# Coreference Resolution*

Programming Assignment #3

Due: Wednesday, November 6th at 5pm

**IMPORTANT NOTE #1**: You may complete this assignment individually or in pairs. *We strongly encourage collaboration*. Your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the course website: `http://cs224n.stanford.edu/grading.html`.

**IMPORTANT NOTE #2**: Please read this assignment soon. Go through the Setup section to ensure that you are able to access the relevant files and compile the code. Especially if your programming experience is limited, start working early so that you will have ample time to discover stumbling blocks and ask questions.

## 1 Overview

In this assignment, you will implement two coreference resolution systems. The first system relies on hand-written rules while the second is based on a discriminative statistical classifier. The second model requires feature engineering just like PA1.

## 2 Setup

We've put everything for this assignment in the directory `/afs/ir/class/cs224n/pa3/`. The starter code can be found in `/afs/ir/class/cs224n/pa3/`. Copy over the starter code to your local directory and make sure you can compile it without errors:

```
cd
mkdir -p cs224n/pa3
cd cs224n/pa3
cp -r /afs/ir/class/cs224n/pa3/java .
cd java
ant
```

The data for this assignment is located in `/afs/ir/class/cs224n/pa3/data`. You can open up the *.dat files in a standard editor to inspect the contents, but in general it is easier to use the provided `view` script. This will print out a document with the mentions highlighted according to the coreferent cluster they correspond to. The script accepts either a training document index or a document id (without the *.dat extension) to search for a document in either the training or dev sets. For example

```
./view 7
```

will open the 7th training document and

```
./view wb.a2e.00.a2e_0025
```

will open the corresponding document in the dev set.

The most important class for this assignment is `cs224n.assignments.CoreferenceTester`. The `main()` method of this class takes several command line options. You can try running it with:

```
java -Xmx500m -cp "extlib/*:classes" cs224n.assignments.CoreferenceTester \
-path /afs/ir/class/cs224n/pa3/data/ \
-model baseline -data dev -documents 100
```

*Revision: October 23, 2013

This command will run a baseline model, which is a simple exact-match heuristic. Two mentions are marked as coreferent if their string representations match exactly. `CoreferenceTester` takes a number of command line parameters. These include:

- `-data {dev,test}` : Determines the test set to use. You should debug your code on the dev set, and report final results (and only final results) on the test set.
- `-documents [1, inf)` : Determines the number of training documents to use. The default value is 100 documents; you should make sure to report results with this default value.
- `-mistakes [1, inf)` : Print debug information on the mistakes your system made on the first $n$ dev set examples.
- `-mentionExtractor {gold, predicted}` : Determines whether to use gold (human-marked) or predicted (machine-marked) mentions. The default is gold.
- `-path`: The path to the data directory.
- `-model`: The class name of the coreference system you want to run (e.g. `RuleBased`).

The systems you will write implement the `CoreferenceSystem` class and can be found under the `cs224n.corefsystems` package. Please write your code in the provided .java files so we can run our scripts on your classes. To implement a `CoreferenceSystem` class, you must fill in the following methods:

- `train(Collection<Pair<Document, List<Entity>>> trainingData)` : Trains the model using the supplied collection of documents and their gold entity clusters. The list of mentions are stored in the document; the gold coreference clusters are stored in the `List<Entity>` object. Look at the implementation in the `baseline` model for a simple example of how to iterate over the documents.
- `runCoreference(Document doc)` : Runs your coreference system on the given document. The method returns a list in the form of a `List<ClusteredMention>` object in which every mention in the document is tagged with the Entity it refers to. Note that every mention must be tagged with an entity; that is, every mention belongs to some cluster even if it is a singleton cluster consisting of only itself.

You are given a number of classes for storing and manipulating the data. These are all in the `cs224.coref` package. They are:

- `Document` : Encapsulates a document you are to run coreference on. It includes the sentences in the document, as well as the annotated mentions (which you will cluster into coreferent mentions).
- `Entity` : Denotes a real-world entity, as defined by a set of mentions.
- `ClusteredMention` : Denotes a mention which is tagged with the entity it refers to.
- `Mention` : Denotes a mention. This class stores much of the useful information from which you would extract rules and features. It includes the text for the mention, along with its location in the sentence, its parse tree, and named entity type (e.g., person, place, organization, etc.).
- `Sentence` : Denotes a sentence. A document is a collection of sentences; each mention is marked as part of both a document and a sentence.

You should be careful modifying the interfaces (method signatures) of these classes since they are serialized and deserialized. You can safely modify the bodies of the methods if needed.

