Project 2 : Word Sense Disambiguation

Kevin Li(kyl27),Ganesh S(gs563),Remya B(rb628),Yashaswini S(ys429)

Table of contents:

Supervised WSD:

In order to perform the Supervised WSD we follow the approach below:

Step 1 : Initially we create a list of Stop words which do not have context associated with them. Example:- "and","the","or", "," ,-" etc. We add this list of stop words to the Hash Set `stop_words` .

```
public void createStopWords(String filename) {

	try {
		BufferedReader bbr = new BufferedReader(new
							FileReader(filename));
		String ln = "";

		while ((ln = bbr.readLine()) != null) {
			stop_words.put(ln);
		}
		bbr.close();
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

Step 2: After we have created the list of Stop words ,we scan each line from the "training_data.data" and perform the following steps:

1. Tokenize the line using the tokenize method of `TokenizerModel class.`

```
InputStream is = new FileInputStream("en-token.bin");
TokenizerModel model = new TokenizerModel(is);
Tokenizer tokenizer = new TokenizerME(model);
String tokens[] = tokenizer.tokenize(line);
```

2. Loop through each of the generated tokens and find its root word using PorterStemmer class, and also eliminate the token if it belongs to the list of Stop Words, by checking if there is an entry for that word in the stop_words list.

```
PorterStemmer stem = new PorterStemmer();
stem.setCurrent(a);
stem.stem();
String result = stem.getCurrent();
result = result.toLowerCase();
boolean value = stop_words.containsValue(result);
```

3. If we do not find the entry for the word in the stop_words, we add it to our list of context_words for the particular target word and sense number. We also increment the frequency and its distance from the target word.

```
if (!value) {
	if (senseContext.word_list.containsKey(result)) {
		senseContext.word_list.get(result).frequency++;
		senseContext.word_list.get(result).distance_sum +=
							findDistance;

		senseContext.word_list.get(result).distance =
		(double)senseContext.word_list.get(result).distance_sum /
		(double) senseContext.word_list.get(result).frequency;
	}
```

```
        else {
                Stats stats = new Stats(1, 0, 0.0, 0.0, 0);
                stats.distance_sum = findDistance ;
                stats.distance = (double)stats.distance_sum /
                                (double) stats.frequency;
                senseContext.word_list.put(result, stats);
        }
}
```

4. We maintain the list of context words for each sense in a separate object of class SenseContext.

```
public class SenseContext {
        public int senseNumber;
        public int occurrences;
        public int lineNumber;
        HashMap<String, Stats> word_list = new HashMap<String, Stats>();
}
```

5. The class SenseContext has the list of all words for a particular SenseNumber, in a HashMap called word_list which has the context word as the Key and the Stats class as the Value, which in-turn contains all the associated details of context word, like - frequency, distance_sum , average distance , its relevance and rank.

```
        public class Stats {
                int frequency;
                int distance_sum;
                double distance;
                double relevance;
                int rank;
        }
```

Example:

Consider if the training_data has the following sentences:

- exist.v | 1 | The at least 50,000 shares %% existed %% as of Oct. 13 or
- exist.v | 2 | `` Sometimes , " says his wife , Zalubice %% exist %% in a desert of sometimes

For each of the context word and the sense number we remove the stop_words like "The","at","and" etc and also calculate it frequency ,distance from target word and average distance as shown below:

Target :exist Sense Number : 1

Context Word	Frequency	Distance Sum	Average Distance(Frequency/Distance Sum)
share	1	1	1
oct	1	3	3

Target :exist Sense Number : 2

Context Word	Frequency	Distance Sum	Average Distance(Frequency/Distance Sum)
desert	1	3	3
zalubic	1	1	1
wife	1	3	3
sometim	2	13	6.5
sai	1	5	5

Naïve Bayes

To implement the Naïve Bayes approach, we iterate through the HashMaps of tokens for both the training and test contexts, for each target word to be disambiguated. We determined the probability associated with each token based on its number of occurrences with the training contexts for the target word, divided by the number of entries for a given sense of the target word. The computation for the probabilities for each token resulted in very low values, so we stored the probabilities in log-form. The mathematical reasoning is identical to that presented in the project description.

Results

Baseline	Naïve Bayes
750 correct / 933 total	773 correct / 933 total

Extensions and Discussion

The Naïve Bayes approach makes strong use of the baseline probabilities of each target word to be disambiguated. This produces relatively strong results regardless of what features to factor into the probability calculations. Possible features to incorporate are: distance from target word, baseline token frequency, and POS.

Code for Naïve Bayes

```
// each sense context for given target word, from training data
for(int i = 0; i < trainedContextSenses.size(); i++) {
	SenseContext trainedSenseContext = trainedContextSenses.get(i);
	double prob = 0.0;

