CLAIRLIB Documentation v1.04 Includes Bioinformatics & Political Science

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.

5.15 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.

5.15.1 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.

5.15.2 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 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.

5.15.3 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
```

5.15.4 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.

5.15.5 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
```

6 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

6.1 Generating Corpora

6.1.1 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
```

6.1.2 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
```

6.1.3 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
```

6.1.4 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
```

6.1.5 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
```

6.2 Gathering Corpus Statistics

6.2.1 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
```

6.2.2 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"
```

6.3 Generating Networks

6.3.1 Generate a network from a corpus

```
input: indexed corpus
output: network graph

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

6.3.2 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
```

6.4 Gathering Network Statistics

6.4.1 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
```

6.4.2 Generate plots from a network

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

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

6.4.3 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
```

6.5 Other Useful Tools

6.5.1 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
```

6.5.2 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
```

6.5.3 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
```

6.5.4 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
```

This version of Clairlib also includes several modules designed for use in Bioinformatics and for the analysis of Political Science text. These include:

- 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.

7 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/.

7.1 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.

7.2 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');
```

7.2.1 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();
```

7.2.2 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);
```

7.3 Clair::Network

7.3.1 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);
```

7.3.2 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");
```

7.3.3 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");
```

7.3.4 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.

7.3.5 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");
```

7.3.6 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);
```

7.4 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.

7.5 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);
```

7.6 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});
```

7.7 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’);
```

7.8 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";
```

7.9 Clair::Utils::CorpusDownload

7.9.1 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 build-CorpusFromFiles 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);
```

7.9.2 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);
```

7.10 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";
}
```

7.11 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");
```

7.12 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.

7.12.1 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.

7.12.2 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");
```

7.12.3 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");
```

7.13 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";
```

7.14 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";
    }
}
```

7.15 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
```

7.16 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";
}
```

7.17 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".

7.18 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();
```

7.19 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 satisfy %filter. See the description of get_graf(%filter) for more information.

7.20 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
);
```

8 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";
```