LPmade - The Manual

Ryan N. Lichtenwalter

November 10, 2011

Contents

1 Introduction

LPmade is a complete cross-platform software solution for multicore link prediction and related tasks and analysis. Its first principal contributions are a scalable network library supporting high-performance implementations of the most commonly employed unsupervised link prediction methods. Link prediction in longitudinal data requires a sophisticated and disciplined process for correct results and fair evaluation, so the second principle contribution of LPmade is a sophisticated GNU `make` script that completely automates link prediction, prediction evaluation, and network analysis. Finally, LPmade streamlines and automates the process of creating multivariate supervised link prediction models as proposed in [7] with a version of WEKA [8] modified to operate effectively on extremely large data sets. With mere minutes of manual work, one may start with a raw stream of records representing a network and progress through hundreds of steps to complete plots, gigabytes or terabytes of output, and actionable or publishable results.

There is no shortage of graph libraries: the Boost Graph Library, SNAP, igraph, JGraphT, GraphCrunch, GOBLIN and many others. Some offer extreme generality, some offer extreme efficiency, some offer modeling utilities, and some have a dizzying array of algorithms. LPmade is not just yet another graph library. Its software components are, by necessity, designed for high performance, and it does offer a wide array of graph analysis algorithms. It is, however, first and foremost an extensive toolkit for performing link prediction to achieve both research and application goals.

Link prediction is of increasing interest in both research and corporate contexts. Virtually every major conference and journal in data mining or machine learning now has a significant network science component, and these very often include treatment of link prediction. Link prediction is of great use in domains ranging from biology to corporate recruiting, but it is a difficult area in which to develop models because of extreme class imbalance, the longitudinal nature

of the data, the difficulties inherent in effective evaluation, and other issues raised in [7]. This software represents not only the first library to focus on link prediction specifically, incorporating highly general and extensible forms of the predictors introduced in [6]. It also streamlines and parameterizes the complex link prediction work flow so that researchers can start with source data and achieve predictions in minimal time whether they are using pre-built components or their own extensions to the software. The software and manual are available at `http://nd.edu/~dial/software/LPmade.tar.gz`.

2 License

Portions of this software are released under the GNU GPL version 3. You may copy, distribute, and modify this software as you like subject to the terms of the GNU GPL. In addition, we request that any publications for which LPmade is used to obtain results cite *LPmade* by Ryan N. Lichtenwalter and Nitesh V. Chawla in the Journal of Machine Learning Research.

In short, LPmade is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. LPmade is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with LPmade. If not, see `http://www.gnu.org/licenses/`.

The bundled `gnuplot` is released under its own license. The `costcurve` and `hmeasure` binaries are unlicensed and may be freely used, modified, or distributed. For specific license terms, please see `./licenses.txt`.

3 Software Contents

LPmade contains the following components and other software:

- netlib - a network library written expressly for LPmade containing analytical tools and link prediction code
- metrics - a classifier evaluation library written expressly for LPmade containing highly optimized evaluation code
  - ROC and AUROC - receiver operating characteristic curve and curve area calculators
  - PR and AUPR - precision-recall curve and curve area calculators
  - MSE - mean-squared error calculator
- wd - the working directory area

  - Makefile.common - a large GNU `make` script for controlling complex link prediction work flows
  - Makefile.options - the GNU `make` script in which paths and important environment information should be placed
  - condmat - moderately small sample network data from Mark Newman; includes task-specific Makefile for purposes of illustration
  - disease-g - moderately small sample network data from Darcy Davis; includes task-specific Makefile for purposes of illustration

- gnuplot - a commonly used and widely available software package for producing high-quality plots
- WEKA - a commonly used data mining and machine learning toolkit

4 Installation

The procedure for installation is simple. The distribution is constructed such that it is unnecessary to seek and learn how to build and integrate any libraries. The only possible prerequisite for successful operation on supported systems is version 1.5 or higher of the Java Runtime Environment (JRE). We recommend Oracle/Sun Java, but the OpenJDK may work. This is only required if your goals include using the bundled non-standard WEKA distribution.

4.1 Downloading

You must first download the LPmade gzipped tarball into some desired location on your system. You can find the complete LPmade distribution with bundled `gnuplot` and WEKA at `http://nd.edu/~dial/software/LPmade.tar.gz`. After successfully downloading the software with a command such as `wget`, you should inflate it with `tar`. One possible procedure is included below.

```
wget http://nd.edu/~dial/software/LPmade.tar.gz
tar -xzf LPmade.tar.gz
```

4.2 Building

Navigate to the directory into which you expanded the gzipped tarball. Here, you will find two files containing the license terms and a file called 'make_all.sh'. To build LPmade and prepare it for use, you must execute 'make_all.sh'.

```
./make_all.sh
```

This will perform the following actions within the working directory.

- Compile all components of the network and link prediction library.
- Compile all evaluation code.

- Compile the bundled version of `gnuplot` and install into the gnuplot subdirectory.

After this process completes without errors, you can make use of any of the binaries in the evaluation or network libraries individually or you can use the work flow management scripts.

To use the bundled WEKA distribution, you must first set the `JAVA` variable to the absolute path of the `java` binary on your system. This binary must from JRE or JDK version 1.5 or higher. See `http://www.java.com/en/` for further instructions regarding Java.

5 The Network Libary

The network library contains two major components. One component is the WeightedNetwork class. The other component is a set of files that provide access to many different link prediction methods.

