RELISON

Release 1.0.0

Javier Sanz-Cruzado, Pablo Castells

Dec 15, 2022

GET STARTED

RELISON is a comprehensive framework for social network analysis and recommendation in Java. This framework allows the analysis of multiple structural properties of a social network graph (from simple ones like the nodes degree or the distance between nodes to more complex ones like betweenness or random walk probabilities), detect the communities in a social network, predict and recommend which links are more likely to appear in the network in the future, and analyze the flow of information that travels through the network.

RELISON has an special focus on the link recommendation (also known as contact recommendation) problem in social networks, i.e. recommending those people in a social network with whom a given user might be interested to connect. Among its main goals, RELISON aims to provide efficient tools for executing and evaluating such approaches, considering not only their accuracy, but also other aspects like their novelty, diversity, and the effects such recommendations have on global properties of the networks (as changes in the structural properties or in the speed or diversity of the information diffusion).

CHAPTER

ONE

INTRODUCTION

RELISON is a comprehensive framework for social network analysis and recommendation in Java. This framework allows the analysis of multiple structural properties of a social network graph (from simple ones like the nodes degree or the distance between nodes to more complex ones like betweenness or random walk probabilities), detect the communities in a social network, predict and recommend which links are more likely to appear in the network in the future, and analyze the flow of information that travels through the network.

RELISON has an special focus on the link recommendation (also known as contact recommendation) problem in social networks, i.e. recommending those people in a social network with whom a given user might be interested to connect. Among its main goals, RELISON aims to provide efficient tools for executing and evaluating such approaches, considering not only their accuracy, but also other aspects like their novelty, diversity, and the effects such recommendations have on global properties of the networks (as changes in the structural properties or in the speed or diversity of the information diffusion).

These framework is divided in 6 different components:

- **RELISON-core:** Basic graph definitions and generators.
- **RELISON-sna:** Social network analysis metrics and community detection.
- **RELISON-content:** Classes and definitions for user-generated contents in social networks.
- **RELISON-linkpred:** Link prediction and contact recommendation functionalities.
- **RELISON-diffusion:** Simulation of information diffusion dynamics.
- **RELISON-examples:** Examples showcasing the functionality of the library.

CHAPTER

TWO

INSTALL RELISON

RELISON is a library working with the following operating systems:

- Linux
- Windows 10

It requires Java version 14 or later.

2.1 Install from source

There are many ways you can install and use RELISON on your system. We provide information here on how to do this:

- *Executable*
- *Docker*
- *Maven*
- *Java*

2.1.1 Executable

If you want to use the default command line programs, we recommend to download the JAR file containing all the dependencies. This file is available from the following link:

https://github.com/ir-uam/RELISON/releases/download/v1.0.0-maven/relison.jar

You can download it with the following command:

```
curl -L https://github.com/ir-uam/RELISON/releases/download/v1.0.0-maven/relison.jar --
↪output relison.jar
```

Then, you can use any of the available programs by executing the following line on your terminal:

```
java [VM_OPTIONS] -jar relison.jar PROGRAM_CODE [arguments]
```

where

- [VM_OPTIONS] represent the list of configuration parameters for the Java virtual machine.
- PROGRAM_CODE is the name of the program to execute.
- [arguments] are the list of arguments of program.

2.1.2 Docker

A Dockerized version of the command line RELISON program is available on Docker Hub. In order to run a container, execute the following two docker commands:

```
docker pull javiersanzcruza/relison:latest
docker run -p 8888:8888 javiersanzcruza/relison:latest
```

This will create a Docker container with a Jupyter notebook acessible from http://localhost:8888. The executable .jar for RELISON will be available on the /tmp/notebooks/RELISON/ directory.

2.1.3 Maven

The recommended way to use RELISON is through Maven. This will allow you to automatically download all the necessary dependencies to work with this framework. There are two ways to do this:

Maven Central

A first option allows to download the project from Maven Central. In order to add the library to your project, just add the following dependency to the .pom file.

```
<dependency>
  <groupId>io.github.ir-uam</groupId>
  <artifactId>RELISON-[module-name]</artifactId>
  <version>1.0.0</version>
</dependency>
```

where [module-name] indicates the name of the module you want to import.

2.1.4 Java

In case you do not want to use Maven, you just need to compile the library into JAR files, or obtain them from Maven Central: https://search.maven.org/search?q=RELISON%20io.github.ir-uam

In addition, you might need to obtain the following libraries (and their dependencies):

- FastUtil v.8.5.2. (https://fastutil.di.unimi.it/)
- Matrix Toolkits Java v.1.0.4. (https://github.com/fommil/matrix-toolkits-java)
- **RankSys v.0.4.3. (http://ranksys.github.io/)**
  - RankSys-core
  - RankSys-fast
  - RankSys-formats
  - RankSys-rec
  - RankSys-nn
  - RankSys-novdiv
  - RankSys-novelty
  - RankSys-mf

- Jung v.2.1.1. (http://jung.sourceforge.net/)
- **Apache Lucene v.8.4.1. (https://lucene.apache.org/)**
  - Lucene-core
- **Terrier v.5.1. (http://terrier.org/)**
  - terrier-core
  - terrier-realtime
  - terrier-learning
- Weka v.3.6.6. (https://www.cs.waikato.ac.nz/ml/weka/)
- Cloning v.1.9.2. (https://mvnrepository.com/artifact/uk.com.robust-it/cloning/1.9.2)
- Yaml Beans v.1.06 (https://github.com/EsotericSoftware/yamlbeans)
- JUnit v. 4.13.1.

CHAPTER

THREE

LICENSE

CHAPTER

FOUR

CONTACT

Questions? Please contact

- Javier Sanz-Cruzado (javier.sanz-cruzadopuig@glasgow.ac.uk)
- Pablo Castells (pablo.castells@uam.es)

CHAPTER

FIVE

NETWORKS

The RELISON library allows the creation, reading, writing and manipulation of networks. Mathematically, a network is modelled as a graph, $G = \langle \mathcal{U}, E \rangle$, where $\mathcal{U}$ is the set of users in the network, and $E \in \mathcal{U}^2$ is the set of links in the network. For each user $u \in \mathcal{U}$, we define his neighborhood, $\Gamma(u)$ as the set of people in the network sharing a link with him.

In order to use the basic graph structures, the following package must be included in the pom.xml file in your Maven project.

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-core</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

SIX

PROPERTIES OF A GRAPH

The RELISON library allows different types of network, depending on its properties. We have to select three options:

Number of edges between users

The first option to consider in a network is the number of edges we allow between the same pair of users. If we consider many of them, we are talking about *multigraphs*, whereas, if we just consider one, we are using a *simple* graph.

Direction of the edges

The second consideration defines whether all the edges are reciprocal or not. If they are, there is no difference between the `(u,v)` and `(v,u)`, and we are talking about an *undirected* network. This is the case of the Facebook friendship network, where a link has to be accepted by both users before it is created. In case the direction of the edges matters, we are talking about a *directed* network, as the Twitter follows network.

Direction of the edges

The third and final consideration is referred to the properties of the edges. Sometimes, it is possible to define a function w, which assigns a weight to each of the edges in the network. This weight might define very different properties: the frequency of interaction between the users, the cost of travelling from one node to another, etc. If the network considers these weights, we consider it a *weighted* network, while an *unweighed* network just takes binary weights (weight equal to 1 if the edge exists, 0 otherwise).

In the table below, we include the basic interfaces for defining any of those graphs, depending on the properties of the network.

Table 1: Basic classes for the definition of network graphs

Multigraph	Directed	Weighted	Class
			UndirectedUnweightedGraph
		✓	UndirectedWeightedGraph
	✓		DirectedUnweightedGraph
	✓	✓	DirectedWeightedGraph
✓			UndirectedUnweightedMultiGraph
✓		✓	UndirectedWeightedMultiGraph
✓	✓		DirectedUnweightedMultiGraph
✓	✓	✓	DirectedWeightedMultiGraph

CHAPTER

SEVEN

CREATION OF EMPTY GRAPHS

When integrated in another Java library, the RELISON library provides two options for generating empty graphs.

