Trust Modelling in Dynamic Environments

Yue Xia
20705952
Yu Su
20502755

***Abstract*—Word sense disambiguation is an important area of the machine learning field. In this project, we implemented a semi-supervised machine learning algorithm - Yarowsky algorithm - to determine the meaning of a word in different contexts in both Scala and PySpark using distributed computing techniques.**

I. Introduction

A fundamental characteristic of language is that words can have more than one distinct meaning. The lexical ambiguity is especially obvious to anyone who gets the joke when they hear a pun. One thing worth to mention is that the most frequently used 121 English nouns account for about 20% occurrences in real text, have on average 7.8 meanings each [1]. However, for anyone who can fluently use English, the potential for ambiguous readings tends to be completely unnoticeable. This fact demonstrates that even though words have multiple meanings in principle, there is very little ambiguity to a real person when reading an actual text.

Step into the time of information explosion, nowadays people are trying to make machines "smart" enough to do anything for human beings. Making the machines to determine the meaning of every word in context is not a trivial task, computer scientists and software engineers thus came up with different algorithms to realize this goal. This type of problem in computational linguistics filed is called word sense disambiguation (WSD), which is essentially a task of classification. In this classification process, word senses are classes, the context provides the evidence, and each occurrence of a word is assigned to one or more classes based on the evidence [2]. Some common algorithms that currently used to handle WSD problem are including HyperLex algorithm, Lesk algorithm, Yarowsky algorithm, etc. In this project, we will focus on explaining (section II), implementing (section III) and executing (section IV) Yarowsky algorithm. We will implement Yarowsky algorithm in Spark with both Scala (for CS651) and Python (for CS631) drivers. Our implementation and the data to test the algorithm can be found in GitHub https://github.com/aixeuy/DI-C-project, and will be discussed in detail in section IV.

II. Original Yarowsky algorithm

The original Yarowsky algorithm is a semi-supervised learning algorithm for WSD built by Professor David Yarowsky from University of Pennsylvania. He claims that this algorithm rivals the performance of supervised techniques that require hand annotations when trained on unannotated English text. This algorithm is based on two properties of human language:

1) One sense per collocation: some collocations are excellent features of the context that can be strong predictors for one sense or another.

2) One Sense Per Discourse: ambiguous tokens of the same type tend to be correlated within a document (i.e. discourse)

Moreover, as mentioned above, since a small proportion of English nouns are frequently used and account for about one fifth of occurrences in real text, we can at least conclude that English language is highly redundant. Therefore, the sense of a word is effectively overdetermined by 1) and 2) above. Section II.A and II.B discuss in detail how the two properties help to build Yarowsky Algorithm, section II.C explains the original algorithm step by step.

A. *One sense per collocation*

Prof. Yarowsky observed and quantified the strong tendency that words will exhibit only one sense in a given collocation in his work in 1993. The distance and adjacency of collocation, the predicate-argument relationship, and the burstiness of the words, all affect the influence of collocation. This property of language is highly reliable and is extremely useful for WSD problems. Based on this property, a supervised algorithm called decision-list algorithm was built in 1994, and it is used as a component of Yarowsky algorithm.

A decision list is an ordered set of rules with form that if some object has a particular feature f, then predict class label k. The rules can be ordered based on weights assigned. The weights are evaluated using log-likelihood ratio $\text{Log}(\frac{Pr(Sense\_A|Collocation\_i)}{Pr(Sense\_B|Collocation\_i)})$, and we apply the rule with the highest weight. Note that these weights are the parameters of the classifier that to be learned by training algorithm. The Yarowsky algorithm is self-trained with decision list base classifier, and we will discuss in detail in section II.C.

