Assignment #3: Extracting noun groups using machine learning techniques

## **Objectives**

The objectives of this assignment are to:

* Write a program to detect partial syntactic structures
* Understand the principles of supervised machine learning techniques applied to language processing
* Use a popular machine learning toolkit

## **Organization and location**

The third lab session will take place on

* Group 1: Wednesday, September 23 from 10:15 to 12:00 in the Alpha room
* Group 2: Wednesday, September 23 from 10:15 to 12:00 in the Beta room
* Group 3: Wednesday, September 23 from 13:15 to 15:00 in the Beta room

You can work alone or collaborate with another student.

Each group will have to:

* Write and train a machine learning program to detect noun groups.
* Evaluate the results and comment them briefly.

## **Programming**

### **Choosing a training and a test sets**

1. As annotated data and annotation scheme, you will use the data available from [CoNLL 2000](http://www.cnts.ua.ac.be/conll2000/chunking/).
2. Download both the training and test sets (the same as in the previous assignment) and decompress them.   
   (Local copies available here: [[train.txt](http://fileadmin.cs.lth.se/cs/Education/EDAN20/corpus/conll2000/train.txt)] [[test.txt](http://fileadmin.cs.lth.se/cs/Education/EDAN20/corpus/conll2000/test.txt)].)
3. Be sure that you have a running [Weka](http://www.cs.waikato.ac.nz/ml/weka/)system: start it by typing weka. You must use version 3.7.10 or higher. You can increase the memory by typingjava -Xmx2g -jar weka.jar, where the mx parameter corresponds to the maximum heap size.
4. As a refresher on Java maps, i.e. name/value structures or dictionaries, read this program and run it on the Selma corpus [[link to the code](http://fileadmin.cs.lth.se/cs/Education/EDAN20/programs/chunker/src/ora/WordSort.java)].   
   The code is taken from Patrick Niemeyer and Daniel Leuck, Learning Java, 4th edition, 2013, O'Reilly, page 392. [[link to the book](http://shop.oreilly.com/product/0636920023463.do)].

### **Baseline**

Most statistical algorithms for language processing start with a so-called baseline. The baseline figure corresponds to the application of a minimal technique that is used to assess the difficulty of a task and for comparison with further programs.

1. Read the baseline proposed by the organizers of the [CoNLL 2000 shared task](http://www.cnts.ua.ac.be/conll2000/chunking/) (In the *Results* Sect.). What do you think of it?
2. Implement this baseline program. You may either create a completely new program or start from an existing program that you will modify [[Program folder](https://github.com/pnugues/ilppp/tree/master/programs/labs/chunking/chunker_java_ml/src)]. In this case:
   1. Modify and use the BLChunkerIncomplete program to read the training set of CoNLL 2000 and collect the most frequent chunk tag for each part-of-speech tag. You will have to complement the countAssoc() andselectBestAssoc() methods. You will also have to modify the paths in Constants.java so that they match your folder organization.
   2. Complement the tag() method to assign a part-of-speech value its most frequent chunk tag.   
      Apply it to the test set and save the result in a result file.
3. Measure the performance of the system. Use the [eval.sh](https://github.com/pnugues/ilppp/tree/master/programs/corpus/conll2000)script and the [conlleval.txt](http://www.cnts.ua.ac.be/conll2000/chunking/)evaluation program used by the CoNLL 2000 shared task.

### **Replicating the baseline with decision trees**

In this exercise, you will replicate the baseline experiment with a popular machine-learning toolkit: [Weka](http://www.cs.waikato.ac.nz/ml/weka/). Weka is written in Java and comes with many ready-to-use algorithms.

You will not write programs, but merely understand the overall process involving a machine-learning toolkit. You will train and apply models using the Java code supplied in the previous section.

1. Weka uses a parameter file to learn a model. The parameter data are described by a header in the form of attributes and values. Download the [chunk\_header.arff](https://github.com/pnugues/ilppp/tree/master/programs/corpus/conll2000)header and modify it to fit the baseline experiment. Do you need the word attribute to replicate the baseline?   
   The header is hard-coded in the Java program.
2. Read and run the MLChunker class from this [[program folder](https://github.com/pnugues/ilppp/tree/master/programs/labs/chunking/chunker_java_ml/src)] to create a Weka-compatible training file from the training set:
   * Reuse the Java program to load the training file and read the lines.
   * The Java program extracts two arguments: The part of speech and the chunk type from each line and write them in an ARFF file. The fields are separated by a comma.
3. From your ARFF training file, learn a decision tree using Weka and produce a model.
   * Load your ARFF file by selecting the *Preprocess button*
   * Choose and create a classifier by pressing the *Classify* button and then the *Choose* button. Use the **J48** decision trees.
   * Save the model by right-clicking on the item in the *Result list*
4. Embed your model in the Java program and measure its efficiency.
   * Read the tag() method in MLChunker.java and WekaGlue.java to have Weka predict the chunk tag of your feature vector.
   * Add the weka.jar library to your program. You will find it in the /usr/local/cs/EDAN20 folder or you can download it from the Weka site. (Use version 3.7.10 or higher).
   * Run MLChunker to read the words and parts of speech of each line of the test set and predict the chunk tag. The input to this predicate is the list of (Word, POS) pairs. The predicate will determine the chunk type and write the results in the CoNLL format: Word POS Chunk.
   * Evaluate the results using the evaluation script.

### **Using more features**

In this exercise, you will extend the MLChunker program using more features. You will start from the Java program you downloaded and add one or two more features so that you improve the performance of your chunker.

The program that won the CoNLL 2000 shared task (Kudoh and Matsumoto, 2000) used a window of five words around the chunk tag to identify, *c i*. They built a feature vector consisting of:

* The values of the five words in this window: *w i-2*, *w i-1*, *w i*, *w i+1*, *w i+2*
* The values of the five parts of speech in this window: *t i-2*, *t i-1*, *t i*, *t i+1*, *t i+2*
* The values of the two previous chunk tags in the first part of the window: *c i-2*, *c i-1*

The two last parameters are said to be dynamic because the program computes them at run-time. Kudoh and Matsumoto trained a classifier based on support vector machines. Read [Kudoh and Matsumoto's paper](http://www.cnts.ua.ac.be/conll2000/pdf/14244kud.pdf) and the [Yamcha](http://www.chasen.org/~taku/software/yamcha/) software site.

1. What is the feature vector that corresponds to the baseline?
2. Extract a set of features from the training set to improve the baseline. Use for instance ( *t i-1*, *t i*) or better, ( *t i-1*, *t i*, *t i+1*). In the beginning and end of the sentence, the feature vector will overflow as there is no *t -1*. Use padding codes such as BOS and EOS (beginning of sentence, end of sentence)
3. Train and embed your classifier in your Java program and measure its efficiency on the test set. You need to reach a global accuracy F1 score of 81 to pass this laboratory.