Graph Alignment, Identification and Analysis (GAIA) User Manual

LINQ's Group

University of Maryland, College Park
http://linqs.cs.umd.edu/trac/gia

© *Draft date March 18, 2010*

Contents

Chapter 1

Introduction

The Graph Alignment, Identification and Analysis (GAIA) library and tool is an open source software designed and developed for performing machine learning and analysis over graphs. The software is being developed using the Java programming language by the LINQs Group at the University of Maryland, College Park. In this user guide, we first discuss the motivations and philosophy used in developing GAIA in Section **??** and Section **??**. We then present GAIA as a tool in Chapter **??** to demonstrate how GAIA can be used most common experiments without writing any code and only having to specify a configurations file for the experimental parameters, model parameters, feature construction, and statistics and other output of the experiment. In Chapter **??**, we go beyond using GAIA simply as a tool and discuss how to use GAIA as a library. This allows users to develop new implementations of different parts of GAIA (i.e., different models, statistics, input formats, samplers, experiments, etc.) for custom experiments or to try out different models. Next, in Chapter **??**, we discuss the network visualization tool implemented within GAIA.

1.1 Motivation

GAIA was developed to address a need in the machine learning and network analysis community to have a framework to share, evaluate, and apply different algorithms and models over graphs. Prior to GAIA, testing or applying network algorithms required re-implementing the code. Not only does this make it difficult to do any meaningful comparisons, hindering reproducibility of experiments, this also results in wasted effort in the form of redundant code. Moreover, this made it difficult to analyze the interplay between different machine learning tasks over network needed to look at the problem of Graph Identification.

With these problems in mind, GAIA was developed as a general framework for implementing algorithms and methods for use over general graph. Special care was made to make GAIA general enough to handle all different types and sizes of graphs, as well as algorithms for the network based problems defined in Diehl et al.**?**, namely:

- data representation-Examining ways to represent different types of linked data.
- object ranking-Using attributes and link structure to order a set of objects within a graph
- group detection-Cluster objects in a graph into groups that share a common characteristic
- collective classification-To predict labels of objects in a graph, making use of correlations associated with how those objects are connected in the graph
- link prediction-Predicted the existence of links in a graph
- entity resolution-Resolving objects in a graph to the underlying entity it refers to
- subgraph discovery-Find interesting or commonly occurring subgraphs in one or more graphs
- graph classification-Predict the labels of whole graphs
- generative models-Analyze ways in which graphs with common graph properties can be generated using various generation models
- graph identification-Identify the desired information graph (with the proper and complete set of nodes and edges for analysis) from a given data graph (which maybe noisy, incomplete, and at the wrong level of abstraction)
- graph alignment-Looks at ways objects within a graph correspond to objects in a second graph. One common name for graph alignment is ontology alignment.

As of this time, algorithms have been implemented for generative models, object classification, link prediction, and entity resolution.

1.2 General Philosophy

Given the motivation presented in Section **??**, GAIA was developed with the following guidelines:

- The library should be modular to allow for different parts of GAIA to be developed and changed independently. This will allow users to develop just the parts they are interested in analyzing and still be able to use the other parts of the code.

- The graph is the connecting piece among the different modules. For this end, the graph interface must be general enough to handle graphs of all types and sizes. Specifically, the graph interface allows for all the nodes, edges, and the graph to have features. Moreover, the graph interface supports many different types of features, including strings, numerics, categorical, and list values. The features can also have probabilities associated with each of them. Next, the graph interface also allows for both directed and undirected hyperedges. Finally, the graph interface is independent of the underlying implementation. This means different implementations of the graph interface can be used to efficiently handle the problem at hand.

- The code should be well documented, both in javadoc and inline comments, to make using and debugging code easy. Also, the code should be sufficiently readable that the code can be used as a guide to understanding the algorithms implemented therein.

- The code is designed to maximize code reuse. Commonly used functionality and code should be inherited or stored as utilities.

- The code should simplify comparison of different algorithms. Evaluation statistics should be implemented so those don't have to be rewritten.

- Though the code is a library, the code should be set up so that it can be used by a user with little to no coding or configuration. This involves setting up Experiment classes to handle many common tasks over networks, as well as making sure the algorithms implemented have as many parameters over values to have a default value set.

- The code should be designed to allow repeatability of experiments. All implementations should be set up such that the exact same experiment can be rerun given the used data and configuration file.

1.3 Caveats

Although we are striving to make implementations within GAIA as general as possible, some implementations may have limitations with regard to how memory and runtime. This maybe due to limitations of the algorithm being implemented itself, as well as practical limitations faced when attempting to include as many implementations of algorithms within GAIA as possible. If running into difficulty with runtime or running out of memory, make sure to take a look at the Java documentation written for the implementation in question.