	// loop through keysets of contexts
	for(String testToken : testSenseContext.word_list.keySet()) {
		if(trainedSenseContext.word_list.containsKey(testToken)) {
			prob += Math.log(
			trainedSenseContext.word_list.get(testToken).frequency
			/ (double) trainedSenseContext.occurrences);
		} else { // add a very small prob value
			prob += Math.log(0.0001
				/ (double) trainedSenseContext.occurrences);
		}
	}
	senseProbabilities[i] =
		new SenseProb(trainedSenseContext.senseNumber, prob +=
			Math.log((double) trainedSenseContext.occurrences /
			(double) totalTrainingEntries));
}
```

Discounted Cumulative Gain (DCG)

We implemented what we consider to be a better method for finding the best sense to match an ambiguous word with multiple valid senses. This approach is called *Discounted Cumulative Gain*, or DCG. DCG takes into account Zipf's Law, which states that the frequency of a given word in a corpus is

inversely proportional to its frequency rank. The approach works by ranking each token in the context of the target word based on its relevance to the target word. The relevance for each token is a function of its average distance to the target word and its frequency in all context entries for the target word.

The resulting tokens are sorted based on their relevance, with the token at rank 1 being the most relevant to the target word.

For a given set of tokens, we compute the DCG as follows:

$$DCG = \sum_{i=1}^{N} \frac{2^{rel(i)} - 1}{\log_2(i+1)}$$

Where rel(i) is the relevance of the token with rank i.

We experimented with many different functions for determining the relevance of a given token to the target word. It is clear that the relevance function should increase with respect to the frequency of the token, and decrease with respect the average distance from the token to the target word. Eventually, the best result we obtained was with the following relevance function:

$$rel(i) = \frac{freq(i)}{dist(i)}$$

Where freq(i) is the number of occurrences of the token with rank i, and dist(i) is the average distance from the token with rank i to the target word.

We compute the DCG on the set of context tokens for each different sense of the target word, relative to the set of the context tokens in the test entry. The set of (training) context tokens that produce the highest DCG will represent the most likely sense for the target word. Note that the DCG score does not account for the baseline probability of each sense. This is would be a further extension to pursue.

Example

Suppose we have two training entries for the word “bank”, as well as the subsequent test entry:

bank.n | 1 | finance money %% bank %% storage

bank.n | 2 | river %% bank %% shore

bank | 0 | river storage %% bank %% finance account shore

The DCG computation for the sense 1 would result in a higher score. The token “finance” has a distance of 1, and a frequency of 1. The token “money” does not exist in the test context. The token “storage” also has a distance and frequency of 1.

The DCG computation for the sense 2 would result in a lower score. It also has two tokens (“river” and “shore”) that exist in the test context, but both tokens are a distance of 2 away from the target word.

Since tokens are considered more relevant if they occur closer to the target word, the sense 1 would be considered a closer match to the sense of the target word in the test entry.

Results

Using the DCG based approach, we achieved slightly better performance compared to the baseline approach of choosing the most common sense. These results were computed by using the training dataset, and testing on the validation dataset.

Baseline	DCG
750 correct / 933 total	783 correct / 933 total

Extensions and Discussion

The DCG computation takes into account that words that in the context of the target word are exponentially more related to the target word with respect to their frequency in the context. We discovered that this relationship is very accurate in predicting the relevancy for each token with respect to the target word. However, the function used to determine the relevancy score is highly subjective and is difficult to optimize. This function must be increasing with respect to the frequency of the token, and decreasing with respect to the distance from the target word. We did not account for POS tagging of the token, nor did we account for the baseline frequency of the token, both of which are very important in determining its relevance. We also did not account for the baseline probability of the most common sense for each target word. These aspects can be used to further enhance the performance of the DCG approach.

Code for DCG

```
public double dcg(HashMap<String, Stats> senseContext) {
	double dcg = 0.0;
	distance(senseContext);
	relevance(senseContext);
	rank(senseContext);

	for(String token : senseContext.keySet()) {
		if (test.containsKey(token)) {
			dcg += (Math.pow(2.0, test.get(token).relevance) - 1)
					/ (Math.log(senseContext.get(token).rank + 1)
					/ Math.log(2));
		}
	}
	return dcg;
}

