CLAIRLIB Documentation v1.08 Volume 1

Dragomir Radev, University of Michigan
Mark Hodges, University of Michigan
Anthony Fader, University of Michigan
Mark Joseph, University of Michigan
Joshua Gerrish, University of Michigan
Mark Schaller, University of Michigan
Jonathan dePeri, Columbia University
Bryan Gibson, University of Michigan
Chen Huang, University of Michigan
Amjad Abu Jbara, University of Michigan
Prem Ganeshkumar, University of Michigan

http://www.clairlib.org

February 26, 2010

Contents

1 Introduction

The University of Michigan CLAIR (Computational Linguistics and Information Retrieval) group is happy to present version 1.08 of the Clair Library.

The Clair library is intended to simplify a number of generic tasks in Natural Language Processing (NLP), Information Retrieval (IR), and Network Analysis (NA). Its architecture also allows for external software to be plugged in with very little effort.

We are distributing the Clair library in two forms: Clairlib-core, which has essential functionality and minimal dependence on external software, and Clairlib-ext, which has extended functionality that may be of interest to a smaller audience. Depending on whether you choose to install only Clairlib-core or both Clairlib-core and Clairlib-ext, some of the content of this manual will not apply to your installation. Throughout this document, for the sake of brevity, we will usually say "the Clair library" or the more abbreviated "Clairlib" to refer to the software we're distributing.

This work has been supported in part by National Institutes of Health grants R01 LM008106 "Representing and Acquiring Knowledge of Genome Regulation" and U54 DA021519 "National Center for Integrative Bioinformatics," as well as by grants IDM 0329043 "Probabilistic and link-based Methods for Exploiting Very Large Textual Repositories," DHB 0527513 "The Dynamics of Political Representation and Political Rhetoric," 0534323 "Collaborative Research: BlogoCenter - Infrastructure for Collecting, Mining and Accessing Blogs," and 0527513 "The Dynamics of Political Representation and Political Rhetoric," from the National Science Foundation.

1.1 Functionality

Much can be done using Clairlib on its own. Some of the things that Clairlib can do are listed below, in separate lists indicating whether that functionality comes from within a particular distribution of Clairlib, or is made available through Clairlib interfaces, but actually is imported from another source, such as a CPAN module, or external software.

1.1.1 Native to Clairlib-core

- **Tokenization**:

  Convert a sequence of strings into a sequence of tokens.
- **Summarization**:

  Extract content from an information and present the most important parts to the user in a condensed form.
- **LexRank**:

  Multi-document summarization, classification, and many other tasks.
- **Biased LexRank**:

  Semi-supervised passage retrieval.
- **Document Clustering**:

  Unsupervised assignment of documents into groups.
- **Document Indexing**:

  Transforming a document into an indexed form.
- **PageRank**:

  Assigning a numerical weighting to each element of a hyperlinked set of documents.
- **Biased Pagerank**
- **Web Graph Analysis**:

  Study of link patterns emerging between documents.

- **Network Generation**:

  Generate random networks. Currently, this includes generation of Erdos-Renyi random graphs.

- **Power Law Distribution Analysis**

- **Network Analysis**:

  - clustering coefficient
  - degree distribution plotting
  - average shortest path
  - diameter
  - triangles
  - shortest path matrices
  - connected components
  - maximum flow

- **Cosine Similarity**:

  Similarity between two documents, represented by the vectors, by finding the cosine of the angle between them

- **Random Walks on Graphs**

- **Statistics**: Analyzing and generating distributions

  - Distributions: Including Geometric, Gaussian, LogNormal, Zipfian and T-distribution
  - Tests

- **Tf**:

  Compute the Term Frequency

- **Idf**:

  Compute Inverse Document Frequency

- **Perceptron Learning and Classification**

- **Phrase Based Retrieval and Fuzzy OR Queries**

- **Harmonic Functions** Computing harmonic functions using the Relaxation and Monte Carlo methods

- **Synthetic Collection** Generation of synthetic document collections

- **Gene Interaction Extraction** Extract interactions between genes from biomedical texts.

1.1.2 Imported and available via Clairlib-core

- Parsing
- Stemming
- Sentence Segmentation
- Web Page Download
- Web Crawling
- XML Parsing
- XML Tree Building

- XML Writing
- Statistical Parsing
- Gene Tagging
- N-Grams extraction

1.2 Native to Clairlib-ext

- Interfacing with Weka, a machine-learning Java toolkit
- Latent Semantic Indexing
- Parsing using a Charniak Parser
- Using the Automatic Link Extractor (ALE)
- Using Google WebSearch

1.3 Authors

Dragomir Radev, Mark Hodges, Anthony Fader, Mark Joseph, Joshua Gerrish, Mark Schaller, Jonathan dePeri, Bryan Gibson, Chen Huang, Amjad Abu Jbara, and Prem Ganeshkumar.

1.4 Contributors

Timothy Allison, Michael Dagitses, Aaron Elkiss, Gunes Erkan, Scott Gifford, Justin Joque, Patrick Jordan, Jung-bae Kim, Samuela Pollack, and Adam Winkel

1.5 Changes

1.08 August 2009

- Updated Clair::SynthCollection to generate synthetic documents based on (1 to 4)-grams.
- Modified extract N-grams to optionally use CMU-LM.
- Updated make_synth_collection.pl to fully utilize Clair::SyntheticCollection.
- Fixed some Tokenizer issues.
- Added summarize_document.pl to the utilities.
- Added summarize_collection to the utilities.
- Added learn.pl to the utilities.
- Added classify.pl to the utilities.
- Added extract_features.pl
- Added bigrams_to_rand_doc.pl to the utilities.
- Added make_synth_collection_Menczer.pl to the utilities.
- Added Clair::RandomWalk for random walk on graphs.
- Added Clair::Harmonic for computing harmonic functions based on the Relaxation and Montecarlo methods
- Added random_walk.pl to the utilities.

- Added harmonic.pl to the utilities.
- Added directory_to_URL_network.pl to the utilities.
- Fixed a bug in the crawling code.
- Added new tutorials.
- Added new sections to the documentation.
- Added Clair::Bio::GIN for gene interaction extraction.
- Added an interface to Stanford parser in Clair::Utils::Parse
- Added tag_genes.pl to the utilities.
- Added extract_interactions.pl to the utilities.

1.07 June 2009

- Added Clair::Network::Spectral for spectral partitioning using Fiedler Vector.
- Made Clairlib independent of MEAD (MEAD is no more required for Clairlib).
- Added Naive Bayes learning and classification.
- Added tests for feature extraction, learning, classification.
- Fixed a bug in Clair::Cluster::create_lexical_network().
- Added sampling options to Clair::Cluster.
- Added ”No IDF” option and sampling capabilities to corpus_to_cos.pl utility.
- Fixed documentation typos.
- Added new tutorials to the documentation.
- Fixed bug in Clair::Utils::CorpusDownload.
- Added ’manual weights’ option to make_synth_collection util.
- Fixed bug in extract_ngrams.

1.06 March 2009

- Added Clair:Network:FordFulkerson
- Added change_perl_path.pl to the utilities.
- Added new scripts to interface ACL Anthology Network.
- Fixed a bug in split_into_sentences() of Clair::Document

1.05 July 2008

- Fixed formatting bugs in CorpusDownload.pm
- Added get_predecessor_matrix() function in Network.pm
- Added get_shortest_path() function in Network.pm
- Added erase_corpus.pl script
- added erase_isolated_nodes.pl script
- added –ignore-isolated-nodes in convert_network.pl
- added several options to print_network_stats.pl: a. –self-loop,
- completed the descriptions of print_network_stats.pl: added the note of –force into usage.
- added sentence_to_docs.pl , lines_to_docs.pl under util folder

1.04B June 2008

- Added -no-duplicated-edges in convert_network.pl
- Added largest connected component in cos_to_stats.pl
- Added full avergage shortest path in print_network.pl
- fixed divide by zero error in Network.pm, Betweeness.pm

1.04A April 2008

- Added Clair::Network::GirvanNewman algorithm to do hierarchical clustering
- Added Clair::Network::KernighanLin algorithm to do graph partition

1.04 Feburary 2008

- Added Clair::Network::AdamicAdar to compute the adamic/adar value for a given network corpus
- Added Clair::ChisqIndependent to compute p-value and degree of freedom for Chi square

1.03 August 2007

- Added functionality to perform community finding within weighted, undirected networks
- Added util/chunk_document.pl to break documents into smaller files by word number
- Added option to retain punctuation for idf and tf queries
- Added option to print out full lists of idf and tf values for a corpus
- LexRank moved from Clair::Network to Clair::Network::Centrality::LexRank
- LexRank use now follows the same use pattern as the other centrality modules

1.02 July 2007

- Distribution reorganized in standard format
- Improved and expanded installation documentation (INSTALL)
- Improved POD (inline) documentation
- Additional examples
- Updated PDF documentation

1.01 May 2007

- Added Phrase-based Retrieval and Fuzzy OR Queries
- Extended Clairlib-ext with interfaces for the Cluster class and the Document class to the Weka machine learning toolkit
- Added LSI functionality
- Extended parsing of strings / files into Documents
- Added perceptron learning and classification for documents

1.0 RC1 April 2007

- Moved all Clair modules beneath the Clair::* namespace, updated documentation
- Improved Network Analysis, added Clustering Coefficients code
- Added Network Generation and Statistics modules

0.955 March 2007

- Made it possible to distribute clairlib in two distributions, one containing core code and another containing code that may be dependent on other resources
- Cleaned up unit tests

0.953 February 2007

- Fixed bugs in Clair::Cluster, Clair::Document involving stemming
- Cleaned up t/ and test/ directories
- Created util/ directory
- Added scripts to util/ directory to:
  - Run a Google query and save the returned URLs to a file
  - Download files from a URL and build a corpus
  - Segment a document into sentences and build a corpus of the sentences
  - Take all documents in a directory and create a corpus
  - Index the corpus (compute TF*IDF, etc.)
  - Compute cosine similarity measures between all documents in a corpus
  - Generate networks corresponding to various cosine thresholds
  - Print network statistics about a network file
  - Generate plots of degree distribution and cosine transitions

- New methods in Clair::Network:

```
print_network_info
get_network_info_as_string
get_cumulative_distribution
cumulative_power_law_exponent
find_components
newman_clustering_coefficient
linear_regression
```

2 Getting Started

2.1 Downloading

Clairlib can be downloaded from http://www.clairlib.org.

2.2 Installing

This guide explains how to install both Clairlib distributions, Clairlib-Core and Clairlib-Ext. To install Clairlib-core, follow the instructions in the section immediately below. To install Clairlib-Ext, first follow the instructions for installing Clairlib-Core, then follow those for Clairlib-Ext itself. Clairlib-Ext requires an installed version of Clairlib-Core in order to run; it is not a stand-alone distribution.

3 Install and Test Clairlib-Core

System Requirements

Clairlib-Core requires Perl 5.8.2 or greater. Before you proceed, confirm that the version of Perl you are running is at least this recent by entering the following at the shell prompt.

```
perl -v
```

Install CPAN Libraries

Clairlib-Core depends on access to the following Perl modules:

BerkeleyDB

Carp

File::Type

Getopt::Long

Graph::Directed

Hash::Flatten

HTML::LinkExtractor

HTML::Parse

HTML::Strip

IO::File

IO::Handle

IO::Pipe

Lingua::Stem

Lingua::EN::Sentence

Math::MatrixReal

Math::Random

MLDBM

PDL

POSIX

Scalar::Util

Statistics::ChisqIndep

Storable

Test::More

Text::Sentence

XML::Parser

XML::Simple

There are multiple approaches to locating and installing these modules; using the automated CPAN installer, which is bundled with Perl, is perhaps the quickest and easiest. To do so, enter the following at the shell prompt:

```
$ perl -MCPAN -e shell
```

If you have not yet configured the CPAN installer, then you'll have to do so this one time. If you do not know the answer to any of the questions asked, simply hit enter, and the default options will likely suit your environment adequately. However, when asked about parameter options for the `perl Makefile.PL` command, users without root permissions or who otherwise wish to install Perl libraries within their personal **$HOME** directory structure should enter the suggested path when prompted:

```
Your choice:  ] PREFIX=˜/perl
```

This will cause the CPAN installer to install all modules it downloads and tests into **$HOME/perl**, which means that all subdirectories of this directory that contain Perl modules will need to be added to Perl's `@INC` variable so that they will be found when needed (see next section for further explanation).

As a side note, if you ever need to reconfigure the installer, type at the shell prompt:

```
$ perl -MCPAN -e shell
cpan>o conf init
```

After configuration (if needed), return to the CPAN shell prompt,

```
cpan>
```

and type the following to upgrade the CPAN installer to the latest version:

```
cpan>install Bundle::CPAN
cpan>q
```

If asked whether to prepend the installation of required libraries to the queue, hit return (or enter `yes`). After quitting the shell, type the following to install or upgrade `Module::Build` and make it the preferred installer:

```
$ perl -MCPAN -e shell
cpan>install Module::Build
cpan>o conf prefer_installer MB
cpan>o conf commit
cpan>q
```

Finally, install each of the following dependencies (if you are at all unsure whether the latest versions of each have already been installed) by entering the following at the shell prompt:

```
$ perl -MCPAN -e shell
cpan>install BerkeleyDB
cpan>install Carp
cpan>install File::Type
cpan>install Getopt::Long
cpan>install Graph::Directed
cpan>install HTML::LinkExtractor
cpan>install HTML::Parse
cpan>install HTML::Strip
cpan>install IO::File
cpan>install IO::Handle
cpan>install IO::Pipe
cpan>install Lingua::Stem
cpan>install Math::MatrixReal
cpan>install Math::Random
cpan>install MLDBM
cpan>install PDL
cpan>install POSIX
cpan>install Scalar::Util
cpan>install Statistics::ChisqIndep
cpan>install Storable
cpan>install Test::More
cpan>install Text::Sentence
cpan>install XML::Parser
cpan>install XML::Simple
```

Configure Clairlib-Core

Download the Clairlib-Core distribution (**clairlib-core.tar.gz**) into, say, the directory **$HOME**. Then to install Clairlib-Core in **$HOME/clairlib-core**, enter the following at the shell prompt:

```
$ cd $HOME
$ gunzip clairlib-core.tar.gz
$ tar -xf clairlib-core.tar
$ cd clairlib-core/lib/Clair
```

Then edit Config.pm, which is located in **clairlib-core/lib/Clair**. You will see the following message at the top of the file:

```
#################################
# For Clairlib-core users:
# 1. Edit the value assigned to $CLAIRLIB_HOME and give it the value
#    of the path to your installation.
# 2. Edit the value assigned to $MEAD_HOME and give it the value
#    that points to your installation of MEAD.
# 3. Edit the value assigned to $EMAIL and give it an appropriate
#    value.
```

Follow those instructions. In the case of our example, we would assign

```
$CLAIRLIB_HOME=$HOME/clairlib-core
```

and

```
$MEAD_HOME=$HOME/mead
```

where **$HOME** must be replaced by an explicit path string such as **/home/username**. Also, note that the following MEAD variables reflect the structure of a standard MEAD installation and should typically be kept as assigned:

```
$CIDR_HOME "$MEAD_HOME/bin/addons/cidr";
$PRMAIN    "$MEAD_HOME/bin/feature-scripts/lexrank/prmain";
```

Test and Install the Clairlib-Core Modules

Before testing and installing the Clairlib-core modules, you'll need to modify Perl's `@INC` variable to ensure that it includes 1) paths to all Clairlib dependencies (the required libraries installed above), and 2) the path to Clairlib's own modules (in the case of our example, **$HOME/clairlib-core/lib**). The simplest way to do this is by modifying the contents of your `PERL5LIB` environment variable from the shell prompt:

```
$ export PERL5LIB=$HOME/clairlib-core/lib:$HOME/perl/lib     (*)
```

Here **$HOME/clairlib-core/lib** is the path to Clairlib's own modules, while **$HOME/perl** is the path to Clairlib's required modules, installed above (assuming that path is their location). However, doing this requires that you export `PERL5LIB` each time you invoke the shell environment, so a better way to modify `@INC` is the following:

```
$ cd $HOME
```

Edit **.profile** or the appropriate configuration file for your shell environment, or create it if it does not already exist. Add `(*)` to to the file, or prepend the necessary paths using colons, as in `(*)`. Save the file and enter:

```
$ . .profile
```

This way you will not have to export `PERL5LIB` each time you invoke the shell. Enter

```
$ echo $PERL5LIB
```

to confirm that your modifications have been applied.
Now you may test your Clairlib-Core installation. Enter its directory, in the case of our example:

```
$ cd $HOME/clairlib-core
```

Then enter the following commands to test the Clairlib-Core modules:

```
$ perl Makefile.PL
$ make
$ make test
```

If you would like to have the Clairlib-Core modules installed for you, and you have the necessary (root) permissions to do so, you may install them by entering the following command:

```
$ make install
```

If, on the other hand, you have only local permissions, but you have a personal perl library located at, say, **$HOME/perl** (as described earlier), then you can install Clairlib-Core there by entering the commands:

```
$ perl Makefile.PL PREFIX=~/perl
$ make install
```

Using the Clairlib-Core Modules

To use the Clairlib-Core modules in a Perl script, you must add a path to the modules to Perl's `@INC` variable. You may use either 1) **$CLAIRLIB_HOME/lib**, where `$CLAIRLIB_HOME` is the path defined in **Config.pm**, or 2) **$INSTALL_PATH**, where `$INSTALL_PATH` is a path to the location of the installed Clairlib-Core modules (if you installed them as in the previous section, immediately above). Either of these paths can be added to `@INC` either by appending the path to the `PERL5LIB` environment variable or by putting a `use lib PATH` statement at the top of the script. See the beginning of the previous section above for a detailed explanation of how to modify the `PERL5LIB` variable.

4 Install and Test Clairlib-Ext

The Clairlib-Ext distribution contains optional extensions to Clairlib-Core as well as functionality that depends on other software. The sections below explain how to configure different functionalities of Clairlib-Ext. As each is independent of the rest, you may configure as many or as few as you wish. This section provides instructions for the installation and testing of the Clairlib-ext modules itself.

Sentence Segmentation using Adwait Ratnaparkhi's MxTerminator

To use MxTerminator for sentence segmentation, download the installation package from:
ftp://ftp.cis.upenn.edu/pub/adwait/jmx/jmx.tar.gz.
Putting the tarball in, say, **$HOME/jmx**, enter the following to unpack:

```
$ cd $HOME/jmx
$ gunzip jmx.tar.gz
$ tar -xf .tar
```

Uncomment and modify the following lines in **clairlib-core/lib/Clair/Config.pm**. Point `$JMX_HOME` to the top directory of your MxTerminator installation, and point `$JMX_MODEL_PATH` to the location of your MxTerminator trained data, as for example

```
# $JMX_HOME                "$HOME/jmx";
# $SENTENCE_SEGMENTER_TYPE "MxTerminator";
# $JMX_MODEL_PATH          "$HOME/jmx/eos.project";
```

where `$HOME` must be replaced by a literal path string such as **/home/username**. Note that the **/bin** directory of a Java installation must be located in your search path, or MxTerminator will not work.

Parsing using a Charniak Parser

To use a Charniak parser with Clairlib, uncomment the following variables in **clairlib-core/lib/Clair/Config.pm** and point them to it, as for example:

```
# Default parser and data paths for the Charniak parser for use in Parse.pm
# (Note that CHARNIAK_DATA should end with a slash and that the other
# paths include the executable)
# $CHARNIAK_PATH      "/data0/tools/charniak/PARSE/parseIt";
# $CHARNIAK_DATA_PATH "/data0/tools/charniak/DATA/EN/";

# Default path to Chunklink
# $CHUNKLINK_PATH "/data2/tools/chunklink/chunklink.pl";
```

Using the Weka Machine Learning Toolkit

To use the Weka Machine Learning Toolkit, a Java machine learning library, with Clairlib, download Weka from **http://www.cs.waikato.ac.nz/ml/weka/** and uncomment the following line in **clairlib-core/lib/Clair/Config.pm**. Point the variable to the location of Weka's **.jar** file, as for example:

```
# $WEKA_JAR_PATH "$HOME/weka/weka-3-4-11/weka.jar"
```

where `$HOME` must be replaced by an explicit path string such as **/home/username**. Note that the **/bin** directory of a Java installation must be located in your search path, or MxTerminator will not work.

Using the Automatic Link Extractor (ALE)

If you have MySQL installed and wish to use ALE, uncomment the following variables. Point `$ALE_PORT` at your MySQL socket, and provide the root password to your MySQL installation:

```
# $ALE_PORT "/tmp/mysql.sock";
# $ALE_DB_USER "root";
# $ALE_DB_PASS "";
```

Using Google WebSearch

To use the Google WebSearch module, first install the CPAN module `Net::Google` (refer to the of Clairlib-Core installation instructions for further explanation) Then, uncomment the following line and provide a Google SOAP API key. Unfortunately, Google no longer gives out SOAP API keys but has moved to an AJAX Search API. If you have a SOAP API key, you can still use it, and WebSearch will still work.

```
# $GOOGLE_DEFAULT_KEY "";
```

Using CMU-LM tool kit.

The CMU-Cambridge Statistical Language Modeling toolkit is a suite of UNIX software tools to facilitate the construction and testing of statistical language models. CMU-LM is used by clairlib for N-grams extraction. It can be downloaded from http://mi.eng.cam.ac.uk/˜prc14/toolkit.html. Then, add the CMU-LM path to $PATH (or modify ˜/.profile):

```
export PATH=/path/to/CMU-CAM-LM/bin:$PATH
```

Using GENIA Tagger

The GENIA tagger analyzes English sentences and outputs the base forms, part-of-speech tags, chunk tags, and named entity tags. It is used in Clair::Bio::GIN. To be able to use it, download it from http://www-tsujii.is.s.u-tokyo.ac.jp/GENIA/tagger/ then uncomment and point the following line in Clair::Config to point to the Genia tagger home.

```
# $GENIATAGGER_PATH = "/path/to/geniatagger";
```

Using the Stanford Parser

To use the Stanford parser in Clairlib, download it from http://nlp.stanford.edu/software/lex-parser.shtml and install it as instructed in its documentation, then uncomment the following line and point it to the parser home directory.

```
# $STANFORD_PARSER_PATH = "/path/to/stanford/parser";
```

Configure Clairlib-Ext

Download the Clairlib-Ext distribution (**clairlib-ext.tar.gz**) into, for example, the directory **$HOME**. Then to install Clairlib-Ext in **$HOME/clairlib-ext**, enter the following at the shell prompt:

```
$ cd $HOME
$ gunzip clairlib-ext.tar.gz
$ tar -xf clairlib-ext.tar
$ cd clairlib-ext
```

To test the Clairlib-Ext modules, you must first have installed the Clairlib-Core modules. Confirm that you have, then enter the following:

```
$ perl Makefile.PL
$ make
$ make test
```

If you would like to have the Clairlib-Ext modules installed, and you have the necessary (root) permissions to do so, you may install them by entering:

```
$ make install
```

If, on the other hand, you have only local permissions, but you have a personal perl library located at, say, **$HOME/perl** (as described earlier), then you can install Clairlib-Ext there by entering the commands:

```
$ perl Makefile.PL PREFIX=~/perl
$ make install
```

Using the Clairlib-Ext Modules

To use the Clairlib-Ext modules in a script, you must add a path to the modules to Perl's `@INC` variable. You may use either 1) **$CLAIRLIB_EXT_HOME/lib**, where **$CLAIRLIB_EXT_HOME** is the path to the top directory of your Clairlib-Ext installation, or 2) **$INSTALL_PATH**, where **$INSTALL_PATH** is a path to the location of the installed Clairlib-Ext modules (if you installed them as in the previous section). Either of these paths can be added to `@INC` either by appending the path to the `PERL5LIB` environment variable or by putting a `use lib PATH` statement at the top of the script. See the beginning of section V of the Clairlib-Core installation instructions for a detailed explanation of how to modify the `PERL5LIB` variable.