5.1 The WeightedNetwork Class

The WeightedNetwork class is built from two main data structures. The first is the name mapping data structure. The mapping is implemented as a sorted vector of external vertex identifiers. Vector indices correspond to internal vertex names and vector values correspond to external vertex names. This design allows for $O(1)$ internal-to-external translation and $O(\log(|V|))$ external-to-internal translation while requiring the minimum possible space. The second primary data structure is the adjacency list. This data structure is implemented as a vector of vectors. Indices in the first-level vector correspond to vertices and the sorted values in the second-level vector correspond to pairs containing neighbor-weight tuples. This allows $O(\log(|V|))$ worst-case edge queries. In sparse networks where $O(|E|) = O(|V|)$, the edge queries are amortized $O(1)$. The adjacency list data structure is duplicated to represent both in-neighbors and out-neighbors to support rapid querying of both in-neighbor sets and out-neighbor sets. This is important to speed up several algorithms including the Adamic/Adar link prediction method. Internal names are represented as unsigned integers. Weights are represented as doubles.

The class supports weighted, directed networks. It therefore supports unweighted networks by setting all weights to 1 and undirected networks by maintaining duplicate copies of the adjacency list. The WeightedNetwork class is designed to be immutable through its public interface. Any public functions that manipulate the network itself return a new copy of the network.

The WeightedNetwork class provides optimized implementations of several standard asymptotically optimal network analysis and manipulation algorithms. It provides Tarjan's algorithm for finding strongly connected components in directed graphs and Brandes' algorithm [1] for computing betweenness centrality. It also provides algorithms based on both breadth-first search and Dijkstra's algorithm to support hop-based or weight-based distance computations.

The class does not currently include any template arguments. In the future, template arguments may be added to support undirected and unweighted networks explicitly, thus decreasing unnecessary memory overhead. Such design decisions must be balanced against the learning curve required to use the library. It is very likely that future versions of the library will contain a template argument for different external vertex identifier types. Although accepting strings as external identifiers and maintaining them in memory is wasteful compared performing and writing a preliminary map to disk, for small networks, this facility is a desirable convenience the implementation of which has virtually no impact on code complexity.

5.1.1 Existing Binaries

The library currently contains the following binaries, most of which accept the network file at standard input:

- `adjacency_list` - Produces a standard adjacency list with the external name of the vertex followed by a space-separated list of comma-separated neighbor-weight tuples. Each vertex occupies a separate line.
- `all_pairs` - Produces the complete list of $\binom{|V|}{2}$ pairs in the network ordered first by source and then by target.
- `assortativity_coefficient` - Computes the assortativity coefficient of the network, which is the tendency of vertices with a given degree to be connected to other vertices with a similar degree.
- `average_clustering_coefficient` - Computes the arithmetic mean of clustering coefficients of all vertices in the network.
- `betweenness` - Computes betweenness centralities in $O(|V||E|)$ time for all vertices in the network using Brandes' algorithm. This implementation ignores edge weights.
- `clustering_coefficient_distribution` - Computes the clustering coefficient of all vertices in the network.
- `clustering_spectrum` - Computes the clustering spectrum of the network, the arithmetic mean of clustering coefficients by degree.
- `combine_networks` - Accepts as parameters two or more network files and outputs a network that contains all vertices and edges in either network.
- `count_edges` - Outputs the number of edges in the network. This is most efficiently achieved by using `grep` to search for 'Edges' in the network file.
- `count_mutuals` - Outputs the number of bidirectional relationships in the network.

- `count_vertices` - Outputs the number of vertices in the network. This is most efficiently achieved with the command `head -n 1` on the network file.
- `duplicate` - Creates an identical copy of the input network and writes it to output. The functionality is most efficiently achieved with either `cp` or `cat` depending on the desired goal.
- `eccentricity_distribution` - Computes the eccentricity of each vertex, the longest shortest path from it to any other vertex in the network.
- `free_choice_mutuality` - Computes the mutuality score of the network based on the free choice model.
- `label_predictions` - Accepts a `.pred` file from `predict` and generates the corresponding `.prob` file by stripping vertex identifiers and appending labels.
- `largest_scc` - Creates a new network consisting of the largest strongly connected component in the input network.
- `largest_wcc` - Creates a new network consisting of the largest weakly connected component in the input network.
- `out_degree_distribution` - Produces the out-degree distribution of the network.
- `pagerank` - Computes PageRank [2] scores for every vertex in the network with the PageRank parameter d.
- `predict` - The binary that provides access to all the prediction methods in the link prediction library as described separately in this manual.
- `pseudo_diameter` - A fast approximation algorithm that executes repeated breadth-first searches from the most distant vertex found in the previous step in an attempt to approximate the diameter of the network. The algorithm terminates when the distance does not increase from one iteration to the next. In rare circumstances, the algorithm may overestimate, but it cannot underestimate.
- `remove_edges` - Produces a copy of the network with the edges provided in the parameter file removed.
- `remove_isolates` - Produces a copy of the network with all isolate vertices removed.
- `reverse_edges` - Produces a copy of the network with all edge directions reversed.
- `scc_distribution` - Produces a distribution of the size of strongly connected components in the network.