In addition, you are provided with some useful utilities. Most of these are methods in one of the above classes, and are documented there. However, some particularly useful tools are:

- `Document.prettyPrint(Collection<Entity>clusters)` : Prints out a document in a structured form, marking coreferent phrases. If your terminal supports colors, phrases will print in color; otherwise each mention will be tagged with the unique ID of its entity.
- `Entity.orderedMentionPairs()` : Returns an `Iterable` object which cycles over every pair of mentions for the entity. For example, if an entity has mentions A,B and C, it returns:
{(A,B), (A,C), (B,A), (B,C), (C,A), (C,B)}.

# 3 Coreference Systems

You should implement a two naive baseline systems, a better baseline, a sophisticated rule-based system, and a classifier-based system.

## 3.1 Your job: Naive Baselines

You should familiarize yourself with the framework code and the debugging tools. Then you should implement two basic systems to better understand how the performance of coreference systems is measured The first system should assign every mention to a single entity. The second system should assign each mention to its own cluster (i.e., a **singleton**).

Notice the behavior of the MUC score on the two baselines: is it what you would expect?[1] Is it similarly bad for both baselines (and should it be)? In your future experiments, you should keep in mind that the different scoring methods behave differently.

## 3.2 Your job: Better Baseline

You should implement a better baseline system to familiarize yourself with the task and the code framework. Feel free to experiment with different methods for clustering, which can optionally be guided by statistics you gather from training. These will also be useful in your actual system, either as a basis for your rule-based system, or as insight for features to a classifier.

An example of a baseline system is a head-matching system: At training time the system could accumulate statistics on the heads of the parses of mentions, and remember which heads are coreferent with each other. This will gather information like *the leader* and *the president* are coreferent, if *leader* and *president* were heads of coreferent mentions at training time.

You can also consider handling pronouns separately (I, she, etc. likely deserve special attention), modeling distance, or a number of other ideas.

## 3.3 Your job: Rule-Based

Rule-based systems are surprisingly effective for coreference resolution.[2] Rule-based systems have at least two general advantages in comparison to vanilla machine learning approaches:

1. Rule-based systems can often encode sophisticated linguistic phenomena more effectively than a classifier-based approach. For example, if a feature is known to be true, but the data are too sparse to support it (or to not overfit to it).

2. Rule-based systems can have a more sophisticated control flow. Particularly, tricks like making multiple passes over the data are much easier to incorporate in a rule-based system.

Ignoring the `train()` method (or using it to collect statistics), implement a rule-based system for coreference resolution. As a first approximation, you should implement simple heuristics, like the baseline. Eventually you should implement a more structured model. One possibility is to run the document through the system multiple times, slowly moving from high precision (e.g. exact match, head match) to low precision (e.g. gender agreement) rules.

In your rule-based system, you should make use of both of the advantages mentioned in the previous section. You should include at least some rules which would not be effective features in a classifier, and justify why this is the case. You should also implement some form of sophisticated control flow, and justify your decision. Some examples of rules or types of rules you could use include the following. You should, however, also implement different and/or more rules than are mentioned here.

- Explore the behavior of pronouns in coreference; are there rules for what kinds of things they should be coreferent with?

- Explore consistency with named entity types (e.g. people are rarely coreferent with places)

- Implement Hobbs algorithm

[1] Skim the MUC score description at `http://acl.ldc.upenn.edu/M/M95/M95-1005.pdf` and try to predict what will happen.
[2] `http://nlp.stanford.edu/pubs/conllst2011-coref.pdf`

- Explore behavior of quoted text; try identifying the speaker and using that information.
- Explore gender coherence (*he* can be coreferent with *him* but not *her*).

#### 3.3.1 Evaluation

For reference, our solution obtained a MUC F1 of 0.749 and a B3 F1 of 0.705 on the test set. Any solution in the ballpark of these figures should be fine.

When reporting results, you should justify your design decisions carefully and support your rules with linguistic theory and a favorable effect on the results. Since there is less of a notion of correct for your choice of system here, a proportionally greater part of your grade will come from (a) **the quality of your final results**, and (b) **the justification of your system and your error analysis**.

### 3.4 Your job: Classifier-Based

In this part you will be implementing a statistical classifier-based system.[3] As input your classifier will accept a set of positive and negative examples of coreferent pairs. The positive examples are pairs consisting of a mention and the previous mention it is coreferent with (if any), tagged as being coreferent. If it is coreferent with multiple previous mentions, the default code takes only the most recent one (though feel free to experiment with taking all of them!).