Change Perl Path

To be able to run *.pl files, you may need to change the perl path in all *.pl files to the perl path on your machine. To make this easy for you, we provide a utility script "change_perl_path.pl" that changes the perl path in all the *.pl files in a specified directory and all its subdirectories to the path that you specify. For example, to change the perl path in all *.pl files in the "util" directroy to /usr/local/perl/

```
$ change_perl_path.pl util/ /usr/local/perl/
```

Support and Documentation

After installing Clairlib, you may access documentation for a module using the `perldoc` command, as for example:

```
$ perldoc Clair::Document
```

Each Clairlib distribution also includes a PDF tutorial. Online API documentation is available at:

```
http://belobog.si.umich.edu/clair/clairlib/pdoc
```

5 Structure of the Clairlib Code

The Clairlib code is divided into many modules, located in subdirectories within the `lib/Clair` directory.

5.1 Key Modules

Some of the key functionality is in the `lib/Clair` directory itself:

- `Clair::Document` - Represents a single document
- `Clair::Cluster` - Represents a collection of many documents
- `Clair::Network` - Represents a network, like a graph. The nodes of the network may often be of type `Clair::Document`, but do not have to be.
- `Clair::Gen` - Works with Poisson and Power Law distributions
- `Clair::Util` - Provides utility functions needed when using the Clair library
- `Clair::Config` - Provides configurable constants needed by the Clair library (library paths, etc.)

Other modules in the top directory include the following:

- `Clair::Features` - Carry out feature selection using Chi-squared algorithm with Clair::GenericDoc
- `Clair::Debug` - A simple class that Exports debugmsg and errmsg subs.
- `Clair::Learn` - Implement various learning algorithms here. Default algorithm is Perceptron.
- `Clair::Index` - Creates various indexes from supplied Clair::GenericDoc objects.
- `Clair::Classify` - Take in the model file generated by Learn.pm and then carry out the classification.
- `Clair::StringManip` - Majority of the string manipulation routines required by other packages.
- `Clair::Centroid` - Compute the centroid of a cluster.
- `Clair::Corpus` - Class for dealing with TREC corpus format data.
- `Clair::CIDR` - Single pass document clustering.
- `Clair::SyntheticCollection` - Generate synthetic clusters of documents.
- `Clair::Extensions` - Versioning File for the Clairlib-ext distribution.
- `Clair::IDF` - Handle IDF databases.
- `Clair::SentenceFeatures` - A collection of sentence feature subroutines.
- `Clair::RandomWalk` - Random walk on graphs.
- `Clair::Harmonic` - Compute harmonic functions.

5.2 Corpora Processing Modules

Within the `lib/Clair/Utils/` directory, several modules are provided to work with corpora:

- `Clair::Utils::CorpusDownload` - Download corpora from a list of URLs or from a single URL as a starting point, compute IDF and TF values
- `Clair::Utils::Idf` - Retrieve IDF values calculated by CorpusDownload
- `Clair::Utils::Tf` - Retrieve TF values calculated by CorpusDownload
- `Clair::Utils::TFIDFTUtils` - Provides utility functions needed for the IDF/TF calculations
- `Clair::Utils::Robot2` - configurable web traversal engine (for web robots & agents)
- `Clair::Utils::LinearAlgebra`
- `Clair::Utils::Stem` - An implementation of a stemmer
- `Clair::Utils::MxTerminator` - Split text into sentences.
- `Clair::Utils::ALE` - The Automatic Link Extrapolator

5.3 Clairlib-ext Modules

The Clairlib-ext distribution also contains the following modules in lib/Clair/Utils/:

- `Clair::Utils::WebSearch` - Performs Google searches and downloads files
- `Clair::Utils::Parse` - Parse a file using the Charniak parser or use the Chunklink tool.

5.4 Network and Graph Processing

Clairlib includes a large collection of network and graph processing modules:

- `Clair::Network` - Network Class for the CLAIR Library
- `Clair::NetworkWrapper` - A subclass of `Clair::Network` that wraps the C++ version of Lexrank.
- `Clair::Network::AdamicAdar` - Adamic/Adar Algorithms, calculate the Adamic/Adar value of a network.
- `Clair::Network::Sample` - Network sampling algorithms
  - `Clair::Network::Sample::RandomEdge` - Random edge sampling
  - `Clair::Network::Sample::RandomNode` - Random node sampling
  - `Clair::Network::Sample::ForestFire` - Random sampling using Forest Fire model
  - `Clair::Network::Sample::SampleBase` - Abstract class for network sampling
- `Clair::Network::Reader` - Different network file type readers
  - `Clair::Network::Reader` - Abstract class for reading in network formats
  - `Clair::Network::Reader::GraphML` - Class for reading in GraphML network files
  - `Clair::Network::Reader::Pajek` - Class for reading in Pajek network files
  - `Clair::Network::Reader::Edgelist` - Class for reading in edgelist network files
- `Clair::Network::Generator` - Random network generators
  - `Clair::Network::Generator::GeneratorBase` - Network generator abstract class
  - `Clair::Network::Generator::ErdosRenyi` - ErdosRenyi network generator abstract class

- `Clair::Network::Writer` - Different network file type writers
  - `Clair::Network::Writer` - Abstract class for exporting various Network formats
  - `Clair::Network::Writer::GraphML` - Class for writing GraphML network files
  - `Clair::Network::Writer::Pajek` - Class for writing Pajek network files
  - `Clair::Network::Writer::Edgelist` - Class for writing edge list network files
- `Clair::Network::Centrality` - Network centrality measures
  - `Clair::Network::Centrality` - Abstract class for computing network centrality
  - `Clair::Network::Centrality::Degree` - Class for computing degree
  - `Clair::Network::Centrality::Closeness` - Class for computing closeness centrality
  - `Clair::Network::Centrality::Betweenness` - Class for computing betweenness centrality
- `Clair::Network::CFNetwork` - Class for performing community finding using Newman 2004 modularity algorithm
- `Clair::Network::KernighanLin` - Class for performing community finding and graph partition using KernighanLin algorithm
- `Clair::Network::GirvanNewman` - Class for performing community finding using Girvan/Newman Algorithm
- `Clair::Network::Spectral` - Class for performing spectral graph partitioning using Fiedler vector Algorithm
- `Clair::Network::FordFulkerson` - Class for finding maximum flow using Ford/Fulkerson Algorithm

The Network modules uses the Graph CPAN module by default, but this other graph libraries such as Boost can be used:

- `Clair::GraphWrapper` - Abstract class for underlying graphs
- `Clair::GraphWrapper::Boost` - GraphWrapper class that provides an interface to the Boost graph library

5.5 Distributions and Statistics Modules

There are also packages for dealing with discrete and continuous distributions:

- `Clair::RandomDistribution::RandomDistributionBase` - base class for all distributions
- `Clair::RandomDistribution::Gaussian`
- `Clair::RandomDistribution::LogNormal`
- `Clair::RandomDistribution::Poisson`
- `Clair::RandomDistribution::RandomDistributionFromWeights`
- `Clair::RandomDistribution::Zipfian`
- `Clair::Statistics::Distributions::TDist`
- `Clair::Statistics::Distributions::DistBase`
- `Clair::Statistics::Distributions::Geometric`

5.6 ALE Modules

- `Clair::ALE::Default::Tokenizer` - ALE's default tokenizer.
- `Clair::ALE::Default::Stemmer` - ALE's default stemmer.
- `Clair::ALE::Tokenizer`
- `Clair::ALE::Stemmer` - Internal stemmer used by ALE
- `Clair::ALE::Conn` - A connection between two pages, consisting of one or more links, created the the Automatic Link Extrapolator.
- `Clair::ALE::Link` - A link between two URLs created by the Automatic Link Extrapolator.
- `Clair::ALE::_SQL` - Internal SQL adapter for use by ALE
- `Clair::ALE::URL` - A URL created by the Automatic Link Extrapolator
- `Clair::ALE::NormalizeURL`

5.7 Political Science Modules

- `Clair::Polisci` - Polisci modules
  - `Clair::Polisci::AU::XMLHandler`
  - `Clair::Polisci::US::XMLHandler`
  - `Clair::Polisci::US::Connection` - Read records from the US polisci database
  - `Clair::Polisci::Speaker` - An object representing a hansard speaker
  - `Clair::Polisci::Record` - An object representing a hansard record
  - `Clair::Polisci::Graf` - An object representing a hansard graf
  - `Clair::Polisci::AustralianParser` - A class for parsing Australian hansard html.

5.8 Mead Interfacing Modules

- `Clair::MEAD::DocsentConverter` - Document =¿ Mead Cluster converter
- `Clair::MEAD::Summary` - Access to a MEAD summary
- `Clair::MEAD::Wrapper` - A perl module wrapper for MEAD

5.9 Bio Modules

- `Clair::Bio` - Bio utilities
- `Clair::Bio::EUtils::ESearchHandler` - An XML handler for parsing ESearch results
- `Clair::Bio::EUtils::ESearch` - A Perl interface to the ESearch utility
- `Clair::Bio::EUtils` - A base class for Bio::EUtils objects
- `Clair::Bio::Connection` - Connect to the Bio database using SOAP
- `Clair::Bio::GeneRIF` - Perl module for parsing GeneRIF files
- `Clair::Bio::GIN` - Gene interaction extraction.
- `Clair::Bio::GIN::Data` - Interface to data files used for interaction extraction.
- `Clair::Bio::GIN::Interaction` - Data structure for representing a gene interaction.

5.10 Information Retrieval Modules

- `Clair::IR` - Basic Information Retrieval operations

5.11 LinkPolicy Modules

- `Clair::LinkPolicy` - Different document linking policies
  - `Clair::LinkPolicy::MenczerMacro` - Class implementing the Menczer Micro link model
  - `Clair::LinkPolicy::LinkPolicyBase` - Base class for creating corpora from collections
  - `Clair::LinkPolicy::RadevPAMixed` - Class implementing the RadevPAMixed link model
  - `Clair::LinkPolicy::MenczerPAMixed` - Class implementing the MenczerPAMixed Micro link model
  - `Clair::LinkPolicy::RadevMicro` - Class implementing the Radev Micro link model
  - `Clair::LinkPolicy::BarabasiAlbert` - Class implementing the Barabasi Albert link model.
  - `Clair::LinkPolicy::WattsStrogatz` - Class implementing the Watts/Strogatz link model
  - `Clair::LinkPolicy::ErdosRenyi` - Class implementing the Erdos Renyi link model

5.12 Sentence Segmentation Modules

- `Clair::SentenceSegmenter` - Sentence segmentation
  - `Clair::SentenceSegmenter::SentenceSegmenter`
  - `Clair::SentenceSegmenter::Text`

5.13 Generic Document Modules

- `Clair::GenericDoc` - Generic document representations and parsing modules
  - `Clair::GenericDoc` - A class to standardize and create generic representation of documents.
  - `Clair::GenericDoc::html` - A submodule that strips out html tags.
  - `Clair::GenericDoc::shakespeare` - specialized to parse shakespeare html files.
  - `Clair::GenericDoc::octet_stream` - A submodule that parses xml and converts it into a hash
  - `Clair::GenericDoc::sports` - A specialized module for parsing docs for hw2
  - `Clair::GenericDoc::xml` - A submodule that parses xml and converts it into a hash
  - `Clair::GenericDoc::plain` - A submodule that returns the document as is.

5.14 Other Modules

- `Clair::CIDR::Wrapper` - A wrapper script for the original cidr script
- `Clair::Nutch::Search` - A class for performing simple Nutch searches.
- `Clair::Interface::Weka` - Interfacing with Weka, a machine-learning Java toolkit.
- `Clair::Index::mldbm` - A submodule that gets dynamically loaded by Index.pm.
- `Clair::Index::dirfiles` - Builds the index into the filesystem namespace.
- `Clair::Algorithm::LSI` - Latent Semantic Indexing.

- `Clair::Info::Query` - A module that implements different types of queries.
- `Clair::Info::Stats`

Many of the above modules are described in more details in the following section.

6 Graph Formats in Clairlib

In this section, we talk about the graph formats Clairlib accepts as input and those that it can produce as output. We describe each format and show how to use Clairlib to input or output graphs in it.

Clairlib accepts the following graph formats as input:

- Edgelist Format
- Graph Modeling Language - GML
- Graph Markup Language - GrpahML
- Pajek Format

And it's able to output the following graph formats:

- Edgelist Format
- Graph Markup Language - GrpahML
- Pajek Format
- Graclus Format

6.1 Edgelist Format

In the edgelist format, a graph is represented by its edges. Each line in the edgelist file corresponds to an edge in the graph. The edge can optionally have a weight. For example, if a graph file has four nodes (n1, n2, n3, and n4) and has two edges (n1-n2, n1-n3) and each edge is weighted 0.5, the edgelist file of this graph should be:

```
n1 n2 0.5
n1 n3 0.5
n4
```

The default delimiter used to separate the components of each line is a single space but any other delimiter can be specified by the user. Notice that isolated nodes are allowed and represented by a line with a single node (as n4 in the example above.) Multiple edges are also allowed and take the following format:

```
n1 n3 0.5
n1 n3 0.9
n1 n3 0.1
```

To read a graph stored in an edgelist file format, Clairlib provides *Clair::Networks::Reader::Edgelist* module which reads the file line by line and creates a *Clair::Network* instance. You need first to create an instance of *Clair::Network::Reader::Edgelist* then use it to read the file.

```
$reader = new Clair::Network::Reader::Edgelist(directed=>0, delim=>"˜");
$net = $reader->read_network($filename);
```

The *directed* argument indicates that the graph is undirected if its value is 0 and directed if 1. Note that there is nothing in an edglist file that tells whether the graph is directed or not. By default, Clairlib will consider it as being undirected unless the user set the *directed* parameter to 1 in which case the first node in the line will be the source and the second the target.

The *delim* argument tells the reader that the nodes in the file are separated by ” ”. The default delimiter is ”[t]+”. *$filename* is the graph file name.

If you have a *Clair::Network* instance (e.g. *$net*) and want to write it to a file in edgelist format, create an instance of *Clair::Network::Writer::Edgelist* first then use it to write the network to the file.

```
$export = new Clair::Network::Writer::Edgelist();
$export->write_network($net, $filename);
```

6.2 Graph Modeling Language - GML

Graph Modelling Language (GML) is a hierarchical ASCII-based file format for describing graphs. GML is a portable file format supported by several graph programs. A GML file consists of a hierarchical key-value list. An example of a simple GML file is:

```
graph [
  comment "This is a sample graph"
  directed 1
  IsPlanar 1
  node [
    id 1
    label
    "Node 1"
  ]
  node [
    id 2
    label
    "Node 2"
  ]
  edge [
    source 1
    target 2
    label "Edge from node 1 to node 2"
  ]
]
```

For more information about the GML format, visit *http://www.infosun.fim.uni-passau.de/Graphlet/GML/*.

To read a graph stored in a GML file format, Clairlib provides *Clair::Networks::Reader::GML* module which reads the file and creates a *Clair::Network* instance. You need first to create an instance of *Clair::Network::Reader::GML* then use it to read the file.

```
$reader = new Clair::Network::Reader::GML();
$net = $reader->read_network($filename);
```

There is no support for outputting graphs in GML format in this version of Clairlib.

6.3 Graph Markup Language - GraphML

GraphML is a comprehensive and easy-to-use file format for graphs. Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs. An example of simple GraphML file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
  </graph>
</graphml>
```

For more information about the GraphML format, visit *http://graphml.graphdrawing.org/*.

To read a graph stored in a GraphML file format, Clairlib provides *Clair::Networks::Reader::GraphML* module which reads the file and creates a *Clair::Network* instance. You need first to create an instance of *Clair::Network::Reader::GraphML* then use it to read the file.

```
$reader = new Clair::Network::Reader::GraphML();
$net = $reader->read_network($filename);
```

If you have a *Clair::Network* instance (e.g. *$net*) and want to write it to a file in GraphML format, create an instance of *Clair::Network::Writer::Edgelist* first then use it to write the file.

```
$export = new Clair::Network::Writer::GraphML();
$export->write_network($net, $filename);
```

6.4 Pajek Format

Pajek is a popular network analysis program for Windows. The data (vertices and edges) as well as other attributes (e.g. vertex colour, label font size) of a network is stored in a plain text files. An example of simple Pajek file is:

```
* Vertices n
  1 "Vertice 1"
  2 "Vertice 2"
  ...
* Edges
  1 2
  1 3
  ...
```

The upper part of the file specifies the vertices of the network. In particular, n is the number of vertices in the network. This should agree with the number of lines following under the first line. For each line specifying a vertex, the first value is the vertex number (counting from 1), while the second is the vertex label. Besides, there are a set of optional attributes of the vertices that can be specified after the vertex label.

For more information about the Pajek software and file format, visit *http://vlado.fmf.uni-lj.si/pub/networks/pajek/*.

To read a graph stored in a Pajek file format, Clairlib provides *Clair::Networks::Reader::Pajek* module which reads the file and creates a *Clair::Network* instance. You need first to create an instance of *Clair::Network::Reader::Pajek* then use it to read the file.

```
$reader = new Clair::Network::Reader::Pajek();
$net = $reader->read_network($filename);
```

If you have a *Clair::Network* instance (e.g. *$net*) and want to write it to a file in Pajek format, create an instance of *Clair::Network::Writer::Pajek* first then use it to write the file.

```
$export = new Clair::Network::Writer::Pajek();
$export->write_network($net, $filename);
```

6.5 Graclus Format

Graclus is a graph clustering software. A Graclus file contains a matrix in adjacent list format. For example, if a graph has 3 nodes(1,2,3) and 2 edges (1-2,2-3) and the edges are weighted (0.1, 0.2) respectively, the corresponding Graclus files should be:

```
3 2         <--- # of nodes and edges and format
2 0.1      <--- nodes adjacent to 1 and weights
1 0.1 3 0.2 <--- nodes adjacent to 2 and weights
2 0.2 <--- nodes adjacent to 3 and weights
```

For more information, visit *http://www.cs.utexas.edu/users/dml/Software/graclus.html*.

This version of Clairlib supports only outputting graphs in Graclus format. If you have a *Clair::Network* instance (e.g. *$net*) and want to write it to a file in Graclus format, create an instance of *Clair::Network::Writer::Graclus* first then use it to write the file.

```
$export = new Clair::Network::Writer::Graclus();
$export->write_network($net, $filename);
```

7 Graphical User Interface

The graphical user interface is a distinguishing feature that was recently added to Clairlib and constituted a quantum leap in its development. It consists of a number of interactive and user-friendly visual tools, the main purpose of which is to make the rich set of Clairlib functionalities easier to access by a larger number of users from various levels and backgrounds.

It is also intended to be used by educators to enhance the teaching process. It can also help students do their assignments, projects, and research experiments in an interactive environment. We believe that visual tools facilitate understanding and make learning a more enjoyable experience. Focusing on this purpose, the GUI is tuned for simplicity and ease of use more than high computational efficiency. Therefore, while it's suitable for small and medium scale projects, it's not guaranteed to work efficiently for large projects that involve very large datasets and require heavy processing. The command-line interface is more appropriate for such projects.

The GUI encompasses three main components: the Network Editor/Visualizer/Analyzer, the Text Processor, and the Corpus Processor.

7.0.1 Network Component

The Network component allows users to:

- Build a new network visually using a set of simple drawing and editing tools.
- Open existing networks stored in files in different formats.
- Visualize a network and interact with it.
- Analyze the network and compute its statistics such as diameter, clustering coefficient, degree distribution, etc.

- Simulate some network operations such as random walks and label propagation.

This component uses the open source package, JUNG[1] as an infrastructure for the visualization functionality.

7.1 Text Processing

The text processing component allows users to process textual data in plain, XHTML, PDF, or DOC format imported from a file stored on the disk or brought directly from the web. Most of the text processing capabilities implemented in Clairlib are accessible through this component. It uses some functionalities of the open-source package of the Stanford Parser[2] to perform some tasks such as parse tree visualization.

7.2 Corpus Processing

The corpus processing component allows users to build a corpus of textual data out of a collection of files in plain, XHTML, or PDF format, or by crawling a website. Several tasks could then be performed on a corpus such as indexing, querying, summarization, information extraction, hyperlink or cosine network construction, etc.

Although these three components can be run independently, they are designed to easily interact with each other. For example, the corpus processing component allows a computed cosine network to be forwarded to the network component for analysis or visualization.

8 Tutorials

In this section, you'll go through some tutorials that will help you explore the various functionalities of Clairlib.

8.1 Clairlib Web Site Processing Utilities

This tutorial explains how to use Clairlib tools to create a network from a group of files and process that network to extract information. Unlike the next tutorial (7.2), this tutorial one doesn't use the Clairlib API directly, but is based on executable tools only.

8.1.1 Introduction

This tutorial will walk you through downloading files, creating a corpus from them, creating a network from the corpus, and extracting information along the way.

8.1.2 Generating the corpus

The first thing we will need is a corpus of files to run our tests against. As an example we will be using a set of files extracted from Wikipedia. We'll first download those files into a folder:

```
mkdir corpus
```

We'll use the 'wget' command to download the files. The -r means to recursively get all of the files in the folder, -nd means don't create the directory path, and -nc means only get one copy of each file:

```
cd corpus
wget -r -nd -nc http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
```

Now that we have our files, we can create the corpus. To do this we'll use the 'directory_to_corpus.pl' utility. The options used here are fairly consistent for all utilities: –corpus, or -c, refers to the name of the corpus we are creating. This should be something fairly simple, since we use it often and it is used to name several of the files we'll be creating. In this case, we call our corpus 'chemical'. –base, or -b, refers to the base directory of our

[1]http://jung.sourceforge.net/
[2]http://nlp.stanford.edu/software/lex-parser.shtml

corpus' data files. A common practice is to use 'produced'. Lastly –directory, or -d, refers to the directory where our files to be converted are located:

```
directory_to_corpus.pl --corpus chemical --base produced \
 --directory corpus
```

Now that our corpus has been organized, we'll index it so we can then start extacting data from it. To do that we'll use 'index_corpus.pl'. Again, we'll specify the corpus name and the base directory where the index files should be produced:

```
index_corpus.pl --corpus chemical --base produced
```

We've now got our corpus and our indices and are ready to extract data.

8.1.3 Tfs and Idfs

First we'll run a query for the term frequency of a single term. To do this we'll use 'tf_query.pl'. Let's query 'health':

```
tf_query.pl -c chemical -b produced -q health
```

This outputs a list of the files in our corpus which contain the term 'health' and the number of times those terms occur in that file. To get term frequencies for all terms in the corpus, pass the –all option:

```
tf_query.pl -c chemical -b produced --all
```

This returns a list of terms, their frequencies, and the number of documents each occurs in.

In order to see the full list of term frequencies for stemmed terms, pass the stemmed option:

```
tf_query.pl -c chemical -b produced --stemmed --all
```

Next we'll run a query for the inverse document frequency of a single term. To do this we'll use 'idf_query'. Again, we'll query 'health':

```
idf_query.pl -c chemical -b produced -q health
```

We can also pass the –all option to idf_query.pl to get a list of idf's for all terms in the corpus:

```
idf_query.pl -c chemical -b produced --all
```

8.1.4 Creating a Network

We now have a corpus from which we can extract some data. Next we'll create a network from this corpus. To do this, we'll use 'corpus_to_network.pl'. This command creates a network of hyperlinks from our corpus. It produces a graph file with each line containing two linked nodes. This command requires a specified output file which we'll call 'chemical.graph':

```
corpus_to_network.pl -c chemical -b produced -o chemical.graph
```

Now we can gather some data on this network. To do that we'll run 'print_network_stats.pl' on our graph file. This command can be used to produce many different types of data. The easiest way to use it is with the –all option, which run all of its various tests. We'll redirect its output to a file:

```
print_network_stats.pl -i chemical.graph --all > chemical.graph.stats
```

If we now look at ’chemical.graph.stats’ we can see statistics for our network including numbers of nodes and edges, degree statistics, clustering coefficients, and path statistics. This command also creates three centrality files (betweenness, closeness, and degree) which are lists of all terms and their centralities.

8.1.5 Code

This is a list of all of the commands used in this tutorial:

```
mkdir corpus
cd corpus
wget -r -nd -nc http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
directory_to_corpus.pl --corpus chemical --base produced \
 --directory corpus
index_corpus.pl --corpus chemical --base produced
tf_query.pl -c chemical -b produced -q health
tf_query.pl -c chemical -b produced --all
idf_query.pl -c chemical -b produced -q health
idf_query.pl -c chemical -b produced --all
corpus_to_network.pl -c chemical -b produced -o chemical.graph
print_network_stats.pl -i chemical.graph --all > chemical.graph.stats
```

8.2 Information Retrieval Tutorial

This tutorial is about building a basic information retrieval system using Clairlib API. We will use a well-known IR test corpus, the Cranfield Collection, which contains 1,400 documents on aerodynamics. The main task is to process the Cranfield documents, and build an inverted index from them. After that, we will build a system that takes a query, processes it, finds and ranks documents matching that query, and returns a ranked list of results.

8.2.1 Corpus Description

The corpus used in this tutorial is the Cranfield Collection. You can download the Cranfield corpus from this link *http://belobog.si.umich.edu/clair/clairlib/cranfield.tar.gz.* After downloading a copy of the file, create a new directory *cranfield* and extract the file into it.