- `shortest_path_distribution` - Produces the distribution of $\binom{|V|}{2}$ shortest path lengths in the network.
- `sim_rank` - Computes the SimRank scores of the network. This is only feasible for small networks because of $O(|V|^2)$ space requirements.
- `snowball_sample` - Creates a snowball sample of the network to the given distance from the given vertex using out-edges.
- `snowball_sample_in` - Creates a snowball sample of the network to the given distance from the given vertex using in-edges.
- `stream_to_network` - Given a stream of edges or events and the separator and index of the source and target, generate a properly formatted network file.
- `threshold_edges` - Produce a copy of the network with all edges below the threshold removed.
- `wcc_distribution` - Produces a distribution of the size of weakly connected components in the network.
- `weighted_stream_to_network` - Given a stream of edges or events such that each edge or event comes with a weight and the separator and index of the source, target, and weight, generate a properly formatted network file.

5.1.2 Testing

The network library includes a suite of tests. These can be invoked with the command `make test` in the `./netlib` directory. The testing architecture is also easily extensible. To write additional tests or to write a test for an extension, users must create files with input, a file with correct or expected output, and a file that describes how the binary should be invoked. If these files all have the same prefix as the binary name, the test will automatically be run when `make test` is issued. Please see the files in the `./netlib/test` directory for examples.

5.2 The Link Prediction Library

The unsupervised link prediction methods are all contained within the directory `./netlib/LinkPredictor`. All classes are subclasses of the abstract class LinkPredictor. All derived classes are initialized with a WeightedNetwork object. The class includes one pure virtual function, `generateScore`, which any useful derived class must implement. It also includes the two utility functions `printOutNeighborScores` and `printInNeighborScores` to print scores corresponding to all possible new links that would span a given distance. A distance of 0 indicates that the predictor should generate scores for all potential links.

The library currently includes Adamic/Adar, clustering coefficient of the source, common neighbors, geodesic distance, degree of source, PageRank of

source, total weight from source, degree of target, PageRank of target, total weight from target, Jaccard coefficient, Katz, existence of a reciprocal edge, always positive, count of all paths, preferential attachment, PropFlow, rooted PageRank, count of shortest paths, SimRank, and weighted rooted PageRank. These take various parameters according to the logical or published requirements of the methods. All predictors can use either in-edges or out-edges. To use out-edges, use `printInNeighborScores` after reversing the direction of all edges. The facility to do this is provided in `WeightedNetwork::reverseEdges`. All predictors are accessible for use through the binary built from './netlib/LinkPredictor/mains/predict.cpp'. To add a predictor to this binary, follow the many examples already present in the file.

Each predictor is initialized to include some basic information about the last source and target for which it generated a prediction. For purposes of efficiency, some predictors save results from one prediction request to the next. For instance, path-based predictors that execute from a given source vertex to provide values for many possible targets keep the scores for each target. Subsequent calls with the same source simply return the stored value for the given target. It is therefore best to order calls so that the pairs with the same source vertex are contiguous. Other predictors such as SimRank must generate all predictions in the process of generating one. In networks where SimRank is feasible, all predictions must occupy space in a matrix in memory.

5.3 Using Link Predictors

All unsupervised link prediction methods are accessible for use through the `predict` binary in './netlib/bin'. All pairs predicted by using this binary are in terms of external names, which are associated with scores indicating the estimated likelihoods of link formation. In the automated system, files of this format are called `.pred` files. These files appear as follows:

```
5 7 0.4
5 9 0.1
6 2 3.8
```

This binary accepts several parameters, which are printed upon invocation with missing or incorrect parameters. A sample usage is provided below:

```
predict -d O -f file.net -n 2 Katz 5 0.005
```

The order of the parameters specified to predict is not significant, but the `predict` parameters must precede the requested predictor and its parameters. The order of parameters to the requested predictor is significant. Some users may find it helpful to note that in some ways this binary and command-line usage is similar to that used by WEKA for selection of its operational classes. Predictor names are case sensitive.

5.3.1 Parameters

Parameters to the `predict` binary itself are as follows:

- `-d DIRECTION`: The direction of edges to use; must be in I,O
- `-f NETWORK`: File containing the network in which to predict
- `-n DEGREE`: Degree of neighborhood in which to predict
- `-c`: Specifies that node-pairs for prediction are available at standard input

The `-d DIRECTION` Option This option is used to indicate whether the requested predictor should use in-edges or out-edges. It is not required, and the default is to use out-edges. In-edge predictions are not actually accomplished with a different prediction routine. Instead, the direction of all edges in the network is reversed, but this reversal is designed to work so that calls to a predictor with `-d I -n 2` and `-d O -n 2` both produce the predictions for the same edges in the same order.

The `-f NETWORK` Option This option allows for specification of the network. It is required. The network must be in the graph file format described in this manual in Section 5.4.

The `-n DEGREE` Option This option allows for automatically generating all predictions based on a given geodesic distance. For instance `-n 2` will generate the set of predictions correspond to all source-target pairs that are two hops away from each other in the file specific by `-f`. Notably, when this option is set to 0, predictions are generated for all pairs that are not already connected by an edge regardless of distance. The `-n` option should not be used with the `-c` flag.

The `-c` Flag This flag indicates that the set of pairs is available at standard input. The format must be source then target separated by a space. As indicated above, pairs specified this way should be in terms of external names. These pairs may be specified in any order, but some predictors cache multiple target results after performing computations for a particular source, so it may be more efficient to order pairs so that identical source vertices are contiguous. The `-c` flag should not be used with the `-n` option.

