Comparing methods for Coreference solving

By Jonas Henrikson, 2006

Abstract

In 2005, Magnus Danielsson created a coreference solver for a project called Carsim. The purpose of this project was to automatically analyze news reports about traffic accidents, extract the relevant details of the events in an accident, and recreate a simulation of these events. In analyzing the news reports, an important part of the program is the ability to understand when seperate references to e.g. a car or a person involved in the accident are actually references to the same car or person; i.e. the ability to understand when coreference occurs in the text. In the coreference solving program by Magnusson, the method used to make final decisions of whether two nouns (or, more precisely, two noun phrases) are indeed coreferential was the ID3 decision tree algorithm. The purpose of my project is to generalise the code to make it possible to substitute the ID3 algorithm with other solver methods, so as to be able to compare the efficiency and accuracy of different methods.

Original program structure

Instead of changing the coreference implementation in the actual Carsim program, I used an evaluation version of the coreference solver. This evaluator selects a number of random documents from a collection of accident reports, performs the coreference calculations, and compares the results to manually identified coreferences. The final results are shown as three percentages: recall, precision and the so-called f-measure. Recall is the percentage of how many of the actual coreferences are identified by the solver, precision is the percentage of identified coreferences that actually *are* coreferences, and the f-measure is a sort of mean value calculated from the first two percentages.

The evaluator includes a lot of code for identifying noun phrases and their grammatical features, etc, but fortunately most of these processes are carried out before the final result of whether a pair of noun phrases are coreferential or not is calculated. Not that much of the code was originally dependent on the decision tree algorithm, and most of the code concerning the ID3 solver method were contained in a seperate directory: the "decisiontree" packade. My first step was to identify what segments of code outside the decisiontree package were specifically dependent on the ID3 solver.

First of all, the Evaluator class is where most of the central work is done, and this is where the ID3 object was created. Each document used in the evaluation is represented in the Evaluator by an instance of the CorefDocument class, and since this is where the information about the noun phrases are stored as private vectors, and since these vectors are the data to be "analyzed" by the decision tree algorithm, the Evaluator sent the ID3 object to each CorefDocument object which in turn used a TreeExecutor object to execute the desicion tree solver with regard to the noun phrases in the document. The TreeExecutor class was part of the decisiontree package, and this class is where the final call to the ID3 algorithm occured, to decide whether a given pair of noun phrases actually corefers or not.

So, in short, the only classes outside the decisiontree package which contained specific uses of classes inside the decisiontree package were the Evaluator and CorefDocument classes.

My changes – Step 1: Generalisation

What I did first was to make the CorefDocument independent of the decisiontree package. Instead of having a method in the CorefDocument class to which the Evaluator sent the ID3 tree, and which used the document's private noun phrase vectors and the TreeExecutor class, I added simple "get"-methods making it possible for the Evaluator to access the otherwise private noun phrases, so that the uses of decisiontree package classes could be contained in the Evaluator class. (See the appendix for specific examples of this and other changes to the code.) By doing this, I could move the use of the TreeExecutor class to the Evaluator, and hence the CorefDocument class became completely free of ties to the decisiontree package.

Next, I discoreved that a couple of the classes contained in the decisiontree package were actually not specifically dependent on any of the other decisiontree package classes, and therefore I moved them out of the package and put them among the rest of the standard evaluator classes. These were the NPFeatureTransferer and ApplyHardConstraints classes. Only one line was changed in each of these files: the first line of the file saying which package the classes were part of.

Then I looked closer at the TreeExecutor class and noticed that this class was only minimally dependent on the decision tree solver method. What was happening at this point in the changes being made was that the Evaluator created an ID3 tree and a TreeExecutor, sent the ID3 tree to the TreeExecutor, and the Treeexecutor at one (and only one) point in the code asked the ID3 object whether a pair of noun phrases coreferred or not. What I did here was to create a simple interface called CoreferenceSolver, and a simple class called TreeSolver which implements the CoreferenceSolver interface. I moved the creation of the ID3 object to the TreeSolver class, and I changed the TreeExecutor class so that instead of using the ID3 tree directly, it now used the CoreferenceSolver inferface. By doing this, the TreeExecutor class became fully independent from the decision tree solver method, and because of this I thought it to be apropriate to change the name of this class from TreeExecutor to SolverExecutor, and to move it out of the decisiontree package and put it among the other standard evaluator classes, just as with the NPFeatureTransferer and ApplyHardConstraints classes mentioned above.

Now there was only one line left in the Evaluator class that specifically used the decision tree solver method: the line where the TreeSolver object was created (which of course was represented by and used as a CoreferenceSolver object in all other places). No other files outside the decisiontree package now had any ties to the ID3 solver method, except the new and simple TreeSolver class. Generalisation complete!

My changes – Step 2: Implementing another solver method

The alternative solver method I implemented was an SVM classifier. Only two things were needed to make this work. First, a new class implementing the CoreferenceSolver interface had to be created, which I naturally called SVMSolver, and second, the one line in the Evaluator class which created a TreeSolver object was changed to instead create an SVMSolver object. The only issue here was the difference in format of the input to the two solver methods. While the ID3 solver method used two seperate vectors to build the decision tree – one with positive examples of coreferences and one with negative examples – the SVM solver method used only one vector to build the SVM classifier. This one vector had the same content as both of the positive and negative examples combined, but with a boolean value in the beginning of each example to flag it as positive or negative. Also, while the ID3 solver used standard Vector objects, the SVM solver used ArrayList objects. I decided to handle this type conversion in the constructor for the new SVMSolver class, so that nothing else needed to be changed in any other class. Once this type conversion was implemented, the SVM solver method fit perfectly into the program and could be tested and evaluated in exactly the same way as the original ID3 solver method.