B. *One sense per discourse*

Observed by Gale, Church and Yarowsky, words tend to exhibit only one sense in a given document (or discourse), which means we can take advantage of this regularity in conjunction for each word. This property is weaker than the collocation property mentioned above since it can be overridden when local evidence is strong, thus should not be used as a hard constraint. Prof. Yarowsky tested the one sense per discourse hypothesis on a set of 37,232 examples over a 3-year period for the claim's accuracy and applicability. Accuracy measures how often a word takes on the majority sense for a discourse when it occurs more than once in this discourse. Applicability measures how often a word occurs more than once in a discourse. The following table demonstrates that the claim of one sense per discourse holds for at least these 10 words with high reliability[3].

p(more than one occurrence)

p(most frequent sense | more than one occurrence)

Word	Senses	Accuracy	Applicblty
plant	living/factory	99.8 %	72.8 %
tank	vehicle/contnr	99.6 %	50.5 %
poach	steal/boil	100.0 %	44.4 %
palm	tree/hand	99.8 %	38.5 %
axes	grid/tools	100.0 %	35.5 %
sake	benefit/drink	100.0 %	33.7 %
bass	fish/music	100.0 %	58.8 %
space	volume/outer	99.2 %	67.7 %
motion	legal/physical	99.9 %	49.8 %
crane	bird/machine	100.0 %	49.1 %
Average		**99.8 %**	**50.1 %**

C. Original Yarowsky Algorithm

In the real world, instead of occurring in one collocation that indicates the sense, a word tend to occur in multiple such collocations. Beginning with a small set of seed examples that representing different meanings of a word, one can expand the seed examples using the two properties of language mentioned above with additional data. Steps of the original Yarowsky algorithm are as following:

Step1:

Identify all occurrences of the given polysemous word from the text corpus and store their context as lines in an untagged training set.

Step2:

Identify (can hand-tag) a relatively small number of training examples to represent each sense of the word. There are several strategies that require minimal or no human participation in identifying seeds, including 1) use a single defining collocate for each class, 2) use words in dictionary definitions and 3) label salient corpus collocations. We use these training examples as seeds and call it the initial state (see Figure 1).

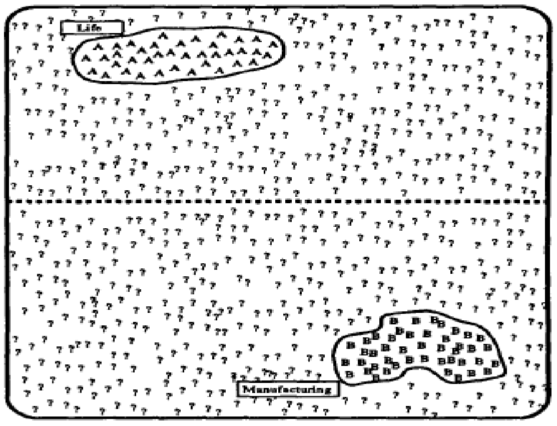

Figure 1: Sample Initial State

A = SENSE-A training example
B = SENSE-B training example
? = currently unclassified training example
Life = Set of training examples containing the collocation "life".

Setp3:

Train a supervised classifier on the labeled examples from step 2. As mentioned in earlier sections, the Yarowsky algorithm uses decision list as the base classifier, which is ranked by the purity of the distribution. Apply the resulting classifier to the entire sample set, i.e. label all examples, and add the labels where the supervised classifier was highly confident, i.e. labels above threshold, to the growing seed sets (see Figure 2).

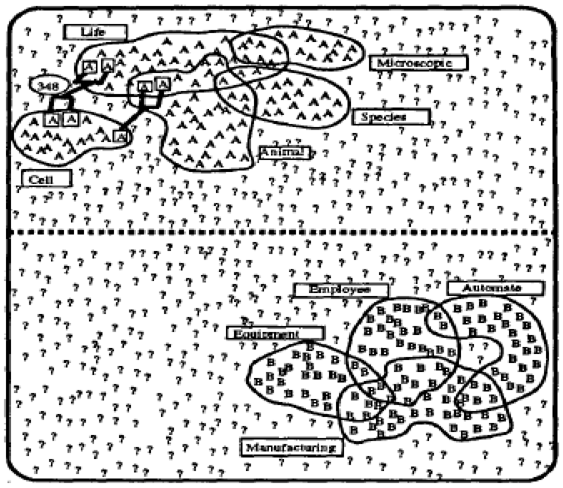

Figure 2: Sample Intermediate State (following Steps 3b and 3c)

Step4:

When the training parameters are held constant and the algorithm converges to a stable residual set, stop. Most training examples will exhibit multiple collocations indicative the same sense as shown in Figure 3, however, the decision list algorithm ensures that we only take the most reliable piece of evidence instead of a combination of all matching collocations. This strategy effectively avoided many problems and conflicts.