5.3.2 Predictors

After specifying the options and flags to the predict binary, users should select the predictor they wish to use. Predictors distributed with the current version of the libary include the following:

- AdamicAdar

- CommonNeighbor
- ClusteringCoefficient
- Distance
- IDegree
- IPageRank
- IVolume
- JaccardCoefficient
- JDegree
- JPageRank
- JVolume
- Katz
- Mutuality
- One
- PreferentialAttachment
- PropFlow
- RootedPageRank
- ShortestPathCount
- SimRank
- WeightedRootedPageRank

AdamicAdar The Adamic/Adar link prediction method as presented in [6]. This predictor is an inverse frequency measure. The contribution of a common neighbor to the score is weighted in proportion to the rarity of the neighbor. This predictor accepts no additional arguments.

CommonNeighbor The common neighbors link prediction method as presented in [6]. This predictor is the simple count of common neighbors. In directed networks, a common neighbor exists if there is a neighbor relationship between the source and target and the neighbor in terms of the direction specified by the `-d` option to predict. This means common neighbors may exist even if the source and target are more than two hops away with respect to paths along the directed edges. This predictor accepts no additional arguments.

ClusteringCoefficient The clustering coefficient of the source. For purposes of consistency with the rest of the library, this should be called IClusteringCoefficient, and this change may occur in future releases. This predictor accepts no additional arguments.

Distance The geodesic distance between the source and the target. This predictor is only useful in scenarios when the `-c` flag is provided to `predict` or when `-n 0` is used. This predictor accepts no additional arguments.

IDegree/JDegree The degree of the source or target. The `-d` option to `predict` determines whether to use in-degree or out-degree. This predictor accepts no additional arguments.

IPageRank/JPageRank The PageRank score of the source or target. The `-d` option to `predict` determines whether to use in-degree or out-degree. This predictor requires one argument: the PageRank d value.

IVolume/JVolume The total sum of the weights in or out of the source or target. The `-d` option to `predict` determines whether to use in-degree or out-degree. This predictor accepts no additional arguments.

JaccardCoefficient The Jaccard coefficient link prediction method as presented in [6]. This predictor produces a score corresponding to the quotient of the intersection of the two neighbor sets and the union of the two neighbors sets. This predictor accepts no additional arguments.

Katz The Katz link prediction method as presented in [6]. Because this score is computed by performing traversals on the graph rather than matrix operations, and because the networks of interest are typically large, this predictor accepts as its first parameter a maximum distance away from the source. Its second parameter is the damping parameter β.

Mutuality This predictor outputs 1 if the edge exists in the other direction or 0 otherwise. It is meaningless in undirected networks. This predictor accepts no additional arguments.

One This predictor always outputs 1. This predictor is not useful for actually generating predictions, but it may be used as a null model or as a way to quickly generate the set of pairs for a given value of the parameter `-n`. This predictor accepts no additional arguments.

PreferentialAttachment Preferential attachment link prediction method as presented in [6]. As implemented in this library, it is the product of the out-degree of the source and the in-degree of the target. This predictor accepts no additional arguments.

PropFlow The PropFlow link prediction method as presented in [7]. This should serve as the reference implementation of PropFow. This predictor accepts a maximum distance to explore the network in the underlying breadth first search before terminating.

RootedPageRank Rooted PageRank link prediction method as presented in [6]. This predictor accepts the random walk restart parameter α.

ShortestPathCount This predictor outputs the number of shortest paths from the source to the target. This means that it executes a breadth first search terminating at the level at which the target is found and counts the number of times the target is encountered at that level. This predictor accepts a maximum distance to explore the network in the underlying breadth first search before terminating.

SimRank The SimRank link prediction method as presented in [6]. Because the method requires $O(|V|^2)$ space, it is feasible only for small networks. It accepts the C parameter.

WeightedRootedPageRank This is the same as rooted PageRank except that it uses edge weights if they exist to inform the transition probability from one vertex to another. This predictor accepts the random walk restart parameter α.

5.3.3 Labeling

After generating a `.pred` file containing the source external name, target external name, and score, it is necessary to obtain labels to evaluate the quality of the scores. The `label_predictions` binary will do just this. For purposes of evaluation, `.pred` files are not particularly useful, so it is often logical to simply pipe the output of the link predictor to the `label_predictions` binary. An example follows:

```
predict -f predict.net -n 2 IDegree | label_predictions label.net
```

This binary will produce a file containing the original score and the label, either 1 if a link exists in the label network or 0 if not. These two data elements are separated by spaces. In the automated system, files of this format are called `.prob` files. From here, the files can either be sorted by score to form `.sprob` files or combined into a data set for supervised classification.

5.4 File Format

The graph file format accepted by the network library is very similar to the Pajek format. There is currently no facility for comments in the format. There is currently no validity checking on the format in library I/O operations. The

format differs from the Pajek format in that it indicates the number of edges within the file after that marker line, it employs UNIX line endings, and internal vertex names start with 0 instead of 1. A sample follows below:

```
*Vertices 3
0 5
1 8
2 4832
*Edges 3
0 1 4.2
1 0 2.1
2 1 6
```

The first line indicates the number of vertices. The following lines list first the internal name and then the external name. After a number of such mappings corresponding to the number of vertices, there is a line indicating the number of edges. Following this are lines that list the internal name of the source and target for the edge followed by its weight. For undirected networks, each undirected edge is listed twice.

