Project 2 Report: Word Sense Disambiguation

Shibo Zang(sz428) Hongfei Li(hl963)

1. Approach

For our WSD system, we implement our idea described in the proposal， which is to use supervised learning algorithm -- Naive Bayes as the classifier of Word Sense Disambiguation and to use WordNet dictionary to select feature vectors.

We divided our system into several sub-functions:

1.1 Data cleaning and data preprocessing:

We first eliminate punctuations and meaningless words in the corpus such as prepositions, conjunctions, articles, pronouns, interjections, quantifiers, auxillaries, etc., using part-of-speech tagging. The tagging type is showned below.

```python
def construct_unwanted_tags():
    unwantedTags = set()
    l = ['$', '\'\'', '(', ')', ',', '--', '.', ':', 'CC', 'CD',
    'DT', 'EX', 'IN', 'PDT', 'LS', 'MD', 'POS', 'PRP',
    'PRP$', 'RP', 'SYM', 'TO', 'UH', 'WDT', 'WP', 'WP$', 'WRB', '``']

    for t in l:
        unwantedTags.add(t)
    return unwantedTags
```

From above codes, we see that the unwanted set are turned into a set for better look up performance in the next steps.

Once we have the unwanted word tags, we will start parsing the data file and remove words with unwanted tag, and transform the remaining words to their simplest form. The following codes are used to do so. Details of the following code will be discussed in Section 2.

```Python
def remove_unwanted_tags(string, unwantedTags):
    wordTagPairs = pos_tag(word_tokenize(string))
    returnTags = []
    for word in wordTagPairs:
        if(word[1] not in unwantedTags) and len(word[0]) != 0
            and len(word[0]) != 1 and word[0] != 'n\'t'
            and word[0] != '\'s' and word[0] != '\'ll':
            returnTags.append(word)
    return returnTags

def sentence_to_present_tense_and_single(tags):
    for i in range(len(tags)):
        wn_tag = penn_to_wn(tags[i][1])
        tags[i] = WordNetLemmatizer().lemmatize(tags[i][0], wn_tag)
    return tags
```

The input corpus are processed using the above methods and `processed_trainig.xml` is generated for training purpose.

1.2 Data Structures

In the experiment, we built individual model for each target words. The primary data structure we used is Dictionary in Python. As we used Naive Bayes as the classifier, we need two dictionary to store prior probabilities and individual feature probabilities respectively. The first dictionary stores prior probability of each senseid of a particular word (lexelt_item) $P(s_i)$.

```
priorProbabilityDictionary
{
    lexelt_item:
    {
        sense_id: probability
    }
}
```

We can see that the key of outer dictionary is lexelt_item, and its correpsonding value is a inner dictionary, whose key value pair is sense_id and its probability.
And the second dictionary stores individual feature probabilities $P(f_i|s_i)$. And the structure is:

```
featureProbabilityDictionary
{
    lexelt_item:
    {
        sense_id:
        {
            feature: probability
        }
    }
}
```

As you can see, this is a nested dictionary. The outermost dictionary consists of multiple lexelt_items. For each lexelt_item, its value is another dictionary, whose key is sense_id and value is the innermost dictionary, composed by feature - probability pairs.

The advantage of using dictionary instead of list is due to its high efficiency. The searching and insertion complexity are both O(1).

1.3 Naive Bayes Classification Model

1.3.1 Naive Bayes Model

To train a Naive Bayes Classification Model, we need to compute prior probabilities for each sense_id and individual feature probabilities for each feature given a certain sense_id.

```python
def naive_bayes_training(root):
    #  Compute P(senseid_i)
    prior_prob = {}

    trained = 0
    for lexelt in root:
        if trained == training_amount:
            break

        for instance in lexelt:
            for answer in instance:
                if answer.tag == 'context':
                    continue

                senseid = answer.attrib['senseid']
                if senseid == 'U':
                    continue

                item = lexelt.attrib['item']
                if item in prior_prob:
                    lexelt_dict = prior_prob[item]
                    if senseid in lexelt_dict:
                        lexelt_dict[senseid] += 1
                    else:
                        lexelt_dict[senseid] = 1

                else:
                    prior_prob[item] = {senseid: 1}

    senseid_amount = {}

    for lexelt in prior_prob:
        count = 0
        for senseid in prior_prob[lexelt]:
            count += prior_prob[lexelt][senseid]
        for senseid in prior_prob[lexelt]:
            senseid_amount[senseid] = prior_prob[lexelt][senseid]
            prior_prob[lexelt][senseid] /= float(count)

    print "Prior prob done"
```

From above codes, we iterate each <answer> ... </answer> label in the preprocessed xml file. We can easily get senseid from these labels and count their occurencies. Then the prior probability for one sense_id would be its count divided by all sense_id count of a lexelt_item.