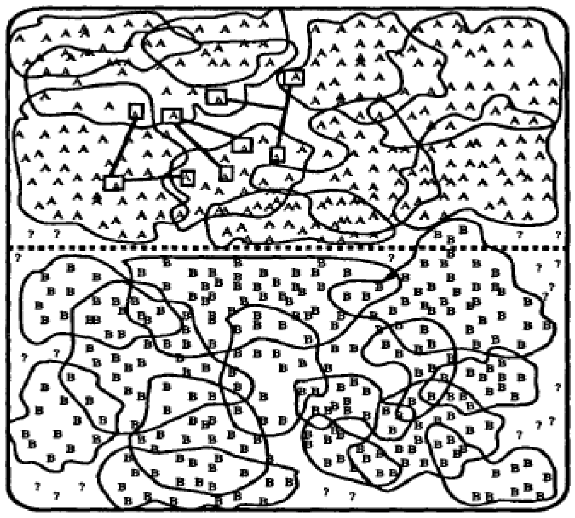

Figure 3: Sample Final State

Setp5:

Use the previously defined optimal classifier to annotate the original untagged corpus with sense tags and probabilities or to classify to new data[5].

III. Algorithm and Implementation

A. Algorithm

As mentioned before, the Yarowsky algorithm is a semi-supervised learning algorithm that determines the sense of a chosen word in different contaxes. It inputs a set of sentences. All of them contain the target word which has multiple meanings. Some of the inputs are initially labelled. The algorithm iteratively trains classifier on labelled data and labels the rest if the confidence is high. Finally, all sentences in the input will be classified.

Ideally, there is nothing that prevents the Yarowsky algorithm from classifying more then two senses of a word. But the core approach to distribute the computation in the implementation would be the same. Thus for simplicity, we only consider an easier problem: classification of two senses of a word. The details of our Yarowsky's Algorithm are shown in Algorithm I. It takes the following inputs:

- $S_{classified}$: A set of sentences used as the seeds in the semi-supervised learning. Their labels are known.
- $S_{unclassified}$: A set of sentences used in training whose labels are unknown.
- n: The number of words in n-gram.
- m: Number of n-grams; To avoid overfitting, we select the m best n-grams according to entropy based information gain.
- $threshold$: The threshold to label unclassified sentences.
- $numIter$: The maximum number of iterations in training each model.
- α: Step size for training.
- δ: The stop condition for training a model. Stop training when gradient change is smaller than δ

Algorithm 1 Yarowsky

```
Require: $S_{classified}$, $S_{unclassified}$, $n$, $m$, $threshold$, $numIter$, $\alpha$, $\delta$
1: $count$ = infinity
2: $countNew$ = $S_{unclassified}$.size
3: $F$ = null
4: $model$ = null
5: while $countNew \neq 0$ and $countNew \neq count$ do
6:     $F$ = extractNGramFeatures($S_{classified}$, $n$, $m$)
7:     $FS_{classified}$=convertToFeatures($S_{classified}$,$F$)
8:     $model$ = train($FS_{classified}$, $numIter$, $\alpha$, $\delta$)
9:     $S_{unclassifiedNew}$ = {}
10:    for $s$ in $S_{unclassified}$ do
11:        $score$ = classify(convertToFeatures($s$,$F$),$model$)
12:        if abs($score$) < $threshold$ then
13:            $S_{classifiedNew}$.add($s$)
14:        else
15:            if $score > 0$ then
16:                $s$.label = '+'
17:                $S_{classified}$.add($s$)
18:            else
19:                $s$.label = '-'
20:                $S_{classified}$.add($s$)
21:            end if
22:        end if
23:    end for
24:    $S_{unclassified}$ = $S_{unclassifiedNew}$
25:    $count$ = $countNew$
26:    $countNew$ = $S_{unclassified}$.size
27: end while
28: write($F$, $model$)
```

The function extractNGramFeatures on line 5 extracts n-grams from the labelled sentences and then selects m n-gram features with highest entropy based information gain. The information gain for an n-gram feature f is calculated by:

$$IG(S,f) = Entropy(S) - \frac{|S_f|}{|S|}Entropy(S_f) - \frac{|S_{-f}|}{|S|}Entropy(S_{-f}) \tag{1}$$

$$Entropy(S) = -\sum_{j\in\{+,-\}} p_j log_2 p_j \tag{2}$$

Where S is a set of sentences with labels. S_f denotes the subset of S consisting of sentences containing the n-gram f. While S_{-f} are the sentences where f is not present. p_j in the second equation denotes the probability of a sentence in S being labelled as j (+ or - in this case).