Chapter 2

Installing GAIA

2.1 Getting GAIA

2.1.1 Viewing the Source Code

One way to explore the GAIA source code is to go to http://linqs.cs.umd.edu/trac/gia and click on Browse Source. This is a web interface allowing you to view the source code without downloading anything. The GAIA project uses SVN as its revision management system. Thus, the repository is laid out using standard SVN conventions where "trunk" contains the main revision of the code, "branch" contains different branches of the code, and tags contains tagged points of the code representing stable revisions.

2.1.2 Downloading GAIA

The simplest way to obtain GAIA is to download it as a jar file from http://linqs.cs.umd.edu/trac/gia. There are two jar files. The first, gia.jar, contains the binary for the GAIA. The second file, gia-src.jar, contains the binary, as well as the source code for GAIA. Both jar files were generated using the Eclipse Export feature. Thus, the jar file with the source code can be directly loaded into Eclipse.

Note, some features of eclipse may require third party libraries. See Section **??** for what those are and where to download them.

2.1.3 Checking out GAIA

Another way to acquire GAIA is to check it out directly from the SVN. GAIA can currently be checked out using the command:

```
svn co http://linqs.cs.umd.edu/svn/gia
```

Note that by default, access to the SVN is read only so any modification to the code cannot be checked in directly. For information on contributing your code to the project, see Section **??**.

2.2 Third Party Libraries

Some features of GAIA require third party libraries which are not including in the GAIA jar file and must be downloaded separately. Links to the specific versions of the libraries are available at http://linqs.cs.umd.edu/trac/gia/wiki/ThirdPartyLibraries.

2.3 Contributing to GAIA

Contributions to the GAIA code base is encouraged and welcome. Look at Chapter **??** for details about the GAIA code base and conventions used in the code. Contributions can be sent via email to gia-devel@cs.umd.edu. Please include documentation about the changes and features you'd like to contribute and a jar file of the changes.

Chapter 3

Basic User Guide

In this chapter, we will talk about how to use GAIA as a tool. GAIA was designed so users could run basic experiments without having to write code. Instead, the user can use one of the Experiment classes which was created to do most of the common machine learning tasks. As input, a configuration file is passed which contains the different parameters required for the experiment and algorithms the user wishes to test.

In this chapter, we begin by providing some basic terminology and concepts used throughout GAIA in Section **??**. We then go through an example experiment in Section **??**.

3.1 Basic Definitions

3.1.1 Graph

A Graph is the primary object in GAIA on which the different modules of GAIA perform over. GAIA is set up such that multiple Graphs can be instantiated at any time using any available implementation of the Graph interface. Now, within each graph, there are objects known as Graph Items. A Graph Item is a general term to describe all the objects within the graph itself namely, the nodes and edges in the graph. The GAIA Graph interface allows for heterogenous graphs with many different kinds of nodes and edges (see Section **??**). Furthermore, it allows for both undirected and directed edge types, as well as hyperedges. This means that undirected edges can consist of one or more connected nodes and directed edges can consist of one or more source nodes and one or more target nodes. Implementations of graphs and input formats vary though so make sure to look at them prior to using.

3.1.2 Decorable

Any object in GAIA which can hold attributes are considered Decorable. The Decorable objects in GAIA include all Graphs, Nodes, and Edges. Each Decorable object has a corresponding schema ID and schema which specifies what features and what type each feature is. For example, a Node might have the schema ID of “person” which corresponds to a schema that includes the features numeric feature “age,” a categorical feature (“male” or “female”) “gender,” and string value “name.” Similarly, a directed edge may have a of schema ID “friendof” which corresponds to a schema which contains a numeric feature “friendshipstrength.”

Aside from the schema ID, all Decorable items also hold an ID which is unique for each object. Graphs are unique given a string identifier, we call the object ID, and its schema ID. For example, two graphs can be a graph with the object ID of “myspace” and schema “socialnetwork” and another can be a graph with object ID “facebook” with schema “socialnetwork”. Similarly, Graph Items (nodes and edges) are unique given the ID of the graph it belongs to, the object’s schema ID, and a string value representing an identifier for the object. For Graph Items, the object identifier needs to only be unique for a given graph and schema pair. For example, an object id of “eagle” can be used for both a node of schema “animal,” as well as a node of schema “sports-team” in a given graph. Likewise, two nodes with object id “eagle” with schema “animal” is valid and can exist as long as they are in two different graphs.

3.1.3 Feature

Decorable items can hold features. There are two classes of features, Explicit and Derived. Explicit features are features whose values are explicitly set. For example, a feature like ”name” for a person is set to “John” and “Anna.” On the other hand, Derived features are features whose values are not set but are instead derived from other features or the graph itself. For example, a feature called “degree” for nodes may not be set, and instead is calculated for a node by counting how many edges the node is connected to.