The individual feature probability calculation part is implemented as follows:

Python

```python
def naive_bayes_training
    #  Compute P(feature_i|senseid_i)
    trained = 0
    feature_candidates_dict = {}
    for lexelt in root:
        if trained == training_amount:
            break

        lexelt_item = lexelt.attrib['item']
        if lexelt_item not in feature_candidates_dict:
            feature_candidates_dict[lexelt_item] = {}

        for instance in lexelt:
            if instance[0].attrib['senseid'] == 'U':
                continue

            length = len(instance) - 1
            context = instance[length].text.strip().split()
            feature_candidates = context[-(select_range/2):]

            for head in instance[length]:
                if head.tail is None:
                    continue

                feature_candidates += head.tail.strip().split()
                if len(feature_candidates) > select_range:
                    break

            feature_candidates = feature_candidates[:select_range]

            for answer in instance[:length]:
                senseid = answer.attrib['senseid']
                if senseid == 'U':
                    continue

                for feature in feature_candidates:
                    if feature in feature_candidates_dict[lexelt_item]:
                        if senseid in feature_candidates_dict[lexelt_item][feature]:
                            feature_candidates_dict[lexelt_item][feature][senseid] +=
                        else:
                            feature_candidates_dict[lexelt_item][feature][senseid] =
                    else:
                        feature_candidates_dict[lexelt_item][feature] = {senseid: 1}
        print lexelt_item + ": Feature candidates done"

    print "Feature candidates all done"
    feature_dict = {}
    for lexelt_item in feature_candidates_dict:
        feature_dict[lexelt_item] = {}

        for feature in feature_candidates_dict[lexelt_item]:
            feature_dict[lexelt_item][feature] = get_word_definition_overlap_count(fe
                                                    lexelt_item.split('.')[0])
            print feature + ' overlap ' + lexelt_item + ': ' + str(feature_dict[lexel

        sorted_features = sorted(feature_dict[lexelt_item],
                                 key=feature_dict[lexelt_item].get, reverse=True)[:feature

        print lexelt_item.split('.')[0] + ' selected features: ' + str(sorted_feature

        feature_dict[lexelt_item] = {}

        for feature in sorted_features:
            senseid_count_dict = feature_candidates_dict[lexelt_item][feature]

            for senseid in senseid_count_dict:
                if senseid not in feature_dict[lexelt_item]:
                    feature_dict[lexelt_item][senseid] = {}
                feature_dict[lexelt_item][senseid][feature] = senseid_count_dict[sens
                                                    / float(senseid_amount[senseid])
                print feature + ' ' + str(senseid) + ' ' + str(senseid_count_dict[ser
                        + ' ' + str(senseid_amount[senseid])

        print lexelt_item + ": prob calc done"

    for lexelt_item in feature_dict:
        all_senseid_count = 0
        for senseid in feature_dict[lexelt_item]:
            all_senseid_count += senseid_amount[senseid]
        feature_dict[lexelt_item]['<unk>'] = 1 / float(all_senseid_count)

    with open('prior_prob.json', 'w') as f:
        f.write(json.dumps(prior_prob))
    with open('feature_dict.json', 'w') as f:
        f.write(json.dumps(feature_dict))

    return (prior_prob, feature_dict)
```

To compute feature probabilities given a certain sense_id $P(f_i|s_i)$, we will count how man times a feature occur given a sense_id, and divide it by the amount of times that sense_id occurs.

In all, the method to compute prior probabilities and feature probabilities are the same with the procedure given by Instruction.pdf.