```
   wget http://www.clairlib.org/clair/clairlib/cranfield.tar.gz
   mkdir cranfield
   cd cranfield
   tar -xvzf ../cranfield.tar.gz
```

8.2.2 Parse corpus files and store metadata

Parse XML file

The first thing we need to do is to write a subroutine that parses a cranfield XML file and store its content in a hash.

```
use XML::Simple;
sub parse_file{
   my $file=shift;
   # create object
   my $xml = new XML::Simple;
   # read XML file
   my $data = $xml->XMLin($file);
   return $data;
}
```

Next, we write another subroutine to process all the collection files, store them in a suitable format for indexing, and store their metadata. We need this step because the default format of the files isn't compatible with the input format that Clairlib expect. We will explain the parts of this subroutine first and show the whole code later.

Create dbm files to store metadata

We need first to create four db files to store the metadata: title, bibliography, author, and length.

```
   %title_meta=();
   dbmopen(%title_meta, "titles", 0666);
   %bibl_meta=();
   dbmopen(%bibl_meta, "bibl", 0666);
   %author_meta=();
   dbmopen(%author_meta, "authors", 0666);
   %length=();
   dbmopen(%length, "doclength", 0666);
```

This will create four db files and bind each of them to a hash.

Parse all files and store metadata

Then, we read all the files from their location, parse each file, store its content in a new txt file, and store its metadata in the db files through the hashes.

```
if(-d $destination){
    `rm -r $destination`;
}
`mkdir $destination`;
@files = <$cranfield_path/*>;
foreach my $doc(@files){
   my $hash = parse_file("$doc");
   my $text =  $hash->{"TITLE"} . "\n" . $hash->{"TEXT"}. "\n" .
               $hash->{"AUTHOR"};
   my $docno = $hash->{"DOCNO"};
   $docno =~ s/[^0-9]*([0-9]+)[^0-9]*/$1/g;
   open (FILE, ">$destination/$docno.txt") or die "Can't create file";
   print FILE $text;
   close (FILE);
   $title_meta{$docno} = $hash->{"TITLE"};
   $author_meta{$docno} = $hash->{"AUTHOR"};
   $text =~ s/\.//g; $text =~ s/,//g;
   my @doclength = split /\s+/, $text;
   $length{$docno} = scalar @doclength;
}
```

The first four lines of the code above check whether the directory *$destination* already exists or not. If it exists, it will be deleted and then a new empty directory is created. The rest of the code loops through the collection files and processes them as explained before.

Put all together

The whole code for processing the collection is shown next

```
sub process_collection{
   my ($cranfield_path, $destination) = @_;
   print "parsing xml and storing metadata...\n";
   @files = ‘ls $cranfield_path‘;
   %title_meta=();
   dbmopen(%title_meta, "titles", 0666);
   %bibl_meta=();
   dbmopen(%bibl_meta, "bibl", 0666);
   %author_meta=();
   dbmopen(%author_meta, "authors", 0666);
   %length=();
   dbmopen(%length, "doclength", 0666);
   if(-d $destination){
      ‘rm -r $destination‘;
   }
  ‘mkdir $destination‘;
   @files = <$cranfield_path/*>;
   foreach my $doc(@files){
       my $hash = parse_file("$doc");
       my $text =  $hash->{"TITLE"} . "\n" . $hash->{"TEXT"}. "\n" /
        . $hash->{"AUTHOR"};
       my $docno = $hash->{"DOCNO"};
       $docno =~ s/[^0-9]*([0-9]+)[^0-9]*/$1/g;
       open (FILE, ">$destination/$docno.txt") or die "Can’t create file";
       print FILE $text;
       close (FILE);
       $title_meta{$docno} = $hash->{"TITLE"};
       $author_meta{$docno} = $hash->{"AUTHOR"};
       $text =~ s/\.//g; $text =~ s/,//g;
       my @doclength = split /\s+/, $text;
       $length{$docno} = scalar @doclength;
   }
}
```

8.2.3 Compute TF and IDF

By now, we have the files ready for indexing and stored in *$destination*. The next step is to use the *Clair::Utils::CorpusDownload* module to read these files and create a clairlib corpus. We start by creating a new object of the *Clair::Utils::CorpusDownload*.

```
$corpus = Clair::Utils::CorpusDownload->new(corpusname => "cranfield",
          rootdir => "produced");
```

rootdir is the path to the directory where the corpus and its TFIDF index will be built and stored. Using the *$corpus* object, we call *build_corpus_from_directory()* subroutine which builds a corpus from a set of files located on the computer.

```
corpus->build_corpus_from_directory(dir=>$data_source, cleanup => 0,
                                    skipCopy => 0);
```

This will read all the txt files that we created above and store them in the *produced* directory in TREC ”Text REtrieval Conference” format. *cleanup=0* is used to retain metafiles produced during corpus build. Then, we build the Inverse Document Frequency (IDF) and the Term Frequency (TF).

```
    $corpus->buildIdf(stemmed => 1);
    $corpus->build_docno_dbm();
    $corpus->buildTf(stemmed => 1);
```

The *stemmed=1* option indicates that the TF and the IDF are computed using stemmed values. Notice that we have to call *build_docno_dbm()* before the *buildTF*. *build_docno_dbm()* builds the *DOCNO-to-URL* and *URL-to-DOCNO* databases. The whole code for the *create_index()* subroutine that creates the corpus and builds the index is

```
use Clair::Utils::CorpusDownload;
use Clair::Utils::Tf;
sub create_index{
   my $data_source = shift;
   my $corpus = Clair::Utils::CorpusDownload->new(corpusname => "corpus",
                                                  rootdir => "produced");
   $corpus->build_corpus_from_directory(dir=>"$data_source",
                        cleanup => 0,  skipCopy => 0);
   $corpus->buildIdf(stemmed => 1);
   $corpus->build_docno_dbm();
   $corpus->buildTf(stemmed => 1);
   $corpus->build_term_counts(stemmed => 1);
}
```

Now, put the three subroutines (*parse_file(), process_collection(), and create_index()*) in a module file and name it *IR.pm*

Create a Perl script file *index.pl* and add the following to it

```
#! /usr/bin/perl
use IR;
my $collection_path = shift;
my $destination = "data";
process_collection($collection_path,$destination);
create_index($destination);
```

8.2.4 Query description

We need our system to be able to handle the following types of queries:

```
cat          : returns any document that has the word "cat" in it

cat dog      : any document that has one or more of these words
           ("fuzzy or" is assumed by default)

cat dog rat  : up to 10 words in a query

"tabby cat"  : phrases of up to 5 words in length

!"tabby cat" : negations of single words or phrases

!cat !dog    : multiple negations per query

"small tabby cat" "shaggy dog" : multiple phrases in a query

!cat
```

In this tutorial, we will not worry about nested queries in parentheses. Each query should return zero or more matching documents. If multiple documents match, we should sort them by decreasing score. The score is defined as the number of query terms (or phrases) that match in the ”fuzzy or”, including repetitions and counting phrases as the number of words that are included in them. Negated terms are not included in the score. For example, if the query is: (cat dog ”pack rat”) and we have three documents D1, D2, and D3 that contain at least one of the query terms as follows:

- D1: cat cat dog cat mouse
- D2: pack rat cat rat rat
- D3: rabbit elephant dog dog cat pack rat cat

their scores should be as follows:

- 4
- 3 (”pack rat” counts as two terms, but ”rat” alone doesn’t count)
- 6

8.2.5 Processing and handling the queries

We start by showing the code then the explanation follows:

```
#!/usr/bin/perl
use Clair::IR;
print "Enter your query or type q to quit\n>";
my $query= <>;
while ($query ne "q\n"){
  my @searchTerms=();
  while ($query =~ s/!\"(.+?)\"//){
    push(@searchTerms, "!".$1);
  }
  while ($query =~ s/\"(.+?)\"//){
    push(@searchTerms, $1);
  }
  push(@searchTerms, split(/\s+/, $query));
  my ($ref, $locref) = execute_query(@searchTerms);
  my %results = %$ref;
  my %locations = %$locref;
  @sortedResults = reverse sort {$results{$a} <=> $results{$b}} keys %results;
  foreach my $result(@sortedResults){
     my $sum =  get_summary($result, $locations{$result});
     print "Doc: $result  \tScore: $results{$result}\t$sum\n";
  }
  print "\n>";
  $query=<>;
}
```

Add the code above to a Perl script file and name it *query.pl*

The code for the *get_summary* subroutine is

```
sub get_summary{
   my ($docId, $position) = @_;
   my $text = `cat data/$docId.txt`;
   $text =~ s/\s\./\./g;
   my $return = "";
   my $start = 0;
   @words = split /\s+/,$text;
   if ($position > 11){
       $start = $position - 11;
   }
   for($count = $start; $count <= ($start+20); $count++){
       if (exists $words[$count]){$return .= "$words[$count] ";}
       if ($count == $start +10){$return .= "\n\t\t";}
   }
   return $return;
}
```

Add *get_summary* to *IR.pm*

Read in the query and parse it

The code above starts by asking the user to enter a query to search for, or ”q” to exit.

```
print "Enter your query or type q to quit\n>";
my $query= <>;
```

For each query, create a hash to store the query terms in @searchTerms.

The search term is a single word (e.g. cat), a negated word (e.g. !cat), a phrase (e.g. ”cat dog”), a negated phrase (e.g. !”cat dog”). To parse the query into a set of search terms, we start by extract the negated phrases from the query (e.g. !”cat dog”) and add each phrase as a single negated entry to the searchTerms hash.

```
while ($query =~ s/!\"(.+?)\"//){
    push(@searchTerms, "!".$1);
}
```

For example, if the query is **cat !dog !”dog rat” ”monkey cat”** The above while loop will extract **!dog rat** from the query and add it as one entry to the **searchTerms** hash and leaves **cat !dog ”monkey cat”** in the query. Next, we extract the unnegated phrases from the query (e.g. ”monkey cat”)

```
while ($query =~ s/\"(.+?)\"//){
    push(@searchTerms, $1);
}
```

The above while loop extracts the phrase **monkey cat** and add as a single search term leaving **cat !dog**. After that, we add all the remaining words (negated and unnegated) to *@searchTerms*

```
push(@searchTerms, split(/\s+/, $query));
```

8.2.6 Execute the query and return the results

After adding all the search terms to *@searchTerms* as explained in the previous subsection, *@searchTerms* array is passed to a subroutine, *execute_query*

```
my ($ref, $locref) = execute_query(@searchTerms);
```

execute_query() takes an array of search terms and returns two hashes; one of DocIds of matching documents along with their scores, and the other of the location of the first result.

```
sub execute_query{
   my @searchTerms = @_;
   my $tf = Clair::Utils::Tf->new(rootdir => "produced",
                   corpusname => "corpus", stemmed => 1);
   my %results = ();
   my %out = ();
   my %location = ();
   my @negation;
   my $negationOnly = 1;
   foreach $term(@searchTerms){
       if ($term=~ s/!//g){
           push(@negation, $term);
       }else{
           $negationOnly = 0;
           my @words = split(/ /, $term);
           $numWords = @words;
           my $urls = $tf->getDocsWithPhrase(@words);
           foreach my $key (keys %$urls){
               $ref = $urls->{$key};
               $toAdd = keys(%$ref) * $numWords;
               $key =~ s/.*\/([0-9]+)\.txt/$1/g;
               if (!exists $location{$key}){
                   my ($position, $storedVal) = each %$ref;
                   $location{$key} = $position;
               }
               if (exists $out{$key}){
                   $out{$key}+= $toAdd;
               }else{
                   $out{$key}=$toAdd;
               }
           }
       }
   }
   if ($negationOnly == 1){
       %out = getAllDocKeys($tf);
       foreach my $key(keys %out){
           $location{$key} = 0;
       }
   }
   if ((scalar @negation) > 0){
       my @negatedDocs = negationResults(\@negation, $tf);
       foreach $removeDoc(@negatedDocs){
           if (exists $out{$removeDoc}){
               delete $out{$removeDoc};
           }
       }
   }
   return \%out, \%location;
}
```

Sort the results by scores

```
my %results = %$ref;
my %locations = %$locref;
@sortedResults = reverse sort {$results{$a} <=> $results{$b}} keys %results;
[edit]Print out the results
foreach my $result(@sortedResults){
   my $sum =  get_summary($result, $locations{$result});
   print "Doc: $result  \tScore: $results{$result}\t$sum\n";
}
```

8.2.7 Test the system

Now, you should have created three files

- IR.pm which includes four subroutines:
  - *parse_file*
  - *create_index*
  - *process_collection*
  - *execute_query*
  - *get_summary*
- index.pl
- query.pl

Make sure that these files are in the same directory. To test the system, first run index.pl to create the index.

```
perl index.pl cranfield
```

Then, run query.pl

```
perl query.pl
```

Then, enter a search query (e.g. chemical reaction)

```
 Enter a query to search for, or enter "q" to exit
>chemical reactions
```

The output will something like

```
Doc: 1061   Score: 8
            to take place according to a single-step chemical reaction.
            the solution of the problem is based on the simultaneous.
Doc: 488    Score: 7
            linearizing is achieved by expanding equation of rate of chemical
            reaction in a taylor series and neglecting higher-order terms.
Doc: 1296   Score: 7
            non-equilibrium expansions of air with coupled chemical reactions.
            analysis and solutions of the streamtube gas dynamics involving
            coupled chemical rate
...
```

8.3 Automatic Link Extractor

ALE (Automatic Link Extractor) is a collection of tools and Perl libraries providing easy database access for indexing information about the links in HTML documents and retrieving information from those indices. The basic process used is to give a series of documents to the ALE indexer, then ask questions with the command-line search tool or the Perl modules. In this tutorial, you'll learn how to use run Clairlib ALE to index the pages of **http://www.kzoo.edu** as a sample.

8.3.1 Clairlib and System Configurations

To be able to use ALE, you need to:

- Install MySQL (http://www.mysql.com/) on your machine (if it's not already installed). Create a new database named ”clair” and create a new user that has full privileges on it.
- Uncomment the following three lines in **Config.pm** module (located in *$CLAILIB_PATH/lib/Clair*)

```
     $ALE_PORT  = "/tmp/mysql.sock";
     $ALE_DB_USER = "user";
     $ALE_DB_PASS = "pass";
```

  Point *$ALE_PORT* to your your MySQL socket and set *$ALE_DB_USER* and *$ALE_DB_PASS* to your MySQL user information.

- set the following ENVIRONMENT VARIABLES:
  - **ALESPACE:** which is the subdirectory where all data should be stored, and a prefix for all directory names. If you are working with data independent of other projects, you should try to set ALESPACE to something unique, perhaps starting with your username. It defaults to ”default”.
  - **ALECACHEBASE:** which determines the root of the location where ALE can find the documents its working with, in wget format.
  - **MYSQL_UNIX_PORT:** gives the path to the UNIX socket where the MySQL database ALE should use is running on.

```
     export ALESPACE=KZOO
     export ALECACHE=/data0/ale/cache
     export MYSQL_UNIX_PORT=/tmp/mysql.sock
```

8.3.2 Download Website Files

All the website pages should be downloaded to your machine before indexing it. The following script uses **wget** to do this:

```
#!/bin/sh
umask 002
if [ ! -d "$ALECACHE" ]
then
  if mkdir "$ALECACHE"
  then
    :
  else
    exit $?
  fi
fi
cd "$ALECACHE"
exec 2>&1
exec wget  --timeout 5  --reject gif,png,jpg,
jpeg,gz,tar,gzip,exe,sit,hxq,bin -S -x -U ALE/0.1 "$@"
```

put the code above in a file named ”aleget” and then run it to download www.kzoo.edu

```
aleget -r ’http://www.kzoo.edu/index.php’
```

8.3.3 Index the Files

To index the links information of the website files, you’ll use Clair::ALE::Extract module. The following script does this.

```
!#/usr/bin/perl
use Clair::ALE::Extract;
use Clair::Config qw($ALE_PORT $ALE_DB_USER $ALE_DB_PASS);
if (not defined $ALE_PORT or not -e $ALE_PORT) {
    die "ALE_PORT not defined in Clair::Config or doesn’t exist";
}
$ENV{MYSQL_UNIX_PORT} = $ALE_PORT;
my $e = Clair::ALE::Extract->new();
my $alecache=$ENV{’ALECACHE’};
my $doc_dir="$alecache/www.kzoo.edu/";
open(ALL,’find $doc_dir -name "*.html" -print|’);
my @files2;
foreach $file (<ALL>){
  chomp($file);
  push @files2, $file;
  print "file = ",$file;
}
$e->extract( drop_tables => 1, files => \@files2 );
```

The code above creates three tables in the MySQL database and stores the indexing information in them.

8.3.4 Search ALE for connections by various criteria

Clair::ALE::Search module allows you to search the Automatic Link Extractor for connections that meet the criteria you give. Valid criteria are:

- **limit:** Return at most this many connections.

- **source_url:** The first URL in the connection. Use **no_source_url** to exclude connections where the first URL is this one. The argument to this should just be a simple string.
- **dest_url:** The last URL in the connection. Use **no_dest_url** to exclude connections where the last URL is this one. The argument to this should just be a simple string.
- **link_text:** The text that links two pages. For multi-hop links, put a number after link. To exclude links with this text, use **no_link_text**.
- **link_word:** An individual word that links two pages. For multi-hop links, put a number after link. To exclude links which contain these words, use **no_link_word**.

To search for connections, create a new *Clair::ALE::Search* object and pass the desired criteria as arguments to the constructor

```
use Clair::ALE::Search;
my $search = new Clair::ALE::Search(source_url=>"http://www.kzoo.edu/");
```

The *queryresult()* subroutine can be access via the *$search* object. It returns the next result from the query, or *undef* if there are no more results. We can make use of this subroutine to loop through the results of our query as follows:

```
use Clair::ALE:Conn;
while (my $conn = $search->queryresult)
{
  my $conn = $search->queryresult;
  $conn->print;
}
```

This will print the information of all the connections that match the query. The output of the code above should be something like:

```
(Connection)
Hop 1
   (Link)
   From:
       (URL)
       url: http://www.kzoo.edu/college/history
        id: 73
   To:
       (URL)
       url: http://www.kzoo.edu/map.html
        id: 145
   Link ID: 310
   Link Text: Campus Map
(Connection)
Hop 1
   (Link)
   From:
       (URL)
       url: http://www.kzoo.edu/college/history
        id: 73
   To:
       (URL)
       url: http://www.kzoo.edu/directory.html
        id: 36
   Link ID: 312
   Link Text: Directories
```

You can get the number of links in a connection using

```
$num_links = $conn->{numlinks};
```

And you can get an array of all the links in the connection using

```
 @links = $conn->{links}
```

For each "link" in the array *@links* you can get the source URL, the destination URL, the link text, and the link ID

```
 use Clair::ALE:Link;
 $source_url=@link[0]->{from};
 $destination_url=@link[0]->{to};
 $text=@link[0]->{text};
 $ID=@link[0]->{id};
```

8.4 Spectral Partitioning Using Fiedler Vector

In this tutorial you will use Clairlib module **Clair::Network::Spectral** to partition the Karate Club network into two subgraphs. **Clair::Network::Spectral** implements Spectral Partitioning using Fiedler Vector (i.e. the second smallest eigenvector of the Laplacian matrix).

8.4.1 Dataset

The dataset used in this tutorial is the Karate Club network, a network of friendships between the 34 members of a karate club at a US university, as described by Wayne Zachary in 1977. The network file is in GML format:

```
Creator "Mark Newman on Fri Jul 21 12:39:27 2006"
graph
[
  node
  [
    id 1
  ]
  .
  .
  node
  [
    id 34
  ]
  edge
  [
    source 2
    target 1
  ]
  .
  .
  edge
  [
    source 34
    target 33
  ]
]
```

You can download the network from **http://www.clairlib.org/mediawiki/index.php/Corpora**

8.4.2 Read in the network file and create a Clair::Network object

First you need to create a Clair::Network::Reader::GML object

```
use Clair::Network::Reader::GML;
my $reader=new Clair::Network::Reader::GML();
```

Then, pass the network filename to the read_network subroutine via the $reader object. This will return a **Clair::Network** object.

```
use Clair::Network;
my $filename = "karate.gml";
my $net = $reader->read_network($filename);
```

8.4.3 Create a Clair::Network::Spectral object

Create a new **Clair::Network::Spectral** by calling the constructor and passing the following two parameters:

- **Network:** a Clair::Network object.
- **Splitting Method (optional):** to specify the method used to choose the splitting value. This can take one of the following options:
  - **Bisection:** The splitting value is the median of the Fiedler Vector components.

  – **Gap:** The splitting value is in the middle of largest gap within the Fiedler Vector components.
  – **Sign (Default):** The splitting value is 0.

The splitting value is used to partition the network into two parts by choosing all nodes whose corresponding Fiedler Vector component is larger than splitting value to be in one partition and the remaining nodes to be in the other partition. For the purpose of this tutorial we will use the ”bisection” method.

```
use Clair::Network::Spectral;
$spectral = new Clair::Network::Spectral($net,"sign");
```

8.4.4 Partition the network

To partition the network, simply call get_partitions subroutine via $spectral object. get_partitions returns the nodes of each of the partitions in an array.

```
my(@parta,@partb) = $spectral->get_partitions();
```

You can print the result of partitioning the Karate Club network in by looping through the two resulting arrays or simply by using the dumper.

```
use Data::Dumper
print "PartA = ",Dumper(@parta),"\n";
print "PartB = ",Dumper(@partb),"\n";
```

The output should be as follows:

```
PartA = [
          '25',
          '15',
           .
           .
          '31',
          '30'
        ];
PartB = [
          '13',
          '8',
           .
           .
          '18',
          '12'
       ];
```

You can get the splitting value that was used to do the partitioning by calling get_splitting_value().