Results / Evaluation / Conclusion

To compare the two methods of coreference solving, I ran two Evaluations of each method. One evaluation actually contains four iterations of the whole process, and shows a total result. The resluts from these evaluations are as follows.

	Recall	**Precision**	**F-measure**
Total results for **ID3**, 1'st and 2'nd run:	*86.1%*	*81.6%*	*83.8%*
Total results for **SVM**, 1'st run:	*86.1%*	*79.6%*	*82.7%*
Total results for **SVM**, 2'nd run:	*85.3%*	*79.8%*	*82.4%*

Interestingly, the results from the two ID3 evaluations were identical, even though the documents used are randomly selected. I have no explanation for this. As you can see, the SVM solver method can be equal to the ID3 solver method in terms of recall, although it shows bigger variation in this value. When it comes to precision, however, the ID3 solver takes the lead. This 2% drop in precision on the SVM's part might look like very slight, but this difference becomes much more striking when formulated as a 10% increase in incorrectly identified coreferences.

Philosophical comment on coreference

In the philosophy of language, coreference remains a problem in the sense that it is difficult to explicate just how coreference works and is comprehended by language users. However, it is not a central problem. The question of what coreference is is not as problematic as the question of how it works, and the latter question is of much bigger interest in computational linguistics. Up to and including my candidate course in theoretical philosophy, coreference has hardly been mentioned as a difficulty, while on the other hand you are probably already aware that coreference is a very difficult problem in computational linguistics. Apparently, there is a difference in how this issue is viewed within these two disciplines. In philosophy, it is generally thought that the comprehension of coreference is highly dependent on a wide base of knowledge about the world. In computational linguistics, by contrast, one usually tries to find a solution by focusing on grammatical features and basic semantic category information. Some mean that it is important to realize that while this approach is perfectly legitimate, a big part of the problem remains unsolved.

Appendix – Relevant code segments

It is recommended that you are somewhat familiar with the source code when reading this appendix.

Originally, the ID3 tree was created in the Evaluator class with these three lines:

```
ID3 id3 = new ID3(yesVector, noVector, "true", "false");
String generatedTree = id3.getResultTree();
DTreeNode treeInMemory = id3.getRootNode();
```

With the creation of my new CoreferenceSolver interface and TreeSolver class, these lines were removed and replaced by the following line. This line is the only thing that needs to be changed in the Evaluator class if one wants to switch solver methods.

```
CoreferenceSolver coreferenceSolver = new TreeSolver(yesVector, noVector);
```

To use the SVM solver instead, one would simply change the above line to this:

```
CoreferenceSolver coreferenceSolver = new SVMSolver(yesVector, noVector);
```

This method was removed from the CorefDocument class:

```
public void addToEvaluation(EvalMaker localEv, EvalMaker globalEv, DTreeNode treeInMemory) {
            TreeExecutor t = new TreeExecutor(nounPhraseFeatures, treeInMemory);
            Vector postiveChains = t.getPositiveChains();
            localEv.addValues(chainsAsInts, postiveChains);
            globalEv.addValues(chainsAsInts, postiveChains);
}
```

In its stead, these two methods were added:

```
public Vector getChainsAsInts() {
            return this.chainsAsInts;
}
public Vector getNounPhraseFeatures() {
            return this.nounPhraseFeatures;
}
```

Because of these changes to the CorefDocument class, this line was removed from the Evaluator class:

```
doc.addToEvaluation(localEvaluator, globalEvaluator, treeInMemory);
```

...and was substituted by the following lines. Note that these are very similar to the contents of the removed addToEvaluation method shown above.

```
Vector nounPhraseFeatures = doc.getNounPhraseFeatures();
SolverExecutor solverExec = new SolverExecutor(nounPhraseFeatures, coreferenceSolver);
Vector chainsAsInts = doc.getChainsAsInts();
Vector postiveChains = solverExec.getPositiveChains();
localEvaluator.addValues(chainsAsInts, postiveChains);
globalEvaluator.addValues(chainsAsInts, postiveChains);
```

Here is the very simple new CoreferenceSolver interface:

```
public interface CoreferenceSolver {
  public boolean corefers(Vector features);
}
```

And here is the TreeSolver class which implements it. Again, note the similarities with the first three lines mentioned in this appendix, which were removed from the Evaluator class.

```
public class TreeSolver implements CoreferenceSolver {
            private ID3 id3;
            private DTreeNode root;

            public boolean corefers(Vector features) {
                        return ExecutingTree.corefers(features, root);
            }
            public TreeSolver(Vector yesVector, Vector noVector) {
                        this.id3 = new ID3(yesVector, noVector, "true", "false");
                        this.root = id3.getRootNode();
            }
}
```

The new SVMSolver class is a bit too big to be included in this appendix, while other changes are too small to deserve mention. Please refer to the source code for further code reading pleasure.