1.3.2 Model Parameters

For training model, there are three main parameters.

```python
# Feature range around target word
select_range = 6

# The number of features for a lexelt
feature_num = 20

# The training number for each lexelt
training_amount = 20

# The weight given to unigram
alpha = 0.3
```

- select_range: while analysing a target word, select_range decide how man surrounding words we will look each time. 6 means we will look at 3 precedinging words and 3 following words for each <head>some_word</head>.
- feature_num: this argument decide how many feature words will be selected finally. The default value is 20, which means that for a single lexelt_item, we will have 20 feature words after training is done.
- training_amount: this argument is considered for efficiency. In our experiment, we found that the most unefficient operation is to compute word definition overlapping count, which will be discussed in the following paragraphs. However, for every lexelt_item, there are more than 100 contexts to be analyze, which would incur large amounts of computing overlapping count operations. Thus, we decided to train our model basing on the first $training\_amount$ context. And we will show that the performance of classifier would not be decreased by the limitation of training data size.
- alpha: while computing the word overlapping counts, we use the formula $alpha * unigram\_count + (1 - alpha) * bigram\_count$ as the point from feature candidate word to target word, and we will select those words who have the highest point.

1.3.3 Feature Selection

Feature selection is very important for Naive Bayes Classifier. We implemented feature selection by the method we proposed in the proposal.

First, we look at three previous words and three following words (set by $select\_range$) around the target word. We will process these 6 words based on their definitions in the dictionary and reward those words which have consecutive overlap with the definition of target word. How to decide which words should be selected as features? We use WordNet to help us define the definition of these words and compare them one by one.

In order to use dictionary to help us pick feature words, a helper method `get_word_definition_overlap_count` is written. This method is used to calculate a score that represents how close two words' meanings are. The approach is that firstly we combines are definitions of two words into two large strings, then we calculate number of bigrams and unigrams that overlaps in the two large strings. Secondly, since it is highly possible that duplilcate words will exist in a large string, and number of occurences does not matter in our case, we used a set of words to represent the combined definitions of a word. Lastly, once we have bigram and ungram overlap counts, we used weight index alpha to calculate score. Current weight for unigram is 0.3 and for bigram is 0.7.

```python
# Unigram weight
alpha = 0.3


def get_word_definition_overlap_count(a, b):
    unwantedTags = utils.construct_unwanted_tags()
    defs_a = wn.synsets(a)
    defs_b = wn.synsets(b)

    # Unigram overlap count
    unigramOverlapCount = 0
    bigramOverlapCount = 0
    aUnigramSet = set()
    bUnigramSet = set()
    aBigramSet = set()
    bBigramSet = set()
    # Construct a ngram sets
    for d in defs_a:
        aUnigramArray = utils.process_string(
            d.definition().lower(), unwantedTags).split()

        # Unigram
        add_list_to_set(aUnigramSet, aUnigramArray)
        # Bigram
        for i in range(len(aUnigramArray) - 1):
            aBigramSet.add(aUnigramArray[i] + aUnigramArray[i + 1])

    # Construct b ngram sets
    for d in defs_b:
        bUnigramArray = utils.process_string(
            d.definition().lower(), unwantedTags).split()
        # Unigram
        add_list_to_set(bUnigramSet, bUnigramArray)

        # Bigram
        for i in range(len(bUnigramArray) - 1):
            bBigramSet.add(bUnigramArray[i] + bUnigramArray[i + 1])

    for word in bUnigramSet:
        if(word in aUnigramSet):
            unigramOverlapCount += 1
    for word in bBigramSet:
        if word in aBigramSet:
            bigramOverlapCount += 1
    return unigramOverlapCount * alpha + bigramOverlapCount * (1 - alpha)
```

```
def add_list_to_set(setS, l):
    for a in l:
        setS.add(a)
```

In the end, we select 20 features (set by feature_num) with the highest overlapping count for each word.

1.3.4 Unknown word smoothing

Another important part of our system is how to smooth unknown word. As the theorem by Naive Bayes, $s = argmax_{s \in S(w)} P(s) \prod P(f_j|s)$, we found that some feature f_k might not occur in the training contexts. Thus, its probability $P(f_k|s_i)$ is regarded as 1 and leads to inappropriate results.

We implement our smoothing method according to the formula: $P(< unk > |s_i) = \frac{1}{\sum count\_of\_s_i}$. By realizing the smoothing method, we increased our classification accuracy over 20% on our validation set.

1.4 Classification Procedures

We will look at previous 10 words and following 10 words around the target word. We will process these 20 words based on their definitions in the dictionary and reward those words (increase their weights in the feature vector) which have consecutive overlap with the definition of target word. And of course we also need to process the content of words in dictionary, which means to keep nouns, adjectives, verbs, and adverbs. Finally we would get the feature vectors we want and apply them to the Naive Bayes classifier.