```
my $splitting_value = $spectral->get_splitting_value();
```

The result should be 0.0625504557748447. You can also get the Fiedler Vector by

```
  my $splitting_value = $spectral->get_fiedler_vector();
```

You can change the splitting method anytime and then redo the partitioning using the new method.

```
  $spectral->set_splitting_method("gap");
```

8.5 Graph Partitioning Using the GirvanNewman Algorithm

In this tutorial, you will use *Clair::Network::GirvanNewman* to partition the Karate Club network into two partitions. For more information about Karate Club network, see **Spectral Partitioning Using Fiedler Vector** tutorial (the previous subsection).

8.5.1 Read in the network file and create a Clair::Network object

First you need to create a *Clair::Network::Reader::GML* object

```
use Clair::Network::Reader::GML;
my $reader=new Clair::Network::Reader::GML();
```

Then, pass the network filename to the *read_network* subroutine via the *$reader* object. This will return a *Clair::Network* object.

```
use Clair::Network;
my $filename = "karate.gml";
my $net = $reader->read_network($filename);
```

8.5.2 Create a Clair::Network::GirvanNewman object

Create a new *Clair::Network::GirvanNewman* by calling its constructor

```
use Clair::Network::GirvanNewman;
$GN = new Clair::Network::GirvanNewman($net);
```

8.5.3 Partition the graph

To partition the graph, simply call *partition()* subroutine via the *$GN* object. *partition()* returns the result as a hash.

```
my $graphPartition = $GN->partition();
```

$graphPartition is a hash with ”node id” as key and ”partition number” as value. The hierarchy structure for each node is represented as (0—1—2—1—...). So the number between ”—” is the partition the node belongs to in a specific hierarchy.

You can use the dumper to print the contents of *$graphPartition*

```
use Data::Dumper
print Dumper($graphPartition);
```

The output will be something like. This is what’s called the dendrogram

```
$VAR1 = {
         '33' => '0|1|1|2|3|3|4|4|7|8|8|9|10...',
         '32' => '0|1|1|2|1|1|2|2|5|6|6|7|8...',
         . . .
         . . .
         '19' => '0|1|1|2|3|3|4|4|7|8|8|9|...',
         '5' => '0|0|0|0|4|4|5|6|3|1|1|2|...'
        };
```

8.6 Graph Partitioning Using Krenighan-Lin Algorithm

In this tutorial, you will use Clair::Network::KrenighanLin to partition the Karate Club network into two partitions. For more information about Karate Club network, see **Spectral Partitioning Using Fiedler Vector tutorial**.

8.6.1 Read in the network file and create a Clair::Network object

First you need to create a *Clair::Network::Reader::GML* object

```
use Clair::Network::Reader::GML;
my $reader=new Clair::Network::Reader::GML();
```

Then, pass the network filename to the *read_network* subroutine via the *$reader object*. This will return a *Clair::Network* object.

```
use Clair::Network;
my $filename = "karate.gml";
my $net = $reader->read_network($filename);
```

8.6.2 Create a Clair::Network::KrenighanLin object

Create a new *Clair::Network::KrenighanLin* by calling its constructor

```
use Clair::Network::KrenighanLin;
$KL = new Clair::Network::KrenighanLin($net);
```

8.6.3 Partition the graph

To partition the graph, simply call *generatePartition()* subroutine via the *$GN* object. *generatePartition()* returns the result as a hash.

```
my $graphPartition = $KL->generatePartition();
```

$graphPartition is a hash with ”node id” as key and ”partition number (0/1)” as value.

You can use the dumper to print the contents of *$graphPartition*.

```
use Data::Dumper
print Dumper($graphPartition);
```

The output will be something like

```
 $VAR1 = {
          '33' => 0,
          '32' => 0,
            .
            .
          '19' => 0,
          '5' => 1
        };
```

8.7 Generate Network from WordNet Semantic Lexicon

This tutorial will show how to use the Clairlib utility 'wordnet_to_network.pl', which integrates Princeton University's WordNet semantic lexicon to generate a graph file. It creates a network based on nouns and their synset words and saves the graph in the given output file. To use the utility, simply enter the command

```
wordnet_to_network.pl --output WordNetNouns.graph --verbose
```

8.8 Generate a Collection of Synthetic Documents

In this tutorial, we will use the Clairlib utility 'make_synth_collection.pl' to generate a collection of synthetic documents. It takes a pre-existing corpus and uses the 'produced' directory as the starting point to build a synthetic collection similar to the original. The utility takes some statistical information as arguments, which it uses to generate the synthetic collection. To see the options the utility provides, type the command

```
make_synth_collection.pl --help
```

For this tutorial, we will generate a collection with the name SynthCollection from the files downloaded in Section 6.1 of this document. First, if you have already completed Section 6.1, rename the directory 'corpus' to 'source' to make things clearer:

```
mv corpus source
```

If you haven't already completed that section of the tutorial, use the first few commands in Section 6.1 to download the required documents into 'source' and generate and index a corpus from 'source':

```
mkdir source
cd source
wget -r -nd -nc http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
directory_to_corpus.pl --corpus chemical --base produced \
 --directory source
index_corpus.pl --corpus chemical --base produced
```

Now, we will use the 'chemical' corpus in the directory 'produced' as a base to generate a synthetic collection. The collection's size will be 20 documents. The terms used in the synthetic documents will be chosen from the input corpus using a Zipfian distribution (with alpha = 1) with respect to term frequency. (That is, terms ranked higher in term frequency in the original corpus are more likely to appear in the synthetic collection. The probability is inversely proportional to rank * alpha, as dictated by Zipf's Law.) The lengths of the synthetic documents will mirror the lengths of the original documents.

```
make_synth_collection.pl --output SynthCollection --directory synth_out \
 --corpus chemical --base produced --size 20 --term-policy zipfian \
 --term-alpha 1 --doclen-policy mirror --verbose
```

This will generate the synthetic collection in the directory 'synth_out/'. The –size argument specifies the number of documents in the synthetic collection. –term-policy is the method used to pick terms from the source corpus to include in the synthetic collection. –doclen-policy is the method used to determine the lengths of the documents. The other 'term' and 'doclen' arguments are statistical variables that vary depending on the policies used. That is, different –term-policy and –doclen-policy arguments warrant the inclusion of different variables as arguments to the utility (as delineated in the –help command).

One special term-policy is 'manualweights', which requires the user to provide a file containing a list of weights corresponding to all unique terms in the source corpus sorted from most to least frequent. One way of determining the rank of each term and modify it is to look at the file 'source_tc.txt', generated in the output directory (in this tutorial, 'synth_out').

Another interesting functionality of make_synth_collection.pl is its ability to not only generate random documents based on a distribution of terms and their frequencies, but also by n-grams. Rather than simply taking into account the term count data of the source corpus, make_synth_collection.pl can use the CMU-LM toolkit to extract n-grams from a corpus of files, then generate a collection of documents based on n-gram frequencies. Clairlib currently supports 2-, 3-, and 4-grams (as well as 1-grams, as demonstrated earlier in this section).

For this section of the tutorial, we will first create a corpus out of a file included with Clairlib, 11sent.txt. The following set of commands is reused in Section 7.13 of the tutorial:

```
cp $CLAIRLIB-HOME/corpora/11sent/11sent.txt ./
lines_to_docs.pl --input 11sent.txt --output 11sent_source
directory_to_corpus.pl --corpus 11Sent --base 11sent_produced \
 --directory 11sent_source
index_corpus.pl --corpus 11Sent --base 11sent_produced
```

Now, we have a corpus based on 11sent.txt (called 11Sent) in the directory 11sent_produced/. We can now use make_synth_collection.pl to generate 11 documents similar to the 11Sent corpus, based on 3-grams extracted from the source:

```
make_synth_collection.pl --output NgramCorpus --directory trigram_synth_out \
--corpus 11sent --base 11sent_produced --size 11 --ngram 3 --filetype text \
--doclen-policy mirror --verbose
```

The collection of documents found in the directory trigram_synth_out/ is composed entirely of 3-grams extracted from 11sent.txt. Notice that the command we used to generate this collection doesn't include a '–term-policy' argument. Clairlib uses RandomDistributionFromWeights in the finite state machine to decide on the next n-gram to use. Because this process is different from the one used during document generation using 1-grams, a '–term-policy' argument is not needed.

Note that when we generated a synthetic collection using 1-grams (i.e., terms), we did not supply an '–ngram' argument. This is because make_synth_collection.pl defaults to unigram-based document generation when that argument is not given. Also note that a '–filetype' argument is required whenever '–ngram' is greater than 1. The acceptable values for this argument are (text, html, stem), in compliance with Clair::Document.

8.9 Generate Random Synthetic Network

In this tutorial, we will generate a random Erdos-Renyi graph with restrictions using 'generate_random_network.pl'. This utility can either take a pre-existing graph file and randomize it or create a file from scratch. Two different models can be generated: gnm and gnp. A gnm graph has a set number of nodes and edges, with random attachment. A gnp graph is a set number of nodes with an edge existing between two nodes with probability p. For the purposes of this tutorial, we will first generate an undirected gnp graph by providing the number of nodes and the probability that an edge exists between two nodes.

```
generate_random_network.pl --undirected --output random_one.graph \
 --type erdos-renyi-gnp -n 20 -p 0.42 --stats
```

If we specify an input file, the utility will use the file's number of nodes (and number of edges, if we generate a gnm graph) to generate the random graph. The delimiter expression used to distinguish vertices is by default ”[ˆ]+” but it can be changed with ’–delimiter $DELIMITER’ where $DELIMITER is the character used in the input file. We will demonstrate this utility’s ability to take an input file by generating a directed gnm graph from the graph we generated earlier.

```
generate_random_network.pl --input random_one.graph \
 --output random_two.graph --type erdos-renyi-gnm --stats
```

8.10 Extract N-Grams From a Batch of Files

In this tutorial, we will extract n-grams from a batch of files and list them in a single text file along with the frequency with which they appear. To do this, we will use the utility ’extract_ngrams.pl’ and the text file 11sent.txt. ’extract_ngrams.pl’ takes either text files or html as input and can take multiple files. It extracts all n-grams (given n), and it has a few more options, as explained by

```
extract_ngrams.pl --help
```

To find all 3-grams in 11sent.txt (with sentences segmented) and sort them in descending order, enter

```
 extract_ngrams.pl -r "11sent.txt" -f text \
 -w 11sent_3grams.txt -N 3 --segment --sort --verbose
```

This should create the file ’11sent_3grams.txt’ containing all 3-grams in 11sent.txt, with each sentence treated as a discrete entity. As the help command indicates, we can also extract n-grams from multiple files using the asterisk (’*’) in the input file expression. For example, to extract n-grams in all files starting with ”doc”, use ”doc*”.

The ”extract_ngrams.pl” script gives you the option to use either Clairlib or CMU-LM to extract the n-grams. By default, Clairlib will be used. If you want to use the CMU-LM instead, pass the ”engine” option with value ”CMU_LM”

```
extract_ngrams.pl -r "11sent.txt" -f text -w 11sent_3grams.txt \
-N 3 --segment --sort --engine CMU_LM --verbose
```

8.11 Generate Cosine Similarity Statistics for a List of Sentences

In this tutorial, we will take an input file containing a list of sentences, convert it into a corpus with one line per document, then generate detailed cosine similarity statistics for that corpus. We will use the utility *list_to_cos_stats.pl*, a rather simple utility whose options can be seen using the command

```
list_to_cos_stats.pl --help
```

Once again, we will use the file 11sent.txt, which is 11 lines long, each line containing a sentence. The utility will parse the text file and create 11 documents in the corpus, each one containing one line. Then, the utility will create files containing detailed cosine similarity statistics for this corpus. The level of detail we want for the cosine threshold statistics is 0.05 (the argument *–step*), and the utility will create the directories *sentences_produced* and *sentences_data*.

```
list_to_cos_stats.pl --corpus sentences --base sentences_produced \
 --data sentences_data --input 11sent.txt --step 0.05
```

8.12 Create a Lexical Network from a Corpus

In this tutorial, we will create a lexical network from a corpus. We will use the utility *corpus_to_lexical_network.pl*, which will take as input a pre-indexed corpus and produce a graph where each node is a word and an edge exists between two words if they occur in the same sentences. Multiple occurrences are weighted more. To see this utility's usage details, use the command

```
corpus_to_lexical_network.pl --help
```

First off, we will copy the file *11sent.txt* (*http://belobog.si.umich.edu/clair/clairlib/11sent.txt*) to our current working directory. Then, we will create a collection of documents with one line from 11sent.txt each using a utility called *lines_to_docs.pl*. Then, we will create a corpus from that collection and index it.

```
lines_to_docs.pl --input 11sent.txt --output 11sent_source
directory_to_corpus.pl --corpus 11Sent --base 11sent_produced \
 --directory 11sent_source
index_corpus.pl --corpus 11Sent --base 11sent_produced
```

Next, we can directly use the utility to stem the corpus create a lexical network, *11sent.graph*:

```
corpus_to_lexical_network.pl -c 11Sent -b 11sent_produced \
 -o 11sent.graph --stem --verbose
```

Now, our output file, *11sent.graph* contains a lexical network of stemmed terms.

8.13 Create a Linked Corpus Out of a Pre-generated Synthetic Collection

In this tutorial, we will create a linked corpus out of a pre-generated synthetic collection using the utility *link_synthetic_collection.pl*. First off, we need to generate a synthetic collection to pass to this utility. We will use *make_synth_collection.pl* to create a collection of documents called *SynthCollection* in the directory *synth_out*, as demonstrated earlier in this tutorial:

```
mkdir source
cd source
wget -r -nd -nc http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
directory_to_corpus.pl --corpus chemical --base produced \
--directory source
index_corpus.pl --corpus chemical --base produced
make_synth_collection.pl --output SynthCollection --directory synth_out \
--corpus chemical --base produced --size 20 --term-policy zipfian \
--term-alpha 1 --doclen-policy mirror --verbose
```

Now, we can use *link_synthetic_collection.pl* to link this collection of documents and create a corpus. *link_synthetic_collection.pl* provides a number of policies to use when generating the synthetic corpus. Each link policy requires various arguments, as explained by the command:

```
link_synthetic_collection.pl --help
```

For this tutorial, we will use the Watts-Strogatz option, which requires the arguments *-p* (link probability) and *-k* (number of neighbors per node).

```
link_synthetic_collection.pl -n SynthCorpus -b synth_corpus\
 -c SynthCollection -d synth_out -l watts -p 0.42 -k 3
```

This command will look in the directory *synth_out/* for a collection of documents called *SynthCollection*. Then, it will create a directory called *synth_corpus/* and link *SynthCollection* to create a corpus called *SynthCorpus*.

8.14 Cosine Similarities and Lexrank Distribution Tutorial

In this tutorial, we will calculate the cosine similarities and Lexrank distribution for every pair of lines in a file. To do this, we will use the *compute_lexrank.pl* utility. It will create a node for each line in the provided text file and place them in a cluster. Then, it will calculate the cosine similarity matrix and Lexrank transition probabilities for each pair of nodes in the cluster and sort the nodes in descending order of Lexrank centrality. For the purposes of this demonstration, we will use 11sent.txt, provided in *http://www.clairlib.org/mediawiki/index.php/Corpora*

```
compute_lexrank.pl 11sent
```

This will create 3 files in the same directory as 11sent.txt:

- *11sent.cos* contains the cosine similarity matrix
- *11sent.lr* contains each line arranged in descending order of Lexrank centrality
- *11sent.prob* contains the transition probabilities used in Lexrank for each pair of nodes.

8.15 Text Summarization

8.15.1 Single Document Summarization

In this tutorial you will learn how to use Clairlib to summarize a text document or html page. The basic idea behind text summarization is to identify the most important pieces of information from the document, omitting irrelevant information and minimizing details, and assemble them into a compact coherent report. One way to determining which pieces are most important is by splitting the document into pieces (e.g. sentences) and then computing some textual feature for each sentence, such as length, centroid, position, similarity to the first piece. The values of the features of each sentence are then combined in some way to score it. The sentences with the highest and scores are finally returned as a summary. This method of automatic summarization is called **Sentence Extraction** and it is the one implemented in Clairlib and discussed in this tutorial.

For the purpose of this tutorial, we will use an HTML formatted news article about a Gulf Air Airbus that crashed off the coast of Bahrain in the Persian Gulf on August 23, 2000 as a sample document. The article can be downloaded from **http://belobog.si.umich.edu/clair/clairlib/gulf.html**

Document Summarization Process
The process of summarizing a single text or html document in Clairlib involves the following steps:

- Read in the file and create a Clair::Document object.
- Split the document into sentences.
- Calculate the sentences features.
- Combine the features values to score the sentences.
- Return the sentences with highest scores.

The details of how to do these steps using Clairlib follows.

Read in the file and create a Clair::Document object
The first step is to create a new *Clair::Document* object. The file name is passed to the constructor as a parameter.

```
  use Clair::Document;
  $file = "gulf.html";
  my $doc = new Clair::Document(type=>"html", file=>"gulf.html");
  $doc->strip_html();
```

The last line in the code above removes all html tags from the document and saves the resulting string as the text of the Document object.

Split the document into sentences
As was mentioned in the introduction of this tutorial, summarization starts by splitting the document into pieces (sentences, in our case). This can be easily done by calling the *split_into_sentences* subroutine of the Document object.

```
$doc->split_into_sentences();
```

The *split_into_sentences* subroutine splits the document into an array of sentences and store it internally in the Document object.

Calculate the sentences features
In this tutorial, we will calculate four features for each sentence, the length, the position, the centroid, and the similarity to the first sentence in the document. The *compute_sentence_features* subroutine of the Document object takes a hash of features as input and computes the given features for each of the document sentences. The hash of features should have the name of the feature as the key and a reference to a subroutine that calculates the feature as the value. *Clair::SentenceFeatures* module provides several subroutines to compute the features of a sentence. We will make of these functions to calculate our desired features.

```
use Clair::SentenceFeatures qw(length_feature
                               position_feature
                               sim_with_first_feature
                               centroid_feature);
# define the features hash
my %features = (
   ’length’ => \&length_feature,
   ’position’ => \&position_feature,
   ’simwithfirst’ => \&sim_with_first_feature,
   ’centroid’ => \&centroid_feature
);
$doc->compute_sentence_features(%features);
```

Since the computed values are for different features (and thus are of different scales), those values need to be normalized (i.e make all the values of all the features between 0 and 1). The *normalize_sentence_features* subroutine does this. It takes as input an array of the names of the features to be normalized.

```
@features_names = keys %features
$doc->normalize_sentence_features(keys %features);
```

Combine the feature values to score the sentences
The next step is to combine the values of the features computed in the previous section to score the sentences. The *score_sentences* subroutine of the Document object scores the sentences using a given combiner subroutine. The combiner subroutine should be passed a hash containing feature names mapped to their values and should return a real number as a score. By default, the sentence scores will be normalized unless a *normalize* argument is passed and set to 0. Alternatively, if a weights argument is specified and hash is specified and hash of weights for the features is passes, then the returned score will be a linear combination of the features specified in the hash according to their given weights. This option will override the combiner parameter. In this tutorial we’ll use the weights option and weight all the features equally.

```
# Create the weights hash
my %weights=();
# weight all the features equally.
$weights{"lenght"}=1;
$weights{"position"}=1;
$weights{"simwithfirst"}=1;
$doc->score_sentences( weights => \%weights );
```

Return the sentences with highest scores
Now we have all the sentences scored based on the desired features. The last step is to pick the sentences with highest scores and return them as the summary of the original document. The *get_summary* subroutine of the Document object returns the highest scored sentences. You can limit the maximum number of the returned sentences by passing a *size* argument and you can also choose whether to preserve the sentences order as in the original document or to order them on their scores.

```
@summary = $doc->get_summary(size => 5, preserve_order => 1);
```

The returned *summary* is an array of hash references. Each hash reference represents a sentence and contains the following key-value pairs:

- index - The index of this sentence (starting at 0).
- text - The text of this sentence.
- features - A hash reference of this sentence's features.
- score - The score of this sentence.

You can use the *Data::Dumper* to see the structure and the values of the returned array

```
use Data::Dumper;
print Dumper(@summary);
```

You can also loop through the array and print the summary sentences.

```
foreach my $sent (@summary) {
   print "$sent->{text} ";
}
```

summarize_document.pl utility
If you need to summarize a document and output the summary to a file you can also use the *summarize_document.pl* utility as follows:

```
summarize_document.pl --input gulf.html --type html --output summary.txt \\
--length 0.7 --position 0.8 --max_sentences 5
```

For more information about how to run this script run

```
summarize_document --help
```

8.15.2 Multi-Document Summarization

In this subsection of the tutorial you will learn how to summarize multiple documents into a single condense paragraph. As we did in the **Single Document Summarization**, we will use the **Sentence Extraction** method but instead of considering the sentences from one document, we will consider the sentences of a set of documents.

We will use a set of news articles all talking about the Gulf Air Airbus that crashed off the coast of Bahrain in the Persian Gulf on August 23, 2000 as a sample set of documents. The collection can downloaded from

http://belobog.si.umich.edu/clair/clairlib/gulf.tar.gz.

Multi-Document Summarization Process The process of summarizing multiple documents in Clairlib involves the following steps:

- Create a cluster of all the documents.
- Compute the values of the desired features for all the sentences of all the documents in the cluster.
- Combine the features values to score the sentences.
- Return the sentences with highest scores as the summary.

The details of how to implement these steps using Clairlib follows.

Create a cluster of all documents
First, extract the articles files

```
tar xvf gult.tar.gz
```

Let’s assume that files are extracted in ”./gulf”. Prepare a list of all the articles files.

```
@docs = glob("./gulf/*");
```

Then, create a new *Clair::Cluster* object.

```
 use Clair::Cluster;
 my $cluster = new Clair::Cluster();
```

Add the documents to the cluster

```
$cluster->load_file_list_array(\@docs, type => "text", filename_id => 1);
```

The *load_file_list_array* subroutine reads in the files, creates a new *Clair::Document* object of type *text* for each file, assigns an incrementing number as a unique id for each Document object, and adds the Document objects to the cluster.

Compute the values of the desired features for all the sentences of all the documents in the cluster
We will compute three features for each sentence, the length, the position, and the similarity to the first sentence in the document. *compute_sentence_features* subroutine of the Cluster object takes a hash of features as input and computes the given features for each of the sentences of all the document in the cluster. The hash of features should have the name of the feature as the key and a reference to a subroutine that calculates the feature as the value. *Clair::SentenceFeatures* module provides several subroutines to compute the features of a sentence. We will make of these functions to calculate our desired features.

```
use Clair::SentenceFeatures qw(length_feature
                               position_feature
                               sim_with_first_feature
                               centroid_feature);
# define the features hash
my %features = (
   ’length’ => \&length_feature,
   ’position’ => \&position_feature,
   ’simwithfirst’ => \&sim_with_first_feature
);
$cluster->compute_sentence_features(%features);
```

Since the computed values are for different features (and thus are of different scales), those values need to be normalized (i.e make all the values of all the features between 0 and 1). *normalize_sentence_features* subroutine does this. It takes as input an array of the names of the features to be normalized.

```
@features_names = keys %features
$cluster->normalize_sentence_features(keys %features);
```

Combine the features values to score the sentences
The next step is to combine the values of the features computed in the previous section to score the sentences. The *score_sentences* subroutine of the Cluster object scores the sentences using a given combiner subroutine. The combiner subroutine should be passed a hash containing feature names mapped to their values and should return a real number as a score. By default, the sentence scores will be normalized unless a *normalize* argument is passed and set to 0. Alternatively, if a weights argument is specified and hash is specified and hash of weights for the features is passes, then the returned score will be a linear combination of the features specified in the hash according to their given weights. This option will override the combiner parameter. In this tutorial we'll use the weights option and weight all the features equally.