The first option considers the straightforward use of the class constructors. RELISON provides an efficient implementation for each of the network types described above. The obtain this implementation, it is just necessary to add the prefix `Fast` to the interface names described earlier. No additional argument needs to be provided to the constructors. For instance, if we want to create a directed unweighted and simple graph, we would just create it as:

```
Graph<U> graph = new FastDirectedUnweightedGraph();
```

The second option uses graph generators: the `EmptyGraphGenerator` class builds empty simple graphs, whereas the `EmptyMultiGraphGenerator` class does the same for multigraphs. The constructor has to be configured with two parameters, indicating whether the graph is directed and/or weighted. Following the previous example, we would do the following:

```
Graph<U> graph;
boolean directed = true;
boolean weighted = false;

GraphGenerator<U> ggen = new EmptyGraphGenerator();
ggen.configure(directed, weighted);
graph = ggen.generate();
```

CHAPTER

EIGHT

READING / WRITING GRAPHS

Most times, the networks which shall be used in the library are stored in an external file. RELISON provides classes for reading this type of networks. When we want to read the network, we use a `GraphReader`. This works as follows:

```
String file = "file.txt"; // the route of the file
boolean readWeights = true; // if we want to read the weights.
boolean readTypes = false; // if we want to read the types
GraphReader<U> greader; // we assume here that it has been configured
Graph<U> graph = greader.read(file, readWeights, readTypes);
```

To write a graph into a file, we use a `GraphWriter`. Once it is created, this works as follows:

```
String file = "file.txt"; // the route of the file
Graph<U> graph; // the graph to write
GraphWriter<U> gwriter; // the graph we want to write
boolean writeWeights = true; // if we want to write the edge weights.
boolean writeTypes = false; // if we want to write the edge types
gwriter.write(graph, file, writeWeights, writeTypes);
```

8.1 Basic format

The basic format prints, on each line, an edge of the network. The format of a line is the following (separated by a delimiter):

```
node1 node2 (weight edgeType)
```

where `node1` and `node2` are the identifiers of the nodes, `weight` is the weight value (a double value) and `type` contains an integer value classifying the edge. The last values are optional. The weight should only be provided when we want to read the weights of the network, and the types must be provided when the user wants to read them.

> For instance, let's suppose that we have a Twitter network with 10,000 users and 100,000 edges. The first two users have, as nicknames, "JavierSanzCruza" and "pcastells", respectively, and the first follows the second. Then, if we use "," as delimiters the file would look as:

```
JavierSanzCruza,pcastells,1.0
<...>
```

If we want to read graphs from this format, we use the `TextGraphReader` and `TextMultiGraphReader` classes, depending on whether we want to read a simple network or a multigraph. These classes receive the following arguments:

- `directed`: true if the network is directed, false otherwise.

- `weighted`: true if the network is weighted, false otherwise.
- `selfloops`: true if we want to read edges from a node to itself, false otherwise.
- `delimiter`: the delimiter separating the different fields in the file. In the main programs of RELISON, it is a tab.
- `uParser`: a parser for reading the type of the nodes from text.

When we want to write graphs into this format, we use the `TextGraphWriter` class, which just receives the delimiter in the constructor.

We wanted to note here that this is the format available in the programs provided by the RELISON library, with the fields separated by tabs (the `t` character).

8.2 Pajek format

This format allows reading and writing networks in the Pajek format (more information in the following link). These graphs have the following format (space separated):

```
*Vertices numVertices
vertexId1 "vertexLabel1"
vertexId2 "vertexLabel2"
<...>
vertexIdN "vertexLabelN"
*Edges numEdges
vertexId1.1. vertexId1.2. weight1
<...>
```

Here, for each node, we differentiate two value: the `vertexId` is a numerical value identifying the user, and the `vertexLabel` is the actual identifier of the user. For instance, let's suppose that we have a Twitter network with 10,000 users and 100,000 edges. The first two users have, as nicknames, "JavierSanzCruza" and "pcastells", respectively, and the first follows the second. Then, the Pajek file would be the following: .. code:

```
*Vertices 10000
1 "JavierSanzCruza"
2 "pcastells"
<...>
*Edges 100000
1 2 1.0
<...>
```

If we want to read graphs from this format, we use the `PajekGraphReader` class. This class receives the following arguments:

- `multigraph`: true if the network is modelled after a multigraph, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if the network is weighted, false otherwise.
- `selfloops`: true if we want to read edges from a node to itself, false otherwise.
- `uParser`: a parser for reading the type of the nodes from text.

When we want to write graphs into this format, we use the `PajekGraphWriter` class, which does not receive any argument in its constructor.

Differently from the basic format, this one does not allow reading the types of the edges.

CHAPTER

NINE

GRAPH MANIPULATION

The RELISON library provides methods for the manipulation of the network (adding nodes, edges, changing the weights of edges, etc.). All this methods are provided in the `Graph` interface, but we summarize them here.

9.1 Nodes

The simplest way to modify a network is to add or remove one of its edges.

9.1.1 Add nodes

If we want to add a node to the network, we use the following method:

```
boolean addNode(U user)
```

Arguments:

- *user*: the identifier of the user.

Returns

- If the node is added, it returns true. Otherwise, it returns false. A user can only be added once, so, if a node is added twice, the second time, this method will return false.

9.1.2 Remove nodes

If we want to remove a node from the network, we use:

```
boolean removeNode(U user)
```

Arguments:

- *user*: the identifier of the user.

Returns

- If the node is removed, it returns true. Otherwise, it returns false. If the user does not exist in the network, this method will return false.

9.2 Edges

The second group of elements that we can modify in a network is the group of edges in the network. In this case, we have several methods of interest.

9.2.1 Add edges

To add edges, we can consider several options. We include here the most complete one, although more of them can be seen on the reference of the `Graph` interface, here.

```
boolean addEdge(U orig, U dest, double weight, int type, boolean insertNodes)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `weight`: the weight of the edge.
- `type`: the type of the edge.
- `insertNodes`: true if we want to add the nodes to the network if they do not exist, false otherwise.

Returns

- If the edge is added, it returns true. Otherwise, it returns false. In simple networks, an edge can only be added once.

9.2.2 Update edge weights

If we want to modify the weight of an edge, we have to use the following methods. In simple networks, we have to use:

```
boolean updateEdgeWeight(U orig, U dest, double newWeight)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `newWeight`: the new weight of the edge.

Returns

- If the edge weight is updated, it returns true. If the edge does not exist, it cannot be updated.

In multigraphs, we use the following method instead (the previous one just updates the first created edge between the users):

```
boolean updateEdgeWeight(U orig, U dest, double newWeight, int idx)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `newWeight`: the new weight of the edge.
- `idx`: the number of the edge between the users to modify.

Returns

- If the edge weight is updated, it returns true. If the edge does not exist, it cannot be updated.

9.2.3 Update edge types

If we want to modify the type of an edge, we have to use the following methods. In simple networks, we have to use:

```
boolean updateEdgeType(U orig, U dest, int newType)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `newType`: the new type of the edge.

Returns

- If the edge weight is updated, it returns true. If the edge does not exist, it cannot be updated.

In multigraphs, we use the following method instead (the previous one just updates the first created edge between the users):

```
boolean updateEdgeType(U orig, U dest, double newType, int idx)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `newType`: the new type of the edge.
- `idx`: the number of the edge between the users to modify.

Returns

- If the edge type is updated, it returns true. If the edge does not exist, it cannot be updated.

9.2.4 Remove edges

Finally, if we want to remove an edge, we have to use the following method:

```
boolean removeEdge(U orig, U dest)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.

Returns

- If the edge weight is remove, it returns true. If the edge does not exist, it cannot be removed.

In multigraphs, we use the following method instead (the previous one just removes the first created edge between the users):

```
boolean removeEdge(U orig, U dest, int idx)
```

Arguments:

- `orig`: the first node of the edge.
- `dest`: the second node of the edge.
- `idx`: the number of the edge between the users to remove.

Returns

- If the edge type is updated, it returns true. If the edge does not exist, it cannot be updated.

Also, we provide a method to remove all the edges between two nodes in the multigraph:

```
boolean removeEdges(U orig, U dest)
```

Arguments:

- `orig`: the first node of the edges.
- `dest`: the second node of the edges.

Returns

- If the edge weight is remove, it returns true. If there are not edges between the users, they cannot be removed.

CHAPTER

TEN

ACCESSING THE PROPERTIES OF A NETWORK

In addition to the methods for modifying the structure of a social network, RELISON also provides methods for accessing to the basic information of a network: the nodes in the network, his neighbors, the weights and types of the edges, etc. Here, we summarize how this can be done.