The function `word_sense_disambiguation()` takes two dictionaries as argument -- prior probabilities dictionary and feature probabilities dictionary. From these probabilities given by model, it multiple these probabilities every time it sees a feature vector. And then compare them basing on the formula $s = argmax_{s \in S(w)} P(s) \prod P(f_j|s)$, choosing the highest one as the prediction label.

```python
Python
def word_sense_disambiguation(featureProbabilityDictionary, priorProbabilityDictionar
    sense_ids = featureProbabilityDictionary[instanceString]
    sense_id_scores = {}
    surround_words = []
    if(context.text is not None):
        surround_words = context.text.strip().split()[-10:]

    for head in context:
        if(head.tail is None):
            continue
        surround_words += head.tail.strip().split()
        if len(surround_words) > 20:
            break
    surround_words = surround_words[:20]

    for sense_id, featureProbility in sense_ids.iteritems():
        if sense_id == '<unk>':
            continue

        sense_id_scores[sense_id] = priorProbabilityDictionary[
            instanceString][sense_id]

        for word in surround_words:
            if word in featureProbility:
                sense_id_scores[sense_id] *= featureProbility[word]
            else:
                sense_id_scores[sense_id] *= sense_ids['<unk>']


    # return the sense_id with max score
    return max(sense_id_scores.iteritems(), key=operator.itemgetter(1))[0]
```

In the above code, we calculated the score of each sense id in every given context, and use the sense id with highest score as the meaning of the word in given context. The scoring method is simply multiplying the probabiliy of occurence of the words that surrounds the target word. The probabily of feature words and unknow words are read from `feature_dict.json` and `prior_prob.json` which are calculated from the naive-bayes-trainng section.

2.Software

Python's `xml.etree.ElementTree` class to parse the given data file. However removing `&` from the data files and add a `<root>` to the beginning to the data files were necessary manual process in order to have `ElementTree` function correctly.

```Python
tree = ET.parse('processed_training.xml')
root = tree.getroot()
lexelts = root.findall("./lexelt")
```

Python nltk library to was used to tokenize sentences and do POS tagging.

```Python
wordTagPairs = pos_tag(word_tokenize(string))
```

The above line of code will return a list of word tag pairs such as

```Python
[('delicious','JJ')('apple','NN'),.....]
```

Using the word tags obtained above, nltk's `WordNetLemmatizer` function was used to transform words to their simple form for analysis optimization.
For example, here is a table of words that were transformed.

Original Word	Transformed Word
apples	apple
happier	happy
runned	run
harder	hard

Code example

```python
def sentence_to_present_tense_and_single(tags):
    for i in range(len(tags)):
        wn_tag = penn_to_wn(tags[i][1])
        tags[i] = WordNetLemmatizer().lemmatize(tags[i][0], wn_tag)
    return tags

def is_noun(tag):
    return tag in ['NN', 'NNS', 'NNP', 'NNPS']

def is_verb(tag):
    return tag in ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']

def is_adverb(tag):
    return tag in ['RB', 'RBR', 'RBS']

def is_adjective(tag):
    return tag in ['JJ', 'JJR', 'JJS']

def penn_to_wn(tag):
    if is_adjective(tag):
        return wn.ADJ
    elif is_noun(tag):
        return wn.NOUN
    elif is_adverb(tag):
        return wn.ADV
    elif is_verb(tag):
        return wn.VERB
    return wn.NOUN
```

The wordnet corpus that was built within nltk was used to obtain definitions of words

Code example

```python
from nltk.corpus import wordnet as wn

defs = wn.synsets('apple')
```

3. Results

We conducted several experiments basing on our classification model. By changing our model parameters, we figured out how these parameters affect the classification result. All test results are coming from our validation set, and the validation set is splited from our training set. We have 3 variables that can be varied.

First, let's see how α affects our model. Alpha is used when word definition closeness are calculated. α is the weight that is put on unigram, and $1 - \alpha$ is the weight for bigram.

Alpha	Accuracy
0.2	0.763157895
0.3	0.763157895
0.4	0.75877193
0.5	0.75877193
0.7	0.75877193
0.8	0.75877193

As we can see from the table above, varying α does affect our test accuracy a little bit. But more significant accuracy change could be achieved by varying number of features words that we select.

