

Introduction to BART

May 7, 2008

1 General Introduction

BART, the Beautiful/Baltimore Anaphora Resolution Toolkit, is a tool to perform fully automatic machine-learning based automatic coreference annotation on written text. This section will provide a friendly introduction to the system from a user's perspective.

The system stores all vital information on documents in the token-based stand-off format of MMAX2; it uses the MMAX2 discourse API¹ for this purpose.²

In the standard configuration, only tokenisation is needed, and other steps are performed automatically by suitable components (sentence splitter, part-of-speech tagger, chunker/parser, and named entity recognizer). For learning a new classifier or quantitative evaluation, it is necessary to have gold standard coreference information on a separate markable level.

To run a basic system, you need to have the following external components installed:

- the YamCha chunker and the YamCha model collection (for the chunker-based pipeline)

<http://chasen.org/~taku/software/yamcha/>
(where do the chunking models come from?)

YamCha uses an external SVM package to perform its classification; possible candidates are TinySVM and SVMLight

<http://chasen.org/~taku/software/TinySVM/>
SVMLight/TK, which is a downwards-compatible extension to SVMlight, can also be used as a learner in the coreference resolution.

- Charniak and Johnson's reranking parser
<ftp://ftp.cs.brown.edu/pub/nlparser/>

Other recommended external components include

¹see <http://mmax2.sourceforge.net>

²replace with MiniDiscourse for next release, and mention tokenisation tools somewhere

- SVMlight/TK with java native interface: This allows the use of SVM for classification tasks and the use of tree-valued features. The use of the native Java interface is recommended for improved speed.
- The Carafembic ACE mention tagger performs general mention tagging for ACE mentions. Its use improves the accuracy when using ACE-style corpora in which only ACE mentions (persons, organizations, geopolitical entities, ...) are marked up.

The *preprocessing pipeline* invokes sentence splitter, part-of-speech tagger, chunker and named-entity recognizer and uses this information to tag mention markables (on the *markable* annotation layer of the MMAX2 document). Once documents have been preprocessed, the preprocessing information in the MMAX documents can simply be reused and preprocessing switched off. This is especially convenient when doing repeated experiments on a single dataset.

1.1 Installation and Getting Started

This section will lead us through the steps necessary for running training and testing phases on the MUC6 corpus.

1.1.1 Running without preprocessing

1. In the directory `config`, make a copy of the file `config.properties.sample` and name it `config.properties`.

The `config.properties` file contains configuration options that usually depend on the local system configuration, such as the directories where training/testing data, needed programs, etc. reside.

If you unpacked the MUC sample files into `/path/to/MUC-MMAX`, then you need to set the `trainData/testData` options as follows:

```
trainData = /path/to/MUC-MMAX/muc6/train
trainDataId = MUC6
testData = /path/to/MUC-MMAX/muc6/test
testDataId = MUC6
```

In the run we want to do now, we don't need to run the preprocessing, as the MUC files are already in MMAX format and conveniently preprocessed:

```
runPipeline=false
```

MUC6 marks coreference even outside the main document body, which is why we want to use the mention creation process that uses mentions from the whole document:

```
mentionFactory=elkfed.coref.mentions.FullDocMentionFactory
```

2. To compile the BART sources, we need to have (i) a working JDK (version 5.0 or up) and (ii) Apache Ant³; we also need to setup the classpath so that external libraries (which are part of the BART package) can be found.

First, edit the `setup.sh` file so that `JAVA_HOME` points to the directory where your Java installation is. You then need to source the file with⁴:

```
bash$ source setup.sh
```

We then run `ant` to compile the whole thing:

```
bash$ ant jar
```

We can then use `XMLExperiment` to perform both training and testing⁵:

```
bash$ java -Xmx1024M elkfed.main.XMLExperiment
```

or we can use `XMLTrainer` to create the training data, run `XMLClassifierBuilder` to perform model learning and then use `XMLAnnotator` to test separately:

```
bash$ java -Xmx1024M elkfed.main.XMLTrainer
(lots of output omitted)
bash$ java -Xmx1024M elkfed.main.XMLClassifierBuilder
(some output omitted)
bash$ java -Xmx1024M elkfed.main.XMLAnnotator
(lots of output omitted)
```

1.1.2 Running with the parser pipeline

To try out some preprocessing, we will first use the ACE-02 sample file that is in `sample/ACE-02`⁶. To do this, we first change the `testData` configuration entry in `config.properties`:

```
testData=./sample/ACE-02
```

³available at <http://ant.apache.org/bindownload.cgi>

⁴this only works with bash. Users of other shells such as `tsh` will have to adapt this.

