Network Statistics Documentation

Author: David Turner
Date: 3 May, 2013

Table of Contents

1. Installation

1.1. Dependencies

- Java >= 1.6.x
- R >= 2.15.x
  - R can be installed on Linux, Mac and Windows
    - **Linux:** The easiest way is to install from package manager (r-base), but if is not available then you can download it from http://cran.rstudio.com/bin/linux/
    - **Mac:** Download and install the .pkg file from http://cran.rstudio.com/bin/macosx/
    - **Windows:** Download and install from http://cran.rstudio.com/bin/windows/base/. Once installed, you may need to set up your path variable to be able to run *Rscript.exe*.
  - In addition to R, the following additional libraries are needed (installed when running make)
    - car
    - rjson
    - mvtnorm
    - multcomp
    - plotrix
    - gtools
    - R2HTML

1.2. Building

The following is for Linux and Mac only.

To build the Network Analysis tool, you should run:

```
$ make
```

This will compile the Java files and create the file *Network.jar* as well. The jar file will already be packaged with the archive, but if any changes are made to the Java application, it will be regenerated. It will also install the *rjson* library from the CRAN repositories if not already installed.

To package the all the files (excluding results) run:

```
$ make tar
```

or

```
$ make zip
```

This will create a file *Network.tar.gz* or *Network.zip* which contains all the build files and source.

If you want to clean up all the build files run:

```
$ make clean
```

1.3. Usage

generate.sh (or *generate.bat* on Windows) is used to call all the necessary programs & scripts to process the network. The following instructions are for Linux and Mac. Windows is the exact same, except replace *./generate.sh* with *generate.bat*

SYNOPSIS

./generate.sh [OPTIONS] StartDate EndDate Domain

DESCRIPTION

generate.sh requires at least 2 arguments: The **startYear**, and the **endYear**. The third argument is optional, and is the domain to the server api. If this parameter is not provided, then it will use a cached version of the data. It might also be a good idea to have the domain argument in quotations to avoid any problems with escapable characters.

```
./generate.sh -s 2010 -e 2012 -d "http://myApiDomain.com"
./generate.sh 2010 2012 "http://myApiDomain.com"
```

This could take several minutes depending on the size of the data. Once the script is done running, there will be many csv files created in *output/data*, and charts in the various subfolders of the *output/* directory. Some json files will also be place in *cache/* so that in the future the domain argument does not need to be specified. For example:

```
./generate.sh -s 2010 -e 2012
./generate.sh 2010 2012
```

This will result in using the cached json, rather than fetching a new version from the server.

OPTIONS

-s StartDate, **--startDate** StartDate
: The starting date range for the graph data.

-e EndDate, **--endDate** EndDate
: The end date range for the graph data.

-d Domain, **--domain** Domain
: The location of the server api. If this is not provided, then a cached version of the server api response.

-c, --cache
: If set, it will skip the Java application all together and use the cached csv files which have previously been generated.

2. Server API

The Java application will call a server API to get an array of nodes and edges to construct a graph. The nodes will contain an array of meta data describing each node (could be stats about that person to later apply some statistical tests).

2.1. API Parameters

The API should expect to handle the following GET Parameters:

- **year:** integer representing the year that the data is from

2.2. JSON Structure

The api should return a json object like the following:

```
{
    "nodes": [
        {
            "type": "Type1",
            "name": "Person1",
            "meta": {
                "field1": "2",
                "field2": "2"
            }
        },
        {
            "type": "Type2",
            "name": "Project1",
            "meta": {
                "field1": "1",
                "field2": "5"
            }
        }
    ],
    "edges": [
        {
            "a": "Person1",
            "b": "Person2",
            "type": "Type1"
        },
        {
            "a": "Person2",
            "b": "Project1",
            "type": "Type2"
        }
    ]
}
```

- **nodes:**
  Contains an array of nodes, where each node contains specified **type**, **name**, and **meta** fields. The **meta** field is an object containing key => value pairs between a stat and value. The value should always be in the form of a string, even if it is a numeric value. Also each node of the same **type** should use the same meta keys. If a key does not have a value for a certain node, and empty string "" should be used.
- **edges:**
  Contains an array of edges. Each edge consists of a source **a,** target **b**, and a **type**. The source and target correspond to the **name** of a node, and the **type** should be the **type** of the target node.

