Practical Excercise 2: Word-Sense Disambiguation

For this practical, your task is to implement a program that performs Word-Sense Disambiguation using the Naive Bayes approach.

1 Algorithm description

Naive Bayes uses two phases: training and testing. During the training phase, a statistical model is constructed based on the training corpus. During the testing phase, the model is used to disambiguate. More specifically:

- **Training**: the training procedure should read the training corpus and (1) produce a dictionary containing all the words encountered and the number of occurrences of every word (we shall refer to the occurrences of a word \( v \) as \( C(v) \)); (2) keep track of the number of senses encountered (let this be \( C(s) \)); and (3) count the number of times every word \( v \) appears in the context of a training example labelled by sense \( s \) \( (C(v, s)) \). Once the training phase is over, all these counts should be saved so that the testing procedure can use them.

- **Testing**: given a word \( w \) with \( s_1, \ldots, s_k, \ldots, s_N \) senses, the testing procedure assigns sense \( s \) to \( w \) if sense \( s \) maximises the following expression:

\[
\text{score}(s_k) = \log \frac{C(s_k) + 1}{C(w)} + \sum_{v_j} \log \frac{C(v_j, s_k) + m \cdot 1/V}{C(s_k) + m}
\]

where \( v_j \) ranges over all words in the context of \( w \), \( V \) is the number of words in the dictionary, and \( m \) is the smoothing factor. We will use base 10 logarithm.

Notice that the expression above has been smoothed to improve the results for the cases in which (1) some of the senses of word \( w \) do not appear in the training corpus, and (2) words not seen during training appear in testing.

The testing procedure should also compare the result of the disambiguation (we will refer to this as the *predicted tag*) with the correct sense of the word (its *true tag*). See below for how you can find this tag. The testing procedure should compute the ratio of correctly predicted tags / total number of predictions.

2 Corpus description: SENSEVAL

You will use four types of files, provided by the SENSEVAL project (for more information, go to their webpage http://www.itri.brighton.ac.uk/events/senseval/ARCHIVE/resources.html, however, we have changed some files a bit).
• Files for senses (*.sen). For every word, there is a file containing all the different senses which that word can have. The format of the file is one line for every sense consisting in <sense-id>:<sense-tag>. <sense-id> is a 6-digit code, <sense-tag> is a human-readable tag given to the sense. Your training program should first load this file.

• Files with the training corpus (*.cor). The file is divided into examples, tagged with a 6-digit code 80XXXX. Every example contains exactly one ambiguous word; the context for the ambiguous word will consist of all the words appearing in the example. A typical training example is:

800001
We had reason, therefore, to read the conditions regarding the cemetery. I was <tag "512619">amazed</tag> and saddened to see the many ...

As you can see, the sense for the ambiguous word is denoted by its id in the sequence

<tag "XXXXX">word</tag>

• Files for testing (*.eval). These files are very similar to the training files in format. A typical testing example is

700007
I’ve grown up, basically. Until recently, I was just <tag>amazed</tag> that I was ...

Obviously, the ambiguous word (denoted now by the sequence <tag>word</tag>) does not have a sense (it is your task to find it!). The examples are identified by a 6-digit code 70XXXX.

• Files for evaluating (*.gold). These files contain the true tags of the ambiguous words appearing in the test files. You should compare your results with them to check whether your predicted tag coincides with the true one (so you can do statistics on performance of your system). A typical gold file looks like:

700002:kind
700003:unstint or kind
700005:unstint

Notice that the true tags can contain a disjunction of tags. You should count a “hit” if your predicted tag coincides with one of the tags given in the list of true tags.

3 Data structures

To make your life easier, we provide you with some of the data structures (written in C++) with the appropriate functions. You can find a detailed explanation of their functionality in their header files. The classes are:
• **class dictionary**: here you can keep your count of words \(C(w)\).

• **class senses**: here you can keep your count of senses \(C(s)\).

• **class matrix**: here you can keep your count of word - sense \(C(w,s)\).

4 Filtering

Filtering consists in removing words from the training and testing corpus that provide little information. They are usually called “stop-words”. They are contained in what is known as the “stoplist”. Typically, these words are: also, an, and, do, from, his, etc.

5 The programs

You should write the following three programs:

• **filter <file>**. This program should take the file <file> and produce another in which stop-words have been removed.

• **train <word>**. This program should compute and save the counts \(C(v), C(s), C(v,s)\) for the training corpus corresponding to the word <word>.

• **test <word> <m>**. This program uses the counts computed in the training phase to disambiguate examples in the test corpus file for word <word>. The parameter <m> corresponds to the smoothing factor. The program test should output a line like:

  **Performance for word 'band' with m = 1000 is 77.4% = 234/302**

It should also create a result file with name <word>.result with detailed information about what the predicted tag was, and what its score was (with score we mean the value of the winning sense in the expression written in section 1) for each example. This file should look like (one line for each example):

  700004 :MISS :[predicted] 'mus' :[true] 'brass' :[score] -46.4152

6 Files provided

• Header files: /g/150TP/files/PP2/code/*.h

• Object files: /g/150TP/files/PP2/code/*.o

• Corpus files: /g/150TP/files/PP2/corpus/{*.sen,*.cor,*.eval,*.gold}

• Stoplist: /g/150TP/files/PP2/other/stoplist
7 What to hand in

Please submit your solution electronically to marias@eeecs.tufts.edu before Monday, 26th February 2001, 5pm. You should include in your message:

- Clear, comprehensive documentation on how your programs work.
- Result file generous.res produced in execution of “test generous 100” when no filtering on files for generous is done.
- A file containing the performance achieved for the three words behaviour, bury and generous, with m=1 and m=100, and with filtering/no filtering. In total there should appear 12 lines.
- Code (c++ programs, shell scripts, perl scripts) used in your system.