public void relevance(HashMap<String, Stats> context) {
	for(Stats s : context.values()) {
		s.relevance = s.frequency / s.distance;
	}
}
```

General Discussion:

The Three most informative features that we have discovered during our analysis of Supervised WSD:
1. Frequency of each token
2.Distance to the target word
3.Baseline frequency for each token i.e how common each token is.
Another important feature which could increase the accuracy would be considering the POS of the context words .
We can also increase accuracy by using N-Gram frequencies and probabilities in the computation of DCG function instead of single token frequencies.

Example :
bank.n | 1 | finance money %% bank %% storage

bank.n | 2 | river %% bank %% shore

bank | 0 | river %% bank %% money account

If we use the original DCG algorithm, bank(1) and bank(2) would have equivalent DCG scores. Both senses have one matching token with a distance of one relative to the test entry. If we consider Bi-gram frequencies and probabilities, then for the given test data, the bi-gram "river bank" would increase the DCG score for bank(2) resulting in a better match than bank(1)

Extension 2:

In order to keep track of the different probabilities of each sense associated with each word to be disambiguated, we maintain an array of tuples of (sense number / probability) corresponding to the probability of that sense number being correct. After computing the probabilities for each of the possible senses, we sort this array and return the sense number associated with the highest probability value. By accounting for bigram frequencies and probabilities (ideally also including POS tagging), we might occasionally not return the most likely sense. For instance, suppose that the word to be disambiguated is "bank." Even if there are significantly more words relating to the financial/storage sense in the surrounding context, if the preceeding word is "river" then the financial sense of the word would clearly be incorrect. This could be accounted for by considering bigram statistics and their relationships with the different senses for each target word. For example:

bank | 0 | My father was a financial accountant, working for a large firm situated by the Thames river %% bank %% , and overlooking the water below.

Using the Naive Bayes method, or the basic DCG algorithm, the financial sense would be considered the most likely sense. However, by accounting for bigram frequencies, we can determine that the correct sense in this case must be the "river bank" sense.

The following code segment illustrates how we used an array of SenseProb, a private inner class, to maintain associations between sense numbers and probabilities. This array is then sorted, so we can return the first, second, third, etc... most likely sense(s).

```
for(int i = 0; i < trainedContextSenses.size(); i++) {
                                    ...
       for(String testToken : testSenseContext.word_list.keySet()) {
              ...
       }
       senseProbabilities[i] =
                     new SenseProb(trainedSenseContext.senseNumber, prob);
       System.out.println(prob);

