Natural Language Processing - Project 4
CSE 398/498-013
Prof. Sihong Xie
Due Date: 11:30 PM, Dec 15, 2017

1 Problem Statement

Implement the Lesk algorithm as described in Section 20.4 (and Figure 20.3 in particular) to disambiguate word senses in the lexical sample setting. Lesk is a dictionary-based unsupervised word sense disambiguation algorithm: given an ambiguous word (the "target word") in a sentence, Lesk compares the context of the word to the signatures of the possible senses of the word, and the sense that has the most similar signature to the context is regarded as the correct sense.

Context and signature What words shall enter the context of the target word? There are some options:

(i) All words in the sentence of the target word.

(ii) All words in the sentence with stopwords and punctuations removed.

(iii) All words in a window of size d (an odd number greater than 1) centered at the target word.

(iv) Only nouns, adjectives, adverbs and verbs in the sentence. This would require POS tagging.

You shall lower-case and lemmatize all words in all options.

For signatures, you will have all options except (iii) since there is no target word in a gloss. All sentences in a gloss are used to contruct the signature. Please see Section 20.4 for a running example. At the end, a signature/context is extracted as a set of words.

Similarity Similarity between a context and a signature, if calculated improperly/properly, can lead to junk/perfect disambiguations. Section 20.6 gives a few examples of nice/bad similarity metrics. Given two sets of words, here are your options:

- Cosine similarity. Create two binary vectors, say $\mathbf{v}$ and $\mathbf{w}$, for the two sets of words, say V and W: if a word is in V (W, resp.), then the corresponding entry in $\mathbf{v}$ ($\mathbf{w}$, resp.) is set to 1, otherwise 0. Then

$$\text{sim}_{\text{cosine}}(\mathbf{v}, \mathbf{w}) = \frac{\sum_{i=1}^{N} v_i \times w_i}{\sqrt{\sum_{i=1}^{N} v_i}\sqrt{\sum_{i=1}^{N} w_i}}, \tag{1}$$

  where N is the total number of unique words in $V \cup W$.

- Jaccard similarity. Given the binary vectors as above,

$$\text{sim}_{\text{Jaccard}}(\mathbf{v}, \mathbf{w}) = \frac{\sum_{i=1}^{N} \min\{v_i, w_i\}}{\sum_{i=1}^{N} \max\{v_i, w_i\}} \tag{2}$$

2 Exercises

- Describe the benefits of lower-casing and lemmatizing words in context and signature.
- Prove that the Jaccard similarity is the ratio of the number of common words to N.
- Prove that both similarity metrics are symmetric (e.g., $\text{sim}_{\text{cosine}}(\mathbf{v}, \mathbf{w}) = \text{sim}_{\text{cosine}}(\mathbf{w}, \mathbf{v})$).

3 Experiment

Dataset You will need quite a few files as input.

- Sentences with ambiguous words. In each file in the folder *data/semcor_ txt/*, a sentence is given in one line followed by a positive number, say s, indicating the number of ambiguous words in the sentenece. Each of the following s lines has 4 fields. For example,

  ```
  #2 further ADV further%4:02:02::,further%4:02:03::
  ```

  The number after # (e.g., *2*) indicates the location (0-based indexing) of the target word (e.g., *further*) is. The third field is the POS tag of the target word (e.g., *ADV*). The forth field is a list of WordNet sense keys (e.g., *further%4:02:03::*) concatenated by commas (,). There can be one or multiple sense keys and either way is considered as a single sense during prediction.

- WordNet dictionary. Unzipped the file *data/WNdb-3.0.tar.gz* to get the folder *data/dict/* containing the WordNet files. You shall use the third-party Java library, JWI from MIT, to query WordNet using these files. It is a nice library that just needs the path to this folder and can take care of the other details for you. I have the jar file in the project4.zip file for you.

- Stopword list. The file *data/stopwords.txt* contains a list of stopwords and punctuations, which you may (or may not) want to include in a context or a signature.