⁵the option `-Xmx1024M` is used to allocate more heap space for the Java process. If your computer does not have enough memory, or Java runs into memory problems, you have to adjust this number

⁶This is nonsense from an evaluation point of view, as the ACE and MUC annotation schemes differ considerably. But as preprocessing the whole MUC6 corpus would take longer, we'll just have fun with the sample file.

We then need to change the options so that (i) preprocessing is activated, (ii) the Charniak parser is used and (iii) the directory where the Charniak parser is located is known to the system:

```
runPipeline=true
pipeline=elkfed.mmax.pipeline.ParserPipeline
parser=elkfed.mmax.pipeline.CharniakParser
charniakDir=/path/to/the/reranking-parser
```

To be able to use the Charniak parser, we also need to replace the `parse.sh` script in the `reranking-parser` directory with our modified version.

We can then run `XMLAnnotator`, which uses the model we trained on the MUC data (*again, this is not useful for any serious purpose, but we want to try out the pipeline*) on the corpus that will be run through the preprocessing pipeline for us.

1.2 Additional Configuration

The `config.properties` file in the `config` directory contains a few more settings that influence the behaviour of the system:

- the option `mentionFactory` indicates the name of the class used for creating the mention objects from MMAX markables, which can be used to influence the set of mentions that are created and can then be linked.

Currently, the following `MentionFactory` subclasses exist⁷:

- `FullDocMentionFactory` creates mentions for every markable on the *markable* annotation layer
- `DefaultMentionFactory` creates mentions for every markable that is in the ‘main text’ part (marked by a markable on the *section* annotation layer with attribute *name=text*).
- The value of `trainDataId` / `testDataId` selects the following corpus-specific behaviour:
 - If either `trainDataId` or `testDataId` are set to `MUC6`, the anaphor must be a definite for the expression to be an apposition (in `FE_Appositive`).
- The value of `runPipeline` can be set to `true` if it is desired to (re-)run the preprocessing steps on the corpus, or `false`, if existing annotation layers are to be reused.
- The value of `pipeline` can be used to select a different version of the preprocessing pipeline⁸:

⁷it is necessary to prepend the package name `elkfed.coref.mentions` in all cases

⁸it is always necessary to prepend the package name `elkfed.mmax.pipeline`

- `DefaultPipeline` uses a the Stanford POS tagger, the YamCha chunker and the Stanford named entity tagger.
 - `ParserPipeline` uses the Charniak parser to extract POS tags, BaseNPs as chunks, and also extracts parse trees.
 - `NERTestPipeline` uses the Charniak parser to extract syntactic structure, but uses the Carafembic mention tagger for extracting both nominal and name mentions. Because only ACE entities are extracted and non-ACE noun phrases are ignored, this is the recommended preprocessing when using ACE-style corpora which do not mark all mentions.
- The value of `default_system` determines the feature set and learners to be used. To use different settings, it is possible to either give `XMLExperiment` the name of an XML file containing such a description, or change the value of `default_system` to the name (without the `.xml` suffix) of an existing description from the `elkfed.main` package. In the current distribution of BART, the following XML descriptions are included:
 - `idc0_system` uses exactly the Soon et al. feature set (mention type, gender/number agreement, alias, appositive, semantic class compatibility, sentence distance).
 - `bart_system` uses an extended feature set: besides the information used by IDC0, it also uses parse tree information (tree kernels, syntactic position), as well as some semantic information (web patterns, Wikipedia alias, semantic class values).

The BART system uses tree kernels and requires external information (web queries and information extracted from Wikipedia in a relational database), which means that setting it up requires some work. For more details, please refer to the descriptions of the individual features in section 3.

1.3 XML system descriptions

The encoding/decoding model used as well as the learners and the features used can be influenced by means of XML description files. The two description files that can be used out of the box are loaded from the JAR file; they can be found in the package `elkfed.main`, whereas other examples can be found in the package `elkfed.main.old.xml`. To use an alternative system description, just put it in the current directory and give the filename to `XML{Trainer/Annotator/Experiment}`.