}
Arrays.sort(senseProbabilities);
```

Extension 3

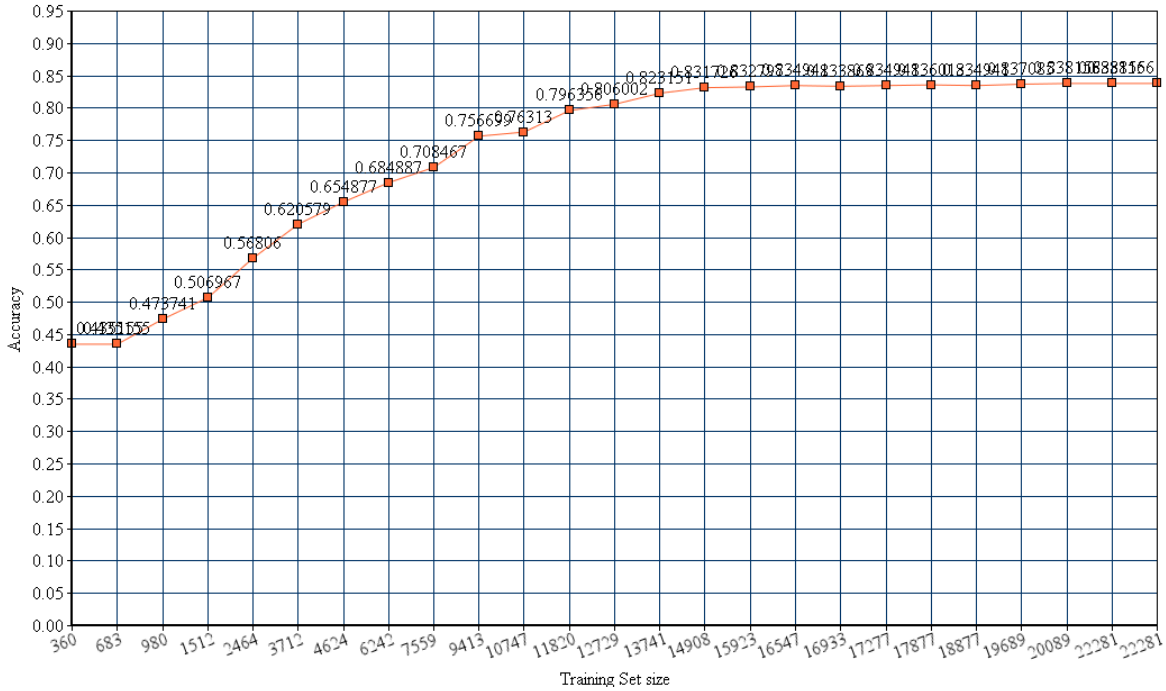
The soft scoring approach depends significantly on how the probility values are calculated/manipluated in the Naive Bayes approach. Since we stored the probabilities in log form, we represent those probabilties here in exponent form.

The average computed probability of each sense was on the order of 2^(-100) to 2^(-400). This is very reasonable, since the average test entry contains 100 - 200 words. If we assume that a given token has a conditional probability of occurrence of 50% for one sense, then the corresponding probability score of that sense would be on the order of 2^(-100) to 2^(-200).

We found that the supervised WSD approach, with DCG, resulted in better accuracy compared to the dictionary based approach. This is likely due to fundamental difference betwee the DCG model and dictionary based model.

Extension 4:

Effect of the training set size on performance.



We have plotted the graph of "Training Set Size" vs "Accuracy" by testing on validation_data.
We have increased the test size from a few hundred to a few thousand lines in training_data and found that the accuracy plateaus at around 15000 lines of training data.
We vary the training set size by limiting the number of entries for each sense of each target word. We have observed that the size of the context token list, for each sense of each target word, also tends to plateau after 15000 lines of training data. That is, the number of unique context tokens for each sense of each target word tends stabilize after 15000 lines of training data.

Softwares/Libraries used:

jwnl.jar
opennlp-maxent.jar
openenlp-tools.jar
opennlp-uima.jar
lucene-analyzers.jar
SnowballStemmer.jar
commons-lang.jar

Kevin Li(kyl27),Ganesh S(gs563),Remya B(rb628),Yashaswini S(ys429)

Kaggle Test results for SupervisedWSD

Team Name : Ganesh and Kevin
Ranking: 6th with 82.619 % accuracy.

Dashboard

Leaderboard - CS4740: 2014 Word-Sense Disambiguation (Supervised)

This leaderboard is calculated on all of the test data.

See someone using multiple accounts?
Let us know.

#	Δ6h	Team Name	Score	Entries	Last Submission UTC (Best - Last Submission)
1	-	NLPHW2S	0.85962	27	Tue, 18 Mar 2014 23:05:47 (-25.2h)
2	-	bayes	0.85477	9	Tue, 18 Mar 2014 15:47:11
3	-	Christopher Merrill	0.84278	3	Mon, 17 Mar 2014 23:15:59
4	-	team homophily	0.83589	17	Tue, 18 Mar 2014 21:33:51 (-22.4h)
5	-	meng369	0.82746	20	Tue, 18 Mar 2014 16:26:04 (-14.4h)
6	-	**Ganesh and Kevin**	**0.82619**	**1**	**Mon, 17 Mar 2014 23:01:30**
7	-	NB thats me	0.82542	3	Mon, 17 Mar 2014 23:21:59 (-2.3h)
8	-	PRANLP2	0.82210	31	Tue, 18 Mar 2014 19:16:21 (-4.4h)
9	↑1	BJF antsy	0.82185	16	Tue, 18 Mar 2014 21:35:59 (-0.3h)
10	↓1	team-name-here	0.81674	22	Tue, 18 Mar 2014 23:00:29 (-16.3h)

Dictionary Based Word Sense Disambiguation

Programming Language*: Java*

Data structures: ArrayList, HashMap, Java APIs for pattern matching, JAWS (Java API for Wordnet Searching)

Libraries used: Weka - rainbow stopwords, MorphaStemmer, Stanford POS Tagger, Apache Commons Lang

Pre-processing

Operations on dictionary:
Parse the dictionary.xml file using DOM parser and store the target word, its sense and words extracted from definitions of the target word in a hash map.

Hash Map structure:

Key	Value
Word.(pos) + Sense id Word.(pos)- Target word with it's pos Eg., complete.v	List of target word glossary, Synonyms, Hypernyms, Hyponyms/Troponyms eg., [complete, carry, out,dispatch, discharge, complete,carry, through, accomplish, execute, carry, out, action, fulfill, fulfil, put, effect,nail, play, participate, game, sport, task]

For each entry in the dictionary, only the glossary provided for the same would not suffice for comparison(as it is limited), hence we use the link to WordNet glosses and get the synsets or word forms, definitions, hypernym/hyponyms in case of noun and hypernym/troponym in case of verb.Once the above are extracted, the stop words are removed and are lemmatized.

- Considering only glossary from the dictionary for complete.v
  - Sense 1 value- [come, bring ,finish ,end]
- Considering all the links to wordnet for complete.v
  - Sense 1 values for the dictionary -[bring, finish, end, complete, finish,end, terminate, bring, end,halt,close, finish, game, baseball, protect, lead,top, top, off, finish, conclude, get, through, wrap, finish, off, mop, polish, off, clear, finish, finish,task, completely, see, through, remain, completion, round, out, finish, out, fill, follow, through, follow, follow, out, carry, out, implement, put, through, through, pursue, conclusion, bring, successful, issue, carry, through, accomplish, execute, carry, out, action, fulfill, fulfil, put, effect, write, require, information, form,complete, fill, fill, make, out]

Code snippet : Extracting from wordnet glossary, hypernym, hyponym, troponym, synonym

