the same name as the test sentences files but ending with the suffix ".lesk" (e.g., **test.txt.lesk**). The output file should contain the sense rankings for each occurrence of the target word in the test sentences file. Each line of the output file should be a ranked list of senses for a test instance, in the order of decreasing overlap scores, from left to right. Print each sense with its score in parentheses. For example, the output for a test instance of the target word "*line*" might look like this:

text(4) phone(3) product(2) cord(0) division(0) formation(0)

where "*text*" is the highest ranked sense with an overlap score of "4", and "*formation*" is the lowest ranked sense with an overlap score of "0".

Be sure that the lines in the output file correspond to the same ordering of target word occurrences in the test sentences file! For example, the first line in the output file should have the sense rankings for the first word occurrence in the test sentences file, the second line in the output file should correspond to the second word occurrence in the test sentences file, etc.

PART 2: Nearest-neighbor Algorithm

For the second part of the assignment, you will implement **nearest-neighbor matching using distributional similarity**. This method will select the best sense for an instance of a target word in a sentence based on the cosine similarity between each sense's *signature vector* and a context vector constructed from the sentence. Your program should accept four arguments: (1) a Training Sentences file, (2) a Test Sentences file, (3) a Stopwords file, and (4) a parameter *k* for the context window size. Your program should be named "**distsim**" and should accept these arguments on the command-line in the order above. For example, we should be able to run your program like so:

```
python3 distsim.py train.txt test.txt stopwords.txt 2
```

If you use Java, the arguments should similarly be accepted on the command-line in the same order.

Training and Test Sentences Files

The Test Sentences file will have exactly the same format as in PART 1. The Training Sentences file will be similar, except that each sentence will be preceded by "GOLDSENSE:<goldsense>\t", where "goldsense" is the correct sense for the target word in the given sentence. We will refer to the gold sense and its corresponding sentence as a training instance. For example, a training instance might look like this:

GOLDSENSE:product Mr. Mottus added that Noxell may have been " spoiled by " the success of its Cover Girl <occurrence>line</> .

Stopwords File and *k* Parameter

The Stopwords file will have the same format as in PART 1.

The *k* parameter will be a non-negative integer representing the size of the context window to use around the target word occurrence. The context vector should be constructed from *k* words immediately to the left of the target word and *k* words immediately to its right. Note that there may be fewer than *k* words in the left or right context if the target word is close to the beginning or end of a sentence (or if *k* is very large).

Special case: if *k*=0, your program should use the entire sentence as the context window.

Nearest Neighbor Matching Algorithm

Training

1. You should collect all of the distinct terms that appear in the training sentences file within the context window *k* of a target word occurrence. For example, if k=2 then collect all of the words that appear within 2 words on the left side and 2 words on the right side of a target word occurrence. Words should be treated as case insensitive, so for example "dog" and "Dog" are considered to be the same term. You should then discard all terms that are stop words, have no alphabetical letters, or have a frequency count of 1. The remaining terms will be your **vocabulary (V)**.
2. You should collect all of the distinct gold senses in the training sentences file. This set will be your *sense inventory* for the target word.
3. For each gold sense s_i, you should create a **signature vector** for that sense. To create this vector, collect all of the sentences that contain instances of the target word labeled as s_i and extract the contexts around each occurrence of the target word based on the window size *k*. Then create a vector of size |V| where each position of the vector represents a word in your vocabulary. The value for each vocabulary term should be its total frequency count in the collection of contexts for sense s_i. NOTE: some words that appear in a context window may not be in your vocabulary (e.g., stopwords and words without alphabetic letters). They still count as part of the context window, but they will not be used in the signature vector.

Testing

1. Given an instance of a target word in a sentence, create a context vector for the word using window size *k*. The context vector should have size |V| and use the same vocabulary and word indices as the signature vectors. Any words in the context window that are not in the vocabulary will not be used in the signature vector. The value for a word should be its frequency in the context window.
2. Compute the cosine similarity between the context vector and each of the signature vectors for the gold senses. The sense whose signature vector has the highest cosine similarity

score is the predicted sense. If the denominator of the cosine equation is zero, then set the cosine score to be zero.

Output: Your program should produce an output file with the same name as the test sentences file but ending with the suffix ".distsim" (e.g., **test.txt.distsim**).

The first four lines of the output file should print statistics about the input files as follows:

Number of Training Sentences = #
Number of Test Sentences = #
Number of Gold Senses = #
Vocabulary Size = #

Each subsequent line of the output file should be a ranked list of senses for a test instance, printed in order of decreasing cosine similarity scores, from left to right. Print each sense with its cosine score in parentheses, with exactly 2 significant digits after the decimal point. (Don't chop off digits yourself – let the printing function do that so it will round appropriately. For example, in python use ".2f" in the print statement.)

For example, the output for an instance of the target word "*line*" might look like this:

text(0.81) phone(0.72) product(0.70) cord(0.33) division(0.15) formation(0.00)

If there is a tie, you should break the tie based on alphabetical order, so that the sense with the smaller string value is ranked higher.

Be sure that the lines in the output file correspond to the same ordering of target word occurrences in the test sentences file! For example, the first line in the output file should have the sense rankings for the first word occurrence in the test sentences file, the second line in the output file should correspond to the second word occurrence in the test sentences file, etc.

IMPORTANT: You may not use ANY external software packages or dictionaries to complete this assignment except for basic libraries. For example, libraries for general I/O handling, math functions, and regular expression matching are ok to use. NumPy is ok to use. But you may not use libraries or code from any NLP-related software packages, or external code that performs any functions specifically related to the assignment task. You must implement these algorithms from scratch, and all submitted code must be your own.

SUBMISSION INSTRUCTIONS

On CANVAS, please submit an archived (.tar) and/or gzipped (.gz) file containing:

1. The source code for your simplified Lesk and Nearest-neighbor algorithms. Be sure to include all files that we will need to compile and run your programs!

2. A README file that includes the following information:
   - how to compile and run your code
   - which CADE machine you tested your program on (this info may be useful to us if we have trouble running your program)
   - any known bugs, problems, or limitations of your program

3. On the CADE machines, run your programs using the input files in the Program #3 folder on CANVAS and submit the output files. Please use k=10 as the context window size for the nearest neighbor algorithm. Specifically, please submit:

test.txt.lesk : the output file generated by running this command (or similar in Java):

```
python3 lesk.py test.txt definitions.txt stopwords.txt
```

test.txt.distsim : the output file generated by running this command (or similar in Java):

```
python3 distsim.py train.txt test.txt stopwords.txt 10
```

Reminder: Your program must be written in Python or Java and it MUST compile and run on the linux-based CADE machines! We will not grade programs that cannot be run on the linux-based CADE machines.

GRADING CRITERIA

Your program will be graded based on new input files! So please test your program thoroughly to evaluate the generality and correctness of your code! Even if your program works perfectly on the examples that we give you, that does not guarantee that it will work perfectly on new input. Please exactly follow the formatting instructions specified in this assignment. We will deduct points if you fail to follow the specifications because it makes our job much more difficult to grade programs that do not conform to the same standards.