Figure 1 shows the system description for the IDC0 system. The root element, `coref-experiment`, has exactly one `system` node, which in turn has a list of classifiers and a list of extractors. In the `soon` system type, the only we will cover here, we only need one classifier, which is used for all anaphor-antecedent pairs.

The following classifiers are implemented:

```

<?xml version="1.0" encoding="UTF-8"?>
<coref-experiment>
<system type="soon">
  <classifiers>
    <classifier type="weka" model="idc0"
      learner="weka.classifiers.trees.J48"
      options="" />
  </classifiers>
  <extractors>
    <!-- general info about antecedent -->
    <extractor name="FE_MentionType_Buggy" />
    <!-- agreement features -->
    <extractor name="FE_Gender" />
    <extractor name="FE_Number" />
    <!-- specialized features for aliases etc. -->
    <extractor name="FE_Alias" />
    <extractor name="FE_Appositive" />
    <!-- string matching features -->
    <extractor name="FE_StringMatch" />
    <!-- semantic class agreement -->
    <extractor name="FE_SemanticClass" />
    <extractor name="FE_SentenceDistance" />
  </extractors>
</system>
</coref-experiment>

```

Figure 1: XML system description: IDC0

- The `weka` classifier uses the WEKA machine learning toolkit for classification; all classifiers from WEKA can be used, and the class name of the corresponding classifier has to be given in the “learner” attribute. Options, as they appear on the command line shown by the WEKA Experimenter, can be specified in the “options” attribute.
- The `svmlight` classifier uses SVMLight, either in its plain variant or in the SVMLight/TK variant. Options to `svm_learn` can be specified in the “options” attribute.
- The `maxent` classifier is a maximum entropy classifier built upon the L-BFGS implementation of Mallet. It is able to perform feature combinations. Binary feature combinations give you a similar accuracy to the SVMLight polynomial-degree-2 classifier, with much reduced training times.

The preliminary interface for this is that the “options” attribute is interpreted

as a combination template, i.e. `options="**"` uses the features alone, whereas `options="** **"` gives binary feature combinations. **This is subject to change. Use with care!**

The extractors are listed in section 3; the name of a feature extractor is specified in the “name” attribute and a matching class is then searched for in the package `elkfed.coref.features.pairs` and in the subpackages `elkfed.coref.features.pairs.{srl/wiki/wn}`

2 Inside BART: architecture and internal APIs

One goal for BART’s architecture has been to provide effective separation of concerns for the following three groups of people who might be interested in working on a system for coreference resolution:

- Those who aim to do *feature* engineering, creating new features that exploit different sources of knowledge.
- Those who aim to explore different *preprocessing* methods, improving the quality of the input to coreference resolution proper.
- Those who aim to explore different methods of representing coreference resolution as a *learning* problem.

To reach this goal, there is a clean separation between the domains of preprocessing, feature extraction, and learning:

The first part of preprocessing is carried out by pipeline components, which add MMAX markables on different annotation layers, and stores the result on the *markable* annotation layer in MMAX. The second part of preprocessing, carried out by `MentionFactory` instances, uses the markables on the *markable* annotation layer to create Java objects with relevant properties, instances of class `Mention`.

Feature extractors are presented are presented instances of the relevant `Instance` subclass — in BART, which exclusively uses binary decisions, this is always `PairInstance`. They then use the information stored in the `Instance`, namely the *anaphor* and *antecedent* properties, which hold references to mention objects. Having each feature extractor in its own class allows for flexible mixing and matching for feature extractors.

The part that is responsible for learning decision functions using a given set of features (referred to as the encoder/decoder) uses a machine learning classifier from the `elkfed.ml` package that is trained with anaphor - potential antecedent pairs from the training set, and the decisions of this classifier regarding single pairs are then used to derive appropriate linking decisions that group mentions into equivalence sets representing entities. The encoder/decoder has to extract pairs that are to be presented to the learner, and delegate the feature extraction to a list of feature extractors. In the testing phase, it has to choose pairs to present to the classifier built in the training phase and to use the classifier decisions to link mentions.

2.1 Important Classes

The most basic building blocks in the Elkfed platform are the interfaces `CorefResolver` and `CorefTrainer` in the package `elkfed.coref`. A coreference resolver get handed a list of `Mention` objects that are to be grouped together in a `DisjointSet`, whereas a `CorefTrainer` just gets handed the list of mentions and is not required to return anything.