10.1 Users in the network

The first element we explain how to access are the nodes in the network. In order to access the complete set of nodes, we can use the following method:

```
Stream<U> getAllNodes()
```

Returns

- An stream object containing the identifiers of all the nodes in the network.

If, instead of accessing to them, we just need to count them, we can use the following method of the graphs:

```
long getVertexCount()
```

Returns

- The number of vertices (nodes, users) in the network graph.

10.2 Edge properties

The next element we can access are the edges. If we know the endpoints of the edge, we can access their different properties. We differentiate four methods here:

10.2.1 Edge existence

The first method allows us to know if there is a link between two users. The method signature is:

```
boolean containsEdge(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- true if there is (at least) an edge, false otherwise.

In multigraphs, there is another method which allows us to obtain the number of edges between a pair of users:

```
int getNumEdges(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- the number of edges between the two nodes.

10.2.2 Edge weight

We might want to access the weight of an edge. For this, in simple graphs, we use the following method:

```
double getEdgeWeight(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- the weight if it exists, `NaN` otherwise.

In the case of multigraphs, the previous method allows us to obtain the sum of all the edge weights. If we want to obtain the individual weights, we have another method:

```
List<Double> getEdgeWeights(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- a list containing all the edge weights if there is (at least) one edge between the users, null otherwise.

10.2.3 Edge types

We might want to access the type of an edge. For this, in simple graphs, we use the following method:

```
int getEdgeWeight(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- the type if it exists, -1 otherwise.

In the case of multigraphs, we have another method, which allows us to retrieve the weights of all the edges:

```
List<Integer> getEdgeTypes(U orig, U dest)
```

Arguments

- `orig`: the first node.
- `dest`: the second node.

Returns

- a list containing all the edge types if there is (at least) one edge between the users, null otherwise.

10.2.4 Edge number

The last method just allows us to determine how many edges our network has:

```
long getEdgeCount()
```

Returns

- the number of edges in the network.

10.3 Neighbors of a node

As accessing the edges in the network just by running over all the possible pairs of users and checking if the edge exists would be very slow, RELISON provides methods for accessing the neighborhood of a user. We explain them here.

10.3.1 Existence of neighbors

The first group of methods study whether a node has (or not) neighbors in the network. Although there are several methods for this, here we only explain the following one:

```
boolean hasNeighbors(U u, EdgeOrientation orient)
```

Arguments

- `u`: the node to study.
- `orient`: the orientation for selecting the neighbors (only useful in directed networks).
  - `IN`: for identifying if the user has incoming neighbors.
  - `OUT`: for identifying if the user has outgoing neighbors.
  - `UND`: for identifying if the user has either incoming or outgoing neighbors.
  - `MUTUAL`: for identifying if the user has neighbors who are both incoming and outgoing.

Returns

- true if the node has neighbors, false otherwise.

10.3.2 Count

The second group of methods allows us to identify how many neighbors a user has.

```
int getNeighbourhoodSize(U u, EdgeOrientation orient)
```

Arguments

- `u`: the node to study.
- `orient`: the orientation for selecting the neighbors (only useful in directed networks).
  - `IN`: for retrieving the incoming neighbors.
  - `OUT`: for retrieving the outgoing neighbors.
  - `UND`: for retrieving either incoming or outgoing neighbors.
  - `MUTUAL`: for retrieving the neighbors who are both incoming and outgoing.

Returns

- the number of neighbors of the user.

There is another method, that allows us to study the number of edges connected to a node. This value is the same as the previous one in the case of simple networks, but it changes when it comes to multigraphs:

```
int degree(U u, EdgeOrientation orient)
```

Arguments

- `u`: the node to study.
- `orient`: the orientation for selecting the neighbors (only useful in directed networks).
  - `IN`: for retrieving the incoming neighbors.
  - `OUT`: for retrieving the outgoing neighbors.
  - `UND`: for retrieving either incoming or outgoing neighbors.
  - `MUTUAL`: for retrieving the neighbors who are both incoming and outgoing.

Returns

- the number of edges connected to the user (the degree), -1 if the user does not exist.

10.3.3 Neighbors

The third group of methods allows us to identify the actual neighbors of a user in the network. The method we introduce here is the following:

```
Stream<U> get(U u, EdgeOrientation orient)
```

Arguments

- `u`: the node to study.
- `orient`: the orientation for selecting the neighbors (only useful in directed networks).
  - `IN`: for retrieving the incoming neighbors.
  - `OUT`: for retrieving the outgoing neighbors.
  - `UND`: for retrieving either incoming or outgoing neighbors.

  - MUTUAL: for retrieving the neighbors who are both incoming and outgoing.

Returns

- an stream containing the identifier of the networks.

10.3.4 Weights

The fourth group of methods allows us to identify the weights of the nodes. We can use the following method:

```
Stream<Weight<U, Double>> getNeighbourhoodWeights(U u, EdgeOrientation orient)
```

Arguments

- u: the node to study.
- orient: the orientation for selecting the neighbors (only useful in directed networks).
  - IN: for retrieving the incoming neighbors.
  - OUT: for retrieving the outgoing neighbors.
  - UND: for retrieving either incoming or outgoing neighbors.
  - MUTUAL: for retrieving the neighbors who are both incoming and outgoing.

Returns

- an stream containing (user, weight) pairs. In multigraphs, a pair is included for each edge.

Additionally, in multigraphs, we can also use the following method:

```
Stream<Weight<U, Double>> getNeighbourhoodWeightsLists(U u, EdgeOrientation orient)
```

Arguments

- u: the node to study.
- orient: the orientation for selecting the neighbors (only useful in directed networks).
  - IN: for retrieving the incoming neighbors.
  - OUT: for retrieving the outgoing neighbors.
  - UND: for retrieving either incoming or outgoing neighbors.
  - MUTUAL: for retrieving the neighbors who are both incoming and outgoing.

Returns

- an stream containing (user, list of weights) pairs.

10.3.5 Edge types

The fifth and last group of methods allows us to identify the weights of the nodes. We can use the following method:

```
Stream<Weight<U, Integer>> getNeighbourhoodTypes(U u, EdgeOrientation orient)
```

Arguments

- u: the node to study.
- orient: the orientation for selecting the neighbors (only useful in directed networks).

  - `IN`: for retrieving the incoming neighbors.
  - `OUT`: for retrieving the outgoing neighbors.
  - `UND`: for retrieving either incoming or outgoing neighbors.
  - `MUTUAL`: for retrieving the neighbors who are both incoming and outgoing.

Returns

- an stream containing (user, type) pairs. In multigraphs, a pair is included for each edge.

Additionally, in multigraphs, we can also use the following method:

```
Stream<Weight<U, Integer>> getNeighbourhoodTypesLists(U u, EdgeOrientation orient)
```

Arguments

- `u`: the node to study.
- `orient`: the orientation for selecting the neighbors (only useful in directed networks).
  - `IN`: for retrieving the incoming neighbors.
  - `OUT`: for retrieving the outgoing neighbors.
  - `UND`: for retrieving either incoming or outgoing neighbors.
  - `MUTUAL`: for retrieving the neighbors who are both incoming and outgoing.

Returns

- an stream containing (user, list of types) pairs.

10.4 Adjacency matrix

In addition to the graph object, we can represent the network using the adjacency matrix of the network. This matrix has in the (u, v) coordinate, the weight of the edge between nodes u and v. We can obtain this matrix using the following method:

```
double[][] getAdjacencyMatrix(EdgeOrientation orient)
```

Arguments

- `orient`: the orientation for selecting the matrix (only useful in directed networks).
  - `IN`: the (u, v) coordinate contains the weight of the (v, u) weight.
  - `OUT`: the (u, v) coordinate contains the weight of the (u, v) weight (natural adjacency matrix)
  - `UND`: the (u, v) coordinate contains the $w(u, v) + w(v, u)$.
  - `MUTUAL`: the (u, v) coordinate contains the $w(u, v) + w(v, u)$ value if both edges exist, 0 otherwise.

Returns

- an array containing the matrix.

We should note that the nodes in the network can be represented by any type of identifier (int, long, string, etc.). So, in order to map these identifiers to the indexes in the matrix, we can use the following method:

```
Index<U> getAdjacencyMatrixMap()
```

Returns

- an index, mapping user identifiers to indexes in the (0, numNodes-1) rank.

CHAPTER

ELEVEN

SOCIAL NETWORK ANALYSIS

RELISON provides functionality for analyzing the structural properties of the social network graphs: vertex, links, communities and graph properties can be easily computed thanks to our framework. We provide several ways to determine the topological properties of networks: with a program which straightforwardly allows you to compute any of the included social network properties in the framework by reading a configuration file, and by integrating the framework in your own library.

In this section, we provide some guide on how to do this.

The social network metrics are included in the SNA module of the framework, which can be imported into any Java library using Maven as follows:

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-sna</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

TWELVE

EXPERIMENTAL CONFIGURATION

In our framework, we include a program that simplifies measuring the structural properties of a given network. This program can be executed as follows, once the binary for the library has been generated:

```
java -jar RELISON.jar sna network multigraph directed weighted selfloops metrics output␣
↪(-communities comm1,comm2,...,commN --distances)
```

where

- `network`: a file containing the social network graph to analyze.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `metrics`: a configuration file for reading the structural properties we want to measure (see *Configuration file* below).
- :code:`output`: a directory in which to store the structural properties.
- **Optional parameters:**
  - `-communities comm1,comm2,...,commN`: a comma-separated list of files containing community partitions of the users.
  - `--distances`: indicates that we want to pre-compute the distance-based metrics in the network (recommended if more than one is used).

12.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different properties we want to measure and analyze. This is a Yaml file with the following format:

```
metrics:
  metric_name1:
    type: vertex/edge/pair/graph/indiv. community/global community
      params:
        param_name1:
          type: int/double/boolean/string/long/orientation/object
          values: [value1,value2,...,valueN] / value
          range:
```

(continues on next page)

(continued from previous page)

```
        - start: startingValue
          end: endingValue
          step: stepValue
        - start: ...
        objects:
          name_of_the_object:
            param_name1:
              type: int/double/boolean/string/long/orientation/object
              ...
            param_name2:
              type: int/double/boolean/string/long/orientation/object
              ...
metric_name2:
  ...
```

In this configuration file, we identify each metric by each name, and, afterwards, we identify its type. We differentiate between six groups of metrics:

- **Vertex metrics:** Properties of individual nodes in the network (e.g. degree, local clustering coefficient).
- **Edge/Pair metrics:** Properties of pairs of users in the network. If they are selected with the "edge" identifier, the metrics are only computed over the set of links in the network.
- **Graph metrics:** Global properties of the network (e.g. global clustering coefficient).
- **Individual community metrics:** Properties of a single community in the partition (e.g. community size, degree).
- **Global community metrics:** Global metrics depending on the community partition (e.g. modularity).

CHAPTER

THIRTEEN

INTEGRATE STRUCTURAL METRICS IN A JAVA PROJECT

By importing the SNA package, it is possible to use different structural metrics in your project, as well as provide novel metrics which can be used within the framework. In this section, we summarize the main methods and functionalities for each of the metric types.

- *Vertex metrics*
- *Edge or pair metrics*
- *Graph metrics*
- *Individual community metrics*
- *Global community metrics*

13.1 Vertex metrics

Vertex metrics allow the computation of metrics for each individual user in the network: properties like degree, local clustering coefficient, etc. All vertex metrics inherit from the `VertexMetric` interface. This metric interface has the following methods, which have to be implemented to develop new node metrics:

```
double compute(Graph<U> graph, U user)
```

This method just finds the metric value for a single node in the network.

Arguments:

- *graph*: the network we want to study.
- *user*: the identifier of the user.

Returns

- the metric value for the given user in the network.

It receives the graph we want to compute the metric for, and the user identifier.

```
Map<U, Double> compute(Graph<U> graph)
```

This method is useful for computing distributions, as it returns the metric value for each single user in the network. Such values are stored in a map, indexed by the user identifier.

Arguments:

- *graph*: the network we want to study.

Returns

- a map, indexed by user identifier, containing the metric values for all the nodes in the network.

```
Map<U, Double> compute(Graph<U> graph, Stream<U> users)
```

This method is similar to the previous one, but it limits the metric to the users in the provided stream. Again, it returns a map, indexed by the user identifier.

Arguments:

- *graph*: the network we want to study.
- *users*: an stream containing the identifiers of a set of users.

Returns

- a map, indexed by user identifier, containing the metric values for all the users in the stream.

```
double averageValue(Graph<U> graph)
```

This method averages the metric over the whole set of users in the network.

Arguments:

- *graph*: the network we want to study.

Returns

- the average value of the metric.

```
double averageValue(Graph<U> graph, Stream<U> users)
```

This method averages the metric over an specified set of users.

Arguments:

- *graph*: the network we want to study.
- *users*: an stream containing the identifiers of a set of users.

Returns

- the average value of the metric for the provided users.

13.2 Edge or pair metrics

This family of metrics takes different pairs of users in the network, regardless of whether a link between them exist or not. Examples of these metrics include distance between users or embeddedness. All these metrics inherit the `PairMetric` interface, which has the following methods:

```
double compute(Graph<U> graph, U u, U v)
```

This method just finds the metric value for a single pair of users in the network.

Arguments:

- *graph*: the network we want to study.
- *u*: the first user in the pair.
- *v*: the second user in the pair.

Returns

- the metric value for the given pair of users in the network.

It receives the graph we want to compute the metric for, and the user identifier.

```
Map<Pair<U>, Double> compute(Graph<U> graph)
```

This method is useful for computing distributions, as it returns the metric value for each pair of users in the network. Such values are stored in a map, indexed by the pair of user identifiers.

Arguments:

- *graph*: the network we want to study.

Returns

- a map, indexed by a pair of user identifiers, containing the metric values for all the pairs in the network.

```
Map<U, Double> computeOnlyLinks(Graph<U> graph)
```

This method is similar to the previous one, but it limits the metric to the set of links in the network.

Arguments:

- *graph*: the network we want to study.

Returns

- a map, indexed by a pair of user identifiers, containing the metric values for all the pairs in the network.

```
Map<Pair<U>, Double> compute(Graph<U> graph, Stream<Pair<U>> pairs)
```

This method is similar limits the metric a provided set of user pairs.

Arguments:

- *graph*: the network we want to study.
- *pairs*: an stream containing the pairs of user identifiers.

Returns

- a map, indexed by a pair of user identifiers, containing the metric values for all the user pairs in the stream.

```
Function<U, Double> computeOrig(Graph<U> graph, U orig)
```

This method generates a function for obtaining the values of all pairs which have, as first node, a provided one.

Arguments:

- *graph*: the network we want to study.
- *orig*: the first node of the pairs.

Returns

- a function that allows to obtain the values for all pairs which have *orig* as first node.

```
Function<U, Double> computeDest(Graph<U> graph, U dest)
```

This method generates a function for obtaining the values of all pairs which have, as second node, a provided one.

Arguments:

- *graph*: the network we want to study.
- *dest*: the second node of the pairs.

Returns

- a function that allows to obtain the values for all pairs which have *orig* as first node.

```
double averageValue(Graph<U> graph)
```

This method averages the metric over all the possible pairs of users in the network.

Arguments:

- *graph*: the network we want to study.

Returns

- the average value of the metric.

```
double averageValueOnlyLinks(Graph<U> graph)
```

This method averages the metric over all the links in the network.

Arguments:

- *graph*: the network we want to study.

Returns

- the average value of the metric.

```
double averageValue(Graph<U> graph, Stream<Pair<U>> pairs)
```

This method averages the metric over an specified set of user pairs.

Arguments:

- *graph*: the network we want to study.
- *pairs*: an stream containing the pairs of user identifiers to consider.

Returns

- the average value of the metric for the provided user pairs.

13.3 Graph metrics

This family of metrics analyzes structural properties of the whole network. They inherit from the `GraphMetric` interface, which has the following methods:

```
double compute(Graph<U> graph)
```

This method finds the value of the structural metric for the given network.

Arguments:

- *graph*: the network we want to study.

Returns

- the value of the metric.

13.4 Individual community metrics

Given a community partition of the network, this family of metrics allows the analysis of the properties of the different communities. They inherit from the `IndividualCommunityMetric` interface, which has the following methods:

```
double compute(Graph<U> graph, Communities<U> comms, int indiv)
```

This method computes the metric for an individual community.

Arguments:

- *graph*: the network we want to study.
- *comms*: the community partition.
- *indiv*: the identifier of the community we want to study.

Returns

- the value of the metric for the given community.