The following non-obvious restrictions apply to valid instances of the format:

1. Internal names are 0-indexed and contiguous. The first listed vertex must have 0 as its internal name.
2. External names must be listed in numerical order.
3. Edges must be listed first in order of source internal name and then in order of target internal name.

In general, users should not create this format directly. It should only be read and written by the library itself. To generate files of this format, users can use the `stream_to_network` or `weighted_stream_to_network` binaries. These accept a simple event list, or equivalently edge list. The unweighted version determines weights by a count of the number of duplicate appearances of an edge. The latter determines weights by summing all values in the weight column for duplicate appearances of an edge. These binaries have no requirements on order and perform all the details of ordering and mapping to generate the proper in-memory representation and subsequent LPmade file format.

6 The Evaluation Library

The evaluation library includes several popular methods for evaluation. It also includes some more experimental or esoteric evaluation measures that may be useful for visualizing and demonstrating the performance of particular predictors. All of the evaluation code has been rigorously optimized. All of the binaries require files containing a sorted list of estimations and labels except the MSE binary, which accepts its input in any order. An example file follows:

```
0.9 1
0.8 0
0.74 1
6.4e-1 1
1.0e-6 0
```

The predictors accept all C-style fixed or point notation output. In the automated system, files of this format are called `.sprob` files.

The evaluation library includes the following evaluation methods:

- roc/auroc - receiver operating characteristic curve and curve area calculators
- pr/aupr - precision-recall curve and curve area calculators
- mse - mean-squared error calculator
- ef - percentage positive in each of a parameterized number of equal-width bins
- topnprec - a threshold curve of top-n precision over varying n
- hmeasure - David J. Hand's H-measure calculation assuming the standard β distribution of costs
- costcurve - the cost improvement calculated as the area between the lower envelope and the predictor cost curve in Bob Holte's cost curve metric

6.1 Receiver Operating Characteristic Curve

The `roc` and `auroc` binaries compute the receiver operating characteristic curve and its corresponding area. Until recently, the area under the ROC curve was a widely accepted evaluation measure for data with imbalanced classes. Now, evaluation measures such as H-measure and area under the precision-recall curve are sometimes preferred. The ROC curve calculator requires the number of negative and positive instances. It is possible to determine these counts quickly and execute the binary by performing the following commands:

```
all=$$(< file.sprob.gz gunzip | wc -l)
pos=$$(< file.sprob.gz gunzip | grep -c '1$$')
let neg=$$all-$$pos
< file.sprob.gz gunzip | ./metrics/roc $$neg $$pos
```

If the curve contains too many points with excessive detail, it can be simplified with the following modified command:

```
< file.sprob.gz gunzip | ./metrics/roc $neg $pos | sed 's/\(.\...\)
    ..../\1/g' | uniq
```

6.2 Precision-Recall Curve

The precision-recall transformation of ROC space as defined in the Davis and Goadrich paper [3]. The method does not curently restrict the ROC points to the convex hull. Code already exists to do this, so it should perform this step in the upcoming release. This measure can provide extra differentiation between predictors in highly imbalanced scenarios with respect to the ROC calculators.

6.3 MSE

This is the standard mean-squared error calculator. It is the only predictor that does not rely on a threshold curve, and thus it does not require that the file associating estimations with labels be in any particular order.

6.4 EF

This measure is useful for showing the behavior of a particular prediction method within the prediction score dimension itself. For instance, if the user wants to evaluate the efficacy of using the degree of the source as a link prediction method, the output from this method provides a list of x-y pairs corresponding to equal-width bins. The abscissa represents the score and the ordinate represents the percentage of predictions with that score that are positive.

6.5 Top-n Precision

The top-n precision is an important measure when the positive class is highly interesting, but the number of false negatives is daunting. In such scenarios, it is often logical to consider only the most confident positive predictions. Without domain-specific knowledge, it is impossible to set such a threshold, so this metric generates a threshold curve of precision values over varying n top predictions.

6.6 H-measure

This is David J. Hand's solution to weaknesses in AUROC [5]. This particular implementation assumes the β distribution described in that publication. Notably, the code for this particular implementation of the H-measure metric was written by Troy Raeder who holds the copyright. This code is released for use, redistribution, and modification with no license restrictions. It is **not** released under GNU GPL version 3.

6.7 Cost Curve

This is based on Drummond's and Holte's solution to weaknesses in AUROC [4]. The lower envelope of a cost curve plot is formed by the intersection of a predictor that always predicts positive and a predictor that always predicts negative. The area between the x-axis and this lower envelope represents cost. The area between the lower envelope and the cost curve for a given prediction

method represents decrease in cost due to the prediction method. The scalar measure produced by the code describes precisely this area. Notably, the code for this particular implementation cost curve area reduction metric was written by Troy Raeder who holds the copyright. This code is released for use, redistribution, and modification with no license restrictions. It is **not** released under GNU GPL version 3.

7 Automation Capabilities

This is perhaps the most innovative and exciting part of the LPmade software distribution. The entire link prediction output building process is controllable with a sophisticated GNU `make` script. Advanced facilities in GNU `make` such as meta-level evaluation in the form of eval-call templates, string substitutions, complex pattern rules, parallel execution, and traps to ensure successful completion all culminate to create an easy yet general and robust build system. LPmade is designed to scale from small networks to very large networks, so despite the speed of the network library, the automation architecture ensures that successfully completed computation is never repeated. Only the minimum computation to perform any given task is performed, and unsuccessful or incomplete computation is retried.