`Mention` objects represent single mentions: they have utility methods that allow to access properties of mentions, and a method `isCoreferent` that allows the training procedure in a ML-based coreference resolution system to see whether a pair of mentions should be coreferent or not.

What happens around these interfaces? Let us begin by the outer side: in the package `elkfed.main`, the classes `Trainer` and `Annotator` are simplified versions of BART's `XMLTrainer` and `XMLAnnotator` classes and contain the necessary code for setting up the actual process.

Objects of type `SoonEncoder` or `SoonDecoder` (to be covered later, below) are handed to instances of `TrainerProcessor`, or `AnnotationProcessor`, respectively, that iterate through documents in the corpus given and then use a `MentionFactory` to create `Mention` objects from the information in the MMAX2 documents.

`SoonEncoder` instances take a list of markables; for every pair m_j, m_i of mentions that are adjacent in a coreference chain, a positive training instance is generated for the pair $\langle m_j, m_i \rangle$, and a negative instance with $\langle m_k, m_i \rangle$ is created for every markable m_k that occurs in between m_i and m_j . These learning instances serve as learning data set for the ML classifier; an object implementing the `InstanceWriter` interface takes these instances and writes them out in a format that is understood by the ML toolkit implementing that classifier, for example in ARFF format for Weka-based learners.

In converse, `SoonDecoder` instances look for an antecedent for a given markable m_i by getting the classification for pairs $\langle m_j, m_i \rangle$ built with some m_j that occurs before m_i , starting with the closest ones; the first pair to be classified as positive is merged and other (potential) antecedents are ignored. The classification of pairs is handled by an object implementing the `OfflineClassifier` interface, which gets a list of pairs and provides the list of decisions for these pairs. In the case of the Weka machine learning toolkit, the classifier is called in-process. For classifiers that are only available as external programs (such as SVMlight when the native interface is not used), always classifying batches of multiple pairs attenuates the speed loss due to the startup time of the external program.

The classification instances that are used for learning and classification are instances of the class `PairInstance`, which get the anaphor and antecedent set by the encoder/decoder, whereas the actual information used for classification is set by objects implementing the `PairFeatureExtractor` interface.

3 Feature Extractors

This section describes the feature extractors that are included in the Elkfed/IDC platform; most, but not all of them are used by BART. Tree-valued features can only be used by the SVMlight learner, string-valued features cannot be used with WEKA learners, and unnormalized continuous features do not work well with polynomial SVMs or MaxEnt classifiers that use feature combinations, so not all sets of features make sense with a given learner.

3.1 Basic Features

3.1.1 MentionType

The feature extractors `FE_MentionType_Buggy` and `FE_MentionType` extract information about the form of the anaphor (definiteness, demonstrative, pronoun), the antecedent (pronoun) and also includes a feature that indicates whether the two mentions are both proper names.

`FE_MentionType_Buggy` checks for the prefix “the” on the mention string to derive definiteness, whereas the *isDefinite* method on `Mention` checks that “the” is actually a word by itself, excluding “them”, “their” and other third person plural pronouns. In the basic Soon et al reimplementation, the information found in the ‘buggy’ version (third-person plural pronouns) is used and leads to improved performance over the corrected version.

3.1.2 Gender agreement

The feature extractor `FE_Gender` uses gender information from the mention to assess gender compatibility. The assigned value can either be true, false, or unknown.

3.1.3 Number agreement

The feature extractor `FE_Number` uses number information to determine number compatibility. This is either true or false.

3.1.4 Alias

`FE_Alias` uses the techniques described in (Soon et al., 2001) to match abbreviations and name variations.

3.1.5 Appositive

`FE_Appositive` adds a feature that is true whenever two mentions are separated exactly by a comma.

3.1.6 String Matching

`FE_StringMatch` strips the determiners off the markable string and then performs a case-insensitive comparison of the rest.

3.1.7 Semantic Class compatibility

`FE_SemanticClass` uses the `SemanticClass` property of the mention to assess the semantic compatibility of anaphor and antecedent (either `TRUE`, `FALSE`, or `UNKNOWN` if either of the two has an unknown semantic class and the lexical heads do not match).

3.1.8 Sentence distance (continuous vs. discrete version)

`FE_SentenceDistance` gives the distance of anaphor and antecedent candidate in sentences. `FE_DistDiscrete` is meant as a discretisation of the values, with two binary features that indicate whether the candidate is in the same sentence or in the previous sentence.

