CS 5740 - Project 2 Report

Philip Su (ps845)
Kelvin Jin (kkj9)
10/23/2015

Overview

In this project, we used the Naive Bayes classification method in order to assign word senses to words in unknown contexts. In order to do this, we first trained two separate Naive Bayes models on the provided training data for collocation and co-occurrence. After this, we used the model to find the most likely sense for each unknown word in our test data. We explain details of our implementation below.

Approach

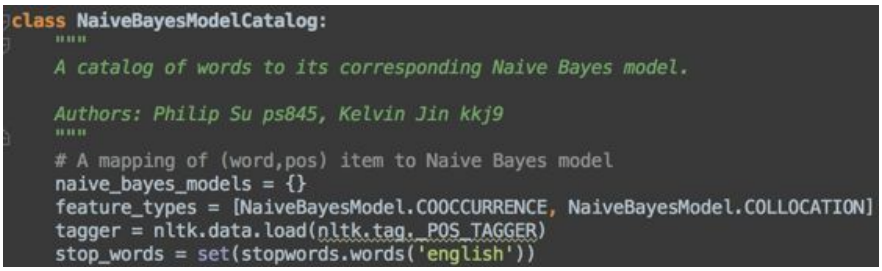

Figure 1. Information stored in the NaiveBayesModelCatalog class

Our top-level class is **NaiveBayesModelCatalog**, a singleton class that contains all of the feature vectors for every context in our training set. It exposes methods to generate and store feature vectors from training data, as well as outputting labels for test data.

Internally, the data is organized such that for each distinct lexeme, all of the feature vectors pertaining to that word is stored in a single **NaiveBayesModel** object. This is because feature vectors for two distinct lexeme pairs have nothing to do with each other.

For example, if the training corpus contained:

```
<lexelt item = “win.v”>...</lexelt>
<lexelt item = “wind.n”>...</lexelt>
<lexelt item = “wind.v”>...</lexelt>
```

then **NaiveBayesModelCatalog** holds the following mapping in **naive_bayes_models**:

```
{
      (“win”, “v”) : NaiveBayesModel(),
      (“wind”, “n”) : NaiveBayesModel(),
```

```
        ("wind", "v") : NaiveBayesModel()
    }
```

Within the training and testing methods of **NaiveBayesModelCatalog**, functionality is split between the catalog and corresponding methods for training and testing in the **NaiveBayesModel** class.

```
def train(self, context_body, senses, feature_type, word):
    """
    Creates a feature vector from the context of a word for a list of sense that have the
    same context. Adds the feature vector to corresponding feature dictionary.

    :param context_body:
    :param senses:
    :param feature_type:
    :param word:
    """
    feature_vector = self.__create_feature_vector(context_body, word, feature_type)
    if feature_type not in self.training_data:
        self.training_data[feature_type] = []
    for sense in senses:
        self.training_data[feature_type].append((feature_vector, sense))
    if feature_type == NaiveBayesModel.COLLOCATION:
        for pos_list in feature_vector.get_values():
            for pos in pos_list:
                self.pos_feature_set.add(pos)
    else:
        for word in feature_vector.get_values():
            self.word_feature_set.add(word)
```

```
def test(self, context_body, word, feature_type):
    """
    Given a context body and a word, calculates the conditional probability of a feature
    given a sense for all senses.
    :param context_body:
    :param word:
    :param feature_type:
    :return A dictionary of conditional probabilities and the
    """
    feature_vector = self.__create_feature_vector(context_body, word, feature_type)
    result = {}
    prior = {}
    if feature_vector == NaiveBayesModel.COLLOCATION:
        category_dim = len(self.pos_feature_set)
    else:
        category_dim = len(self.word_feature_set)
    for sense in self.sense_count_map:
        prob_f_given_s = feature_vector.get_feature_probability_given_sense(
            self.training_data[feature_type], sense, category_dim)
        result[sense] = prob_f_given_s
        prior[sense] = math.log(self.__get_prior(sense))
    return result, prior
```

Figure 2, 3: The train() and test() functions in our NaiveBayesModel class, respectively

While training, the catalog class deals with parsing the XML file and obtaining information such as the lexeme, its sense label, and the context surrounding it. This information is then passed to the model class, which creates a feature vector out of the lexeme's context and stores it along with its label. The flow of logic fairly similar when it comes to testing - important information is extracted from the XML file in the catalog class, and the **NaiveBayesModel** object that corresponds to the test lexeme returns the most likely word sense for that lexeme by finding the sense which maximizes *P(features* | *sense) P(sense)*.

```python
class CollocationFeatureVector:
    """
    Representation of a collocation feature. Collocation feature vectors are a list of
    parts of speech tags around a given word.

    Authors: Philip Su ps845, Kelvin Jin kkj9
    """

    def __init__(self, context_body, word):
        """
        Initializes the feature vector as a list with parts of speech tags around the word.

        :param context_body: context of a word
        :param word: word to determine sense of
        """
        self.feature_vector = []
        context_body = context_body.split()
        # pos_body = nltk.tag.pos_tag(context_body, 'universal')
        pos_body = NaiveBayesModelCatalog.tagger.tag(context_body)
        # Find all indices of word
        word_indices = [i for i in range(len(context_body)) if context_body[i] == word]
        for index in word_indices:
            # Only use words with enough context
            if (index >= NaiveBayesModel.N_CONTEXT and
                        index + NaiveBayesModel.N_CONTEXT < len(context_body)):
                for pos_word_pair in pos_body[index - NaiveBayesModel.N_CONTEXT: index + NaiveBayesModel.N_CONTEXT + 1]:
                    pos = pos_word_pair[1]
                    if pos in NaiveBayesModelCatalog.pos_map:
                        pos = NaiveBayesModelCatalog.pos_map[pos]
                    self.feature_vector.append(pos)
```

```python
class CooccurrenceFeatureVector:
    """
    Representation of a coocurrence feature. Coocurence feature vectors are a set of tokens
    in the context of a word.

    Authors: Philip Su ps845, Kelvin Jin kkj9
    """

    def __init__(self, context_body, word):
        """
        Initializes the feature vector as a list with parts of speech tags around the word.

        :param context_body: context of a word
        :param word: word to determine sense of
        """
        self.feature_vector = set()
        for context in context_body.split():
            if word == context or context.lower() in NaiveBayesModelCatalog.stop_words:
                continue
            self.feature_vector.add(context)
```