```
# Create the weights hash
my %weights=();
# weight all the features equally.
$weights{"lenght"}=1;
$weights{"position"}=1;
$weights{"simwithfirst"}=1;
$cluster->score_sentences( weights => \%weights );
```

Return the sentences with highest scores
Now, we have all the sentences scored based on the desired features. The last step is to pick the sentences with the highest scores and return them as the summary of the original set of documents. The *get_summary* subroutine of the Cluster object returns the highest scored sentences. You can limit the maximum number of the sentences in the summary by passing a *size* argument. You can also choose whether to preserve the order of the sentences as in the original document or to order them on their scores.

```
@summary = $doc->get_summary(size => 5, preserve_order => 1);
```

The returned *@summary* is an array of hash references. Each hash reference represents a sentence and contains the following key/value pairs:

- index - The index of this sentence (starting at 0).
- text - The text of this sentence.
- features - A hash reference of this sentence's features.
- score - The score of this sentence.

You can use the *Data::Dumper* to see the structure and the values of the returned array

```
use Data::Dumper;
print Dumper(@summary);
```

You can also loop through the array and print the summary sentences.

```
foreach my $sent (@summary) {
   print "$sent->{text} ";
}
```

summarize_corpus.pl utility

If you have your documents formatted as a Clairlib corpus, you can summarize those documents by using the *summarize_corpus.pl* utility as follows:

```
summarize_corpus.pl --corpus Gulf --base produced --type text \\
--output summary.txt --length 0.7 --position 0.8 --max_sentences 5
```

For more information about how to use this script, run

```
summarize_corpus --help
```

8.15.3 Using Mead as a summarizer with Clairlib

MEAD is a publicly available toolkit for multi-lingual summarization and evaluation. The toolkit implements multiple summarization algorithms (at arbitrary compression rates) such as position-based, Centroid, TF*IDF, and query-based methods. Methods for evaluating the quality of the summaries include co-selection (precision/recall, kappa, and relative utility) and content-based measures (cosine, word overlap, bigram overlap). You can download the latest version of MEAD from **http://www.summarization.com/mead**.

The *Clair::MEAD::** modules forms an interface to MEAD. We'll use *Clair::MEAD::Wrapper* to access MEAD summarization functionalities.

For the purpose of this tutorial, we will use a set of news articles all talking about the Gulf Air Airbus that crashed off the coast of Bahrain in the Persian Gulf on August 23, 2000 as a sample set of documents. (http://belobog.si.umich.edu/clair/clairlib/gulf.tar.gz).

Summarization Process

The summarization process should go through the following steps:

- Create a cluster of all the documents.
- Create a Clair::Mead::Wrapper object.
- Specify the summarization options.
- Run MEAD and return the summary.

The details of how to implement these steps using Clairlib follows.

Create a cluster of all the documents

First, extract the articles files

```
 tar xvf gult.tar.gz
```

Let's assume that the files are extracted in "./gulf". Prepare a list of all the article files.

```
 @docs = glob("./gulf/*");
```

Then, create a new Clair::Cluster object.

```
  use Clair::Cluster;
  my $cluster = new Clair::Cluster();
```

Add the documents to the cluster

```
$cluster->load_file_list_array(\@docs, type => "text", filename_id => 1);
```

The *load_file_list_array* subroutine reads the files, creates a new *Clair::Document* object of type *text* for each file, assigns an incrementing number as a unique id for each Document object, and adds the Document instances to the cluster.

Create a Clair::Mead::Wrapper object

The *Clair::MEAD::Wrapper* module is a wrapper for MEAD that enables you to access MEAD functionalities. To create a new *Clair::MEAD::Wrapper* object, use

```
use Clair::MEAD::Wrapper;
my $mead = Clair::MEAD::Wrapper->new(
    mead_home => "/path/to/mead";
    cluster => $cluster
);
```

The *mead_home* argument tells Clairlib where to find a MEAD instance. Change its value to the path of your MEAD installation. The *cluster* argument is a *Clair::Cluster* object containing the documents to summarize. We pass the *$cluster* object that we created in the previous subsection as a value for this argument.

Specify the summarization options

To control the way MEAD works, you can specify some options and pass them to mead using the *add_option* subroutine of the Wrapper object. Some of the common options are:

- **-sentences, -s:** produce a summary whose length is either an absolute number or a percentage of the number of sentences of the original cluster. (This is the default.)
- **-words, -w:** produce a summary whose length is either an absolute number or a percentage of the number of words of the original cluster.
- **-percent num, -p num, -compression_percent num:** produce a summary whose length is num% the length of the original cluster. (The default is -percent 20)
- **-absolute num, -a num, -compression_absolute num:** produce a summary whose length is num (words/sentences) regardless of the size of the original cluster. NOTE: if both -percent and -absolute are specified, MEADs behavior may be erratic.
- **-system RANDOM, -RANDOM:** produce a random summary (and name the system RANDOM).
- **-system LEADBASED, -LEADBASED:** produce a lead-based summary, selecting the first sentence from each document, then the second sentence, etc.. (and name the system LEADBASED). NOTE: RANDOM and LEADBASED systems override any classifier, reranker, and features that may be specified.
- **-lang language:** The default is ENG. This option doesnt really do a whole lot currently. Since mead.pl doesnt currently do Chinese summarization, youll probably never have to specify CHIN. To do summarizaton in Chinese, refer to the appropriate section (youll have to use the old-fashioned meadconfig file method).

For example, if we want to produce a summary whose length is 5% of the number of sentences of the original cluster, we use,

```
$mead->add_option("-s -p 5");
```

Run MEAD and return the summary

The final step is to run MEAD to summarize the given cluster based on the specified options

```
my @summary = $mead->run_mead();
```

The *run_mead* subroutine returns the summary in the form of an array of string sentences. You can print the summary by looping through the array.

```
foreach my $sent (@summary) {
   print "$sent->{text} ";
}
```

8.16 Generate a URL-Based Network out of a Hyperlinked Dataset

In this tutorial we will use some Clairlib utils to generate a URL-based network out of a hyperlinked dataset without the need to index it since indexing takes a long time in large datasets. We will show two ways to do this:

8.16.1 By creating a CL corpus first

This way involves three steps:

- Create a corpus out of the dataset
- Generate the links database of the corpus
- Create the URL-based network

Create a Corpus out of the dataset

For the purpose of this tutorial we'll build a corpus by downloading the dataset files from internet. We will use the chemical elements dataset as an example.

```
download_urls.pl -c chemical \
-i http://belobog.si.umich.edu/clair/corpora/chemical \
-b produced
```

If you have the files already downloaded and stored in some directory on your machine, you can use

```
directory_to_corpus.pl --corpus chemical \
--directory source --base produced --type html
```

where "source" the directory where the dataset is located.

Generate the links database of the corpus

To do this, we'll use index_corpus.pl utility. However, we'll pass some parameters that instruct it to skip the indexing part and only build the files needed for the next step (i.e. building the URL network of the corpus)

```
index_corpus.pl --corpus checmial \
--base produced --notf --noidf --nostats
```

The –notf, the –noidf, and the –nostats arguments ask the code to skip the indexing steps.

Create the URL-Based Network

To create the URL network, we'll use the corpus_to_network.pl util as follows

```
corpus_to_network.pl -c chemical -b produced -o chemical.graph
```

Where chemical.graph is the name of the resulting graph file.

8.16.2 Without creating a corpus

To do this, lets download the chemical files and store them into a directory ”chemical_src”

```
mkdir chemical_src
cd chemical_src
wget -r -nd -nc http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
```

Then, we will use the directory_to_URL_network.pl utility as follows:

```
directory\_to\_URL_network.pl --directory chemical_src \
--output chemical.graph
```

8.17 Random Walk on Graphs

In this tutorial you’ll learn how to use Clairlib to perform random walk on graphs. To do this we’ll use random_walk.pl utility. This script takes a graph file in edgelist format as input and performs the random walk on it based on several options as described by

```
random_walk.pl --help
```

For example, if you have the following graph ”example.graph”

```
1 2
1 4
2 1
2 3
4 1
4 3
4 5
5 1
3 1
```

and the transition probabilities between the nodes are as follows ”example.trans”

```
1 2 0.30
1 4 0.70
2 1 0.65
2 3 0.35
4 1 0.20
4 3 0.50
4 5 0.30
5 1 1.00
3 1 1.00
```

The probabilities after walking 100 random steps for 500 rounds, can be found by running

```
random_walk.pl --grpah example.graph --transition_file example.trans \
--steps 100 --rounds 500 --output example100.prob
```

The computed probabilities will be outputed to ”example100.prob” and will be something like,

```
5 0.098
2 0.098
3 0.168
1 0.350
4 0.286
```

To compute the stationary distribution (after walking so many random steps), run the following command,

```
random_walk.pl --grpah example.graph --transition_file example.trans \
--stationary --output example.prob
```

The content of the file ”example.prob” after running the command will be something like this

```
5 0.07
2 0.11
3 0.17
1 0.37
4 0.26
```

8.18 Generate a random document from bigrams using Random Walk

In this tutorial you'll learn how to use Clairlib to generate a random document by first extracting the bigrams of that document then build a network out of these bigrams and perform a random walk on that graph printing the words as the corresponding nodes are visited. The transition probabilities on the edges depend on the frequency of the bigrams.

To do this, we will use the utility *bigrams_to_random_doc.pl* and the text file *11sent.txt*. For information on the usage of *bigrams_to_random_doc.pl*, run

```
bigrams_to_random_doc.pl --help
```

First, extract the bigrams from *11sent.txt* by running

```
extract_ngrams.pl -r "11sent.txt" -f text -w 11sent.2grams \
-N 2 --segment --verbose
```

The *–segment* option here tells the script to segment the text into sentences and add the delimiter *¡s¿* at the end of each sentence.

Then run *bigrams_to_random_doc.pl*

```
bigrams_to_rand_doc.pl --input 11sent.2grams \
--output 11sent.synth --size 5 --delim <s>
```

This will generate a random document of 5 sentences maximum.

8.19 Perceptron Learning and Classification

In this tutorial, you'll learn how to use Clairlib for extracting feature vectors from a set of document, how to train a model using the perceptron learning algorithm given the feature vectors of a training set of documents, and how to classify a set of testing documents given their feature vectors using the the perceptron classification algorithm.

8.19.1 Tutorial Materials

For the purpose of this tutorial, we will use two sets of XML formated documents taking about ”sport”, one of the two sets is for training a model and the other one is for testing the trained model. You can download the training set from http://belobog.si.umich.edu/clair/clairlib/training.tar

wand the testing set from http://belobog.si.umich.edu/clair/clairlib/testing.tar
Extract both datasets to two directories, ”training” and ”testing”

```
tar -xvf training.tar.gz
tar -xvf testing.tar.gz
```

8.19.2 Extracting the feature vectors

We start by extracting the feature vectors for both the training and the testing sets. To do this, we will use the ”extract_features.pl” utility that comes with Clairlib and outputs the feature vectors for a set of documents in svm_light format. For information on the usage of this utility, run

```
extract_features.pl --help
```

To extract the features for the training set, run

```
extract_features.pl --directory training --output features.train \
--parser sports --mode train --select 100 --verbose
```

This will compute the feature vectors for the training documents located in the ”training” directory and then use the chi-square method to select the top 100 most descriminitive features, and finally outputs the vectors to the ”features.train” files in the svm-light format.

To extract the feature vectors for the testing dataset, run the same command with the proper arguments

```
extract_features.pl --directory testing --output features.test \
 --parser sports --mode test --select 100 --verbose
```

8.19.3 Training a model using the Perceptron algorithm

Now, we can pass the training set features file to the perceptron algorithm to train a model. To do this, we will use the ”learn.pl” utility that takes a training features file and trains a model. For information on the usage of this utility run

```
learn.pl --help
```

To use our training dataset for training a model, run

```
learn.pl --train_features features.train --model model_file \
--eta 0.5 --verbose
```

This command will write the trained model to the ”model_file” file.

8.19.4 Classifying the testing set using a trained model

Now, we can use the trained model to classify the testing set. To do this, we will use the ”classify.pl” utility which takes the testing features file and the model file as arguments and outputs the classification result in the following format

```
docid   score   computed_class   correct_class   y|n
```

For more information on the usage of this utility, run

```
classify.pl --help
```

To use this utility to classify the testing set by running

```
classify.pl --test_features features.test --model model_file \
--output results --verbose
```

The result of the classification will be written to the file and some statistics will be printed on the screen.

8.20 Gene Interaction Extraction

In this tutorial you will learn how to use Clairlib in one Bioinformatics application; i.e. Gene Interaction Extraction. Specifically, you will learn to do the following:

- Tag the Genes in a given sentence.
- Tag the interaction words in a sentence.
- Extract the interactions from a given sentence.

To be able to perform the tasks above, you need to make sure that you installed both the Geneia Tagger and the Stanford Parser, and configured Clairlib to properly use them by uncommenting the following lines and pointing the variables in them to the correct paths.

```
 $STANFORD_PARSER_PATH = "/path/to/stanford-parser";
 $GENIATAGGER_PATH = "/path/to/genia-tagger";
```

Then, create an instance of *Clair::Bio::GIN*

```
use Clair::Bio::GIN;
my $gin = new Clair::Bio::GIN();
```

8.20.1 Tag the Genes in a given sentence

If you have the sentence

```
my $sentence = "We provide evidence of a cross-talk between \
nuclear receptor and Ser / Thr protein phosphatases and show \
that vitamin D receptor (VDR) interacts with the catalytic \
subunit of protein phosphatases, PP1c and PP2Ac, and induces \
their enzymatic activity in a ligand-dependent manner.";
```

You can tag the genes in the that sentence by

```
my @genes = @{$gin->tag_genes($sentence)};
foreach my $gene (@genes){
     print $gene,"\n";
}
```

8.20.2 Tag the interaction words in a sentence

Or, you can tag and print the interaction words

```
my @int_words = @{$gin->tag_interaction_words($sentence)};
foreach my $word (@int_words){
     print $word,"\n";
}
```

8.20.3 Extract the interactions from a given sentence

To extract the interactions from the sentence use

```
@interactions = $gin->extract_interactions;
```

The returned array is an array of *Clair::Bio::GIN::Interaction* instances. You can print the information of those interactions using

```
foreach my $interaction (@interactions){
    print "Gene 1: ", $interaction->get_gene1(),"\n";
    print "Gene 2: ", $interaction->get_gene2(),"\n";
    print "Interaction Word: ",$interaction->get_interaction_word(),"\n\n";
}
```

8.21 ACL Anthology Network

One of the useful corpora that was recently added to the Clairlib distribution is AAN, the ACL Anthology Network. AAN is a collection of scientific publications in the NLP area, collected from many ACL venues.The network is currently built using 13,706 of the ACL papers. This includes all papers up to and including those published in November 2008 which were successfully processed. From those papers, we have created several networks. Those networks include:

- **Paper citation:** A directed network composed of nodes. Each node corresponds to a paper and each edge represents a reference from one paper to another.
- **Paper citation network without self citations:** The same as above but edges that represent self citations are dropped.
- **Author citation:** A directed network composed of nodes. Each node corresponds to an author and each edge represents a reference from one author to another.
- **Author citation network without self citations:** The same as Author citation but edges that represent self citations, where an author references himself are dropped.
- **Authors collaboration:** An undirected network where nodes represent authors and edges represent instances where one author coauthored a paper with another author

AAN data files and some useful scripts to process them have been added to Clairlib. All the AAN related files can be found under $PATH-TO-CLAIRLIB/aan/. A full description of the data and statistics files, and the usage of the scripts can be found in the release.README file in the aan path. In this chapter, we will show how to use the scripts to generate AAN networks from data files and then gather useful statistics and information from them.

8.21.1 Generating AAN Networks

This distribution of AAN includes two data files, acl-metadata.txt and acl.txt, from which we can creat the different networks and statistics using in-house scripts. In this section we show you how to use the scripts to generate the aan networks we mentioned above.

Generate Paper Citation Network

```
    aan_make_paper_citations.pl
```

Generate Paper Citation Network excluding self citations

```
    aan_make_paper_citations.pl --nonself
```

Generate Author Citation Network

```
    aan_make_author_citation.pl
```

Generate Author Citation Network excluding self citations

```
    aan_make_author_citation.pl -nonself
```

Generate Author Collaboration Network

```
    bin/aan_make_author_collaboration.pl
```

All networks generated above are formatted using the Edgelist format, which lists a single edge per line. An edge is formatted as ”`Node1_label ==> Node2_label`”.

8.21.2 Basic Statistics

The main script that can be used to generate statistics for any of the networks metioned above is `aan_network_stats.pl` which has the following format:

```
aan_network_stats.pl -input=acit|acoll|pcit
[--delimout=output_delimiter] [--output=output_file]
[-pajek=pajek_file] [--stats] [--graphml=graphml_file]
[--sample=sample_size] [--sampletype=sample_type] [--extract]
[--components] [--undirected] [--paths] [--wcc] [--cc] [--scc] [--triangles]
[--assortativity] [--verbose] [--localcc] [--all] [--betweenness-centrality]
[--degree-centrality] [--closeness-centrality] [--lexrank-centrality]
[--force] [--graph-class=graph_class] [--filebased] [--help]
```

Some examples of how to use this script are:

- To generate the basic statistics of the author citation network:

```
        aan_network_stats.pl -input="acit" --stats
```

- To generate the statistics of the paper citation network and output the result to a file in Pajek compatible format:

```
        aan_network_stats.pl -input="pcit" pajek "pajekfile"
```

- To generate the statistics of the author collaboration network while treating the network as undirected:

```
          aan_network_stats.pl -input="acoll" -undirected
```

- To generate the betweeness,degree and closeness centrality scores for every author based on the author citation network:

```
          aan_network_stats.pl -input="acit" --degree-centrality
          --betweenness-centrality --closeness-centrality
```

- To generate statistics for 100 samples of the authors network where samples are drawn using the randomnode algorithm :

```
          aan_network_stats.pl -input="acit"
          --sample 100 --sampletype randomnode --all
```

You can also count the number of citations and collaborations for authors and papers. There are three scripts that help doing that: aan_author_citations.pl, aan_author_citations.pl, and aan_author_collaborations.pl. Some examples of how to use them are:

- To get the number of all citations for every author provided that they are older than 2005

```
          aan_author_citations.pl -year 2005
```

- To get the number of incoming citations for every paper excluding self citations .

```
          aan_author_citations.pl -incites -nonself
```

- To get the number of collaborations for every author

```
          aan_author_collaborations.pl
```

8.21.3 PageRank Scores

You can get the PageRank scores for papers or authors using `aan_pageranks.pl` script. For example:

- To get the PageRank scores of every paper

```
          aan_pageranks.pl -input="pcit"
```

- To get the PageRank scores of every author

```
          aan_pageranks.pl -input="acit"
```

8.21.4 H-index

You can also get the H-index for every author using `aan_hindex.pl` script. For example

- To get the H-index for every author after excluding self citations

```
aan_hindex.pl -nonself
```

8.21.5 More Information

- For more information about AAN please visit: http://belobog.si.umich.edu/clair/anthology/
- For detailed information about scripts and use instructions, see the `release.README` file located in the AAN path in Clairlib.

9 Paper Replications

In this section we show replications for some papers in the NLP area using Clairlib and compare the results of the each paper with Clairlib results.

9.1 Modeling Statistical Properties of Written Text

In this tutorial we use Clairlib and other tools to replicate the work done by M. ngeles Serrano , Alessandro Flammini, and Filippo Menczer in there paper, Modeling Statistical Properties of Written Text, in which they study some statistical properties of written text including zipf's law, burstiness, Heaps' law describing the sublinear growth of vocabulary size with the length of a document, and the topicality of document collections, which encode correlations within and across documents absent in random null models. They also propose a generative model that explains the simultaneous emergence of all these patterns from simple rules.

In this tutorial we will focus on the following properties:

- Zipf's Law
- Heaps' Law
- Similarity Distribution

We study these properties for both the original datasets and for the synthetic collection generated by the model.

9.1.1 Dataset

The authors of the paper selected three diverse public datasets to illustrate their work. These datasets are:

- **The Industrial Sector dataset (IS):** a collection of corporate Web pages organized into categories or sectors.
- **A sample of the Open Directory dataset(ODP):** a collection of 150,000 Web pages classified into a large hierarchical taxonomy by volunteer editors.
- **The Wikipedia:** a random sample of 100,000 topic pages from the English Wikipedia (Wiki).

These datasets are available for download [http://cnets.indiana.edu/groups/nan/text-modeling here]. For the purpose of this tutorial, we will show the code and the results for the IS dataset only.

9.1.2 Create a Clairlib cluster out of the dataset documents

Download and extract the dataset

First, download the dataset and extract it to some directory.

```
wget http://www.cs.umass.edu/~mccallum/data/sector.tar.gz
tar -xvf sector.tar.gz
```

This will extract the collection files to a directory named ”sector”.

Prepare a list of the dataset files

To list all the files in the dataset, use the UNIX command, ”find”, and pipe the results to the Perl command, ”open”.

```
open FILES,"find sector/ -type f -print|";
```

This will execute the ”find” command and write the resulting filenames to a temporary file in the system memory (a filename per line) then assign this to a file handler, FILES.
Next, read the filenames through the handler FILES into and array

```
my @files = <FILES>;
```

Each element in this array corresponds to a filename and ends with the newline character ’
n’. This character must be chopped

```
chomp(@files);
```

Create the cluster
First, instantiate an empty Clair::Cluster object

```
use Clair::Cluster;
$cluster = new Clair::Cluster();
```

Then, use ”load_file_list_array” function of Clair::Cluster to load the files listed in ”@files” into the cluster

```
$cluster->load_file_list_array(\@files, type => "html");
```

9.1.3 Process the cluster files to stem the words and strip the HTML tags

Start by stripping html from the files. To do so, use the ”strip_all_documents” function of Clair::Cluster

```
$cluset->strip_all_documents();
```

Then, stem the words in all the documents by calling the ”stem_all_documents” function of Clair::Cluster

```
$cluset->stem_all_documents();
```

9.1.4 Basic statistics

In this section, we calculate the basic statistics of the IS collection. These statistics include:

- Number of documents in the collection.
- Number of non-empty documents in the collection.

- Number of words in the collection.
- Number of unique words (vocabulary) in the collection.
- Average length of documents in number of words.
- Variance in the length of documents in number of words.
- Average length of documents in number of unique words.

Count the number of files in the collection

To count the number of files in the cluster, simply call the ”count_elements” function of Clair::Cluster.

```
my $docs_count = $cluster->count_elements();
```

To count the number of non-empty files, we count the number of words in each document and count the number of documents whose length is one word at least. The ”documents” function of Clair::Cluster returns a reference for a hash of all the documents in the cluster. The key of this hash is the document ID and the value is a Clair::Document object.

```
\%documents = \%{ $cluster->documents() };
```

Then, loop through the hash values and find the length of each document by calling the ”split_into_words” function and count the number of words in the resulting array

```
my $num_of_empty_docs = 0;
foreach my $doc (values %documents){
   if(scalar($doc->split_into_words(type => ’stem’))>0){
       $num_of_empty_docs++;
   }
}
```

This will give the following results:

- Number of documents in the collection:
  - Clairlib: 9571
  - Paper: 9571
- Number of non-empty documents:
  - Clairlib: 9567
  - Paper: 9556

The differences in the results here and in the rest of this tutorial comes from differences in stemming and stripping methods.

Count the number of unique words (vocabulary) in the collection

To count the number of unique words in the collection, use the ”tf” function of Clair::Cluster which returns a hash of the frequencies of all the terms (unique words) in the collection, then count the number of elements in this hash

```
my %tf = $cluster->tf();
my $num_unique_words = scalar (keys %tf);
```

This will give the following result:

- Clairlib: 51604
- Paper: 47979

Calculate the average length of documents in number of words

This can be easily done by summing the lengths of all the documents then divide the result by the number of documents.