```
nounSynset = (NounSynset)(synsets[i]);                    //Use synset
String syn1 = nounSynset.getDefinition();
String[] syn =nounSynset.getWordForms();

NounSynset[] hypo = nounSynset.getHyponyms();    //Use homonyms for noun
NounSynset[] hyper = nounSynset.getHypernyms(); //Use hypernyms

VerbSynset[] hyper = nounSynset.getTroponym();   //Use troponym for verbs

DictMinusStopWords=RemoveStopWords(def);
DictLemmatizedWords=Lemmatize(DictMinusStopWords);
```

Add the lemmatized words to the dictionary map

```
dictionaryMap.put(wordsenseArray[entryNo], DictLemmatizedWords);
```

1.2 Operations on test data:

Read the test data line by line and predict the best sense using the below algorithm:

- For each entry, using pattern[regex] extract all the context words surrounding the target word. We add a deadbeef pattern for the target word since we would remove the punctuations in the later stages.

  Snippet:

```
Pattern p = Pattern.compile("^(\\w+\\.\\w)\\s+\\|\\s+(\\d+)\\s+\\|\\s+(.*)");
//Retrieve the context words surrounding the target word
Pattern pContext = Pattern.compile("(.*)\\%%(.*)\\%%(.*)");
String tw = "deadbeef";
```

- For all the words in the example/context remove the stop words(words that are of less meaning) - like a , an ,the,of,when. We have used weka's rainbow stop word list. Also we remove punctuations, $,``and some extra words which will be introduced during POS tagging. We do all of the below steps to improve content overlap between the dictionary definition values and the context words.

  Snippet:

```
if(!weka.core.Stopwords.isStopword(token))
{
        String[] words =  token.replaceAll("(\\p{P})|(\\``)| (\\d+)|(\\$)|(Verb|Noun|Adverb|
                                Adjective)", " ").toLowerCase().split("\\s+");
        for(String word1:words){
                if(!word1.matches("^(\\w)||^(\\s)||^(\\w\\w)"))
                        //Remove a single or two words
                        ContextWords1.add(word1);
        }
}
```

- Lemmatize the remaining words, we chose this approach better that stemming as it gives better results post lemmatizing.

  eg instigated -> instigate ;working -> work ; sold ->sell

  Snippet:

```
for(String word:ContextWords)
{
        new MorphaStemmer();
        String morpha = MorphaStemmer.morpha(word,false);
        ContextWords1.add(morpha);
}
```

- POS Tag the words so that we can lookup the synonyms/hyper/hyponyms for the same from Wordnet.

  eg: sell -> sell_v, plant -> plant_n

  Snippet:

```
for(String token:ContextWord)
{
        List.add(tagger.tagString(token));
}
```

- Use a window size of 5 -10 to pick up words surrounding the target word and using wordNet glosses and POS tagging add definition/synonyms/hypernyms/hyponyms/troponyms (for noun/verb) for each context word. This will improve more chances of overlap between the context and target defintion from the dictionary.

  eg :

  Snippet:

```
if(type == SynsetType.NOUN)
{
        nounSynset = (NounSynset)(synsets[i]);
```