The function convertToFeatures just converts sentences to their n-gram features representation using the selected to m n-grams extracted in F.

On line 8, the train function trains a model with the labelled sentences using gradient descent with a fixed step size. It takes the following inputs and the algorithm is shown in Algorithm 2.

- FS: The training set. Each example consists of n-gram features and labels.
- $numIter$: The maximum number of iterations.
- α: Step size.
- δ: Threshold for the stop condition. Stop training when gradient change is smaller than δ

Algorithm 2 Train

```
Require: $FS$, $S_{unclassified}$, $numIter$, $\alpha$, $\delta$
 1: $model$=map()
 2: for $i = 1$ to $numIter$ do
 3:     $gradient$ = map()
 4:     for $s \in$ FS do
 5:         $score$ = classify($s$, $model$)
 6:         $prob$ = 1 / (1 + exp(-$score$))
 7:         $lb$ = if($s$.label=='+') 1 else 0
 8:         for $f \in s$ do
 9:             $v$ = $gradient$.getOr0($f$) + ($lb$ - $prob$) $\times\alpha$
10:             $gradient$.set($f$, $v$)
11:         end for
12:     end for
13:     $model$ += $gradient$
14:     if sum($gradient$) < $delta$ then
15:         break
16:     end if
17: end for
18: return $model$
```

The classify function in this algorithm as well as Algorithm 1 calculates and returns the classification score of a sentence. It does the same as the spamminess function provided in the assignment.

B. Implementation

We provide the distributed implementation of the Yarowsky Algorithm in both Scala and PySpark. Our implementations follows the algorithm in general, but lots of optimizations are made that parallelize the computation.

The easiest parallelization occurs on line 7 in Algorithm 1. The convertToFeature function just generates the feature representations for the sentences in parallel. Each worker processes a partition of the set of sentences. It is a simple map job though a broadcast variable F which stores the selected features is used. To have better memory usage, each selected n-gram is associated with a unique number. So the feature representation of a sentence is a list of numbers.

Similarly, the for loop from line 10 to 23 in Algorithm 1 can also be parallelized. The sentences are distributed across servers and the servers calculates their scores in parallel. Then a filter can separate out the sentences to be labelled whose scores are large than threshold in magnitude. After, a map can convert scores to labels. See our code for detailed implementation.

The function extractNGramFeatures requires the calculation of entropy which needs the numbers of sentences containing each n-gram with both labels. For example we want the number of sentences labelled as '+' containing 'a' and the number of sentences labelled as '-' containing 'a'. Such numbers are used in calculating p_j for $j \in \{+,-\}$ in Equation 2. To obtain them, the following procedure is used. The input S is a set of sentences with labels.

Algorithm 3 ExtractNGramFeatures

```
Require: $S$
1: $Positives$ = $S$.filter(label is '+')
2: $Negatives$ = $S$.filter(label is '-')
3: $S$.flatMap($s$=>uniqueNGrams($s$).map($f$=>(($f$,$s$.label),1))))
4:  .reduceByKey(_+_)
5:  .map($t$=>(t._1._1, if t._1._2 == '+' t._2 else 0, if t._1._2
    == '-' t._2 else 0))
6:  .reduceByKey(pairwise add)
7:  ...
```

In line 3, for each n-gram in the sentence, the program eimit a key-value pair, the key is the n-gram together with the label of the sentence. The value is 1. Then the reduceByKey on line 4 will gives us the count of sentences containing an n-gram with both labels. Line 5 and 6 bring together the counts of both labels. After line 6, each tuple is of the form: (n-gram, count of +, count of -).