The negative examples are in turn pairs consisting of a mention and a previous mention that it is *not* coreferent with, tagged as being not-coreferent. The strategy that most people have found to work best is to use as negative examples only intervening mentions between two coreferent mentions (and this is what the starter code in `ClassifierBaseline.java` does by default), but feel free to try out alternatives as to which mentions you use as negative examples.

Formally, for a mention $m_i$ which is coreferent with every mention $m_j$ for $j \in J$, we construct training examples:

$$\begin{array}{ll} ((m_i, m_j), \text{ true}) & \text{closest } j \in J, j < i \\ ((m_i, m_k), \text{ false}) & \forall k \notin J, j < k < i \end{array}$$

At inference, we then look backwards through the mentions in the document to search for another mention which is classified as coreferent. If no mentions are classified as coreferent, a new cluster is created.

For example, consider the document with mentions:

$$\langle A_1, A_2, B_1, A_3 \rangle$$

.

The letter denotes the true cluster that mention belongs to, and the number indexes the particular mention in that cluster. Note that these mentions do not come clustered though; $A_1$ might look like *The President* and $A_2$ might look like *Obama*. We define our training data:

$$\begin{array}{l} ((A_2, A_1), \text{ true}) \\ ((B_1, A_2), \text{ false}) \\ ((B_1, A_1), \text{ false}) \\ ((A_3, B_1), \text{ false}) \\ ((A_3, A_2), \text{ true}) \end{array}$$

We would train a classifier on this data. If we were to then evaluate (the same) document, we would make the following decisions:

| **Mention** | **Coreferent With** | | | | **Conclusion** |
|---|---|---|---|---|---|
| | $A_3$ | $B_1$ | $A_2$ | $A_1$ | |
| $A_1$ | - | - | - | - | *New cluster A* |
| $A_2$ | - | - | - | yes | *Cluster into A* |
| $B_1$ | - | - | no | no | *New cluster B* |
| $A_3$ | - | no | yes | N/A | *Cluster into A* |

[3] The classifier is a maximum entropy (maxent) model (also known as a softmax or multiclass logistic regression classifier). We'll cover maxent later in the quarter.

### 3.4.1 Code Overview

All the code for building the classifier is provided for you except the feature set which your classifier will use. You will notice that one feature has already been added in the classifier for you. The feature, entitled `ExactMatch`, adds a boolean feature indicating whether the two mentions currently being compared are exact string matches.

The first thing that was done to add this feature to our classifier was to create an `ExactMatch` class that exists in the `Feature.java` file. Note that `ExactMatch` extends the `Indicator` class, which is used to represent features which have boolean values. You will notice that there are a variety of other `Indicator` classes provided for you that can be used for features with other value types. For instance, if you wanted to represent the number of words between two mentions, you would create a class which extends the `IntIndicator` class. These `Indicator` types are ultimately used to count features using the `Counter` class, so for instance, if you find that two mentions are exactly the same string, the count of the key `ExactMatch(true)` will be incremented, whereas if two mentions are not the same, the count of the key `ExactMatch(false)` will be incremented. As you can see, the hashcode created by each `Indicator` class is simply a combination of the name of the feature class and the value of that feature. Thus, if you wanted to implement your own `Indicator` you should be sure to implement both the `hashCode` and equals functions. In most cases, your actual feature class will merely pass in an appropriate value to its corresponding `Indicator` superclass, just as the `ExactMatch` class does.

Once you have created the class for your feature, you should add it to the `ACTIVE_FEATURES` constant in the `ClassifierBased` class. As you might have noticed, only features which have been added to the `ACTIVE_FEATURES` list will be used by the classifier, so an easy way to test subsets of features would be to comment out the feature classes which you do not want the classifier to use.

Finally, you will implement the code for getting the value of a particular feature in the feature method inside the `ClassifierBased` class. You will notice that in the case of the `ExactMatch` feature, the only computation that needed to be done was passing in the value of the Java provided `equals()` method on the gloss of the two mentions. Note that each time the feature method is called, a new feature is being examined, and thus all feature checks should be surrounded by an else if statement to insure that the code for computing the value of the feature is being performed only when the particular feature is being checked.

For those wanting to create features that examine a pair of other features, you will notice that in the `fillFeatures` method, features which are an instance of `Pair` will cause the feature extractor to extract two separate features which are then merged into a single feature. The skeleton for how to create a pair feature is shown as a comment in the `ACTIVE_FEATURES` array.