```
Map<Integer, Double> compute(Graph<U> graph, Communities<U> comms)
```

This method computes the metric for all the communities in a partition. It returns the value in a map indexed by community identifier.

Arguments:

- *graph*: the network we want to study.
- *comms*: the community partition.

Returns

- a map, indexed by community identifier, containing the value of the metric for each community.

```
double averageValue(Graph<U> graph, Communities<U> comms)
```

This method computes the average value of the metrics over the set of communities.

Arguments:

- *graph*: the network we want to study.
- *comms*: the community partition.

Returns

- the average value of the metric.

13.5 Global community metrics

Given a community partition of the network, this family of metrics allows the analysis of the properties of the whole network which consider the community partition of the graph. They inherit from the `CommunityMetric` interface, which has the following methods:

```
double compute(Graph<U> graph, Communities<U> comms)
```

This method finds the value of the structural metric for the given network.

Arguments:

- *graph*: the network we want to study.
- *comms*: the community partition.

Returns

- the value of the metric.

CHAPTER

FOURTEEN

STRUCTURAL METRICS SUMMARY

RELISON provides the following metrics:

14.1 Vertex metrics

RELISON integrates the following vertex metrics.

- *Betweenness*
- *Closeness*
- *Coreness*
- *Degree*
- *Eccentricity*
- *Eigenvector centrality*
- *Harmonic centrality*
- *HITS*
- *Katz centrality*
- *Local clustering coefficient*
- *PageRank*
- *Reciprocity rate*
- *Length*

14.1.1 Betweenness

The betweenness centrality measures, for each node, the number of shortest paths which pass through the vertex.

$$\text{Betweenness}(u) = \sum_{(v,w):u \neq v,w} \frac{\sigma_{vw}(u)}{\sigma_{vw}}$$

where σ_{vw} is the number of shortest paths between v and w, and $\sigma_{vw}(u)$ is the number of shortest paths which pass through node u.

Reference: M.E.J. Newman, M. Girvan. Finding and evaluating community structure in networks. Physical Review E 69(2), pp. 1-16 (2004)

Parameters

- `normalize`: true if we want to normalize the coefficient, false otherwise.

Configuration file

```
Betweenness:
  type: vertex
  params:
    normalize:
      type: boolean
      values: [true, false]
```

14.1.2 Closeness

This metric computes how close the studied user is to the rest of the nodes in the network. It is computed as:

$$\text{Closeness}(u) = \frac{|\mathcal{U}| - 1}{\sum_{v \neq u} d(u, v)}$$

where $d(u, v)$ represents the distance between the two nodes.

If the graph is not fully connected, the value of this metric would always be equal to zero, so we have restricted its computation to the strongly connected component that user u belongs to.

References:

- M.E.J. Newman. Networks: an introduction (2010).
- L.C. Freeman. Centrality in Networks: I. Conceptual clarification, Social Networks 1, 1979, pp.215-239.

Configuration file

```
Closeness:
  type: vertex
```

14.1.3 Coreness

A k-core in a network is the maximal subgraph of the network such that each vertex in the subgraph has, at least, degree equal to k. The coreness of a node is equal to k if the node belongs to the k-core, but not to the $k + 1$-core.

The coreness of a node measures the maximum k-core it belongs to. A k-core is a subgraph of the network

References:

- Seidman, S.B. Network structure and minimum degree. Social Networks 5(3), pp. 269-287 (1983).
- Batagelj, V., Zaversnik. An O(m) Algorithm for Cores Decomposition of networks. arXiv (2003).

Parameters

- `orientation`: selection of the neighborhood of the node we want to use for computing the degrees. In undirected neighbors, the value of the degree does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: for using the in-degree.
  - `OUT`: for using the out-degree.
  - `UND`: for using the degree $\text{degree}(u) = \text{in-degree}(u) + \text{out-degree}(u)$
  - `MUTUAL`: for using the number of reciprocated links.

Configuration file

The configuration file for the degree is:

```
Coreness:
  type: vertex
  params:
    orientation:
        type: orientation
        values: [IN/OUT/UND/MUTUAL]
```

14.1.4 Degree

This metric measures the number of adjacent links of the studied node. It is computed as:

$$\text{degree}(u) = |\Gamma(u)| = |\{(u, v) \in E\}|$$

where $\Gamma(u)$ is the neighborhood of the node. Note that, in multigraphs, a user can appear several times in $\Gamma(u)$.

Related to this measure, we allow two additional measures: the **inverse degree**, which measures the multiplicative inverse of the degree:

$$\text{inv-degree}(u) = \frac{1.0}{\text{degree}(u)}$$

and the degree of the node in the complementary graph (what we call the **complementary degree**):

$$\text{compl-degree}(u) = |\mathcal{U}| - \text{degree}(u)$$

Note that we cannot compute this metric (and its inverse) in multigraphs, since the complementary of such graph is not properly defined.

Parameters

- `orientation`: selection of the neighborhood of the node we want to use for computing the degree. In undirected neighbors, the value of the degree does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: for computing the in-degree.
  - `OUT`: for computing the out-degree.
  - `UND`: for computing the degree $\text{degree}(u) = \text{in-degree}(u) + \text{out-degree}(u)$
  - `MUTUAL`: for just counting the number of reciprocated links.

Configuration file

The configuration file for the degree is:

```
Degree:
  type: vertex
  params:
    orientation:
        type: orientation
        values: [IN/OUT/UND/MUTUAL]
```

for the inverse degree is:

```
Inverse degree:
  type: vertex
  params:
    orientation:
        type: orientation
        values: [IN/OUT/UND/MUTUAL]
```

for the complementary degree:

```
Complementary degree:
  type: vertex
  params:
    orientation:
        type: orientation
        values: [IN/OUT/UND/MUTUAL]
```

and for its inverse

```
Complementary inverse degree:
  type: vertex
  params:
    orientation:
        type: orientation
        values: [IN/OUT/UND/MUTUAL]
```

14.1.5 Eccentricity

The eccentricity of a node measures the maximum (finite) distance between the node and any other vertex in the network.

$$\text{Eccentricity}(u) = \max_{v:d(u,v)<\infty} d(u,v)$$

where $d(u,v)$ represents the distance between the two nodes.

Reference: P. Dankelmann, W. Goddard, C. Swart. The average eccentricity of a graph and its subgraphs. Utilitas Mathematica 65(May), pp. 41-51 (2004)

Configuration file

```
Eccentricity:
  type: vertex
```

14.1.6 Eigenvector centrality

The eigenvector centrality measures the importance of a node based on the importance of its neighbors. The value of the eigenvector centrality is the u-th coordinate of the vector x, where x is the solution to the equation $Ax = \lambda x$, where λ is the largest eigenvalue o the adjacency matrix A. Or, in other words, the eigenvector centrality of a node is the corresponding coordinate of the eigenvector associated to the largest eigenvalue of the adjacency matrix.

Reference: Bonacich, P.F. Power and centrality: A family of measures. American Journal of Sociology 92 (5), pp. 1170-1182 (1987)

Parameters

- `orientation`: selection of the neighborhood of the node we use for determining the adjacency matrix. In undirected neighbors, the adjacency matrix does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: $A_u v = w(v, u)$
  - `OUT`: $A_u v = w(u, v)$
  - `UND`: $A_u v = w(u, v) + w(v, u)$
  - `MUTUAL`: $A_u v = w(u, v) + w(v, u)$, but only if $w(u, v) \cdot w(v, u) > 0$

Configuration file

```
Eigenvector:
  type: vertex
  params:
    orientation:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.1.7 Harmonic centrality

The harmonic centrality can be studied as an alternative definition of *Closeness*. This metric defines the centrality as the harmonic mean of the distances between the node and the rest of the users in the network. It allows infinite distances (as they sum up just 0).

$$\text{Harmonic}(u) = \frac{1}{|\mathcal{U}| - 1} \sum_{v \neq u} \frac{1}{d(u, v)}$$

where $d(u, v)$ represents the distance between the two nodes.

If the graph is not fully connected, the value of this metric would always be equal to zero, so we have restricted its computation to the strongly connected component that user u belongs to.

References:

- M.E.J. Newman. Networks: an introduction (2010).
- L.C. Freeman. Centrality in Networks: I. Conceptual clarification, Social Networks 1, 1979, pp.215-239.