```
%documents = %{ $cluster->documents() };
my $num_of_words = 0;
foreach my $doc (values %documents){
   $num_of_words  += scalar($doc->split_into_words(type => 'stem'));
}
my $average_word_length = $num_of_words / $docs_count;
```

This will give the following result:

- Clairlib: 316.349
- Paper: 313.46

Calculate the variance in the length of documents in number of words

Create an array of the lengths of the documents then pass it a subroutine that calculates the variance of the values given in that array.

```
my %documents = %{ $cluster->documents() };
my @docs_lengths =();
foreach my $doc (values %documents){
       push @docs_lengths,
       scalar($doc->split_into_words(type => 'stem'));
}
my $docs_length_variance = variance(\@docs_lengths);

sub variance
{
   my $array_ref = shift;
   my $n = scalar(@{$array_ref});
   my $result = 0;
   my $item;
   my $sum = 0;
   my $sum_sq = 0;
   my $n = scalar @{$array_ref};
   foreach $item (@{$array_ref})
   {
       $sum += $item;
       $sum_sq += $item*$item;
   }
   if ($n > 1)
   {
       $result = (($n * $sum_sq) - $sum**2)/($n*($n-1));
   }
   return $result;
}
```

This will give the following result:

- Clairlib: 552975
- Paper: 566409

Calculate the average length of the documents in number of unique words

First, use the "get_unique_words" function of Clair::Document to get an array of the unique words of each of the documents. Sum the sizes of the returned arrays then divide the result by the number of the documents.

```
my %documents = %{ $cluster->documents() };
my $sum_doc_lengths_unique = 0;
foreach my $doc (values %documents){
       $sum_doc_lengths_unique += scalar($doc-> get_unique_words());
}
my $avg_length_unique = $sum_doc_lengths_unique / $docs_count;
```

This will give the following result:

- Clairlib: 136.78
- Paper: 124.26

9.1.5 Distribution of document length

According to the paper, the document length distribution can be approximately fitted by a lognormal distribution. Matlab can be used to do such fitting. First, output the lengths of the documents to a file in Matlab compatible format.

```
OPEN LENGTHS, ">docs_lengths"
        or die "can't open file: $!";
foreach my $length (@docs_lengths){
    print LENGTHS $length,"\n";
}
close LENGTHS;
```

"@docs_lengths" is the array we have just created in the previous section.
Next, start Matlab and run the following commands

```
load docs_lengths;
PD = fitdist(docs_lengths, 'lognormal')
```

This will give the following results:

- Clairlib: mu 4.98, sigma 1.16
- Paper: mu 4.81, sigma (ML) 2.10

Note that the paper shows the maximum likelihood estimate for sigma, while in this tutorial, Matlab estimates the value of sigma as the square root of the unbiased estimate of the variance of the log of the data.

9.1.6 Zipf's Law

Zipf's law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. What we need to do here is to fit a zipf distribution to the frequencies of the words and estimate the value of the exponent alpha. We will use the Matlab code provided by Newman at. el. in their paper. Their code is available in http://www.santafe.edu/ aaronc/powerlaws/ http://www.santafe.edu/ aaronc/powerlaws/.

In the previous section built a hash of terms and their frequencies. Write these frequencies to a file in a Matlab compatible format.

```
open FREQ, ">frequencies";
foreach my $freq (values %tf){
    print FREQ $freq,"\n";
}
close FREQ;
```

Then, use the Matlab code we've just mentioned to estimate the value of alpha.

```
load frequencies;
[alpha, xmin, L] = plfit(frequencies);
```

This will give the following result for alpha:

- Clairlib: 1.59
- Paper: 1.78

To plot the distribution on a log-log graph, use this Matlab command

```
loglog(frequencies,'.','color','green');
```

9.1.7 Heaps' Law

Heaps' law describes the sublinear growth of vocabulary size with the length of the document. We'll use Matlab to plot the distribution. To do this, write, for each document in the cluster, its length in the number of words and the length in the number of unique words (vocabulary).

```
open HEAP, ">heap" or die "Can't open file $!";
%documents = %{ $cluster->documents() };
foreach my $doc (values %documents){
   my $length_words = scalar($doc->split_into_words(type => 'stem'));
   my $length_unique_words = scalar($doc-> get_unique_words());
   print HEAP "$length_words $length_unique_words\n";
}
close HEAP;
```

To plot the distribution, use this Matlab code

```
load heap;
s_size=1000;
hh=sort(heap);
s=zeros(s_size,2);
st=floor(size(heap,1)/s_size)
k=1;
for i=1:s_size
   s(i,1)=hh(k,1);
   s(i,2)=hh(k,2);
   k=k+st;
end
x=heap(:,1);
y=heap(:,2);
loglog(x,y,'.','color','blue');
ylabel('w(n)','FontSize',16);
xlabel('n','FontSize',16)
```

9.1.8 Similarity Distribution

To calculate the similarity between all the pairs of documents of the cluster, use the ”compute_cosine_matrix” function of Clair::Cluster. Since the number of documents is great, the similarity computation for all the pairs will take a very long time. Therefore, you can optionally pass a ”sample_size” parameter in which case the similarity will be computed only for a random sample of ”sample_size” pairs.

```
my %cos_matrix = $cluster->compute_cosine_matrix \
(type=>"stem", sample_size=>5000);
```

To plot the similarity distribution, write the cosine values to a file in Matlab compatible format.

```
open COS,">cos" or die "Can’t open file $!";
foreach my $doc1 (keys %cos_matrix) {
     foreach my $doc2 (keys %{ $cos_matrix{$doc1} }) {
            print COS $cos_matrix{$doc1}{$doc2};
     }
}
```

Then, use the following Matlab code to plot the similarity distribution on a semilog graph.

```
load cos;
[n,xout]=hist(cos,100);
xsim = [1:100]/100;
ps=n/size(cos,1);
semilogy(xsim,ps,’.’,’color’,’green’)
axis([0.0 1.0 10ˆ-6 10ˆ0])
ylabel(’p(s)’,’FontSize’,16);
xlabel(’s’,’FontSize’,16)
```

9.1.9 Generative Model

The paper authors proposed a model to generate synthetic documents by drawing the words from the original dataset vocabulary. The document lengths in number of words are drawn from a lognormal distribution. Here we only show the code that implements the second version of the model that has memory mechanism to capture topicality. For more details about the model and its algorithm, refer to the paper. Clair::RandomDistribution::* modules are used to draw the random values from zipfian, lognormal, and power distributions.

```
use Clair::RandomDistribution::LogNormal;
use Clair::RandomDistribution::Zipfian;
use Clair::RandomDistribution::Exponential;
$rnd_length = new Clair::RandomDistribution::LogNormal \
(mean=>4.81, std_dev => 1.45, dist_size=>$num_unique_words);
$rnd_term = new Clair::RandomDistribution::Zipfian \
(alpha=>1, dist_size=>$num_unique_words);
$rnd_z = new Clair::RandomDistribution::Exponential \
(lambda=>0, dist_size=>$num_unique_words);
%r0=();
$rank=1;
foreach $k (sort sortvocab(keys %tf))
{
     $r0{$k}=$rank;
     $rand++;
}
$r_star=1;
for($i=0;$i<$size;$i++)
{
   ‘mkdir synth‘;
   open(FILE,">synth/SI_synth_$i") or die "can’t open file";
   print "r* = $r_star";
   my %counts=();
   $current_r=1;
   foreach $ke (sort (sortvocab keys %tf))
   {
       if($current_r>=$r_star){
           $counts{$ke}=0;
       }
   }
   $len=$rnd_length->draw_rand_from_dist();
   $first=1;
   %doc = ();
   @terms = ();
   foreach $term (sort sortfreq (keys(%counts)))
   {
         push @terms, $term;
   }
   for(my $j=0;$j<$len;$j++)
   {
        $random_term = $rnd_term->draw_rand_from_dist();
        $newterm = $terms[$random_term];
        #print "Term=$newterm\n";
        $counts{$newterm}++;
        $doc{$newterm}=$counts{$newterm};
        $cmp=$ramdom_term-1;
        while ($counts{$newterm}>$counts{$terms[$cmp]} && $cmp>=0){
               $cmp--;
        }
        ($terms[$newterm],$terms[$cmp]) =  \
        ($terms[$cmp], $terms[$newterm]);
   }
   foreach $term (keys %doc)
   {
     print FILE  $term, " ";
   }
   close(FILE);
   print "Finished Synth Doc $i\n";
   $r_star= 1; #$rnd_z->draw_rand_from_dist();
}

sub sortfreq2{
   if($counts{$a}==$counts{$b}){
       if ($r0{a}>$r0{$b}){
            return 1;
       }else{
            return -1;
       }
   }else{
         return $counts{$b} <=> $counts{$a};
   }
}

sub sortfreq{
   $counts{$b} <=> $counts{$a};
}

sub sortvocab{
   $tf{$b} <=> $tf{$a};
}
```

9.2 Network Properties of Written Human Language

In this tutorial we use Clairlib to replicate the work done by Masucci and Rodgers in their paper, *Network Properties of Written Human Language*, in which they investigate the nature of written human language within the framework of complex network theory. Specifically, they analyze the topology of Orwells novel, 1984, focusing on network properties.

9.2.1 Network Properties

The properties that we will calculate for the network that represents the text are:

- **Number of nodes:** each node corresponds to a word or a puncutation.
- **Number of edges:** two nodes are linked by an edge if they are neighbors.
- **Reciprocity value:** which quantifies the non-random presence of mutual edges between pairs of vertices.
- **Mean degree (word frequency):** the number of different words this word is connected. The degree and the frequency of a word have the same meaning, and are equal, because every time a new word is added to the text it is the only vertex of the network to acquire an edge.
- **Degree distribution:** the probability distribution of words degrees over the whole network.
- **Zipf's power law exponent:** the exponent of the power law that relats the word's frequency of occurance to its rank (where words are ranked on their frequency of occurrence.)
- **Growth exponent:** the exponent of the growth in the number of words with respect to time.

9.2.2 Corpus

The corpus used in this tutorial is the well-known novel, Nineteen Eighty-Four (abbreviated to 1984) by English author George Orwell. The novel is available in txt format at **http://www.clairlib.org/mediawiki/index.php/Corpora**.

9.2.3 Convert Text to Network

The first step is to build a network from the novel text. We treat the text as a finite directed network in which the words are the vertices and two vertices are linked if they are neighbors. Punctuation is also considered as vertices. To convert the text to network, we do following steps: Read in the file:

```
$file = "1984.txt";
$text = ‘cat $file‘;
```

Split the text into separate words and store them in an array:

```
$text =˜ s/ \’/ /g;
$text =˜ s/\’ / /g;
$text =˜ s/([ˆA-Za-z])/ $1 /g;
my @words = split /\s+/, $text;
```

Normalize the words’ case by converting each to lower case:

```
my @res = ();
foreach my $w (@words) {
    push @res, lc($w);
}
```

Build the network and write it to a file:

```
open(OUTFILE, ">1984.graph");
my $i=0;
for($i =0; $i<$#words; $i++)
{
   print OUTFILE "$words[$i] $words[$i+1]\n";
}
close (OUTFILE);
```

9.2.4 Basic Network Statistics

These statistics include number of nodes, number of edges, mean degree, and degree power law distribution exponent. All these can be calculated using the clairlib utility script, print_network_stats.pl.

```
  print_network_stats.pl --input 1984.graph --all --force
```

Following are the results that we got versus the results got by Masucci and Rodgers in their paper.

Property	Paper	Clairlib
Number of Nodes	117687	117196
Number of edges	8992	8576
Mean degree	13.1	13.65
Degree power law distribution exponent	-2.1	-1.9

9.2.5 Calculate the reciprocity value

```
use Clair::Network qw($verbose);
use Clair::Network::Reader::Edgelist;
$reader = Clair::Network::Reader::Edgelist->new();
$delim = "[ \t]+";
$filebased = 0;
$fname = "1984.graph";
$net = $reader->read_network($fname,
                               delim => $delim,
                               directed => 1,
                               filebased => $filebased,
                               edge_property => "lexrank_transition",
                               multiedge => 1);
$n = $net->num_nodes();
$l = scalar($net->get_edges());
$a = $l/$n/($n-1);
$mutual = $net->get_mutual_edges_num();
$r = $mutual/$l;
$rho =  ($r - $a)/(1-$a);
print "r=$r\na=$a\nrho=$rho\n";
```

The result is:

- Clairlib 0.021
- Paper 0.0204

9.2.6 Calculate degree growth

```
my $i=0;
my %list = ();
my %hist = ();
my $count = 0;
for($i =0; $i<$#words; $i++)
{
        if(exists $list{$words[$i]})
        {
                $hist{$count} = $i+1;
        }
        else{
          $hist{$count++}= $i+1;
                $list{$words[$i]} = 0;
        }
}
my $reader = Clair::Network::Reader::Edgelist->new();
my $delim = "[ \t]+";
my $filebased = 0;
my $net = $reader->read_network("1984.graph",
                               delim => $delim,
                               directed => 1,
                               filebased => $filebased,
                               edge_property => "lexrank_transition",
                               multiedge => 1);
my @fit = $net->linear_regression(\%hist, log => 1);
print "$fit[0]\n";
```

The result is:

- Clairlib 1.53
- Paper 1.8

9.3 Patterns in syntactic dependency networks

In this tutorial we use Clairlib to replicate the work done by Cancho et. al. in their papers, *”Patterns in Syntactic Dependency Networks”* and *”Universality in syntactic dependency networks”*, in which they study the architecture of syntactic graphs and show that they display small world patterns, scale-free structure, a well-defined hierarchical organization, and assortative mixing. Specifically, three different European languages will be used: Czech, Romanian, and German.

9.3.1 Data Set

Three datasets were used in the paper, Czech, Romanian, and German corpora. The Romanian is the only one freely available. It is available at the Dependency Grammar Annotator website (http://www.phobos.ro/roric/DGA/dga.html). In this tutorial we will deal only with the Romanian corpus.

9.3.2 Network Properties

Using the Romanian corpus, we will build a Syntactic Dependency Network. This network has its nodes correspond to words and its edges correspond to binary relations (syntactic dependencies) between words. Most of the edges are directed and the arc goes from the head word to its modifier or vice versa depending on the convention used. Head and modifier are primitive concepts in the dependency grammar formalism.

After building the network we will calculate some of the properties of this network. These properties are:

- Number of nodes: Where each node corresponds to a word.
- Number of edges: Where each edge represents a syntactic dependency between a head word and a modifier.
- Clustering coefficient: The probability that two nodes (words) that are neighbors of a given node are neighbors of each other.
- Diameter: Average minimum distance between pairs of nodes.
- Power law exponent: The exponent of the power law distribution (if it exists).
- Size of the giant component
- Average shortest path

9.3.3 Convert Corpus to Network

The first step is to parse the corpus and build a syntactic network from it. Read in the corpus XML files:

```
my @files = glob($xml_dir . "/t*.xml");
```

Where *$xml_dir* is the path where the corpus XML files are located. All the xml files has are named *t*.xml* (e.g. *t10.xml* and *tp7.xml*).

Parse the XML files and build the network

```
 use XML::Twig;
 my $twig = new XML::Twig();
 my %all_words = ();
 my $num_sentences = 0;
 my $num_words = 0;
 my $total_words = 0;
 # The maximum nuber of words used to build the network
 my $word_limit = 100000;
 foreach my $file (@files) {
 $twig->parsefile($file) or die "Couldn’t parse $file\n";
 my $root = $twig->root;
 # loop through the sentences in this document
 my @sentences = $root->children;
 $num_words = 0;
 foreach my $sentence (@sentences) {
   # loop through the tokens
   # build up an associative array of the word and head, indexed by position
   # which is then used to resolve the dependencies
   my %tok_table = ();
   my @tokens = $sentence->children;
   $num_sentences++;
   $num_words += scalar(@tokens);
   # skip sentences with less than two words
   if ((scalar(@tokens) >= 2) && ($num_words < $word_limit)) {
     foreach my $token (@tokens) {
       my $word = $token->field("orth");
       my $ctag = $token->field("ctag");
       $word = lc($word);
       $all_words{$word}++;
       my $ordno = $token->field("ordno");
       my $syn = $token->first_child("syn");
       my $head = $syn->field("head");
       $tok_table{$ordno}{"word"} = $word;
       $tok_table{$ordno}{"head"} = $head;
     }
     # Add the edges to the network
     foreach my $key (keys %tok_table) {
       my $word = $tok_table{$key}{"word"};
       my $head = $tok_table{$key}{"head"};
       if (not defined $tok_table{$head}) {
       } else {
         my $head_word = $tok_table{$head}{"word"};
         $lex_network->add_edge($word, $head_word)
       }
     }
   }
 }
 $total_words += $num_words;
}
```

The full code that reads in the files and builds the network is available in *http://belobog.si.umich.edu/clair/clairlib/parse_dg.txt*. You can use this script by running this command

```
parse_dg.pl -o romanian-newspaper-dga.graph
```

This will create a network and write it to a file, *romanian-newspaper-dga.graph*.

9.3.4 Network Statistics

These statistics include number of nodes, number of edges, clustering coefficient, diameter, power law distribution exponent, size of giant component, and average shortest path. All these can be calculated using the clairlib utility script, *print_network_stats.pl*.

```
  print_network_stats.pl -i romanian-newspaper-dga.graph --force
```

The *force* option instructs the script to ignore the 2000 nodes limit. Following are the results that we got versus the results got by Cancho et. al. in their paper.

Property	Paper	Clairlib
Number of nodes	5563	5598
Number of edges	-	14516
Average Degree	5.1	5.12
Size of giant component	5563	5594
Power law exponent	-2.19	-2.19
Clustering Coefficient	0.09	0.09
Average shortest path	3.4	3.49

10 Useful Utilities

In this section we will be using Clairlib utilities to create corpora, generate networks, extract plots and statistics, and demonstrate how to perform other useful tasks. The chapter is organized into the following sections:

1. Generating Corpora
2. Gathering Corpora Statistics
3. Generating Networks
4. Gathering Network Statistics
5. Other Useful Tools

10.1 Generating Corpora

10.1.1 Generate a corpus by downloading files

```
output: indexed corpus

mkdir corpus
cd corpus
wget -r -nd -nc \
  http://belobog.si.umich.edu/clair/corpora/chemical
cd ..
directory_to_corpus.pl -c chemical -b produced -d corpus
index_corpus.pl -c chemical -b produced
```

10.1.2 Generate a corpus by crawling a site

```
output: indexed corpus

crawl_url.pl -u http://www.asdg.com/ -o asdg.urls
download_urls.pl -c asdg -i asdg.urls -b produced
index_corpus.pl -c asdg -b produced
```

10.1.3 Generate a corpus from a Google search

```
output: indexed corpus

search_to_url.pl -q bulgaria -n 10 > bulgaria.10.urls
download_urls.pl -i bulgaria.10.urls -c bulgaria-10 -b produced
index_corpus.pl -c bulgaria-10 -b produced
```

10.1.4 Generate a corpus of sentences from a document

```
input: collection of documents
output: indexed corpus

sentences_to_docs.pl -d $CLAIRLIB/corpora/1984/ -o docs
directory_to_corpus.pl -c 1984sents -b produced -d docs
index_corpus.pl -c 1984sents -b produced
```

10.1.5 Generate a corpus using Zipfian distribution

```
input: indexed corpus
output: synthetic corpus

make_synth_collection.pl --policy zipfian --alpha 1 -o synth \
  -d synth_out -c chemical -b produced --size 11 --verbose
```

10.2 Gathering Corpus Statistics

10.2.1 Run IDF queries on a corpus

```
input: indexed corpus
output: idf query data

idf_query.pl -c chemical -b produced -q health
idf_query.pl -c chemical -b produced --all
```

10.2.2 Run TF queries on a corpus

```
input: indexed corpus
output: tf query data

tf_query.pl -c chemical -b produced -q health
tf_query.pl -c chemical -b produced --all
tf_query.pl -c chemical -b produced --stemmed --all
tf_query.pl -c chemical -b produced -q "atomic number"
```

10.3 Generating Networks

10.3.1 Generate a network from a corpus

```
 input: indexed corpus
 output: network graph

 corpus_to_network.pl -c chemical -b produced -o chemical.graph
```

10.3.2 Generate synthetic network using Erdos/Renyi linking model

```
 output: synthetic graph

# With n nodes and m edges

 generate_random_network.pl -o synthetic.graph \
   -t erdos-renyi-gnm -n 100 -m 88

# With n nodes and random edge with probability p

 generate_random_network.pl -o synthetic.graph \
   -t erdos-renyi-gnp -n 100 -p .1

# Based on another graph

 generate_random_network.pl -o synthetic.graph \
   -i $CLAIRLIB/corpora/david_copperfield/adjnoun.graph \
   -t erdos-renyi-gnp -p .1
```

10.4 Gathering Network Statistics

10.4.1 Generate plots and statistics from a corpus

```
 input: indexed corpus
 output: plots and stats

 corpus_to_cos.pl -c chemical -o chemical.cos -b produced
 cos_to_cosplots.pl -i chemical.cos
 cos_to_histograms.pl -i chemical.cos
 cos_to_stats.pl -i chemical.cos -o chemical.stats
```

10.4.2 Generate plots from a network

```
 input: network file
 output: degree distribution plots

 network_to_plots.pl -i chemical.cos --bins 100
```

10.4.3 Putting it all together: plots and stats generated from a document

```
 input: sample news data
 output: plots and statistics
 optional: Matlab

 sentences_to_docs.pl -i \
   $CLAIRLIB/corpora/news-sample/lexrank-sample.txt \
   -o lexrank-sample
 directory_to_corpus.pl -c lexrank-sample  -b produced \
   -d lexrank-sample
 index_corpus.pl -c lexrank-sample -b produced
 corpus_to_cos.pl -c lexrank-sample -b produced \
    -o lexrank-sample.cos
 cos_to_histograms.pl -i lexrank-sample.cos
 cos_to_cosplots.pl -i lexrank-sample.cos
 cos_to_stats.pl --graphs -i lexrank-sample.cos \
   -o lexrank-sample.stats
 print_network_stats.pl --triangles -i lexrank-sample-0.26.graph
 stats2matlab.pl -i lexrank-sample.stats -o lexrank-sample.m
 network_growth.pl -c lexrank-sample -b produced
 stats2matlab.pl -i lexrank-sample.wordmodel.stats \
   -o lexrank-sample-wordmodel.m

# Now make the synthetic collection

 make_synth_collection.pl --policy zipfian --alpha 1 -o synth \
   -d synth_out -c lexrank-sample -b produced --size 11 --verbose
 link_synthetic_collection.pl -n synth -b produced -c synth \
   -d synth_out -l erdos -p 0.2
 index_corpus.pl -c synth -b produced
 corpus_to_cos.pl -c synth -b produced -o synth.cos
 cos_to_histograms.pl -i synth.cos
 cos_to_cosplots.pl -i synth.cos
 cos_to_stats.pl -i synth.cos -o synth.stats --graphs --all -v
 stats2matlab.pl -i synth.stats -o synth.m
 network_growth.pl -c synth -b produced
 stats2matlab.pl -i synth.wordmodel.stats -o synth-wordmodel.m

