CS4740 Word Sense Disambiguation

Kang-Li Stephen Cheng (ksc66)

1 Introduction

Word sense disambiguation (WSD) is a task to find the correct sense (meaning) of a word given the *context*. It is an important building block for a number of tasks in natural language processing, with many applications in machine translation, speech synthesis, etc.

In this WSD project, we implement a supervised learning method by using the given English Lexical Sample from Senseval[1] for training and testing the WSD system.

2 Supervised WSD

In the supervised learning approach in WSD, a training corpus of words tagged in context with their senses is supplied to train a classifier which can tag new words in new texts. The training corpus in this project includes a training dataset and a dictionary. A test dataset is also supplied to evaluate/test the performance of the WSD system we build.

2.1 Supervised Machine Learning

In supervised WSD, the input includes:

- a word w in a text window d (called a "document"),
- a fixed set of senses (classes) $S(w) = \{s_1, s_2, \ldots, s_j\}$,
- a training set of m hand-labeled text windows with answers: $(d_1, s_1), \ldots, (d_m, s_m)$.

The output is a learned classifier which predicts the most likely sense $\hat{s}$ given a document d (which usually is a feature vector).

2.2 The Naïve Bayes Algorithm for Classification

The naïve Bayes classification model is based on the simple intuition of the Bayes rule. The document is represented by a feature vector (bag of words). The model takes a word in context as an input and outputs a probability distribution over the pre-defined set of senses S. The naïve Bayes model picks the best sense by

$$\hat{s} = \underset{s \in S(w)}{\operatorname{argmax}} P(s|\vec{f}) \approx \underset{s \in S(w)}{\operatorname{argmax}} P(\vec{f}|s)P(s),$$

where s is the sense, $\vec{f}$ is a feature vector extracted from the context (also called a window) surrounding word w. With assumptions that the position of words in the feature

[1] International Workshop on the Evaluation of Systems for the Semantic Analysis of Text, which is organized by ACL-SIGLEX. `http://www.senseval.org/`

vector does not matter, and the independence of the feature probabilities $P(w_j|s_j)$, where $\vec{f} = (w_1, w_2, \ldots, w_n)$, we have s_{NB}, the sense classified by the Naive Bayes as

$$\hat{s}_{NB} = \underset{s \in S}{\operatorname{argmax}} P(s_j) \prod_{w \in \vec{f}} P(w|s). \tag{1}$$

In the supervised training, we compute $\hat{P}(s)$, the prior probability of sense s, by

$$\hat{P}(s) = \frac{N_s}{N}, \tag{2}$$

where N_s is number of sense s found, and N is the total number of documents in the dataset, respectively. Equation (1) has to be "smoothed", particularly because if the classifier has not seen a word w in the training documents, the probability $P(w|s)$ will be zero. The remedy is use a Laplace (or "add-1") smoothing as follows:

$$\hat{P}(w_i|s) = \frac{\text{count}(w_i, s) + 1}{\text{count}(s) + |\vec{f}|}\ , \tag{3}$$

where $|\vec{f}|$ is the size of the feature vector.

3 Implementation of WSD in LingPipe Toolkits

3.1 Software and Coding

In this project, we use the LingPipe Java toolkits for building the WSD model. LingPipe is a toolkit for NLP-oriented applications. It also includes complete models for POS tagging, NER. The AGPL (royalty free license) version with source codes is available for free download.

The main code WSD.java is a code based on a LingPipe tuturial code called Senseval3.java. A few important changes are made so that the tokenization, stop words eliminations, and stemming of the training dataset are used in processing the training data for building our WSD model. (The LingPipe tutorial tokenizes the training data based on spaces only.)

3.2 The Training Dataset

The training dataset is in XML format. Each word is enclosed in `<lexelt>` pseudo-tags. For example, `activate.v`, has man entries like the following:

```
<lexelt item="activate.v">

<instance id="activate.v.bnc.00024693" docsrc="BNC">
<answer instance="activate.v.bnc.00024693" senseid="38201"/>
<context>
Do you know what it is ,  and where I can get one ?  We suspect you had
seen the Terrex Autospade ,  which is made by Wolf Tools .  It is quite
a hefty spade , with bicycle - type handlebars and a sprung lever at the
rear , which you step on to <head>activate</head> it .
</context>
</instance>
```

Each training data entry for a target word (e.g., `activate.v`) is represented by `Map<String, List<String>>`. Training dataset class extends the Java HashMap as `HashMap<String, Map<String, List<String>>>`.

3.3 Test Dataset

The format of the test dataset is identical to the training data without the answer element providing the sense identification. Since we use the test data only once, the test dataset class consists of three `ArrayList<String>` for the words, the instance Ids, and the contexts as the data for the class.