Configuration file

```
Harmonic:
  type: vertex
```

14.1.8 HITS

The Hyperlink-Induced Topic Search (HITS) algorithm determines the importance of the nodes in the network according to a random walk that. This random walk first traverses an outgoing link, and, afterwards, an incoming one. The algorithm gives two scores to each node: an authority score (indicating the value of the node, according to those who point to him), and a hub score (indicating the value of the node according to the pointed nodes). A good hub is a node which points to good authorities, and a good authority is a node which is pointed by good hubs.

The scores are computed, iteratively, as:

$$\text{auth}(u) = \sum_{v \in \Gamma_{out}(u)} \text{hub}(v)$$

$$\text{hub}(u) = \sum_{v \in \Gamma_{in}(v)} \text{auth}(v)$$

where $\Gamma_{in}(v), \Gamma_{out}(v)$ are, respectively, the set of incoming and outgoing neighbors of v. After each iteration, the scores are normalized.

Reference: J.M. Kleinberg. Authoritative sources in a hyperlink environment. Journal of the ACM 46(5), PP. 604-632 (1999).

Parameters

- `mode`: indicates whether we want to compute the authority scores (true) or the hub scores (false)

Configuration file

```
HITS:
  type: vertex
  params:
    mode:
      type: boolean
      values: [true, false]
```

14.1.9 Katz centrality

The Katz centrality of a node estimates the importance of the nodes according to the paths between the node and the rest of vertices in the network. It considers all the possible paths involving the user (not only the shortest ones, but all the possible ones). Scores are computed as:

$$Katz(u) = \sum_{k=1}^{\infty} \sum_{v \in \mathcal{U}} \alpha^k A_{uv}^k$$

where A is the adjacency matrix of the graph. In Katz centrality, the importance of the paths is weighted, so lower distance paths are more important than those at large distances. Parameter :math:alpha is used to determine how important long paths are.

Reference: Katz, L. A new status index derived from sociometric analysis. Psychometrika 18(1), pp. 33-43 (1953).

Parameters

- `orientation`: selection of the neighborhood of the node we use for determining the adjacency matrix. In undirected neighbors, the adjacency matrix does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: $A_u v = w(v, u)$
  - `OUT`: $A_u v = w(u, v)$
  - `UND`: $A_u v = w(u, v) + w(v, u)$
  - `MUTUAL`: $A_u v = w(u, v) + w(v, u)$, but only if $w(u, v) \cdot w(v, u) > 0$
- `alpha`: a dump factor for giving less importance to long paths. $\alpha \in (0, 1)$

Configuration file

```
Katz:
  type: vertex
  params:
    orientation:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    alpha:
      type: double
      range:
      - start: 0.1
        end: 0.99
        step: 0.1
```

14.1.10 Local clustering coefficient

The local clustering coefficient measures, the proportion of neighbors of the user who are connected.

$$CC(u) = \frac{\{(v,w) \in E | v \neq w \wedge v \in \Gamma(u) \wedge w \in \Gamma(u)\}}{|\Gamma(u)|(|\Gamma(u)| - 1)}$$

We also allow computing this metric in the complementary graph (what we call **complementary local clustering coefficient**)

Reference: D.J. Watts, S.H. Strogatz. Collective dynamics of 'small-world' networks. Nature 393(6684), pp. 440-442 (1998).

Parameters

- `vSel`: selection of the orientation for selecting the first neighbor of the user. This allows the following values:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `wSel`: selection of the orientation for selecting the second neighbor of the user. This allows the following values:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

For the original local clustering coefficient metric, the configuration file is:

```
Local clustering coefficient:
  type: vertex
  params:
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    wSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

whereas for the value in the complementary network, it is:

```
Complementary local clustering coefficient:
  type: vertex
  params:
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    wSel:
```

(continues on next page)

(continued from previous page)

```
    type: orientation
    values: [IN/OUT/UND/MUTUAL]
```

14.1.11 PageRank

PageRank is a random walk algorithm designed to estimate the importance of the nodes in a network, according to its graph structure. The value for a node represents the probability that a random walker who traverses the network randomly choosing the links to follow travels through the node.

The importance of the node depends on three factors: a) the number of incoming nodes, b) the importance of those nodes and c) the number of outgoing edges of these nodes. As the node receives more and more attention from other nodes in the network, the importance of the node increases. For each of these incoming edges, the nodes receive an importance score proportional to the importance of the origin. And, finally, the transferred importance is decreased proportionally to the number of outgoing edges of that node.

The PageRank of a node is defined, recursively, as:

$$PageRank(u) = \frac{r}{|\mathcal{U}|} + (1-r) \sum_{w \in \Gamma_{in}(u)} \frac{PageRank(v)}{|\Gamma_{out}(v)|} + \frac{1-r}{|\mathcal{U}|} \sum_{w:|\Gamma_{out}(w)|=0} PageRank(w)$$

where :math:r represents the probability that the random walker chooses to teleport to any node in the network.

Again, we also provide tools to compute the **complementary PageRank**, i.e. the PageRank value of the nodes in the complementary network.

Reference: S. Brin, L. Page. The anatomy of a large-scale hypertextual web search engine. 7th International Conference on World Wide Web (WWW 1998), Brisbane, Australia, pp. 107-117 (1998).

Parameters

- `r`: the teleport probability. $r \in (0, 1)$

Configuration file

The Yaml file for the original PageRank might be as follows:

```
PageRank:
  type: vertex
  params:
    r:
      type: double
      range:
      - start: 0.1
        end: 0.99
        step: 0.1
```

whereas the file for the complementary variant is:

```
Complementary PageRank:
  type: vertex
  params:
    r:
```

(continues on next page)

(continued from previous page)

```
      type: double
      range:
      - start: 0.1
        end: 0.99
        step: 0.1
```

14.1.12 Reciprocity rate

This metric measures the proportion of the edges involving the node which are reciprocal. This measure is only useful for directed networks (it has value equal to 1 for all nodes in undirected ones).

Parameters

- `orientation`: selection of the set of edges to consider:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time. In this case, the metric is always equal to 1.

Configuration file

The Yaml file for the original PageRank might be as follows:

```
Reciprocity rate:
  type: vertex
  params:
    orientation:
      values: [IN/OUT/UND]
```

14.1.13 Length

This metric measures the length of a node, i.e. the sum of the weights of the links between this node and its neighbors:

$$\text{Length}(u) = \sum_{v \in \Gamma(u)} w(u, v)$$

where :math:Gamma(u) represents the neighborhood of user :math:u and :math:w(u,v) is the weight of the corresponding link between :math:u and :math:v.

Parameters

- `orientation`: selection of the set of edges to consider:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The Yaml file for the original PageRank might be as follows:

```
Length:
  type: vertex
  params:
    orientation:
      values: [IN/OUT/UND/MUTUAL]
```

14.2 Pair metrics

RELISON integrates a set of metrics which can be computed for any pair of users. All the metrics which can be applied to links are included here, but they can be computed for any possible set of users.

RELISON integrates the following edge metrics:

- *Betweenness*
- *Clustering coefficient increment*
- *Distance*
- *Distance without link*
- *Embeddedness*
- *Extended neighbor overlap*
- *Geodesics*
- *Neighbor overlap*
- *Reciprocity*
- *Preferential attachment*
- *Shrinking average shortest path length*
- *Shrinking diameter*
- *Weighted neighbor overlap*
- *Weight*

14.2.1 Betweenness

The betweenness centrality measures, for each pair of users, the number of shortest paths which pass through the edge joining the pair of users. If the edge does not exist, it returns 0.

$$\text{Betweenness}(u, v) = \sum_{(w,x)} \frac{\sigma_{wx}(u, v)}{\sigma_{wx}}$$

where σ_{wx} is the number of shortest paths between w and x, and $\sigma_{wx}(u, v)$ is the number of shortest paths which pass through edge (u, v).

Reference: M.E.J. Newman, M. Girvan. Finding and evaluating community structure in networks. Physical Review E 69(2), pp. 1-16 (2004)

Parameters

- `normalize`: true if we want to normalize the coefficient, false otherwise.

Configuration file

```
Betweenness:
  type: edge
  params:
    normalize:
      type: boolean
      values: [true, false]
```

14.2.2 Clustering coefficient increment

This metric computes how much the global clustering coefficient of the network (see **ADDLINK**) would increase if a link between the two users was added to the network. If the link already exists, this value is equal to zero. It just considers the option where we select distance-2 paths as triads.

Reference: J. Sanz-Cruzado. Contact recommendation in social networks: algorithmic models, diversity and network evolution. PhD thesis (2021).

Configuration file