3. Configuration

Both the Java application and R look for the file *config.json* in the root directory of the project to tell it some information about the data that it will be processing. A sample configuration is found in the file *config.json.example*, and is shown below for reference:

```
{
    "types": [
        "Type1",
        "Type2"
    ],
    "nodes": [
        "Node1",
        "Node2"
    ],
    "edges": [
        {
            "type": "Edge1",
            "source": "Node1",
            "target": "Node2",
            "desc": "Edge Description"
        }
    ],
    "meta": {
        "Type1": {
            "groups": [
                {
                    "id": "group1",
                    "name": "Group 1",
                    "desc": "Group 1 Description"
                }
            ],
            "variables": [
                {
                    "id": "Field1",
                    "name": "Field 1",
                    "desc": "Field 1 Description"
                }
            ]
        }
    },
    "transformable": {
        "Type1": [
            "Field1",
            "Field2"
        ],
        "Type2": [
            "Field1_1",
            "Field2_1"
        ]
    },
    "transformations": {
        "Type1": [
            {
```

```
                "x": "Field3",
                "t": "SQRT"
            }
        ],
        "Type2": []
    },
    "tests": {
        "Person": [
            {
                "y": "Field2",
                "x": "Field1",
                "test": "boxplot"
            }
        ],
        "Project": [
            {
                "y": "Field1_1",
                "x": "Field2_1",
                "yOp": "AVG"
            }
        ]
    }
}
```

- **types:** An array containing the types of nodes in the graph. These must correspond the same way as they appeared in the server API.
- **nodes:** An array containing the names of the different node types.
- **edges:** An array containing objects describing all the edge types in the graph. The fields for each object are as follows:
  - **type:** The type of edge (should be short string)
  - **source:** The node type of the source node (Should match one of the strings in **nodes**)
  - **target:** The node type of the target node (Should match one of the strings in **nodes**)
  - **desc:** The description of the edge.
- **meta:** Contains a hash of all the fields in the results. The key is associated with the type of node, which then contains a second hash containing **groups** and **variables**. **groups** should contain all the non numerical fields, as well as any categorical field which may be numerical. **variables** should contain all the numerical fields. Both **groups** and **variables** contain the following fields:
  - **id:** The id of the field (same as in API, and elsewhere in the config). This should be a single word with no special characters.
  - **name:** The name of the field. This should be a more human readable version of the **id** an may contain any characters.
  - **desc:** The description of the field.

- **transformable:** Contains a hash of fields which are allowed to be transformed. The key is associated with the type of node. The data transformations are done are determined automatically by determining the best transformation from a pre defined list of transformations (listed in the **transformations** point). The best transformation is determined by performing a Shapiro normality test on all the different transformations and summing up the statistic for each year and whichever one has the maximum value, then it will go with that transformation for each year.
- **transformations:** Contains a hash of specific transformations to perform. The key is associated with the type of node. The specific transformations have the following fields:
  - **x:** The name of the field
  - **t:**The type of transformation