Another place to perform parallelization is the train function. Since it uses gradient descent (not stochastic gradient descent that needs sequential execution), we can use a flatMap on the input examples. For each sentence, multiple tuples are emitted, each one corresponding to a feature of the sentence. The key of a tuple is a feature and the value is the partial gradient produced by that single sentence. By summing up the partial gradients produced by all sentences using a reduceByKey, we obtain the total gradient in an iteration over all sentences.

C. Additional Functionalities

Beside the Yarowsky's algorithm, we also implemented other functionalities. The first one is the implementation of parameters search which helps find the optimal hyper parameters:

- n: Size of n-gram.
- m: Size of feature set. To avoid overfitting, we select the m best n-grams with highest information gain.
- $threshold$: The threshold to label unclassified sentences.
- $numIter$: The maximum number of iterations in training each model.
- α: Step size for training.
- δ: The stop condition for training a model. Stop training when gradient change is smaller than δ

The user should provide a range of parameters for the search as well as a step size. Then our program will iteratively try all possible combinations of hyper parameters in the given range. A set of labeled sentences will be separated from the training set. The Yarowsky's algorithm will run on the training set with different combinations of hyper parameters and the resulting model will be tested with the testing set. The parameters with highest testing accuracy will be returned.

Another functionality is to classify an arbitrary sentence. The user is allowed to input a sentence and have the program load a trained model and use it to classify the sentence.

D. Testing Data

The Yarowsky algorithm is tested on a data set containing more than 1500 sentences mostly collected from python's nltk.corpus. The examination of our implementations focuses on the word sense disambiguation of the word 'bank'. Two senses of bank are distinguished. They are the financial bank and the river bank. All collected sentences contain the word 'bank'. And among them, the nltk.corpus package provides senses of bank for around 20 sentences which is too few even for testing. Thus, more sentences that contain 'bank' from Wikipedia's pages about financial bank and river bank are added to our data set. They are also labelled with the assumption that words in the financial bank page refer to financial bank and vice versa. There are about 50 of them. To expand our data set, we could also do more web crawling to extract more sentences containing 'bank'. But we ran out of disk quota by doing so.

The data set we use is stored in a text file while each line represents an example. A line has the form of a sentence containing 'bank' followed by a label. The label is '+' if 'bank' in the sentence has the meaning financial bank and is '-' if it has the other meaning. For sentences where the sense is unclassified, their labels are '0'. There is no separator between the sentence and its label in each line. With correct hyper parameters choose, the algorithm should easily give an accuracy above 0.9. The execution time for the algorithm depends on how many iterations are run. But the time should be within several minutes in training mod. When doing parameters search, it may take much longer depending on the search range.

IV. Execution Instruction

A. Scala

Here we introduce how run the scala program in a linux like system. The instructions can also be found in README.md

After compiling the program by:

```
# mvn clean package
```

We can run the program by:

```
# park-submit --class
ca.uwaterloo.cs451.yarowski.Yarowsky
target/assignments-1.0.jar ...
```

The program takes the following input arguments:

- input: path of training data set. Default: data/bank_final.txt.
- output: path of the classification results of input sentences. In training mod, the program will classify the sentences in the training set and write the results to this path. Default: output.
- model: path of the model. In training mod, will write model to this path. In testing mod, read model from this path. Default: model.
- n_start, n_end, n_step: the start value, end value and step size of n in n-gram (ie. the size of n-gram) used in parameters searching. In training mod, n_start is the value used by the algorithm. Default: 1, 1, 1.
- m_start, m_end, m_step: similar to the previous one, but for the number of selected n-gram. Default: 400, 100, 100.
- num_iter_start, num_iter_end, num_iter_step: similar to the previous one, but for the maximum number of iterations. Default: 100, 100, 100.
- threshold_start, threshold_end, threshold_step: similar to the previous one, but for the threshold to label unclassified sentences. Default: 0.5, 0.5, 0.1.
- alpha_start, alpha_end, alpha_step: similar to the previous one, but for learning rate. Default: 0.002, 0.002, 0.002.
- delta_start, delta_end, delta_step: similar to the previous one, but for minimum gradient to stop training. Default: 0.05, 0.05, 0.05.
- mod: specify the mod, it can be either search, train or test. Default: help.
- test_sent: sentence to test in the testing mod. Default: I go to the bank and withdraw some money.