# If you are on a machine with MATLAB,
# run the following to generate plots:

 mkdir plots
 mv *.m matlab
 matlab -nojvm -nosplash < lexrank-sample-cosine-cumulative.m
 matlab -nojvm -nosplash < lexrank-sample-cosine-hist.m
 matlab -nojvm -nosplash < lexrank-sample-distplots.m
 matlab -nojvm -nosplash < lexrank-sample.m
 matlab -nojvm -nosplash < lexrank-sample-wordmodel.m
 matlab -nojvm -nosplash < synth-cosine-cumulative.m
 matlab -nojvm -nosplash < synth-cosine-hist.m
 matlab -nojvm -nosplash < synth-distplots.m
 matlab -nojvm -nosplash < synth.m
 matlab -nojvm -nosplash < synth-wordmodel.m
```

10.5 Other Useful Tools

10.5.1 Selecting a subset of a corpus for processing

```
input: existing corpus
output: directory containing subset of corpus

corpus_to_cluster.pl -c bulgaria-10 -b produced \
 -f '^https://www.cia.gov/' \
 -f '^http://en.wikipedia.org/' -o filtered
directory_to_corpus.pl -c bulgaria-filtered -b produced \
  -d filtered
```

10.5.2 Convert a network from one format to another

```
input: gml file (or pajek file)
output: edgelist file

convert_network.pl -v \
  -input $CLAIRLIB/corpora/david_copperfield/adjnoun.gml \
  --input-format gml --output ./adjnoun.graph \
  --output-format edgelist
print_network_stats.pl -i ./adjnoun.graph --undirected
```

10.5.3 Extract ngrams from document and create network

```
input: document
output: stats

extract_ngrams.pl -r "$CLAIRLIB/corpora/1984/1984.txt" \
  -f text -w 1984.2gram -N 2 -sort -v
print_network_stats -i 1984.2gram -v --all --sample 100 \
  --sample-type forestfire > 1984.2gram.stats
```

10.5.4 Generate statistics for word growth model from a corpus

```
input: indexed corpus
output: stats
required: Matlab

network_growth.pl -c chemical -b produced
stats2matlab.pl -i chemical.wordmodel.stats -o wordmodel.m
matlab -nojvm -nosplash < wordmodel.m
```

11 Corpus Representation in Clairlib

In section 8.1 we presented the various ways of generating a Clairlib corpus from a data set. In this section, we describe Clairlib representation of corpora; i.e we describe the directory structure of the *produced* directory generated by the command introduced in section 8.1.

11.1 Directory Structure of a Corpus in Clairlib

After running any of the commands that build a corpus from a data set, the following directory structure is built: (Assuming that the user chose the name *produced* for the corpus base directory and *MyCorpus* for the corpus name)

```
produced (1)
|-- corpra (2)
    |-- MyCorpus (3)
        |-- 000
               |-- 00
        |-- 001

|-- Corpus-data (4)
    |-- MyCorpus (5)
    |-- MyCorpus-tf (6)
    |-- MyCorpus-tf-s (7)

|-- download (8)
    |-- MyCorpus (9)

|--MyCorpus.download.uniq (10)
```

In the listing above, the numbers between parentheses are added to the list just to make it easier to reference the corresponding files and directories. The details of this structure is (using the numbers in the list as references):

(1) The base directory of the corpora. One or more corpora can reside in the same base directory.

(2) This directory contains a sub directory for each corpus. Each directory contains the original dataset documents merged and formatted in a standard way (described in the next point.) The name of each directory is the name of the corresponding corpus.

(3) This is the directory of the *MyCorpus* corpus. It contains the documents of the original dataset merged and formatted as follows:

- Each file is formatted the TREC standard format

```
        <DOC>
            <DOCNO> 000-00-000002 </DOCNO>
            <DOCHDR> {the file URL} </DOCHDR>
            {The file content}
        </DOC>
```

- Each 200 document are put together in one file. The files names are serial numbers from 00 to 39
- Each 40 file are put in one directory. The directories names are serial numbers starting from 000.

(4) This directory contains a sub directory for each corpus. Each directory contains the corpus metadata and indexing information (more details in the next point). The name of each directory is the name of the corresponding corpus.

(5) This is the data directory of the *MyCorpus* corpus. It contains the following files

- MyCorpus-docid-to-file: maps documents IDs to file names/
- MyCorpus-docid-to-url: maps documents IDs to their URLs.
- MyCorpus-url-to-docid: maps urls to documents IDs.
- MyCorpus-doclen: maps documents to their lengths.
- MyCorpus-idf: the inverse document frequency of the corpus terms.
- MyCorpus-idf-s: the stemmed Inverse Document Frequency of the corpus terms.
- MyCorpus.links: a list of all the hyperlinks within the corpus in (source, target) format.

- MyCorpus-tc: maps terms to their counts in the corpus.
- MyCorpus-tc-s: maps stemmed terms to their counts in the corpus.

(6) This directory contains the corpus term frequency. The terms are categorized in a 2 level alphabetical hierarchy; i.e terms are categorized on the first character, then the term in each category are categorized on the second character.

(7) The same as in (6) but with the terms stemmed.

(8) This directory contains a sub directory for each corpus. Each directory contains the original dataset copied as is. It preserves the same directory structure of the dataset.

(9) This directory contains the raw files of the *MyCorpus* corpus as they came in the original dataset.

(10) This file maps the corpus files in *./download* to their original location.

- Connection - Connects to the Bio database using SOAP
- EUtils - A container for variables useful inside of Clair::Bio::EUtils::*.
- GeneRIF - Module for parsing GeneRIF files
- AustralianParser - A class for parsing Australian Hansard html.
- Graf - An object representing a Hansard graf.
- Record - An object representing a Hansard record.
- Speaker - An object representing a Hansard speaker.

- AustralianParser - A class for parsing Australian Hansard html.
- Graf - An object representing a Hansard graf.
- Record - An object representing a Hansard record.
- Speaker - An object representing a Hansard speaker.

12 Modules

In this section, we talk briefly about the main components in Clairlib API. A detailed documentation of all the modules can be found online on http://belobog.si.umich.edu/clair/clairlib/pdoc/.

12.1 Clair::Document

Clairlib's Document class can be used to perform some basic analysis and perform some calculations on a single document.

Documents have three types of values: 'html', 'text', and 'stem'. A document must be created as one of the three types. It can then be converted from html to text and from text to stem. Performing a conversion does not cause the old information to be lost. For example, if a document starts as html, and is converted to text, the html is not forgotten, the document now holds an html version and a text version of the original html document.

Creating a new document: A new document can be created either from a string or from a file. To create a document from a string, the string parameter should be specified, while the file parameter should be specified with the filename to load the document from. It is an error if both are specified.

The initial type of a document must be specified. This is done by setting the type parameter to 'html', 'text', or 'stem'. Additionally, an id must be specified for the document. Care should be taken to keep ids of documents unique, as putting documents with the same id into a Cluster or Network can cause problems.

Finally, the language of the document may be specified by passing a value as the language parameter.

```
my $doc = new Clair::Document(file => 'doc.html', id => 'doc1',
                              type => 'html');
```

Using a single Document: strip_html and stem convert an html version of the document to text and a text version to stem, respectively.

The html, text, or stem version of the document can be retrieved using get_html, get_text, and get_stem respectively. For these methods and all those used by document, the programmer is expected to ensure that any time a particular type of a document is used, that type is valid. That is, a document that is created as an html document and is never converted to a text document should never have get_text called or save (described later) called with type specified as anything but 'html'.

```
# We start off with the html version
my $html = $doc->get_html;

# But can now get the text version
my $text = $doc->strip_html();
die if ($text ne $doc->get_text);

# And then the stemmed version
my $stem = $doc->stem();
die if $stem ne $doc->get_stem;

# Note that the html version is unchanged:
die if $html ne $doc->get_html;
```

Several different operations can be performed on a document. It can be split into lines, sentences, or words using split_into_lines, split_into_sentences, and split_into_words which return an array of the text of the document separated appropriately. split_into_lines and split_into_sentences can only be performed on the text version of the document, but split_into_words can be performed on any type of document. It defaults to text, but this can be overridden by specifying the type parameter.

A document can be saved to a file using the save method. The method requires the type to be saved be specified.

Documents may have parent documents as well. This can be used to track the original source of a document. For example, a new document could be created for each sentence of an original document. By using set_parent_document and get_parent_document, each new document can point to the document it was created from.

12.2 Clair::Cluster

Clairlib makes analyzing relationships beween documents very simple. Generally, for simplicity, documents should be loaded as a cluster, then converted to a network, but documents can be added directly to a network.

Creating a Cluster: Documents can be added individually or loaded collectively into a cluster. To add documents individually, the insert function is provided, taking the id and the document, in that order. It is not a requirement that the id of the document in the cluster match the id of the document, but it is recommended for simplicity.

Several functions are provided to load many documents quickly. load_file_list_array adds each file from the provided array as a document and adds it to the cluster. load_file_list_from_file does the same for a list of documents that are given in a provided file. load_documents does the same for each document that matches the expression passed along as a parameter.

Each of these functions must assign a type to each document created. 'text' is the default, but this may be changed by specifying a type parameter. Files can be loaded by filename or by 'count', an index that is incremented for each file. Using the filename is the default, but specifying a parameter count_id of 1 changes that. To allow the load functions to be called repeatedly, a start_count parameter may be specified to have the counts started at a higher number (to avoid repeated ids). Each load function returns the next safe count (note that if start_count is not specified, this is the number of documents loaded).

load_lines_from_file loads each line from a file as an individual document and adds it to the cluster. It behaves very similarly to the other load functions except that ids must be based on the count.

```
my $cluster = Clair::Cluster->new();

$cluster->load_documents("directory/*.txt", type => 'text');
```

12.2.1 Working with Documents Collectively

The functions strip_all_documents, stem_all_documents, and save_documents_to_directory act on every document in the cluster, stripping the html, stemming the text, or saving the documents.

```
$cluster->stem_all_documents();
```

12.2.2 Analyzing a Cluster

The documents in a cluster can be analyzed in two ways. The first is that an IDF database can be built from the documents in the cluster with build_idf. The second is analyzing the similarity between documents in the cluster. First, compute_cosine_matrix is provided which computes the similarity between every pair of documents in the cluster. These values are returned in a hash, but are also saved with the cluster. compute_binary_cosine returns a hash of cosine values that are above the threshold. It can be provided a cosine hash or can use a previously computed hash stored with the cluster. get_largest_cosine returns the largest cosine value, and the two keys that produced it in a hash. It also can be passed a cosine hash or can use a hash stored with the cluster.

```
my %cos_hash = $cluster->compute_cosine_matrix();

my %bin_hash = $cluster->compute_binary_cosine(0.2);
```

12.3 Clair::Network

12.3.1 Creating a Network

There are three ways to create a network from a cluster, based on what statistics are desired from the network. For statistics based on similarity, create_network creates a network based on a cosine hash. Any two documents with a positive cosine relationship will have an edge between them in the network. Optionally, all documents can have an edge by setting the include_zeros parameter as 1. The transition values to compute lexrank are also set, although the values can be saved to a different attribute name by specifying a property parameter.

For statistics based on hyperlink relationship, create_hyperlink_network_from_array and create_hyperlink_network_from_file creates a network with edges between pairs of documents in an array or on lines of a file, respectively.

create_sentence_based_network creates a network with a node for every sentence in every document. The cosine between each sentence is then computed and, if a threshold is specified, the binary cosine is computed. The edges are created based on the similarity values as with create_network.

```
my $network = $cluster->create_network(cosine_matrix => %bin_hash);
```

12.3.2 Importing a Network

Networks can also be read in from various cross-platform graph formats. Currently, the following formats are supported:

- Edgelist
- GraphML
- Pajek

To read in a network, create a Clair::Network::Reader object of the appropiate type and call the read_network method with a filename. A new Clair::Network object will be returned.

Example of reading a Pajek file:

```
use Clair::Network::Reader::Pajek;

my $reader = Clair::Network::Reader::Pajek->new();
my $net = $reader->read_network("example.net");
```

12.3.3 Exporting a Network

You can also export a Network to any of the above formats with the Writer classes.

Example of writing a Pajek format network:

```
use Clair::Network::Writer::Pajek;

my $export = Clair::Network::Writer::Pajek->new();
$export->set_name("networkname");
$export->write_nework($net, "example.net");
```

12.3.4 Analyzing a Network

Once a network has been created, much more analysis is possible. Basic statistics like the number of nodes and edges are available from num_nodes and num_links. The average and maximum diameters can be ascertained from diameter, specifying either a max parameter as 1 or an avg parameter as 1 (max is the default). The average in degree, out degree, and total degree can be computed with avg_in_degree, avg_out_degree, and avg_total_degree respectively.

Shortest Path Length

Clairlib can compute the shortest paths between all pairs of vertices. It returns the results as a hash of hashes of the shortest path matrix.

```
use Clair::Network;

my $net = new Clair::Network();
my $sp_matrix = $net->get_shortest_path_matrix();
```

Average Shortest Path Length

Clairlib can compute the average shortest path length between all pairs of vertices. See the code examples volume of the documentation for usage.

Clustering Coefficient

Clairlib supports two clustering coefficient functions: the Watts-Strogatz clustering coefficient and the Newman clustering coefficient.

Assortativity

Clairlib can compute degree assortativity. It returns a global measure of network assortativity, the degree assortativity coefficient.

```
my $sp_matrix = $net->degree_assortativity_coefficient();
```

Centrality

Clairlib supports several network centrality measures. These measures assign a value to each vertex depending on how "central" that vertex is.

The Centrality modules are in namespace Clair::Network::Centrality. Each module has two centrality member functions, which both return a hash of vertices and their corresponding centrality. The first function returns the base centrality measure. The second returns a centrality normalized to between 0 and 1.

Degree Centrality

Ranking each vertex by vertex degree is the simplest measure of network centrality. This is called degree centrality. For undirected networks, it is simply the degree of each vertex. For directed networks, it is the total degree divided by two.

Closeness Centrality

Closeness centrality measures how close each vertex is to the other vertices. This is found by measuring the length from the target vertex to every other reachable vertex along the shortest path. The reciprocal of this is the closeness centrality.

Betweenness Centrality

Betweenness centrality measures how many shortest paths the target vertex is in between. The betweenness index is the sum of the number of shortest paths between two actors through the target actor, divided by the total number of shortest paths between the two actors.

LexRank Centrality

To compute lexrank from a network built from a cluster using create_network or create_sentence_based_network, compute_lexrank is provided. Initial values or bias values can also be loaded from a file using read_lexrank_initial_distribution and read_lexrank_bias (the default for both is uniform). If the network was not created from a cluster appropriately (or to change the values), transition values can also be loaded from a file using read_lexrank_probabilities_from_file.

```
my %lex_hash = $network->compute_lexrank();
```

PageRank Centrality

Similarly, pagerank can be computed with compute_pagerank. Transition values are already set for a network created with one of the create_hyperlink_network functions, but can be read from a file using read_pagerank_probabilities_from_file otherwise. Initial distribution and personalization values can be read from files using read_pagerank_initial_distribution and read_pagerank_personalization.

The results of these computations are returned by compute_lexrank and compute_pagerank, but can also be saved to a file using save_current_lexrank_distribution or printed to standard out using print_current_lexrank_distribution (for pagerank, save_current_pagerank_distribution and print_current_pagerank_distribution, respectively).

```
$network->print_current_lexrank_distribution();
$network->save_current_lexrank_distribution('lex_out');
```

Community finding using Girvan/Newman algorithm

The Girvan/Newman algorithm will perform hierarchical clustering on network data

To do the clustering, first load a network object derived from Clair Network.

```
use Clair::Network::GirvanNewman;

$GN = Clair::Network::GirvanNewman->new($network);
```

then call:

```
$graphPartition = $GN->generatePartition();
```

The result will be stored in a hash, where the key is the node id and the value is the cluster it belongs to To use the value, call:

```
$str = $$graphPartition->{$nodeID};
```

the cluster is in the format of 0—1—2—... the number between "—" is the cluster it belongs to at different level of the hierarchy. For example, to return the partition $node1 belongs to when partition the network into two, call:

```
@p = split(/\|/, $str);
return $p[1];
```

Finding maximum flow in a flow network using the Ford/Fulkerson algorithm

The Ford/Fulkerson algorithm computes the maximum flow in a flow network.

The idea behind the algorithm is very simple: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on.

To find the maximum flow in a network, first load a network object derived from Clair Network with the capacity of each edge specified as edge attribute.

```
use Clair::Network::FordFulkerson;

$FF = Clair::Network::FordFulkerson->new($network,$source,$destination);
```

then call:

```
($flownet,$max) = $FF->run();
```

This function returns the residual flow network in $flownet and the value of the maxumum flow in $max.

You can change the source and destination node using:

```
$FF->set_source{$s};
$FF->set_destination($t);
```

Community finding using KernighanLin algorithm

The KernighanLin algorithm will divide a undirected weighted graph into two partitions such that the sum of the weight of edges between the two is the minimum (min cut).

To do the partition, first load a network object derived from Clair::Network.

```
use Clair::Network::KernighanLin;

$KL = Clair::Network::KernighanLin->new($network);
```

then call:

```
$graphPartition = $KL->generatePartition();
```

The result will be stored in a hash, where the key is the node id and the value is the partition (0/1) it belongs to To use the value, call:

```
$$graphPartition->{$nodeID};
```

Adamic/Adar Value

The Adamic/Adar value computes the similarity of any two nodes in a network.To compute the Adamic/Adar value, you need to preprocess the corpus to get the attribute you want. The format should be:

```
attribute1
attribute2
...
```

put all the files into one empty folder, and call:

```
use Clair::Network::AdamicAdar;

$aa = Clair::Network::AdamicAdar->new();
$results = $aa->readCorpus($folderName);
```

The result will be stored in a two-dimensional hash. To use the value, call:

```
$results->{$node1}->{$node2};
```

Note: In order to save memory, the results only saved for every one pair of value in ascending order.

Many other network based statistics can be computed. For examples, please see test_network_stat.pl in the test directory.

12.3.5 Network Generation

Random networks can also be generated with the Clair::Network::Generator package. Currently, this includes generation of Erdős-Rényi random graphs.

Clair::Network::Generator::ErdosRenyi

Two models of Erdős-Rényi random networks can be generated. One includes a set number of nodes and edges. The other type includes a set number of nodes with an edge existing between two nodes with a probability p.

Example:

```
use Clair::Network::Generator::ErdosRenyi;
my $generator = Clair::Network::Generator::ErdosRenyi->new();
my $set_edges = $generator->generate(10, 20, type => "gnm");
my $random_number_edges = $generator->generate(10, 0.2, type => "gnp");
```

12.3.6 Network Sampling

Sometimes a network may be too large to process in its entirety. Sampling can be used to extract a representive subset of the network for analysis. Different methods preserve different network properties. Clairlib provides several network sampling algorithms.

- Clair::Network::Sample::RandomNode

  Random node sampling simply chooses a number of nodes from the original graph, choosing nodes uniformly at random. If there is an edge between two nodes that have been selected in the original network, that edge will be included in the sampled network.

- Clair::Network::Sample::RandomEdge

  Random edge sampling chooses edges randomly from the original network, and includes the two incident nodes.

- Clair::Network::Sample::ForestFire

  ForestFire sampling chooses an initial random node, and performs a probabilistic recursive breadth-first search from that initial node. If the ”fire” dies out, it will restart at another random node.

Example:

```
use Clair::Network::Sample::ForestFire;

my $fire = new Clair::Network::Sample::ForestFire($net);
print "Sampling 3 nodes using the Forest Fire model\n";
$new_net = $fire->sample(3, 0.9);
```

12.4 Clair::Statistics

Clairlib provides several statistical tools for analyzing and generating distributions. The distributions include Geometric, Gaussian, LogNormal, Zipfian and student’s T-distribution. There is also experimental support for statistical inference. These distribution and test modules are included under the Clair::Statistics namespace. See the test_statistics.pl recipe for more information. The older Clair::Gen will be folded into this in the next release.

12.5 Clair::Gen

Clair::Gen is for use when working with distributions. It can produce expected Power Law and Poisson distributions, or analyze observed distributions. The read_from_file method reads an observed distribution from a file.

The plEstimate function accepts a distribution as input and produces the best-fitting $\hat{c}$ and $\hat{\alpha}$ values. genPl does the opposite–using $\hat{c}$ and $\hat{\alpha}$ as input, it produces the expected distribution.

For Poisson distributions, poisEstimate and genPois are provided which mirror the functionality of plEstimate and genPl. plEstimate is currently just a stub function, however.

To compare estimated and actual distributions, compareChiSquare is included in the package. This returns the number of degrees of freedom and the p-value.

```
my $g = new Clair::Gen;

$g->read_from_file("trial1.dist");
my @observed = $g->distribution;

my ($c_hat, $alpha_hat) = g->plEstimate(\@observed);
my @expected = g->genPL($c_hat, $alpha_hat);

my ($df, $pv) = $g->compareChiSquare(\@observed, \@expected, 2);
```

12.6 Clair::ChisqIndependent

Clair::ChisqIndependent is for Gen to produce the Chi square for given data. It's subclassed from Statistics::ChisqIndep, and adds one more method: recompute_chisq to compute the p-value of the data when the number of degrees of freedom changes

The recompute_chisq function accepts the modified degree of freedom as input and produces the corresponding p-value.

```
my $chi = new Clair::ChisqIndependent;
my @chi_array = (\@observed, \@expected);

$chi->load_data(\@chi_array);
$chi->{df} -= $extra_df;
$chi->recompute_chisq();

return ($chi->{df}, $chi->{p_value});
```

12.7 Clair::LM

Clair::LM provides functionality for the extraction of N-grams from text and HTML documents. The resulting N-gram dictionary can optionally be pruned of low-frequency N-grams before being written to a human-readable text file and/or serialized to a network-ordered Storable file.

```
use Clair::Cluster
use Clair::Utils::Ngram qw(load_ngramdict
                dump_ngramdict
                extract_ngrams
                write_ngram_counts
                enforce_count_thresholds);

# Read in documents
my $r_cluster = Clair::Cluster->new;
$r_cluster->load_documents("*.html",
                type => 'html',
                filename_id => 1);

# Strip markup and stem resulting document contents, segment into sentences,
#   and extract bigrams, storing the bigram dictionary in $r_ngramdict
my $r_ngramdict = {};
extract_ngrams(cluster   => $r_cluster,
                N         => 2,
                ngramdict => $r_ngramdict,
                format    => 'html',
                stem      => 1,
                segment   => 1,
                verbose   => 1 );

# Remove all bigrams having fewer than 2 occurrences and/or not occurring
#   in the top 100 most frequent
enforce_count_thresholds(N => 2,
                ngramdict => $r_ngramdict,
                mincount  => 2,
                topcount  => 100 );

# Sort bigrams in descending order by count and write bigram dictionary to file
write_ngram_counts(N         => 2,
                ngramdict => $r_ngramdict,
                outfile   => 'test.2grams',
                sort      => 1 );

# Serialize bigram dictionary to (network-ordered) Storable file
dump_ngramdict(N         => 2,
                ngramdict => $r_ngramdict,
                outfile   => 'test.2grams.dump' );

# Restore N-gram dictionary from Storable file
my $N;
($N, $r_ngramdict2) = load_ngramdict(infile => 'test.2grams.dump');
```

12.8 Clair::Util

Clair::Util provides several different methods that are useful but do not fit in other modules. For example, build_IDF_database reads a list of files and writes the IDF values from those files to a database (Berkeley DB). build_idf_by_line can also be used to build an IDF database, in this case, using text pass to it and treating each line as a different document and computing the IDF from those. read_idf opens a database and returns the hash from it. This is particularly useful for examining the contents of an IDF database, but can be easily used for many other tasks as well.