  Possible values for **t** are the following:
  - **IDENTITY:** This is the default value for all non-transformable fields. It simply returns the field unchanged.
  - **SQRT:** Does a square root transformation on the field: `sqrt(field)`
  - **LN:** Does a natural logarithm transformation: `ln(field + 1)`
  - **LOG:** Does a base 10 log transformation: `log(field + 1)`
  - **SQR:** Squares the field: $field^2$
  - **INV:** Does an inverse transformation: $field^{-1}$
- **tests:** Contains a hash of specific tests to perform. The key is associated with the type of node. The specific tests have the following fields:
  - **x:** The name of the field which should appear on the x axis of a plot.
  - **y:** The name of the field which should appear on the y axis of a plot
  - **yOp:** Perform some aggregate function to the y field. Possible options are:
    - **AVG:** Average all the values in the field
    - **MED:** Compute the median of the field
    - **MAX:** Compute the maximum value in the field
    - **MIN:** Compute the minimum value in the field
    - **SUM:** Sum all the values in the field
  - **type:** The type of test to perform. Possible options are:
    - **scatterplot:** do a correlation test, and draw a scatterplot (default)
    - **boxplot:** do a correlation test, and draw a boxplot

- **ttest:** do a t-test between x and y
- **anova:** do an ANOVA test, and draw a boxplot

4. Java Application

The Java application *Network.jar* loads the server API and constructs an undirected graph using the JUNG graphing framework (http://jung.sourceforge.net/). Once the graph is created, it will compute three centrality measures.

- **Betweenness:** Shows how often a node is used as a bridge along the shortest path between two other nodes.
- **Closeness:** The inverse of the average shortest path distance between all other nodes in a graph. In the case of disconnected nodes, the longest shortest path distance of the network is used as an approximation for the distance between the two disconnected nodes.
- **PageRank:** Measures how influential a node is by considering an edge as a vote to a node. The more incoming edges to a node will consider it to be influential, and more influential nodes will have more voting power for other edges.

The centralities are computed for each year and are normalized between [0...1]. Once computed, it will create a csv file for each type of node for each year containing the name of the node followed by all the meta data and the three centralities. These files will be located in the *output/data* directory. In addition, cached json files of the server response will be stored under *cache/*.

5. R Statistics

R is used to perform several statistical tests and analyses on the centrality metrics and meta data generated in the csv files by JUNG and the server API. Make sure that you have run **make** before running the statistics since it will pull in some R library dependencies. Also make sure that you have imagemagick installed (with the mogrify tool), as this is used to finalize the charts generated by R.

5.1. Output

All output from running the statistics will be found in *output/* and its sub directories. An *index.html* file is created in the *output/* folder when running the script. This html file contains all the results for the tests, as well as the charts associated with them.

- **Correlation Charts:** Every numerical field in the data set will have a correlation test done with the field and the centrality values over time. This can be useful to see which centrality measure is most correlated with each field, and can help to determine which measure to use in other tests involving that field.
- **Time Based Tests:** Time based tests (that is, tests which have “Time” as the **x** variable) will generate boxplots. These tests can be useful to see whether or not there is an increasing or decreasing trend of a field over time.
- **Distribution Charts:** Every numeric field in the data set will have a histogram and kernel distribution chart generated. Only the transformed distributions are shown. This can be a good way to verify that the distributions fit a normal distribution.
- **Tests:** The tests specified in the **tests** field in the configuration file test the correlation between two fields. Depending on the type of test done, different types of charts will be generated:
  - **scatterplot (default):** Outputs a scatterplot of the two fields, with a mean squares line. This is useful when the **x** variable is continuous.
  - **boxplot:** Outputs a boxplot of the two fields. The black line in the box represents the median. The box represents the Q1 and Q3 percentiles, while the dotted whisker represents `Q1-1.5*IQR` and `Q3+1.5*IQR`. Any point which falls outside of this range will be drawn as an outlier. The outliers will also be mentioned in the html file. A boxplot should be used whenever the **x** variable is discrete.
  - **ttest:** Outputs a boxplot showing **x** and **y** vectors. This test should be used whenever a comparison between two different means needs to be done.
  - **anova:** Outputs a boxplot between a **y** and groups **x**. The **x** variable should be a group or a discrete variable with not too many unique values (no more than 30). While it is possible to test more, it may be more difficult to interpret. The html file will contain the ANOVA summary table, and the list of intervals and outliers.