Figure 4: Classes used to represent feature vectors

The **CollocationFeatureVector** and **CoocurrenceFeatureVector** classes represent collocational and co-occurrence feature vectors respectively, and handle the calculation of *P(feature | sense)*. Co-occurrence features are stored as a set of all words, except for stop-words, appearing in the context of a word are stored in a set. Collocation features are stored as a list of parts of speech. For example, if we set our window size to four (meaning we consider the four words immediately preceding and succeeding the lexeme) and the word “bank” were surrounded by the following context:

```
('he', 'PRP'), ('went', 'VBD'), ('to', 'TO'), ('the', 'DT'), ('bank', 'NN'), ('to', 'TO'), ('deposit', 'VB'), ('some', 'DT'), ('money', 'NN')
```

The collocation feature vector would be:

```
list['PRP', 'VBD', 'TO', 'DT', 'NN', 'TO', 'VB', 'DT', 'NN']
```

and the co-occurrence feature vector would be:

```
set['he', 'went', 'bank', 'deposit', 'some', 'money']
```

where “to” and “the” are stop-words.

```
def get_feature_probability_given_sense(self, training_data, sense, category_dim):
    """
    Given a set of training collocation feature vectors, calculates the conditional
    probability of a feature vector given a sense. The product of probabilities become small,
    so operations are done as the summation of logarithms.

    :param training_data: a list of training feature vectors
    :param sense: the sense to calculate the conditional probability for
    :param category_dim: the set of all possible values a feature vector can take
    :return: conditional probability given a set of training features and a sense
    """
    prob = 1
    for test_feature in self.feature_vector:
        numerator = 0.0
        denominator = 0.0
        for training_feature_vector, training_sense in training_data:
            if sense == training_sense:
                denominator += 1.0
                if test_feature in training_feature_vector.feature_vector:
                    numerator += 1.0
        # Calculate the conditional probability with Laplace smoothing
        prob += math.log((numerator + NaiveBayesModel.SMOOTHING_CONSTANT) / (
            denominator + category_dim * NaiveBayesModel.SMOOTHING_CONSTANT))
    return prob
```

Figure 5: The method used in calculating the conditional probability of a feature given a sense

Both feature vector objects implement **get_feature_probability_given_sense()**, which calculates *P(feature* | *sense).* For **CoocurrenceFeatureVector**, a count is incremented anytime each word in the test context appears in the word set. For **CollocationFeatureVector**, a count is incremented anytime the parts of speech match at the same index. Probability calculations involve multiplying conditional probabilities together, however, this led to rounding errors so conditional probabilities are instead calculated with the summation of the corresponding logarithmic values.

After the **NaiveBayesModel** calculates the conditional probabilities *P(feature* | *sense*) for both collocation and co-occurrence features for every sense for a given lexeme, the model chooses the most likely sense by multiplying both probabilities and labelling the test data as the sense with the highest probability.

Software

Python’s **xml.etree.ElementTree** was used to parse the XML, using streaming techniques to keep memory usage low, even for large files. NLTK’s part of speech tagger (**nltk.data._POS_TAGGER)** was used to tag context with parts of speech for collocation feature vectors. Additionally, NLTK’s list of stop-words (**nltk.corpus.stopwords**) was used to clean the context paragraph for each word when generating co-occurrence feature vectors.

Results

In order to test the accuracy of our approach without relying on Kaggle, we split 20% of our training data into validation data. We then calculated our validation error in the following table:

Parameters	**Accuracy**
(Baseline)	35.61%
N=5; Included stop-words	50.36%
N=3; Excluded stop-words	59.35%
N=5; Excluded stop-words	58.99%
N=7; Excluded stop-words	58.27%

Table 1. Accuracy of our model based on the feature vector context size, and whether we excluded stop words or not (and baseline for reference). N represents the number of words in front and behind the lexeme that we considered for the feature vector

We varied our model with different parameters such as the number of words considered as part of the feature vector, and also whether stop-words were considered in co-occurrence feature vectors. Because excluding stop-words yielded a significant increase in accuracy, we did not investigate the model's performance when stop words were included any further.

In addition to running our model with different parameters, we also calculated the baseline accuracy to verify that we were not simply choosing the most frequency word sense in each case.

Discussion

During our experiments, we observed that excluding stop-words led to a significant increase in accuracy. The increase in accuracy is a result of the feature vector being less noisy, and the probabilities are calculated only with words that provide significant meaning to disambiguating the sense of the word. In contrast to our expectations, increasing the feature window for collocation features larger than 3 actually led to lower accuracy as shown in Table 1. This could be a result of overfitting on training data. Additionally, the window size of collocation feature vectors seem to have less of an impact on the accuracy than co-occurrence features.

For the test word **bank.n.bnc.00119099**, the sense that was labelled was **bank%1:14:00::**. Within this word sense, the words that appeared most often were "paid/payment, tax, source". From a non-NLP point of view, these context words make sense as they both suggest the word "bank" in relationship with money.