```
Clair::Util::build_idf_by_line("This is a test.\n" .
             "This is considered another document.\n" .
             "A third sample document.",
             "test_dbm_file");

my %idf = Clair::Util::read_idf("test_dbm_file");

print "The idf of ’this’ is: ", $idf{this}, "\n";
```

12.9 Clair::Utils::CorpusDownload

12.9.1 Creating a Corpus

The CorpusDownload module is provided to create a corpus. Create a CorpusDownload object using new(). A corpus name must be provided, and a rootdir is optional, but strongly recommended since the default is ‘/data0/projects/tfidf’. The rootdir must be an absolute path, rather than a relative path. The root directory is where the corpus files will be placed. Many corpora can be made with the same root directory, as long as the corpusname is different for each.

Two functions are provided to create a corpus. buildCorpus is used to download files to form the corpus, while buildCorpusFromFiles is used to form a corpus with files already on the computer. Both require a reference to an array with either the urls or absolute paths to the files for buildCorpus and buildCorpusFromFiles, respectively. These files will then be copied to the root directory provided and a corpus created from them in TREC format.

Because CorpusDownload was designed to use a downloaded corpus, results from a corpus build with buildCorpusFromFiles will list files with “http://” at the beginning, then the full path of the file.

To use a base URL and find files based on links from that file, the function poach is provided as an interface to ‘poacher.’ This returns an array with URLs that can be passed to buildCorpus.

```
$corpus = Clair::Utils::CorpusDownload->new(corpusname => ’new_corpus’,
              rootdir => ’/usr/username/’);

$corpus->buildCorpus(urlsref => $@urls);
```

12.9.2 Computing IDF and TF Values

To compute the IDF and TF values for the corpus, buildIdf and buildTf are provided. Both accept stemmed as a parameter which can be set to 1 to compute the stemmed values or 0 (the default) to compute the unstemmed values. Before buildTf can be called, build_docno_dbm must be called.

```
$corpus->buildIdf(stemmed => 0);
$corpus->buildIdf(stemmed => 1);

$corpus->build_docno_dbm();

$corpus->buildTf(stemmed => 0);
$corpus->buildTf(stemmed => 1);
```

12.10 Clair::Utils::TF and Clair::Utils::IDF

Once IDF values have been computed, they can be accessed by creating an Idf object. In the constructor, rootdir and corpusname parameters should be supplied that match the CorpusDownload parameters, along with a stemmed parameter depending on whether stemmed or unstemmed values are desired (1 and 0 respectively). To get the IDF for a word, then, use the method getIdfForWord, supplying the desired word.

A Tf object is created with the same parameters passed to the constructor. The function getFreq returns the number of times a word appears in the corpus, getNumDocsWithWord returns the number of documents it appears in, and getDocs returns the array of documents it appears in.

```
my $idf = Clair::Utils::Idf->new( rootdir=> ’/usr/username/’,
          corpusname =>’new\_corpus’, stemmed => 0);

print "The idf of ’and’ is ", $idf->getIdfForWord("and"), "\n";

my $tf = Clair::Utils::Idf->new( rootdir=> ’/usr/username/’,
         corpusname =>’new_corpus’, stemmed => 0);

print $tf->getNumDocsWithWord("and"), " docs have ’and’ in them\n";
print "’and’ appears ", $tf->getFreq("and"), "times.\n";

print "The documents are:\n" my @docs = $tf->getDocs("and");
foreach my $doc (@docs) {
  print "$doc\n";
}
```

12.11 Clair::Utils::WebSearch

This applies only to users of Clairlib-ext!

The WebSearch module is used to perform Google searches. A key must be obtained from Google in order to do this. Follow the instructions in the section ”Installing the Clair Library” to obtain a key and have the WebSearch module use it.

Once the key has been obtained and the appropriate variables are set, use the googleGet method to obtain a list of results to a Google query. The following code gets the top 20 results to a search for the ”University of Michigan,” and then prints the results to the screen.

```
my @results = @{Clair::Utils::WebSearch::googleGet("University of \
Michigan", 20)};

foreach my $r (@results) {
  print "$r\n\n";
}
```

The WebSearch module also provides the ability to download a single page as a URI::URL-escaped file using the downloadUrl method. This method needs two parameters: the URL to download and the filename where the downloaded page should be saved.

```
Clair::Utils::WebSearch::downloadUrl("http://www.mgoblue.com/", \
"mgoblue_home.htm");
```

12.12 Clair::Utils::Parse

This applies only to users of Clairlib-ext!

The Parse module provides a wrapper for the Charniak parser and the chunklink tool.

12.12.1 Preparing a File for the Charniak Parser

To be parsed by the Charniak parser, a file must be formatted in a specific way, with sentences on separate lines, placed inside <s>< /s> tags. For example:

```
<s>This is one sentence.</s>
<s>This is another sentence.</s>
```

To make this formatting easier, the the prepare_for_parse function is provided. This function will read a file, split it into sentences (using Clair::Document::split_into_sentences), then put each sentence on its own line, surrounded by <s>< /s> tags, in a new file.

```
Clair::Utils::Parse::prepare_for_parse("input.txt", "output.txt");
```

If a file is already correctly formatted, this step should not be performed.

12.12.2 Charniak Parser

The parse function runs a file through the Charniak parser. The result of parsing will be returned from the function as a string, and may optionally be written to a file by specifying an output file.

Note that a file must be correctly formatted to be parsed. See the previous section, “Preparing a File for the Charniak Parser” for more information.

```
my $parse_output = Clair::Utils::Parse::parse("to_be_parsed.txt",
                          output_file => "output.txt");
```

12.12.3 Chunklink

Chunklink is a very useful tool to analyze file from the Penn Treebank. The Parse module also provides a wrapper to it, with the function Parse::chunklink. This function takes an input file and returns the result as a string, and may optionally also write the results to a file.

```
my $chunkout = Clair::Utils::Parse::chunklink("WSJ_0021.MRG",
                      output_file => "output.txt");
```

12.13 Clair::Utils::Stem

This is an implementation of a stemmer, to take one word at a time and return the stem of it. There are only two functions: new and stem. Creating an object with new initializes the stemmer. Subsequent calls to stem will return the stemmed version of a word. Note that this is not the same stemmer that is used by Document::stem.

```
my $stemmer = new Clair::Utils::Stem;

print "'testing' stemmed is: ", $stemmer->stem("testing"), "\n";
```

12.14 Clair::Bio::Connection

This module connects to the Bio database using SOAP. There are a number of functions. The first is get_ids(). It returns a list of the ids of every paper in the database. The second is get_sentences($id), which Returns a list of hash references containing information about each sentence in the document with PMID $id. The third is get_title($id). It returns the title of the document with PMID $id. The fourth is get_body($id), which returns all of the sentences of the document with PMID $id concatenated together. The function get_citing_sentences($citer, $cited) returns a list of sentences from the document with PMID $citer that cite the document with PMID $cited. This list will have the same structure as in get_sentences. Next, count_citations() returns the total number of citations in the database. If you want information from the abstract of an article, you can use get_abstract_sentences($id) and get_abstract($id), which perform similar processes as the article functions. Then, you can build a network around these articles using some of the other functions present. get_degree_in($id) returns the total number of papers that cite the document with PMID $id. get_degree_out($id) returns the total number of papers the document with PMID $id cites. get_citation_network(@ids) returns a Clair::Network object of ids with an edge between id1 and id2 if id1 cites id2. Generates the network starting from the ids in @ids. To return a Clair::Network object containing the full citation network, use get_full_citation_network(). The last function is dbquery($statement), which executes the given statement on the database and returns the results. The result is an array of array references. See the synopsis for an example. The code to open a connection and perform some sample calculations would look like this:

```
my $c = Clair::Bio::Connection->new();
my $graph = $c->get_citation_network($id);

foreach my $from (sort keys %$graph) {
    foreach my $to (sort keys %{ $graph->{$from} }) {
        print "$from => $to\n";
    }
}
```

12.15 Clair::Bio::EUtils

Clair::Bio::EUtils is a base class for Clair::Bio::EUtils objects. It is a container for variables useful inside of Clair::Bio::EUtils::*. It has two functions, hash2args and build_url.

```
my %args = ( this => "that thing" );
my $base = "http://foo.bar/thing.cgi";
my $url = build_url( base => $base, args => %ags );
print "$url\n"; $ prints http://foo.bar/thing.cgi?this=that%20thing
```

```
my %hash = ( this => "that thing", foo => "bar" );
my $str = hash2args(%hash);
print "$str\n"; # prints this=that%20thing&foo=bar
```

12.16 Clair::Bio::GeneRIF

This module is used to parse GeneRIFs files. A GeneRIF file has the following format: the first line is meta-data that labels each tab-delimited field, and the rest of the lines are those fields. The parsed file is saved into a DBM file after being parsed. The module will look for a DBM file before attempting to parse the text file, unless the 'reload' parameter is passed with a true value to the constructor. Access to the data is done by passing a gene_id value to the get_records_from_id method, which returns a list of all the GeneRIFs with the given gene_id. Additionaly, you may access all GeneRIF entries at once using the get_all_records method. This will most likely be a large hash, backed by a DBM, so it would be best to use the each() function to iterate over its keys and values. This module uses DB_File to store nested data structures. There are a number of methods in this module. The first, new, constructs a new GeneRIF object. The 'path' parameter must point to a GeneRIF text file. The 'reload' parameter is optional and will force the DBM to be recreated (this defaults to false). get_records_from_id returns a list of records with the given gene_id. Each record is an array where the values are in the same order as the fields at the top of the file. Returns () if there are no records. To get the total number of records, use get_total_records. To get all of the records from the GeneRIF file as a hashref mapping gene_ids to lists of records, use get_all_records. The function get_fields returns a list of the field names. These are the keys in each record. An example of the usage of this module would look like this:

```
my @records = $g->get_records_from_id($no);
my @fields = $g->get_fields();
foreach my $i (0 .. $#records) {
    print "Record $i {\n";
    my @rifs = @{$records[$i]};
    foreach my $j (0 .. $#rifs) {
        print "\t$fields[$j] => $rifs[$j]\n";
    }
    print "}\n";
}
```

12.17 Clair::Polisci::AustralianParser

The Clair::Polisci::AustralianParser module is used for parsing Australian hansard html. The basic setup would look like this:

```
use Clair::Polisci::AustralianParser;
my $p = Clair::Polisci::AustralianParser->new(file => "myfile.html");
my $header = $p->get_header();
my $speeches = $p->get_speeches();
$p->write_xml();
```

It contains a few different functions. The function new() creates a new object from the given file. ”out” is an optional reference to a filehandle where the XML will be printed. If “out” is not specified, `$p->write_xml()` will print to STDOUT. For example:

```
my $out = \*OUT;
$p->set_out($out);
my $file = "somefile.html";
$p = Clair::Polisci::AustralianParser->new(file => $file, out => $out);
```

The get_header function returns a hashref containing header key/value pairs.

```
my $header = $p->get_header();
foreach my $key (keys(%$header)) {
    print "$key => $header->{$key}\n";
}
```

The get_speeches function returns an arrayref containing hashrefs to speech info.

```
my $speeches = $p->get_speeches();
foreach my $speech (@$speeches) {
    print "[\n";
    print "\t$speech->{type}\n";
    print "\t$speech->{speaker}\n";
    print "\t$speech->{body}\n";
    print "]\n";
}
```

Finally, the write_xml function converts the data from $header and $speeches into XML and prints it to "out".

12.18 Clair::Polisci::Graf

The Clair::Polisci::Graf module is an object representing a hansard graf. The basic usage looks like this:

```
my $speaker = Clair::Polisci::Speaker->new( ... );
my $graf = Clair::Polisci::Graf->new(
    source => "polisci_us",
    index => 2,
    content => "Four score and seven million years ago...",
    speaker => $speaker
);
```

This is a Graf object used to represent a generic graf from a hansard. A graf is the smallest unit of speech in a hansard. An ordered list of grafs makes up a record. Each graf must have a source, an index, some content, and a speaker. The new() function constructs a new graf from the given parameters. As mentioned, source, index, content, and speaker are all required. Additional information can be associated with this graf by passing it to the constructor as a parameter.

The function to_document() returns the graf as a Clair::Document object. The body of the document is from the graf's content. The implementation would look like this:

```
use Clair::Document;
my $doc = $graf->to_document();
```

12.19 Clair::Polisci::Record

The Clair::Polisci::Record module is a Record object used to represent a generic handard Record. A record is an ordered collection of grafs. This module contains methods to convert a record to cluster of grafs or a document and allows for filtering/projections of grafs based on their properties. A sample script would look like this:

```
use Clair::Cluster;
use Clair::Document;

my $record = Clair::Polisci::Record->new( source => "some_db" );
my $graf = Clair::Polisci::Graf->new( ... );
my $speaker = Clair::Polisci::Speaker->new( ... );
$record->add_graf($graf);
...
my %filter = ( is_speech => 1, speaker => $speaker );
my @grafs = $record->get_grafs(%filter);
my $cluster = $record->to_cluster(%filter);
my $doc = $record->to_document(%filter);
print $doc->to_string();
```

To instantiate a Record, you would use the new() function. This creates a new record from the given source. Additional information can be associated with this graf by passing it to the constructor as a parameter. For example:

```
my $record = Clair::Polisci::Record->new(
    source => "polisci_us",
;
```

There is also a function to add a graf to the record. Its index in the record is determined by `$graf->{index}` and is not guaranteed to be unique within this record.

```
my $graf = Clair::Polisci::Graf->new( ... );
$record->add_graf($graf);
```

The size() function returns the total number of grafs in this record. There is also a get_grafs() function that returns the list of grafs, ordered by their indices, from this record that satisfy the given filter.

```
my %filter = (
    speaker => $speaker,
    is_speech => 1
);
my @grafs = $record->get_grafs(%filter);
```

In the above example, only grafs in $record which satisfy

```
$graf->{is_speech} == 1
```

and

```
$graf->{speaker}->equals($speaker)
```

will be returned in the list.

You can also return the contents of all grafs satisfying %filter concatenated together. See the description of get_graf(%filter) for more information. This is accomplished with the to_string() function. A similar function will do the same thing, only returning a Clair::Document as opposed to a string. This is the to_document() function. Again, See the description of get_graf(%filter) for more information.

The final function is to_graf_cluster(). This returns a cluster whose documents are the content of the grafs of this record that satify %filter. See the description of get_graf(%filter) for more information.

12.20 Clair::Polisci::Speaker

Clair::Polisci::Speaker is a Speaker object used to represent a generic speaker from a hansard. It is basically a container object with a source, an id and an equality relation. Two Speakers are equal if they come from the same source and have the same id. The basic usage would be:

```
my $speaker = Clair::Polisci::Speaker->new(
    source => "polisci_us",
    id => 49238
);
```

The new() function constructs a new speaker from the given source and id. Additional properties can be given to the speaker by adding them to the constructor's parameter list.

```
my $speaker = Clair::Polisci::Speaker->new(
    source => "some_db",
    id => 49032
);
```

13 Mega Code Example

Several code examples are provided with Clairlib, in the 'test' directory and also in the next section of the tutorial. This section takes a thorough look at one of these, 'test_mega.pl.' This script combines many pieces of functionality in Clairlib, so it serves as a good example.

We now walk through this example section by section:

```
# script: test_mega.pl
# functionality: Downloads documents using CorpusDownload, then makes IDFs,
# functionality: TFs, builds a cluster from them, a network based on a
# functionality: binary cosine, and tests the network for a couple of
# functionality: properties

use strict;
use warnings;
use FindBin;
use Clair::Utils::CorpusDownload;
use Clair::Utils::Idf;
use Clair::Utils::Tf;
use Clair::Document;
use Clair::Cluster;
use Clair::Network;
```

We start by declaring the packages we will use. We use FindBin to make the example system independent, because we know the relative location of the library to the script, rather than the more typical situation of knowing the absolute path of the library. Typically, scripts are more likely to change relative paths to the library than the library is to move, so simply hard-coding the path here may be best in most situations.

Next, we determine the "base directory" (where the script is located) and remember the directory where we will put all produced files. We then create a CorpusDownload object, giving it a corpus name of "testhtml" and specifying the produced files directory as the root directory for the corpus. Note that we are specifying an absolute path, not a relative pass for the rootdir parameter (otherwise, some CorpusDownload functions may not work correctly).

```
my $basedir = $FindBin::Bin;
my $gen_dir = "$basedir/produced/mega";

my $corpusref = Clair::Utils::CorpusDownload->new(corpusname => "testhtml",
                rootdir => $gen_dir);
```

We use CorpusDownload::poach to start with a single URL and follow links on that page, then links on those pages, etc. and return those URLs in an array reference. We iterate through those URLs and print them out to the screen. Finally, we pass those URLs to CorpusRef::buildCorpus to download the URLs and create a corpus in TREC format.

```
# Get the list of urls that we want to download
my $uref =                                                        \
$corpusref->poach("http://tangra.si.umich.edu/clair/testhtml/index.ht \
ml", error_file => "$gen_dir/errors.txt");

my @urls = @$uref;

foreach my $v (@urls) {
    print "URL: $v\n";
}

# Build the corpus using the list of urls
# This will index and convert to TREC format
$corpusref->buildCorpus(urlsref => $uref);
```

Our next step is to build the IDF and TF files. This computes the IDF and TF values for every word, then stores them in files from which those values can be easily retrieved. We build the unstemmed IDF, then the stemmed IDF first. Next, we must build the DOCNO/URL database before we build the TF files. Again, we build the unstemmed, and then the stemmed files (this order is not important for either calculation).

```
# ----------------------------------------------------------------
#  This is how to build the IDF.  First we build the unstemmed IDF,
#  then the stemmed one.
# ----------------------------------------------------------------
$corpusref->buildIdf(stemmed => 0);
$corpusref->buildIdf(stemmed => 1);

# ----------------------------------------------------------------
#  This is how to build the TF.  First we build the DOCNO/URL
#  database, which is necessary to build the TFs.  Then we build
#  unstemmed and stemmed TFs.
# ----------------------------------------------------------------
$corpusref->build_docno_dbm();
$corpusref->buildTf(stemmed => 0);
$corpusref->buildTf(stemmed => 1);
```

Now that we have build these values, we want to be able to see what the values are for specific words. We create an Idf object, giving it the same rootdir and corpusname as our CorpusDownload object. We choose whether we want the IDFs for the stemmed or unstemmed versions, choosing unstemmed in this example. We then get and print the IDF values for several words: ‘have,’ ‘and’, and ‘zimbabwe.’ Note that these words should be in lowercase.

```
# ----------------------------------------------------------------
#  Here is how to use a IDF.  The constructor (new) opens the
#  unstemmed IDF.  Then we ask for IDFs for the words "have"
#  "and" and "zimbabwe."
# ----------------------------------------------------------------
my $idfref = Clair::Utils::Idf->new( rootdir => $gen_dir,
                       corpusname => "testhtml" ,
                       stemmed => 0 );

my $result = $idfref->getIdfForWord("have");
print "IDF(have) = $result\n";
$result = $idfref->getIdfForWord("and");
print "IDF(and) = $result\n";
$result = $idfref->getIdfForWord("zimbabwe");
print "IDF(zimbabwe) = $result\n";
```

We now compute the TF values similarly. We create a Tf object, again using the same rootdir and corpusname as we did for CorpusDownload, and again choosing whether we want the stemmed or unstemmed information. Now that we have our Tf object, we can call getNumDocsWithWord to get the number of unique documents that have a word, getFreq to get the number of times a word is in the corpus, and getDocs to get all the URLs of all the documents that have that word in them. We do this with ‘washington’, ‘and,’ and ‘zimbabwe.’

```
# --------------------------------------------------------------------
#  Here is how to use a TF.  The constructor (new) opens the
#  unstemmed TF.  Then we ask for information about the
#  word "have":
#
#  1 first, we get the number of documents in the corpus with
#    the word "Washington"
#  2 then, we get the total number of occurrences of the word         \
"Washington"
#  3 then, we print a list of URLs of the documents that have the
#    word "Washington"
# --------------------------------------------------------------------
my $tfref = Clair::Utils::Tf->new( rootdir => $gen_dir,
                     corpusname => "testhtml" ,
                     stemmed => 0 );

$result = $tfref->getNumDocsWithWord("washington");
my $freq   = $tfref->getFreq("washington");
@urls = $tfref->getDocs("washington");
print "TF(washington) = $freq total in $result docs\n";
print "Documents with \"washington\"\n";
foreach my $url (@urls)  {  print "  $url\n";  }
print "\n";

# --------------------------------------------------------------------
#  Then we do 1-2 with the word "and"
# --------------------------------------------------------------------
$result = $tfref->getNumDocsWithWord("and");
$freq   = $tfref->getFreq("and");
@urls = $tfref->getDocs("and");
print "TF(and) = $freq total in $result docs\n";

# --------------------------------------------------------------------
#  Then we do 1-3 with the word "zimbabwe"
# --------------------------------------------------------------------
$result = $tfref->getNumDocsWithWord("zimbabwe");
$freq   = $tfref->getFreq("zimbabwe");
@urls = $tfref->getDocs("zimbabwe");
print "TF(zimbabwe) = $freq total in $result docs\n";
print "Documents with \"zimbabwe\"\n";
foreach my $url (@urls)  {  print "  $url\n";  }
print "\n";
```

We now change direction, using the fact that CorpusDownload has downloaded all of the html files to a specific directory. The directory location depends upon the root directory, the corpusname and the url of each downloaded file. In this case, all the downloaded files are from the same host and same path in the URL, so they are all in the same folder.

We create a new Clair::Cluster and use load_documents to get all the files from that directory. We give a type of ‘html’ so that every Clair::Document that is created has type ‘html.’ Once we have loaded the documents, we display a message saying how many we have, then strip and stem all the documents.

```
# Create a cluster with the documents
my $c = new Clair::Cluster;

$c->load_documents("$gen_dir/download/testhtml/tangra.si.umich.edu/cl \
air/testhtml/*", type => 'html');

print "Loaded ", $c->count_elements, " documents.\n";

$c->strip_all_documents;
$c->stem_all_documents;

print "I'm done stripping and stemming\n";
```

In order to shorten the computation for the rest of the example, we only want to look at 40 of the documents. To do this, we simply use a foreach loop that inserts the first 40 documents into a new cluster. Which 40 documents are inserted will vary from system to system (and possibly from run to run) since they are not specified or explicitly ordered in any way.

```
my $count = 0;
my $c2 = new Clair::Cluster;
foreach my $doc (values %{ $c->documents} ) {
    $count++;

    if ($count <= 40) {
        $c2->insert($doc->get_id, $doc);
    }
}
```

We now compute the cosine matrix for the new cluster. This will return a hash. By indexing into the hash using a pair of documents, we can get the cosine similarity of those two documents. We next compute the binary cosine using a threshold of 0.15. We could specify the cosine matrix, but not specifying it will result in the use of the cosine matrix from the last compute_cosine_matrix. This returns a hash with the same format as that returned by compute_cosine_matrix.

Next, we create a network based on the binary cosine. Every document with at least one edge (explained next) will become a vertex in the network, and every pair of documents with a non-zero cosine matrix will have an edge between their corresponding vertices.

Using this network, we compute a few statistics, getting the number of documents in the network (remember, this will probably be less than the 40 we started with because it is the number of documents with at least one edge). We also print out the average and maximum diameter of the network we created.

```
my %cm = $c2->compute_cosine_matrix();
my %bin_cos = $c2->compute_binary_cosine(0.15);
my $network = $c2->create_network(cosine_matrix => \%bin_cos);

print "Number of documents in network: ", $network->num_documents,    \
"\n";

print "Average diameter: ", $network->diameter(avg => 1), "\n";
print "Maximum diameter: ", $network->diameter(), "\n";
```