```
                String syn1 = nounSynset.getDefinition();
                String[] syn =nounSynset.getWordForms();
                NounSynset[] hypo = nounSynset.getHyponyms();
                NounSynset[] hyper = nounSynset.getHypernyms();
        }
        If(type == SynsetType.VERB)
        {
                verbSynset = (VerbSynset)(synsets[i]);
                String syn1 = verbSynset.getDefinition();
                String[] syn =verbSynset.getWordForms();
                 VerbSynset[] hypo = verbSynset.getTroponyms();
                VerbSynset[] hyper = verbSynset.getHypernyms();
        }
        If(type == SynsetType.ADJECTIVE)
        {
                 adjectiveSynset = (AdjectiveSynset)(synsets[i]);
                String syn1 = adjectiveSynset.getDefinition();
        }
        If(type == SynsetType.ADVERB)
        {
                adverbSynset = (AdverbSynset)(synsets[i]);
                String syn1 = adverbSynset.getDefinition();
                def.add(syn1);
        }
        else
                 def.add((synsets[i]).getDefinition());
```

- Repeat the procedure of removing the stop words and lemmatizing on the context words list which we obtained above.

1.3 Find the overlapping words and predict the best sense:

- Construct an array list from the dictionary items retrieved the hash map for each sense, and compare the context list being passed for each entry in the test data.

- Eg:
  Target word list- [interconnect, system, thing, people, system, interconnect, social, entity, people, organization]

  Context word list- [rule, principle, condition, customarily, govern, behavior, regard, normative, prescribe, guide, conduct, action, linguistics, rule, describe, prescribe, linguistic, practice, basic, generalization, accept, true, basis, reason, conduct, rule, law, natural, phenomenon, function, complex, system, duration, monarch, government, power, dominance, power, legal, authority, direction, define, game, sport, conduct, systematic, body, regulation, define, life, member, religious, order, mathematics, standard, procedure, solve, class, mathematical, problem, measure, stick, consist, strip, wood, metal, plastic, straight, edge, draw, straight, line, measure, length, current, occur, belong, present, time, syndication, organize, administer, syndicate, sell, article, cartoon, publication, magazine, newspaper, time, fare, agenda, thing, sum, charge, ride, public, conveyance, pay, taxi, passenger, food, drink, regularly, consume, own, series, similar, thing, order, happen, serialize, set, program, periodical, appear, schedule, time, sport, contest, play, successively, team, group, postage, stamp, common, theme, group, coin, currency, select, group, study, collection, mathematics, sum, finite, infinite, sequence, expression, electronics, connection, component, manner, current, flow]
  Intersection list- [system, thing]

  Snippet:

```
List<String> DictList = new ArrayList<String>();
DictList=dictionaryMap.get(wordsensetemp);
IntersectList=intersect(DictList,Context);
List<String> rtnList = new LinkedList<>();
for(String dto : A)
{
    if(B.contains(dto))
    {
        if(!rtnList.contains(dto))
            rtnList.add(dto);
    }
}
return rtnList;
```

Handling consecutive overlaps

If there are consecutive overlaps in the gloss of the target word and the gloss of the target word, they need to be assigned higher weights than the individual overlaps.
How we handle this is:

- We first retrieve the list of overlapping words
- We form consecutive bigrams from the list of overlapping words. We store the bigram as a string
- We covert both target word gloss list and context wordss' gloss list to strings
- We check if the consecutive bigram is a substring of both the target word and context word lists
- If yes, there is a consecutive overlap
- If there is a consecutive overlap, the weight assigned will be (2^2). Otherwise the weight assigned will be 2.

- Eg:
  network.n | 2 | `` They said , ` follow CNN , ' '' he told reporters . That shows how far Ted Turner 's Cable News Network has come since its birth nine years ago , when it was considered the laughingstock of television news . It is bigger , faster and more profitable than the news divisions of any of the three major broadcast %% networks %% . Its niche as the `` network of record '' during major crises draws elite audiences around the world . But for all its success , CNN has hit a plateau .

  Target word list
  [open, fabric, string, rope, wire, weave, regular, interval, mesh, like, fabric, string, substance]

  Context word list
  [broadcast, message, transmit, radio, television, radio, television, show, major, greater, importance, stature, rank, greater, scope, effect, greater, number, size, amount, field, academic, study, concentrate, specialize, scale, mode, greater, seriousness, danger, full, legal, age, division, army, unit, large, sustain, combat, portion, regard, divide, constitute, act, process, divide, administrative, unit, government, business, arithmetic, operation, inverse, multiplication, quotient, number, compute, discord, splits, group, league, interval, rank, string, quality, biology, group, organism, form, open, subdivision, larger, category, botany, taxonomic, unit, plant, phylum, weave,unit, unite, state, air, force, comprise, wing, group, ship, similar, type, act, divide, partition, separation, creation, boundary, divide, niche, position, suit, person, occupy, small, concavity, enclosure, set, back, indent, ecology, status, organism, environment, community, affect, survival, species, network, interconnect, system, thing, people, broadcast, communication, system, consist, group, broadcast, station, transmit, program, fabric, wire, regular, system, intersect, line, rope, channel, electronics, system, interconnect, mesh, like, electronic, component, circuit, record, such, document, phonograph, record, photograph, provide, permanent, evidence, information, past, event, number, win, versus, loss, tie, team, extreme, attainment, worst, performance, attest, sport, sound, record, consist, disk, continuous, groove, reproduce, music, rotate, phonograph, needle, track, groove, sum, recognize, accomplishment, list, crime, accuse, person, previously, convict, compilation, fact, document, serve, legal, evidence, transaction]

  The intersection list
  [open, fabric, string, rope, wire, weave, regular, interval, mesh, like]

  Weight assigned to the sense- 12
  In the example, there is a consecutive overlap in the case of [mesh, like]. We weight this overlap as 4.

  Snippet:

```
size=IntersectList.size();
for(int j=0; j<size-1; j++){
```

```
StringBuilder CompareListString = new StringBuilder();
CompareListString1=null;
compareList.clear();
CompareListString.append(IntersectList.get(j));
CompareListString.append(IntersectList.get(j+1));
CompareListString1=CompareListString.toString();
        if(DictListString1.contains(CompareListString1) &&
ContextString1.contains(CompareListString1)){
        consecutiveCount=consecutiveCount*2;
}
weightMap.put(i, IntersectList.size()+consecutiveCount);
```

EXTENSIONS:

Extention1 - Experiments- Results and Kaggle accuracy:

Experiment - Features used	**Accuracy on validation file(if any)**	**Kaggle Accuracy (Test data)**	**Observation**
Window size - 6 [3+3] Dictionary - Synonyms+glossary Context word - Synonyms+ POS tagged (noun+verb) after choosing window size	4/9 [validationdata had one verb and one noun]	47%	Found that nouns's senses were predicted more accurately by looking at the validation results
Window Size -6 [3+3] Dictionary - Synonym+glossary hypernym+ hypo/troponym[N/V] Context word - Synonyms+ POS tagged (noun+verb)	9/16[validationdata had one verb and one noun]	49%	We realized the dictionary values had lesser words to compare so we increased scope by adding hyper/hyponyms.
Window size -6 Dictionary - Synonym+glossary hypernym+hypo/troponym[N/V] Context word -Synonym+glossary hypernym+ hypo/troponym[N/V] POS tagging done before choosing the window	10/16[validationdata had one verb and one noun]	50.5%	Found that POS tagging if it was done before choosing the context words in the window, chose a better POS as it had the complete sentence for understanding the POS better. Extending the scope of context words [hyper/hyponyms]also helped better the overlap.
Window size -7 Dictionary - Synonym+glossary hypernym+hypo/troponym[N/V] Context word -Synonym+glossary hypernym+ hypo/troponym[N/V] POS tagging done before choosing the window		50.9%	
Window size -10 [5+5] Dictionary - Synonym+glossary hypernym+ hypo/troponym[N/V] Context word -Synonym+glossary hypernym+ hypo/troponym[N/V] POS tagging done before choosing the window		51%	Window size 10 gave us the best results so far.
Window size -10 [5+5] Dictionary - Synonym+glossary		52%	more weight for overlap, better accuracy

hypernym+ hypo/troponym[N/V] and their defintions + meronyms Context word -Synonym+glossary hypernym+ hypo/troponym[N/V] + entailment+usage POS tagging done before choosing the window Added weigths to distinguish overlap words (more weight for overlap)			
Window size -4[2+2] Dictionary - Synonym+glossary hypernym+ hypo/troponym[N/V] and their defintions + meronyms Context word -Synonym+glossary hypernym+ hypo/troponym[N/V] + entailment+usage POS tagging done before choosing the window		55%	Adding hyper/hypo/troponyms' definition along with meronyms to the dictionary values increased our overlap and this by war is the best accuracy that we obtained

Extension-2:

Need for second or third best senses

We might want to keep track of the second best weight in the following scenario:

- The context word set and the target word set is large for Sense 1, which increases the probability of large number of overlaps
- The context word set and the target word set is relatively small for sense 2
- Even with a small set of words to check, Sense 2's weight is close to sense 1's weight, we might want to consider sense 2 as the next best sense for the word.

When we do not consider the POS of the target word during word sense disambiguation, we can keep track of the second and third best sense as well.

- Eg: Target word is a verb
- We consider all the context words irrespective of their POS
- Though the overlap score for the target word is highest with a noun context word, we will store the second sense of the target word which has the highest overlap with a context word that is a verb

When we want to implement a voting method that votes for each sense of the target word based on the confidence that the model assigns to the sense. We will want to keep track of all the senses and their probabilities.

How to implement?

Implementation 1:

The naïve method of doing this is considering the sense with the next highest weight or equal weight as the best sense, to be the next best sense.

Implementation 2:

- We maintain a map of all the senses and their relative weights with respect to the size of definition word set.
  We calculate for each sense,
  (the number of overlaps)/(the number of words in the definition of the context word)
- Whichever sense gets the highest value, we store as the second best sense

Code Snippet

```
float value=((IntersectList.size()+consecutiveCount)/Integer.parseInt(DictList.get(i)));
secondSensePredictionMap.put(i,value);
```