7.1 Introduction to GNU `make`

GNU `make` is a program that determines automatically which pieces of a large work flow need to be constructed or reconstructed due to absence or upstream modification and issues the commands to construct or reconstruct them. The GNU `make` website is `http://www.gnu.org/software/make/`. Full details on the workings and programming of GNU `make` are available in the GNU `make` manual at `http://www.gnu.org/software/make/manual/make.html`. It is a language with some excellent features, but it can be somewhat difficult to master. Fortunately, for virtually all uses, users of LPmade need not be able to do all but the simplest programming tasks with the GNU `make` language. These include defining variables and defining simple rules. The process of using the facilities of `make` necessary for these tasks is described below.

GNU `make` first processes the task-specific Makefile and any Makefiles that it includes, determines the expanded set of rules, and begins computation. After this, the file is no longer read. This means that it is possible to alter any of the Makefiles after a long-running task has been started to start other tasks with different parameters. For instance, one might want to run predictions including preferential attachment for one network but excluding it for another. To do this, execute `make` within the working directory for the first network, modify the `PREDICTORS` parameter in `Makefile.common` as discussed below, then execute `make` within the working directory of the second network.

Many of the tasks will require significant time for large networks. To prevent failures due to SSH disconnections or other network problems, it may be

desirable to invoke `make` with `nohup`. The following example describes how to run a long-running set of tasks with the automation framework that will not fail due to loss of connectivity to the host machine, which we will assume has 8 cores available for dedication to the tasks.

```
nohup make -j 8 classify &
```

7.2 Quick Start

Much of the following text is dedicated to describing the build system, and it is an important reference when problems arise. In general, the process is fairly simple, especially for those familiar with GNU `make`. The following series of steps describes the process to quickly use the automated build system for link prediction and related analysis in a new network data set.

1. Follow the instructions in Section 4 to prepare the system.
2. Create a subdirectory under `./wd` with the name of your network. For purposes of example, let us call it `mynet`, so `./wd/mynet`.
3. Create a `src` subdirectory, `./wd/mynet/src`. Move all necessary source files into this directory.
4. Copy a sample task-specific Makefiles from the condmat or disease-g network. Edit this file so that it indicates how to interpret the raw source data, possibly referencing external scripts, which can go into a subdirectory `./wd/mynet/script/`.
5. Change directory to `./wd/mynet`.
6. Type `make -j <target>` where target is one of the options from Section 7.3.2 or a new target that you define in the task-specific Makefile.

7.3 Makefiles

There are three Makefiles driving the build system. Two of these are important for defining the build process while the third isolates and defines important path variables. The names and purposes of these Makefiles are below:

- `Makefile.options` - defines the location of important binaries necessary for the build process
- `Makefile.common` - defines general rules that apply to all networks LPmade is designed to handle
- `Makefile` - task-specific makefile located in individual network working directories such as `./wd/condmat` or `./wd/disease-g`

7.3.1 The `Makefile.options` File

This file is located in the working directory location, `wd`. It contains important paths and binary locations. Some of these are initially set to logical system defaults. Others are set to locations within the distribution. For those set to locations within the distribution, it is important to note that the relative paths are relative to the `make` invocation location, which is always a subdirectory of `wd`, not the `./wd/Makefile.options` location itself.

- `TEMP` - The path to the temporary directory, which defaults to `/tmp`. In the current version, this variable is only used to provide the `-T` option for `sort`. Since link prediction can produce many millions or billions of instances, it may be important to define a location with more space than the partition on which `/tmp` resides.
- `JAVA` - The path to the actual `java` binary. There is no default, so this variable must be set for performing tasks with WEKA. The `java` binary must be part of a version 1.5 or higher distribution of Java.
- `BIN` - The location where the network library binaries are located. This variable defaults to `../../netlib/bin` and does not need to be changed unless the internal directory structure of the distribution is modified by the user.
- `METRICS` - The location where the evaluation library binaries are located. This variable defaults to `../../metrics` and does not need to be changed unless the internal directory structure of the distribution is modified by the user.
- `GNUPLOT` - The path to the actual compiled `gnuplot` binary. This path will only be accurate after the distribution is built according the specifications in this manual. This variable defaults to `../../gnuplot/bin/gnuplot` and does not need to be changed unless the internal directory structure of the distribution is modified by the user.
- `WEKA` - The path to the actual distributed WEKA JAR file. This variable defaults to `../../weka/weka.jar` and does not need to be changed unless the internal directory structure of the distribution is modified by the user.

7.3.2 The `Makefile.common` File

This file contains all the general rules that pertain downstream of conversion from the raw source data to the stream data used as a source for the automated build system. This file includes several parameters in the form of user-definable variables for the rules it contains. It also defines several targets.

The parameters contained within the `Makefile.common` are listed below with an explanation of their effects:

- `NEIGHBORHOODS` - This variable defines the geodesic distances of the neighborhoods in which predictions should be generated. Consistent with the `predict` binary, a value of 0 indicates that predictions should be generated for all pairs of vertices between which no edge exists.

- `PREDICTORS` - This variable should contain a selection of the link prediction methods available within the `predict` binary as described in Section 5.3.2. This selection of predictors is used for all unsupervised predictions and analysis.

- `FEATURES` - This variable should contain a selection of the link prediction methods available within the `predict` binary. This selection of predictors is used for training and test set construction in supervised classification tasks.