Feature Number	Accuracy
1	0.776315789
5	0.771929825
20	0.75877193
60	0.745614035
120	0.640350877

Surprisingly, the less feature number we have, the higher validation accuracy we got. We are thinking that the model is overfitting when feature number reaches a certain level. However, we can not explain why accuracy decreased when feature number was increased from 1 to 5.

We have also varied feature selection range for training process. This number represents how many words that surrounds the targeting word that were processed.

Feature Range	Accuracy
2	0.789473684
6	0.75877193
8	0.76754386
10	0.763157895

The table above shows that the larger feature range we had, the lower validation accuary we were able to achieve. We think this is because the further away the feature words are from target word, the less correlated the two words are. That's why having a smaller feature range can help us elimininate feature candidates and improve accuracy.

4. Discussion

We implement the Word Sense Disambiguation basing on the Naive Bayes Classifier, and using words that have most common sense as the feature. The motivation of our method is that we think words that share some sense in common may be a good feature to disambiguate word senses, because it indicates that these words have a high correlation with the target word, which means that these words would have a high occurence probability if the target word occurs. For example, from our experiment, word "atmosphere" could have several sense_ids,

```
"atmosphere.n": {
    "atmosphere%1:26:01::": {
      "break": 0.0093457943925234,
      "set": 0.0093457943925234,
      "bar": 0.0093457943925234,
      "form": 0.0093457943925234,
      "show": 0.0093457943925234,
      "work": 0.0093457943925234,
      "high": 0.0093457943925234,
      "good": 0.0093457943925234,
      "right": 0.018691588785047,
      "encounter": 0.0093457943925234,
      "last": 0.0093457943925234,
      "comfort": 0.0093457943925234,
      "contempt": 0.0093457943925234
    },
    "atmosphere%1:26:00::": {
      "level": 0.333333333333333
    },
    "atmosphere%1:07:00::": {
      "head": 0.018518518518519,
      "set": 0.037037037037037,
      "show": 0.018518518518519,
      "high": 0.055555555555556,
      "right": 0.037037037037037,
      "encounter": 0.018518518518519,
      "setting": 0.018518518518519,
      "contempt": 0.018518518518519
    },
    "atmosphere%1:15:00::": {
      "high": 0.052631578947368,
      "planet": 0.052631578947368,
      "good": 0.052631578947368,
      "close": 0.052631578947368,
      "level": 0.052631578947368
    },
    "atmosphere%1:17:00::": {
      "planet": 0.14285714285714,
      "tropical": 0.14285714285714,
      "line": 0.14285714285714
    },
    "<unk>": 0.0052631578947368
  }
```

This dictionary shows the lexelt_item "atmosphere" and its corresponding sense_ids and its high correlated features. We can see that "level" is a very good feature to distinguish senseid "atmosphere%1:26:00::" from others because it has high probability when "atmosphere%1:26:00::" occurs and has low probability when other senses occur. And words like "bar" or "close" might also be good features because they are the unique features for "atmosphere%1:26:01::" and "atmosphere%1:15:00::".

Another example:

```
"play.v": {
    "3165213": {
      "do": 0.125,
      "deal": 0.125,
      "give": 0.125
    },
    "3165211": {
      "move": 0.0833333333333333,
      "lead": 0.0833333333333333
    },
    "3165210": {
      "do": 0.131578947368842,
      "play": 0.026315789473684,
      "run": 0.026315789473684,
      "lead": 0.078947368421053,
      "make": 0.026315789473684,
      "work": 0.026315789473684,
      "start": 0.026315789473684,
      "come": 0.026315789473684
    },
    "3165217": {
      "do": 0.052631578947368,
      "play": 0.052631578947368,
      "get": 0.052631578947368,
      "caught": 0.052631578947368,
      "give": 0.10526315789474,
      "draw": 0.052631578947368,
      "turn": 0.052631578947368,
      "take": 0.052631578947368,
      "go": 0.10526315789474,
      "come": 0.052631578947368,
      "ground": 0.052631578947368
    },
    "3165214": {
      "start": 0.16666666666667
    },
    "3165220": {
      "go": 0.33333333333333,
      "play": 0.33333333333333,
      "playing": 0.33333333333333
    },
    "3165218": {
      "do": 0.14285714285714
    },
    "<unk>": 0.0098039215686275
  },
```

As you know, "play" is a typical polysemant. From the table above, we can easily see that sense_id "3165214", "3165218", can be easily distinguished by their unique word. And for other senses, we could use naive bayes formula to compute their probabilities.