Aside from the two major classes of features, there are also many different types of features corresponding to the type of value the feature can hold. These include string, numeric, categorical, multiple categorical, and multi valued. See Java documentation in linqs.gia.feature for more details.

A major feature of GAIA is the ability to created custom derived features using a declarative language. Although many simple features may have direct implementation available (see linqs.gia.feature.derived), some domains may need more customized derived features. By using the the feature construction feature of GAIA, these features can be created by specifying a feature template (a text file defining how the feature should be computed) and specifying a feature construction file (a text file specifying what features to add in what feature schema). For this chapter, we will demonstrate this capability using one of the predefined derived features. To create custom features using the declarative language, go to Chapter **??**.

3.1.4 Configurable

Configurable items in GAIA are items which you can load and save parameters to. Configurable items include models, experiments, statistics, and derived features. Because these items implement the configurable interface, users can specify an arbitrary number of parameters that maybe needed for each implementation. For example, an model involving object classification will need a parameter to define which feature to predict. Similarly, an experiment may need you to specify which statistics you want to calculate over the predictions (e.g., accuracy, confusion matrix, f-measure).

Parameters are stored as key-value pairs. The key is of type string and the value can be either a string or a number. Parameters can be set in the code by calling setParameter, or more commonly, via a configuration file.

See Java documentation for linqs.gia.configurable for formatting and further details.

3.1.5 Logging

There are many different logging levels defined in GAIA for printing output. This is used as an alternative to printing straight to the system output. This allows users to turn certain messages on and off, depending the situation. Supported logging levels include four major types INFO, MINOR, DEBUG, and DEV. INFO messages are messages which are always printed by default. When running experiments, this may include messages that print what the values for the requested statistics are. MINOR messages are also always printed by default. They print out situations in the code which may not be what the user intended. One example for a MINOR message is that when using certain IO formats, a warning message is printed whenever you try to load an edge in which one of the nodes specified are not defined in the graph. This may be desired in some situations, i.e., you only want to load a part of the whole graph, but in case it isn't, a warning will let you know what is occurring without necessarily halting the execution by throwing an exception. The next level of logging is DEBUG. DEBUG is not printed by default and should only be turned on when trying to understand what is going on with the code. This maybe so you can keep track of what part of the code you're currently in, or to print out messages from different parts of the code for use in debugging unexpected problems. In IO classes, for example, you may want to print which file you are currently loading so that if you're having problems loading the files, you can look directly at the file which is causing the error. Finally, the last level of logging is DEV. DEV is like DEBUG but is only intended for use in development. If you see a DEV message being printed in released code, notify the GAIA developers immediately.

3.1.6 Experiment

When choosing to use GAIA as a tool, you can use one of the predefined Experiment implementations. Each implementation is defined to run a specific type of experiment. Many of the most common experiments are defined in GAIA. To run the experiment, all you need specify is a configuration file (which contains all the parameters used by all Configurable items defined in the experiment, as the command line arguments. An example of this is show in Section **??**.

You can identify which type experiment you want to run by looking at the Javadocs for the defined experiment implementations given under linqs.gia.experiment.

3.2 Example

3.2.1 Object Classification

In this section, we give an example of running an object classification experiment using GAIA. For this example, you will need to download the following:

1. The sample experiment configuration and data files. You can download and unzip these files from http://linqs.cs.umd.edu/trac/gia/wiki/ocexperiment.tgz (if you downloaded the source code for GAIA, the same configuration files are also available from resource/SampleFiles/OCExperimentSample)

2. The GAIA library (gia.jar) from http://linqs.cs.umd.edu/trac.

Place all the contents of the zip file and the jar files in the same directory. To run an experiment, you can now just run the command:

```
./runExp.sh
```

What should be printed out now are log messages indicating the progress and eventual results, including statistics, of running an object classification algorithm with GAIA over the WebKB dataset. We now look closer into exactly what occurred and how.

Looking at ./runExp.sh, we see that it is a bash shell script which just executes the command:

java -classpath=./gia.jar linqs.gia.experiment.OCExperiment experiment.cfg.

Chapter 4

Advanced User/Developer Guide

- Talk about how to develop using the code.
- Talk about the different levels of abstraction.
- Talk about coding standards
- Talk about how to use/include third party libraries
- Talk about the level of documentation
- Talk about looking through the utilities. Discuss some of the more commonly used utilities.
- Discuss basic architecture of the code

Chapter 5

Feature Construction

In this chapter, we describe how to do feature construction using the declarative language.

Chapter 6

Graph Visualization

In this chapter, we describe how to use the graph visualizations available in GIA.

- We're using Prefuse
- Cite dualnet
- Talk about how to use the tool
- Talk about how to create your own visualizations