- `VCP` - This variable should contain the VCP methods to use for link prediction. Currently supported are VCP3Undirected, VCP3Directed, VCP4Undirected, and VCP4Directed. If the `vcp` make target is not used, this variable has no function.

- `ifeq ($(DIRECTED),1) ...` - This conditional allows for the specification of which predictors should be run bidirectionally in directed networks. Predictors for which predictions in both directions are not desired should be appended to the `PREDICTORS` or `FEATURES` variables after the first GNU `make` statement in the conditional block.

- `NUM_BAGS` - The number of bags to use in the bagging framework for supervised classification. The default supervised classification approach defined in `Makefile.common` is consistent with [7], which suggests 10 bags of 10 random forest predictors.

- `PERCENT_POSITIVE` - The real number in (0,100) that defines what percentage of the training set should be positive. The training set within each bag is separately undersampled to this value. The test set is never manipulated in any way.

The targets are provided below with an explanation of their effects. For some of these targets, such as 'sm' and 'classify', the entire set of results is differentiated according to neighborhood. When multiple neighborhoods are specified with the `NEIGHBORHOOD` variable, the build paths for these neighborhoods are independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`. Some targets perform some or all of the work of other targets to accomplish their goals.

- **src** - The target for generating or procuring source files. This target will only function correctly if the rule has been defined by the user in the task-specific Makefile.

- **stream** - The target for generating the stream period files from raw source data. This target will only function correctly if the periods and rules have been defined by the user in the task-specific Makefile. For most purposes, stream periods are independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **net** - The target for generating all network files. This includes the feature network, label network, unsupervised network, test network, and complete network. The creation of these networks is independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **sm** - The target for producing all the unsupervised link prediction methods indicated by the `PREDICTORS` variable in different edge directions depending on the `DIRECTED` variable. This target also labels the predictions, sorts the labeled predictions, and runs the evaluation measures indicated in the target dependency area. The output of this rule is a set of `.prob`, `.sprob`, and evaluation metric files. The build path for each predictor is independent and build paths can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **classify** - The target for producing the supervised predictions described in [7] using the features indicated by the `FEATURES` variable in different edge directions depending on the `DIRECTED` variable. This requires using the feature network for producing training features and the label network for producing training labels. If the 'sm' target has already been built, its `.prob` files are used to construct the testing data. Otherwise the `.prob` files of the 'sm' target only are built to support testing. All of the build paths for the training and testing features are independent. Generating the model for each bag in the supervised classification is also independent. These independent components can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **vcp** - The target for performing supervised predictions using the vertex collocation profile (VCP) method. This target will perform VCP techniques according to the values in the `VCP` variable in `Makefile.common`.

- **plots** - The target to generate plots that aggregate the threshold curve results of the individual predictors. The prediction curves to be plotted are defined with the `PREDICTORS` variable and the metrics to plot are defined in the dependencies for the 'plots' target. Each plotting path is independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **stats** - The target for generating summary statistics for the complete network. In many cases distributions are along the build path for statistics, so both distributions and statistics are generated. Each of the build paths

is independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **threshold** - The target for computing statistics about the strongly connected component breakdown of the network when edges below the thresholds specified in the dependencies are not considered to be a part of the network. Each threshold path is independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **growth** - The target for computing and plotting growth statistics. For this to work reasonably, the `PERIOD` variables in the task-specific Makefile must be defined in a reasonable way according to the domain. For longitudinal data, this might be equal-width time windows. The networks and statistics for each growth period are independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **imbalance** - The target to generate a plot of the class imbalance of the link prediction problem with respect to varying geodesic distances according to the values defined in the `NEIGHBORHOOD` variable. The target to find the values for each neighborhood are independent and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

- **params** - The target to explore the α parameter space for rooted PageRank and weighted rooted PageRank. This may be useful for specific research goals, but it exists mostly as an example of an uncomplicated manner by which one might extend the automated build framework to explore parameter spaces. The target produces a plot of the AUROC performance over varying α. Each value of α requires an independent computation and can be processed simultaneously if make is invoked with a value greater than 1 for `-j`.

7.3.3 The Task-Specific `Makefile` File

This is a task-specific Makefile that must be defined for each data source. Two sample task-specific Makefiles are already present in the `wd` working directory: condmat and disease-g. The purpose of this Makefile is to indicate to the rest of the automation structure how the raw data should be interpreted. Once the raw data is converted into the data stream format accepted by downstream processing, the remainder of the handling is uniform and is conducted by the general rules in `Makefile.common`.

The following elements are required for specification to the task-specific Makefile. Without proper specification, some or all of the build system may fail to function.

- `include ../Makefile.options` - The statement to incorporate variable specifications from the `Makefile.options` file. This statement should be at the top of the task-specific Makefile preceding all other statements. It could be included in `Makefile.common` except that some of the path

definitions may be useful for overriding rules in that file. More coverage of overriding will follow.