3.2 Syntax-based Features

3.2.1 Syntactic position

`FE_SynPos` yields a string that is composed of the first three unique labels of parent nodes. This is meant to indicate the syntactic position — subjects will have a value of `'np.s'`, whereas direct objects will have a value of `'np.vp.s'`, and a noun phrase embedded in a noun-modifying PP would have a value of `'np.pp.np'`.

3.2.2 Tree features

The feature `FE_TreeFeature` is a tree-valued feature that carries information about the syntactic relationship between anaphor and candidate. Its value is a subtree of a parse tree covering both the anaphor and the antecedent candidate. It includes the nodes occurring in the shortest path connecting the pronoun and the candidate, via the nearest commonly dominating node. Also it includes the first-level children of the nodes in the path.

3.3 Knowledge-based Features

3.3.1 Web patterns

The `FE_WebPatterns` feature extractor uses pattern search on the World Wide Web to find instance relations as they exist between `'China'` and `'country'`, or `'Clinton'` and `'president'`. Queries are cached in a local BerkeleyDB-JE database.

The following settings in `config.properties` are necessary for this feature extractor to work:

- **msn_app_id** contains the developer key for Microsoft's Windows Live Search service. The process of getting a developer key for this service is described at the following URL:
http://dev.live.com/blogs/livesearch/archive/2006/03/23/27.aspx

Results of web queries are cached in a Berkeley DB Java Edition database, which is created in the current directory. The current implementation unfortunately precludes concurrent access from multiple processes on the same file system, but a cache that has been established once (by doing the queries needed) can simply be moved to another machine by copying the *.jdb files.

3.3.2 Wikipedia Alias

The `FE_WikiAlias` feature extractor uses information extracted from Wikipedia⁹, namely redirects and links to a given page, but also appearance in lists, to provide evidence for name variations (see the extraction chapter for a more detailed description).

The Wikipedia Alias feature extractor needs to access a MySQL¹⁰ database that contains the *redirects_to*, *links_to* and *lists_dev* tables with information from Wikipedia. The following settings in `config.properties` are necessary for this feature extractor to work:

- **wikiDB_driver** contains the class name of the JDBC driver, usually *com.mysql.jdbc.Driver*
- **wikiDB_user** and **wikiDB_password** contain user name and password of the account that is used to connect to the database
- **wikiDB_dburl** contains the JDBC URL to the database. This should be something like

```
jdbc:mysql://<hostname>:3306/<database>↔
?useOldUTF8Behavior=true&useUnicode=true&↔
characterEncoding=UTF-8
```

 (without the line breaks or ↔ in between the parts).

3.3.3 Wiki (category graph)

The `FE_Wiki` feature extractor uses redirects and the category graph of Wikipedia to assess candidate relatedness, as described in (Ponzetto and Strube, 2006). *TODO: pull in the Wiki report chapter*

3.3.4 Wordnet distance

The `FE_WNSimilarity` feature extractor extracts the WordNet distance between antecedent and candidate heads, according to several distance measures.

⁹see <http://www.wikipedia.org>

¹⁰it is probably possible to use any other JDBC-compatible database

3.3.5 SemClass pair

`FE_SemClassValue` extracts the semantic class values of anaphor and antecedent, both alone and as a pair.

3.3.6 Modifier (in)compatibility

The `FE_Wiki_Inc` feature extractor uses information from the Wikipedia category/graph structure as described above, as well as Wordnet (also see above) to automatically compute the compatibility between the pronominal modifiers of the anaphora and antecedent - if they have matching head nouns.

Attributes and relations are extracted from the markable string of each mention, for example American tourist in Cuba would have associated with it American as an attribute, and from Cuba as a relation, which can then be compared against Cuban tourist to determine the incompatibility of the two mentions. The Wikipedia and Wordnet evaluations are computed separately and a final score of compatibility is assigned based on the two.

This feature extractor needs the Wordnet library and access to the Wikipedia category/graph structure (see the respective subsections for necessary preconditions).

References

- Ponzetto, S. P. and Strube, M. (2006). Exploiting semantic role labeling, WordNet and Wikipedia for coreference resolution. In *Proc. HLT/NAACL 2006*.
- Soon, W. M., Ng, H. T., and Lim, D. C. Y. (2001). A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544.