Program Design The focus of this project is WSD, so please kindly use the Stanford CoreNLP libraries for those tedious but fundamental preprocessing (such as POS tagging, tokenization, lemmatization, etc.). I have the basic jar file of CoreNLP included in project4.zip, but you will need to download the model file (stanford-corenlp-3.8.0-models.jar) from the web since it is pretty large. Don't include any jar files in your submission.

Important: your submission should be a stanalone Java program. Preprocessing using Python or other language, or calling other components from within Java is considered a failure. Use Google to figure out the libraries and how to use them (similar codes for using such libraries will not be considered plagiarisms).

I have designed the functions in the class Lesk for you to implement.

- `public Lesk()`

  The constructor initializes any WordNet/NLP tools and reads the stopwords.

- `public void readTestData(String filename)`

  Read in sentences and ambiguous words in a test corpus to fill the data structures `testCorpus, locations, targets` and `ground_truths`. The format of an input file is specified above. During testing, I will feed different test corpus files to this function.

- `private Map<String, Pair<String, Integer> >`
  `                    getSignatures(String lemma, String pos_name)`

  For a particular combination of lemma (a wordform to be exact) and a POS tag, query WordNet to find all senses and corresponding glosses.

- `private ArrayList<String> str2bow(String str)`

  Convert a String object, usually representing one or more sentences, to a bag-of-words. You may want to do tokenization, lemmatization, lower-casing here and leave the stopword removal to the `predict` function.

- `private double similarity(ArrayList<String> bag1,`
  `                    ArrayList<String> bag2, String sim_opt)`

  Computes a similarity score between two bags-of-words. Use one of the above options (cosine or Jaccard).

- `public void predict(String context_option,`
  `                    int window_size, String sim_option)`

  For each target word (from `ambiguousWords`) in each sentence (from `testCorpus`), create a context and all possible senses (using `getSignatures`) for the target. For each sense, use `similarity` to find the similarity score between the signature and context. Map each sense key to its similarity score in a `HashMap` object, which is then inserted into an `ArrayList`, which is then inserted into `this.predictions`.

  You need to allow different options for context construction and similarity metric though the arguments. Jaccard similarity using ALL_WORDS and ALL_WORDS_R is required.

- `private ArrayList<Double> evaluate(ArrayList<String> groundTruths,`
  `                HashMap<String, Double> predictions, int K)`

  Use `this.predictions` and `this.groundTruths` to generate precision, recall and F1 score at the top K positions. Note that these 3 metrics are calculated for a target word.

  `precision`=(# of correctly predicted senses with the largest K similarities) / K

  `recall`=(# of correctly predicted senses with the largest K similarities) / (# sense keys of the target)

  `f1`=(2 x precision x recall) / (precision + recall)

  A sense key predicted for a target word is considered correct if it matches any one of the sense keys in the ground truth of the target word (note that each target word can have correct multiple WordNet senses).

- `public ArrayList<Double> evaluate(int K)`

  Call the above function to calculate and then average the 3 metrics over all targets.

4 Deliverables

In the folder <your Lehigh id>_p4, put the following items:

- In the file *report_p4.pdf* answer the above exercise questions.
- A file in *results/metrics* generated by the `run_java.sh` script.

The folder should be zipped into <your Lehigh id>_p4.zip and submitted to coursesite.
Important:

- Don't include any jar files in your submission, as I can copy them to your jars folder to compile and run your codes.
- Don't submit the project from your IDE.
- Don't alter the folder names and organizations in the unzipped folder.

5 Grading

The following aspects of your submissions will be graded (15 points in total):

- Functionality: make sure I can compile your codes and produce desirable results using `build_java.sh` and `run_java.sh` (6 points). On top of that: correct Jaccard similarity (2 points), all words in the sentence as context (2 points), stopword removal in both signatures and context (1 points).
- Readibility: how heavy your codes are commented and how clear you document your algorithms and data structures (2 points).
- Speed: there will be a lot of test corpora fed into your program, so speed would be important. My naive implementation takes 15 mins to finish 300+ files (2 points).
- Bonus: implementing and evaluating window based (1 point) or POS tag based (3 points) contexts, or cosine similarity (1 points). Advanced data structures to speed up WordNet sense look-up will earn you 2 points (need to be demonstrated and substantial).