- `DIRECTED` - A variable indicating whether the network is directed or not.
- `WEIGHTED_STREAM` - A variable indicating whether the stream of data has weights associated with each stream event or not. If not, the number of duplicate events is used to construct edge weights. If so, the sum of weights in duplicate events is used to construct edge weights.
- `PERIOD1` ... `PERIOD`N - A user-definable number, N, of periods or splits for the data. The contents of each of these variables should be the differentiating information used to break the network data up into separate stream. For longitudinal data, these variable will correspond to temporal windows in the data. For non-longitudinal data, these variables will respond to a random selection from which folds are constructed. Either way, the raw data or the output of the user-defined raw data transformation script should contain this differentiating information. As an alternative for non-longitudinal data, the function that generates the streams could perform random assignment into each stream period. This concept is somewhat complex, so we refer the reader to the two examples provided with the distribution.
- `FEATURE` - A variable indicating which data streams, as designated by `PERIOD` variables, should be used to construct the network from which training features are computed in supervised classification.
- `LABEL` - A variable indicating which data streams, as designated by `PERIOD` variables, should be used to construct the network from which training labels are computed in supervised classification.
- `UNSUPERVISED` - A variable indicating which data streams, as designated by `PERIOD` variables, should be used to construct the network from which unsupervised method scores are computed in supervised classification. This variable also indicates the data streams that should be used to construct the network from which testing features are computed in supervised classification.
- `TEST` - A variable indicating which data streams, as designated by `PERIOD` variables, should be used to construct the network from which testing labels are computed for both unsupervised methods and supervised classification.
- `COMPLETE` - A variable indicating which data streams, as designated by `PERIOD` variables, should be used to construct the complete network from all available data. This may exclude some data at the option of the user. Otherwise, it should include all periods.

- `PERIODS` - A complete listing of all periods. This should not exclude any periods, but should include all user-defined periods from `PERIOD1` ... `PERIOD`N.
- `./stream/PERIOD1.txt.gz` ... `./stream/PERIOD`N`.txt.gz` - The definition of the rule that converts raw source data into the stream files. This is probably most easily executed with an eval-call loop, as demonstrated in the examples, but users can manually specify each rule if the syntax or semantics of the eval-call loop is confusing.
- `include ../Makefile.common` - The statement to incorporate general rule specifications from the `Makefile.common` file. This statement should follow all the variable and rule declarations listed above.

The following elements are optional for specification to the task-specific Makefile. They are not, strictly speaking, required for proper operation, but they may greatly simplify or organize the task-specific Makefile.

- `SRC` - A variable that contains a complete listing of all files necessary for producing stream output from the raw source data. In the case that multiple files are necessary but all can be downloaded from a common location, this variable can allow for easier specification of source generation rules.
- `SRC2STREAM` - A variable that identifies the location and name of the script that is used to convert the raw source data into a stream of records that can subsequently be divided into periods. This is completely unnecessary since the path to the script, if such a script is even necessary, can just as easily be incorporated directly into the make rules. In some cases, when the information that divides the stream into periods is very simple, an inline call to `grep` or `awk` may even suffice.
- `empty :=` - With the variable definition below, necessary for the representation of an isolated space character in a GNU `make` variable. See the task-specific Makefiles in the example working directories.
- `space := $(empty) $(empty)` - With the variable definition above, necessary for the representation of an isolated space character in a GNU `make` variable. See the task-specific Makefiles in the example working directories.
- eval-call loop - Greatly facilitates the specification of rules by which parts of the stream are divided into the appropriate period stream files. See the task-specific Makefiles in the example working directories.
- `./src/%.txt.gz:` - The rule that indicates how to procure or generate the raw source data. This statement may be useful if the data source is a publically available website. Users may omit this rule so long as the source files required by the stream rules are present. Omission will simply

cause the 'src' target not to function. This statement should follow the `include ../Makefile.common` directive.

8 Compatibility

LPmade was designed to work on any system with common GNU tools, and so long as machines contain these tools, LPmade will work without problems. This may include Linux, BSD, Solaris, and even Windows machines with Cygwin. On 32-bit architectures, the limitations of a 32-bit JRE will impose a maximum size on the data sets that WEKA is able to process. We provide a list here of configurations on which we have tested this implementation of LPmade.

- Architectures
  - x86
  - x64
- Operating Systems
  - Linux
    - RHEL 4
    - RHEL 5
  - Windows/Cygwin

If you use LPmade on a different system, please let us know what your results are, so we can make a note in this section and try to overcome whatever difficulties may arise in the next release. Any common distribution of Linux should contain the tools required by LPmade to properly run.

9 Feature Requests

1. None currently.

If you would like to see a particular feature in LPmade, please make a request to rlichten@nd.edu. We will add them to this list. Contributions of modifications or extensions to the source code are always welcome too.

10 Known Bugs

1. None currently.

If you suspect a bug or encounter any problems using LPmade, please report them to rlichten@nd.edu. We will add them to this list and work to fix them as quickly as possible for subsequent release.

11 Conclusion

We sincerely hope that this manual has been helpful. We look forward to making improvements to LPmade based on community feedback. If you enjoy the software and find it helpful, have a question, want to report a bug, or want to suggest a new feature, please send your emails to rlichten@nd.edu. Thanks for your interest, and happy computing!

References

[1] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Math. Sociology*, 25(2):163–177, 2001.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[3] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[4] C. Drummond and Robert C. Holte. Cost curves: An improved method for visualizing classifier performance. In *Machine Learning*, volume 65, 2006.

[5] David J. Hand. Measuring classifier performance: a coherent alternative to the area under the roc curve. *Machine learning*, 77(1):103–123, 2009.

[6] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

[7] Ryan N. Lichtenwalter, Jake T. Lussier, and Nitesh V. Chawla. New perspectives and methods in link prediction. In *KDD '10: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 243–252, New York, NY, USA, 2010. ACM.

[8] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, California, USA, second edition, 2005.