```
Clustering coefficient increment:
  type: pair
```

14.2.3 Distance

The distance measures the minimum number of steps we have to take for travelling between two nodes. It is also known as the **shortest path length**.

In addition to this metric, we also compute the **efficiency** or **reciprocal shortest path length**, which computes the inverse of the distance between two users:

$$\text{RSL}(u, v) = \frac{1}{\delta(u, v)}$$

where $\delta(u, v)$ represents the distance between users u and v.

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Configuration file

For the original distance, this is computed as:

```
Distance:
  type: pair
```

whereas for the reciprocal distance:

```
Reciprocal shortest path length:
  type: pair
```

14.2.4 Distance without link

The distance measures the distance between two users in the network if we removed the edge between them. If the link does not exist, the distance between the users is the usual one.

Configuration file

```
Distance without link:
  type: pair
```

14.2.5 Embeddedness

The embeddedness of pair of users in a network measures the proportion of the neighbors of the two nodes who are common to both of them. It indicates how redundant a link between the two nodes would be in the network. In case the link exists, it can be considered a measure of the strength of the link.

$$\text{Embeddedness}(u, v) = \frac{|\Gamma(u) \setminus \{v\} \cap \Gamma(v) \setminus \{u\}|}{|\Gamma(u) \setminus \{v\} \cup \Gamma(v) \setminus \{u\}|}$$

where $\Gamma(u)$ is the neighborhood of user u.

In our framework, we can compute two related measures: the first one, the **weakness**, measures the opposite: the number of neighbors of the pair of users who are not common to both:

$$\text{Weakness}(u, v) = 1 - \text{Embeddedness}(u, v)$$

The second one is just the value of the metric in the complementary graph:

$$\text{Compl. Embeddedness}(u, v) = \frac{|\mathcal{U}| - |\Gamma(u) \cup \Gamma(v)|}{|\mathcal{U}| - |\Gamma(u) \cap \Gamma(v)|}$$

References:

- D. Easley, J.M. Kleinberg. Networks, crowds and markets (2010).
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Parameters

All the variants share the same two parameters:

- `uSel`: selection of the orientation for the neighborhood of the starting node of the edge. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the ending node of the edge. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

The natural configuration for the embeddedness of a links takes `uSel = OUT` and `wSel = IN`.

Configuration file

The configuration for the original embeddedness metric is:

```
Embeddedness:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

for the weakness one is:

```
Weakness:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

and for the metric in the complementary graph:

```
Complementary embeddedness:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.2.6 Extended neighbor overlap

This metric counts the intersection of the users at distance two from the first user with the neighborhood of the second:

$$\mathrm{ECN}(u,v) = \left| \left(\Gamma(u) \cup \bigcup_{w \in \Gamma(u)} \Gamma(w) \right) \cap \Gamma(v) \right|$$

Another version, we name **extended neighbor counted overlap**, instead of straightforwardly counting the number of common neighbors, they count the number of times they appear in the intersection:

$$\mathrm{ECCN}(u,v) = |\Gamma(u) \cap \Gamma(v)| + \sum_{w \in \Gamma(u)} |\Gamma(w) \cap \Gamma(v)|$$

Parameters

- `origin`: true if we take the distance 2 neighborhood of the first user, false if we take the neighborhood of the second.
- `uSel`: selection of the orientation for the neighborhood of the first node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the second node of the pair. This allows the following values:

- `IN`: we take the incoming neighbors of the user.
- `OUT`: we take the incoming neighbors of the user.
- `UND`: we take the incoming and outgoing neighbors of the user.
- `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Extended neighbor overlap:
  type: pair
  params:
    origin:
      type: boolean
      values: [true/false]
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

while the configuration for the weighted variant is:

```
Extended neighbor counted overlap:
  type: pair
  params:
    origin:
      type: boolean
      values: [true/false]
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.2.7 Geodesics

The geodesics metric measures the number of minimum distance paths between a pair of users.

$$\text{Geodesics}(u, v) = |\{p \in \text{paths(u,v)} | \text{length}(p) = \delta(u, v)\}|$$

where $\delta(u, v)$ represents the distance between users u and v, $\text{paths}(u, v)$ is the set of all paths between them, and $\text{length}(p)$ is the length of the path p.

Configuration file

For the original distance, this is computed as:

```
Geodesics:
  type: pair
```

whereas for the reciprocal distance:

```
Reciprocal shortest path length:
  type: pair
```

14.2.8 Neighbor overlap

This metric just counts the number of common neighbors between two users in the network.

$$\mathrm{CN}(u, v) = |\Gamma(u) \cap \Gamma(v)|$$

where $\Gamma(u)$ is the neighborhood of user u.

We can also compute what we call the **complementary neighbor overlap** metric, or, in other words, the value of this metric in the complementary graph. Its equation is the following one:

$$\mathrm{CN}(u, v) = |\mathcal{U}| - |\Gamma(u) \cup \Gamma(v)|$$

Parameters

- `uSel`: selection of the orientation for the neighborhood of the first node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the second node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Neighbor overlap:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

while the configuration for the value of the metric in the complementary graph is:

```
Complementary common neighbors:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.2.9 Preferential attachment

The preferential attachment measures to which extent a link between two pairs of users might appear under the preferential attachment model.

$$\text{PA}(u, v) = |\Gamma(u)||\Gamma(v)|$$

where $\Gamma(u)$ represents the neighborhood of the user.

Parameters

- `uSel`: selection of the orientation for the neighborhood of the first node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the second node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Preferential attachment:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.2.10 Reciprocity

Given a pair of users (u, v), this metric just finds whether the link (v, u) appears in the network. If it does, it returns value 1.0. Value 0.0 is returned otherwise.

Configuration file

The configuration file for the original method is the following

```
Reciprocity:
  type: pair
```

14.2.11 Shrinking average shortest path length

The shrinking average shortest path length metric computes the reduction of the average distance between two users in the social network if the pair was added to it. In case the link exists, it returns 0.0.

We have another variation, we name **shrinking neighbors shortest path length** which restricts this calculation to the neighbors of the involved nodes.

Configuration file

The configuration file for the original method is the following

```
Shrinking ASL:
  type: pair
```

whereas for the limited version:

```
Shrinking neighbors ASL:
  type: pair
```

14.2.12 Shrinking diameter

The shrinking average shortest path length metric computes the reduction of the diameter of the network if the pair was added to it. In case the link exists, it returns 0.0.

We have another variation, we name **shrinking neighbors diameters** which restricts this calculation to the neighbors of the involved nodes.

Configuration file

The configuration file for the original method is the following

```
Shrinking diameter:
  type: pair
```

whereas for the limited version

```
Shrinking neighbors diameter:
  type: pair
```

14.2.13 Weakness

See *Embeddedness*

14.2.14 Weight

If it is available, it just measures the weight of an edge in the graph. In unweighted networks, all edges have weight equal to 1. If the weight does not exist, it takes value equal to 0.

Configuration file

```
Weight:
  type: edge
```

14.2.15 Weighted neighbor overlap

This metric just sums the weights of the common neighbors between two users in the network.

$$\text{Weighted-CN}(u, v) = \sum_{x \in |\Gamma(u) \cap \Gamma(v)} w(v, x)$$

where $\Gamma(u)$ is the neighborhood of user u, and $w(v, x)$ is the weight of the edge between w and x.