```
for (Map.Entry<Integer, Float> entry : secondSensePredictionMap.entrySet())
{
        if (sense2 == null || entry.getValue().compareTo(maxEntry2.getValue()) > 0)  {
                sense2 = entry;
        }
}
```

Sense2 is the second best sense

Example:

network.n | 2 | CBS Inc. is cutting `` The Pat Sajak Show '' down to one hour from its current 90 minutes . CBS insisted the move was n't a setback for the program , which is the %% network %% 's first entry into the late-night talk show format since 1972 . `` I have every intention of making this the best possible show and having it run one hour is the best way to it , '' said Rod Perth , who was named vice president of late night entertainment in August . `` This will raise the energy level of the show . ''

Overlapping words for the four senses are as follows:
[system, thing]
[system, group, program]
[system, line, electronics, component]
[]

Though sense 3 got the highest number of overlaps, the ratio of the number of overlaps to the number of words in the target list was higher for sense 2 than sense3. Sense 3 had a large set of target list and context words to overlap. Hence it got a higher number. But the correct sense is 2.

Conclusion

We did observe that it improved the prediction accuracy in certain examples. But the overall results were better with the first best sense

Only with first best sense	Accuracy- 54.3233%
Considering the second best sense	Accuracy- 31.4215%

Extension 3:

Soft scoring approach:

For the soft scoring approach, we normalize the weights of all the senses and then determine the score based on the probability value assigned to the sense. We ran the same on the validation file. We get the correct sense of word for each example in the validation file and assign it the probability value we have calculated. And we calculate the average probability and accuracy.

Experiment	**Average accuracy**
Retaining repeated words from the lists	34.054604 %
Considering only distinct words in both the lists and removing the repeated words + Not considering consecutive overlaps	40.01196 %
Considering only distinct words in both the lists and removing the repeated words + Considering consecutive overlaps	40.227272 %

```
for (Integer value : weightMap.values())
{
        total = total + value;
}
weight=weightMap.get(Integer.parseInt(correctSense));
float avg=(float)weight/(float)total;
probabilityScore.add(avg);

for (Float value : probabilityScore) {
        probabilityTotal = probabilityTotal + value;
```

```
}
accuracy=(float)probabilityTotal/(float)noOfLines * 100;
```

Approach	Average accuracy
Soft scoring approach on validation set	40.227272 %
Hard scoring approach on validation set	72.72727 %

Which approach performs better?

Hard Scoring approach gives us better average accuracy values than Soft scoring approach. We can conclude :

- We can use hard scoring approach when we want a conclusive accuracy value that tells us what percentage of the sense predictions were correct
- We can use soft scoring approach we do not need to check for the precision but just want to know the overall accuracy value

Experiments:

- We ran our model on the entire training set and got 54.3433% accuracy on all the examples**.**
- We first implemented the simple Lesk Dictionary based word sense disambiguation approach to overlap the definition of the target word with the definitions of the context word.
- We extended this to include the wordnet definitions
- We used lemmatizer and stemmer for the words
- We then added hypernyms, hyponyms, meronyms
- We assigned more weight to consecutive overlaps
- We tried with/without repeating words in the context word list and the target word list.
- We have presented the summary of all our experiments in the form of a table.

List of functions and their functionalities in the code

class WordSense	To hold the word and sense of the object
function GetWord()	To fetch the word from the WordSense Object
function GetWord()	To fetch the sense from the WordSense Object
class dictionaryHash	
function dictionaryHash(String dictionaryPath)	• Parses the XML file and extracts the word and all of its senses • Creates a hash with WordSense object(word+sense) as the key and the lemmatised definition word list as the value
Function readData(String fileName)	• Reads the test file one test example at a time • Calls RemoveStopWords() to remove the stop words • Calls patternMatch() to find the context words within the window size specified and calls lemmatise() function to lemmatize them • Calls posTag to assign POS tags to the context words • Call findContext() to find the context of the target word in the example
Function findContext(String testWord, List<String> Context)	• Gets the definition list of the target word • Calls the function intersect() to get the overlap between the target word definition list and the context list. • Checks for consecutive overlaps • Assigns weights to each sense based on the number of overlaps and number of consecutive

	overlaps
Function intersect(List<String> A, List<String> B)	Finds the intersection of the two lists A and B
Function patternMatch(String sCurrentLine)	Extracts the target word, the list of words that precede the target word and the list of words that follow the target word, using regex
Function removeStopWords(List<String> ContextWords)	Removes all the stop words from the given list
Function lemmatize(List<String>ContextWords	Lemmatizes the list of words given
Function POSTag(List<String> ContextWord)	POS tags the list of words
Function returnContextWords(List<String> ContextWord)	Returns the context words inside the specified window size
Function addDefntoContextWord(List<String> ContextWord)	Adding hypernyms, synonyms, hyponyms, meronyms, etc of the context words to increase the probability of the overlap

Kaggle Screenshot for Dictionary Based WSD: user YR