In summary the steps for adding a new feature to the classifier are as follows:

1. Create a class for the feature in the `Feature` class. This might also require creating a new `Indicator` class for the feature.
2. Add the class to the `ACTIVE_FEATURES` constant in the `ClassifierBased` class.
3. Implement the code for obtaining the value of the feature in the feature method inside the `ClassifierBased` class.

### 3.4.2 Feature Engineering

Your only job in this portion is to add features to the classifier. As a first approximation, you can include the features from your baseline system. From there, possible feature ideas include:

- Distance between the mentions (in terms of other mentions)
- Distance between the mentions (in terms of number of sentences)
- Paths through the sentence parse (if same sentence)
- Whether the fixed or candidate mention is a pronoun
- The named entity type of the candidate mention
- Combinations (cross product) of any subsets of features

You do not need to implement all of these features; they are rather meant to be possible ideas of the types of things you could be looking at.

#### 3.4.3 Evaluation

For reference, our solution obtained a MUC F1 of 0.701 and a B3 F1 of 0.685 on the test set. Any solution in the ballpark of these figures should be fine.

You will be evaluated on the features you select and the performance of your system. You should implement features other than the ones provided above for full credit. However, the features you add should be justified in terms of the performance of the system. Error analysis should include **system performance on different feature sets**, as well as **characterizing systematic mistakes the system makes**. You should describe how these features would be fixed, either with more features (and if so, what would they be?), or note that they cannot be easily captured in our framework.

# 4 Administrative Information

## 4.1 Extra Credit

We will give up to 10% extra credit for innovative work going substantially beyond the minimal requirements:

- Substantial extra processing of the data (e.g., applying POS taggers or syntactic parsers) and extraction of features from these annotations.
- Substantial application of linguistic knowledge (far beyond expectations) to the classifier and/or the rule-based systems.
- Read a research paper on coreference resolution, summarize it briefly, and apply the main results to your systems. The Stanford NLP Group has published many papers on coref![4]

## 4.2 Grading Criteria

You should have three functional models: a baseline heuristic system, a rule-based system, and a classifier-based system. Solid implementations of these models, along with a good write-up with interesting data analysis, are required to earn a full grade. There is a **hard limit of 4 pages + 1 page for extra credit**. *We may deduct 10% for each page that exceeds this limit.*

You should report results on the development and test sets, but **you should run your models on the test set only once**. Any tuning should be done on the development set; if performance is worse on the test set you should explore why this might be in your write-up, but **you should not rerun your system**.

Your write-up should include:

- A brief discussion of the naive baseline experiments and a short description of your better baseline system. (Should not exceed half a page)
- A basic overview of the rule-based and classifier-based models and brief description of how each of them works.
- A clear presentation of the rules, algorithms and features you used and alternatives you tried.
- A table showing the results of your rule-based and classifier-based systems. Include the following table with your results on the development and test sets:

| | MUC | | | B3 | | |
|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | Precision | Recall | F1 |
| Rule-based | | | | | | |
| Classifier-based | | | | | | |

- A discussion of the motivations for choices that you made and a description of the testing that you did.

[4] http://nlp.stanford.edu/publications.shtml

- Insightful commentary on what kinds of things your models get right and get wrong, and possible reasons why. Remember we are looking for careful and thorough error analysis to be carried out during the process of improving your model, rather than being something tacked on at the end. Some of the things you might consider include:
  - What obstacles did you come across in trying to get your system's performance to reasonable levels? What sorts of phenomena were hard to express in your framework?
  - What phenomena did one system do a better job than the other of capturing?
  - What features or rules were most useful? Least useful? Most surprisingly useful/not useful?
  - What sorts of mentions were hard to capture? Were these cases that could be captured with more features? Could they be captured with more sophisticated systems? Are they inherent in the task?
- Ideas as to how to best improve the system, based on the above.

### 4.3 Submission Instructions

Please note:

- **We only accept electronic submissions**. *Do not submit a paper copy of the report*.
- If you are working in a group, then **only one member of the group needs to submit the assignment**.

You will submit your program code using a Unix script that we've prepared. To submit your program, first put your files in one directory on Farmshare. This should include all source code files, but should not include compiled class files. **Put your report PDF in the root of your submission directory**. Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```

This script will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to re-submit your assignment, you can run:

```
/afs/ir/class/cs224n/bin/submit -replace
```

We will compile and run your program on Farmshare using `ant` and our standard `build.xml` to compile. If you did not complete the assignment on Farmshare, then please verify your code compiles and runs on it before submission. If there's anything special we need to know about compiling or running your program, please include a `README` file with your submission.