If there are missing arguments, default values will be used. Below are some examples:

Search for the best m and threshold_start and use default values for other hyper paramenters:

```
  # spark-submit --class
ca.uwaterloo.cs451.yarowski.Yarowsky
target/assignments-1.0.jar --input
data/bank_final.txt
--mod search
--m_start 100 --m_end 1000 --m_step 300
--threshold_start 0.5 --threshold_end 0.8
--threshold_step 0.1
```

Search for the best m, use threshold = 0.5 and default values for other hyper paramenters:

```
# spark-submit --class
ca.uwaterloo.cs451.yarowski.Yarowsky
target/assignments-1.0.jar --input
data/bank_final.txt
--mod search
--m_start 100 --m_end 1000 --m_step 300
--threshold_start 0.5 --threshold_end 0.5
--threshold_step 0.1
```

Train a model with m = 700, threshold = 0.6 and default values for other hyper parameters; write classification of input to "output/" and model to "model/":

```
# spark-submit --class
ca.uwaterloo.cs451.yarowski.Yarowsky
target/assignments-1.0.jar --input
data/bank_final.txt  --output output --model model
--mod train
--m_start 700 --threshold_start 0.7999999999999999
```

This should give an accuracy above 0.90 but the algorithm is not deterministic.

Find classification for the sentence "I go to the bank and withdraw some money"

```
# spark-submit --class
ca.uwaterloo.cs451.yarowski.Yarowsky
target/assignments-1.0.jar --input
data/bank_final.txt  --model model
--mod test --test_sent "I go to the bank and
withdraw some money"
```

B. PySpark

PySpark implementation is similar to Scala. Due to the constraints of PySpark, we stopped at the step of hyper-parameter optimization. The code to run the python version program is located in the file: python_version_project/yarowsky.ipynb

The main function for PySpark version is the "run" function. It consumes

- sents0: training set that preprocessed from previous lines
- sents_test: test set that preprocessed from previous lines
- model_path: path to save trained model
- resul_path: path to save the classified text
- save: Boolean value, if to save the value is "True", otherwise "False"
- N: number for N-gram to be extracted
- m: the top m high information gain N-grams to use
- niter: number of iterations to run
- threshold: whether to classify a scored (using logistic regression) word to a sense
- alpha: tuning parameter used in logistic regression calculation
- delta: stopping criteria for training process

The "parameter_search" function is used to find optimal parameters for "run" function. The PySpark version Yarowsky algorithm takes long time to run. A more effective code or higher computing power may resolve this problem.

V. Conclusion

The Yarowsky algorithm is indeed a powerful semi-supervised learning algorithm as the author suggested. After implementation, we observe an over 90% accuracy in both Scala (CS651) and PySpark(CS631) version code applying on "bank" text file. In this project, we classified the word "bank" with two meanings. Nevertheless, the algorithm is applicable to multi-meaning words with a minor change to our code. The two powerful and empirically observed language properties, one sense per collocation and one sense per discourse, ensure that the Yarowsky algorithm works efficiently.

For this project, our algorithm implementation incorporates logistic regression and hyper-parameter tuning that learned in class, utilized tokenize and spamminess (classify) functions from previous assignments. This makes us believe that after digesting the knowledge from classroom, we can move forward to implement algorithms that we never learned from class, or even to further modify it.

Regarding the improvements of the algorithm, multiple techniques can be applied. For example, using an adapted step size or size of feature set m. The accuracy can be further improved significantly by applying those techniques. But since our objective is to adapt the Yarowsky's Algorithm to a distributed computing environment. The stated improvements are not implemented since they are not very relevant. Another easy way is to train the model with more sentences, but our limited disk quota does not allow us to do so.

References

[1] Miller. (1990). Princeton WordNet. Retrieved from https://wordnet.princeton.edu/

[2] Agirre, Eneko, Edmonds, Philip (Eds.)(2007). Word Sense Disambiguation.Retrieved from https://www.springer.com/gp/book/9781402048081

[3] Yarowsky D. Unsupervised Word Sense Disambiguation Rivaling Supervised Methods.(1995) Retrieved from https://www.aclweb.org/anthology/P95-1026