3.4 The Dictionary Dataset

Each entry in the dictionary dataset consists of word.pos (e.g., `appear.v`) to an array of senses, for example:

```
<lexelt item="appear.v">
<sense id="190901" source="ws" synset="appear arise emerge show"
gloss="to come into view; become visible."/>
<sense id="190902" source="ws" synset="appear look seem"
gloss="to seem. "He appears smart, but I have doubts"."/>
<sense id="190903" source="ws" synset="appear come_out"
gloss="to come before the public, as a book or performer."/>
</lexelt>
```

Dictionary class extends `HashMap<String, Sense[]>`, where the `Sense` class contains data for for items defined in the word entry: Id, Source, Synset, and Gloss.

3.5 Processing the Datasets With Tokenizers

The training data file is first tokenized by white-space using LingPipe's regular-expression-based tokenizer factory. Then a normalizing tokenization is applied to the result of space tokenization. Normalizing the data includes conversion of all tokens to lower cases, remove English stop words, and perform stemming[2] to the output.

```
static TokenizerFactory normTokenizerFactory() {
    TokenizerFactory factory = SPACE_TOKENIZER_FACTORY;
    factory = new LowerCaseTokenizerFactory(factory);
    factory = new EnglishStopTokenizerFactory(factory);
    factory = new PorterStemmerTokenizerFactory(factory);
    return factory;
```

3.6 Training the WSD Model

The dictionary and training data are used to train the model by using the naïve Bayes classification. Here is an outline of the process:

1. After the dictionary and training dataset are already processed and we have them as `HashMap` objects, the code loops over all the words in the training data, and extracts each word's senses to a text list, as well as the corresponding senseIds to

[2]Porter's stemming algorithm (Porter's stemmer) computes an approximation of converting words to their morphological base forms.

a String array. The senseIds are then used as the "classes" in calling a trainable classifier.

2. For each sense for the word, the classification corresponding to the sense is used to train the classifier. In this project, it is the naïve Bayes model. Once the loop is over, the classifier is compiled and added to the mapping.

4 Scoring and Experimentation for the Supervised WSD Model

4.1 The Naïve Bayes Model

In order to evaluate the performance of the WSD Model *without* being able to upload the predicted senses to Kaggle at CMS, we download the Senseval 3 Workshop datasets from the official website.[3] Senseval 3 provides the English training and test datasets, the answer key, and the scoring code scorer2.c for senses prediction evaluations. We still train our WSD model by using the CS4740 dictionary and training data files

- `training-data.data`
- `Dictionary.xml`

for building the WSD model, but using the Sesneval 3 test dataset `EnglishLS.test` and the answer key `EnglishLS.test.key` for testing and recording the performance.

For comparison purposes, we use the datasets from Senseval 3 (`EnglishLS.train` and `EnglishLS.dictionary.xml`) to train a "baseline" WSD model. The fine-grained scores treat every sense as distinct. The coarse-grained scores are derived from the fine-grained score by conflating the senses into smaller number of sets. The identical scores

Model	Precision	Recall
WSD-Fine Grain	0.638	0.638
Baseline-Fine Grain	0.638	0.638
WSD-Coarse Grain	0.689	0.689
Baseline-Coarse Grain	0.689	0.689

Table 1: Our supervised WSD Model in comparison with the baseline model. In both models, the Naïve Bayes model are used.

produced by our WSD model and the baseline indicate that the training and dictionary corpora given in CS4740 are almost identical to the Senseval 3 datasets. (Although a simple comparison by using the diff command shows the CS4740 training and dictionary files are different from the Senseval 3 corpora.)

[3] `http://web.eecs.umich.edu/~mihalcea/senseval/senseval3/data.html`

4.2 Additional Experiments on the WSD Model

4.2.1 The Effect of Stemming

The training of the WSD model is based on the normalization of the tokens: convert all tokens to lower cases, eliminate the English stop words and perform stemming on the output. The following table shows the effect of stemming the training data. Interestingly, stemming the training data has very little effect on the supervised WSD model performance. More precisely, stemming actually reduces the score slightly (less than 1% in the score).

WSD Model	Precision	Recall
Stemming-FG*	0.638	0.638
No Stemming-FG	0.640	0.640
Stemming-CG*	0.689	0.689
No Stemming-CG	0.694	0.694

Table 2: The Effect of Stemming the training data on the supervised WSD Model.(*FG and CG stand for fine grain and coarse grain, respectively.)

4.2.2 The Effect of Token N-Grams in Training the WSD Model

Since we treat the feature vector as a bag of words, our naïve Bayes WSD model is just a "token unigram" model. It is interesting to explore the effect of n-gram.

We can use the LingPipe's DynamicLMClassifier class for testing the token n-gram effects. The results of 6-gram, 3-gram are summarized in the following table in comparison with our naïve-Bayes WSD model. The experiments show that the effect of token N-Grams in training the WSD model in comparison with the naïve-Bayes WSD is negligibly small.

WSD Model	Precision	Recall
NaiveBayes-FG*	0.638	0.638
3-Gram-FG	0.638	0.638
6-Gram-FG	0.637	0.637
NaiveBayes-CG*	0.689	0.689
3-Gram-CG	0.689	0.689
6-Gram-CG	0.688	0.688

Table 3: The Effect of N-Grams in supervised training on WSD model performance. (*FG and CG stand for fine grain and coarse grain, respectively.)