We have another version, **logarithmic weighted neighbor overlap** which takes the logarithm of the weight instead:

$$\text{Weighted-CN}(u, v) = \sum_{x \in |\Gamma(u) \cap \Gamma(v)} (1 + \log(w(v, x)))$$

Parameters

- `uSel`: selection of the orientation for the neighborhood of the first node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the second node of the pair. This allows the following values:
  - `IN`: we take the incoming neighbors of the user.
  - `OUT`: we take the incoming neighbors of the user.
  - `UND`: we take the incoming and outgoing neighbors of the user.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Weighted neighbor overlap:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

while the configuration for the value of the metric in the complementary graph is:

```
Log weighted neighbor overlap:
  type: pair
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3 Graph metrics

RELISON integrates the following graph metrics.

- *Average shortest path length*
- *Average reciprocal shortest path length*
- *Clustering coefficient*
- *Clustering coefficient complement*
- *Degree assortativity*
- *Degree Gini complement*
- *Degree Pearson correlation*
- *Density*
- *Diameter*
- *Edge Gini complement*
- *Infinite distances*
- *Radius*
- *Reciprocal average eccentricity*
- *Reciprocal diameter*
- *Reciprocity rate*

14.3.1 Average shortest path length

As its name indicates, this metric just computes the average shortest path length of a graph. In order to deal with infinite distance edges, we propose two options:

- **Option 1:** We only average over those pairs of users at finite distance.
- **Option 2:** We obtain the average shortest path length for each strongly connected component of the network, and then, we average the result for the different components.

Parameters

- `mode`: "Non infinite distances" for option 1, "Components" for option 2.

Configuration file

```
ASL:
  type: graph
  params:
    mode:
      type: string
      values: ["Non infinite distances", "Components"]
```

14.3.2 Average reciprocal shortest path length

This metric computes the harmonic mean of the distances between pairs of users:

$$\text{ARSL} = \frac{1}{|\mathcal{U}|(|\mathcal{U}|-1)} \sum_{u,v} \frac{1}{\delta(u,v)}$$

where $\delta(u,v)$ represents the distance between a pair of users.

Configuration file

```
ARSL:
  type: graph
  params:
    mode:
      type: string
      values: ["Non infinite distances", "Components"]
```

14.3.3 Clustering coefficient

The global clustering coefficient of a social network graph measures the proportion of closed triads in the network:

$$\text{CC}(\mathcal{G}) = \frac{|\{(u,v,w)|u \neq w \wedge (u,v),(v,w),(u,w) \in E\}|}{|\{(u,v,w)|u \neq w \wedge (u,v),(v,w) \in E\}|}$$

Parameters

- `uSel`: selection of the orientation for the first of the two links to consider in the triad (the (u,v) one).
  - `IN`: we take the (v,u) link.
  - `OUT`: we take the (u,v) link.
  - `UND`: we take either the (u,v) or the (v,u) link (the one that exists)
  - `MUTUAL`: we only consider that the pair (u,v) exists if both (u,v) and (v,u) links exist.
- `vSel`: selection of the orientation for the second of the two links in the triad (the (v,w) one).
  - `IN`: we take the (v,w) link
  - `OUT`: we take the (w,v) link.
  - `UND`: we take either the (w,v) or the (v,w) link (the one that exists)
  - `MUTUAL`: we only consider that the pair (u,v) exists if both (u,v) and (v,u) links exist.

The natural clustering coefficient can be chosen by taking `uSel = IN` and `vSel = OUT`.

Configuration file

The configuration file for the original method is the following

```
Clustering coefficient:
  type: graph
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3.4 Clustering coefficient complement

This metric is the complement of the clustering coefficient, as it takes measures the proportion of the open triads in the network.

$$\text{CCC}(\mathcal{G}) = \frac{|\{(u,v,w)|u \neq w \wedge (u,w) \notin E \wedge (u,v),(v,w) \in E\}|}{|\{(u,v,w)|u \neq w \wedge (u,v),(v,w) \in E\}|}$$

Parameters

- `uSel`: selection of the orientation for the first of the two links to consider in the triad (the (u,v) one).
  - `IN`: we take the (v,u) link.
  - `OUT`: we take the (u,v) link.
  - `UND`: we take either the (u,v) or the (v,u) link (the one that exists)
  - `MUTUAL`: we only consider that the pair (u,v) exists if both (u,v) and (v,u) links exist.
- `vSel`: selection of the orientation for the second of the two links in the triad (the (v,w) one).
  - `IN`: we take the (v,w) link
  - `OUT`: we take the (w,v) link.
  - `UND`: we take either the (w,v) or the (v,w) link (the one that exists)
  - `MUTUAL`: we only consider that the pair (u,v) exists if both (u,v) and (v,u) links exist.

The natural clustering coefficient can be chosen by taking `uSel = IN` and `vSel = OUT`.

Configuration file

The configuration file for the original method is the following

```
Clustering coefficient complement:
  type: graph
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

(continues on next page)

(continued from previous page)

```
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3.5 Degree assortativity

The degree assortativity measures to what extent users create links towards similar users in terms of their degree (i.e. if users with small degree create links towards users with small degrees and users with large degree create links towards users with large degree) or not.

In undirected networks, it is computed as:

$$\text{Assortativity}(\mathcal{G}) = \frac{2 \cdot |E| \cdot \sum_{(} u, v)|\Gamma(u)||\Gamma(v)| - \left(\sum_u |\Gamma(u)|^2\right)^2}{4m \sum_u |\Gamma(u)|^3 - \left(\sum_u |\Gamma(u)|^2\right)^2}$$

Reference : M.E.J. Newman. Mixing patterns in networks. Physical Review E, 67 026126 (2003)

Parameters

- `orientation`: selection for the degree to use.
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Degree assortativity:
  type: graph
  params:
    orientation:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3.6 Degree Pearson correlation

The degree assortativity measures the Pearson correlation of the degrees between the origin and destination endpoints of the nodes.

$$\text{Pearson}(\mathcal{G}) = \frac{\sum_{(u,v)\in E} |\Gamma(u)||\Gamma(v)|}{\sqrt{\sum_u |\Gamma(u)|^2 \cdot \sum_v |\Gamma(v)|^2}}$$

Parameters

- `uSel`: selection of the orientation for the neighborhood of the starting node of the edges. This allows the following values:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.
- `vSel`: selection of the orientation for the neighborhood of the ending node of the edges. This allows the following values:
  - `IN`: we take the incoming neighbors of the users.
  - `OUT`: we take the outgoing neighbors of the users.
  - `UND`: we take the incoming and outgoing neighbors of the users.
  - `MUTUAL`: we take those neighbors who are both incoming and outgoing at the same time.

Configuration file

The configuration file for the original method is the following

```
Degree Pearson:
  type: graph
  params:
    uSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
    vSel:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3.7 Degree Gini complement

The degree Gini complement indicates how balanced the degree distribution of the network is. Values close to one indicate that the degree distribution is flat, whereas values close to 0 show that a few users concentrate all the links in the network.

$$\text{DegreeGiniCompl}(\mathcal{G}) = 1 - \frac{1}{|\mathcal{U}| - 1} \sum_{i=1}^{|\mathcal{U}|} (2i - |\mathcal{U}| - 1) \frac{|\Gamma(u_i)|}{\sum_v |\Gamma(v)|}$$

where $\Gamma(u)$ is the neighborhood of user u and u_i is the i-th node in the network with the smaller degree.

Parameters

- `orientation`: selects the type of degree we use (only affects directed networks).
  - `IN`: we take the in-degree of the users.
  - `OUT`: we take the out-degree of the users.
  - `UND`: we take the undirected degree of the users (in-degree + out-degree)
  - `MUTUAL`: we take as the degree the number of mutual links.

Configuration file

The configuration file for the original method is the following

```
Degree Gini Complement:
  type: graph
  params:
    orientation:
      type: orientation
      values: [IN/OUT/UND/MUTUAL]
```

14.3.8 Density

The density of a network measures the proportion of the possible number of edges between nodes which exist in the network. This metric does not consider selfloops.

$$\text{Density}(\mathcal{G}) = \frac{|E|}{|\mathcal{U}|(|\mathcal{U}|-1)}$$

Configuration file

The configuration file for the original method is the following

```
Density:
  type: graph
```

14.3.9 Diameter

The diameter of a network measures the maximum (finite) distance between two users in the network.

$$\text{Diameter}(\mathcal{G}) = \max_{(u,v):\delta(u,v)<\infty} \delta(u,v)$$

It is equivalent to the maximum eccentricity of the network.

Configuration file

The configuration file for the original method is the following

```
Diameter:
  type: graph
```

14.3.10 Edge Gini complement

The edge Gini complement computes how balanced the number of links between different pairs of user is. This metric has only sense over multigraphs, where multiple links between users are allowed. The metric formulation is similar to:

$$\text{EdgeGini}(\mathcal{G}) = 1 - \frac{1}{|\mathcal{U}|(|\mathcal{U}|-1)} \sum_{i=1}^{|\mathcal{U}|(|\mathcal{U}|-1)} (2i - |\mathcal{U}|(|\mathcal{U}|-1) - 1) \frac{|\{(u,v)_i \in E\}|}{|E|}$$

where $(u,v)_i$ is the i-th pair of users with an smaller number of links.

We differentiate three variants:

- **Inter edge Gini complement:** This metric does not consider the selfloops between the users. It takes the previous equation (considering that E does not have selfloops).
- **Semi-complete edge Gini complement:** This metric stores selfloops as a different category for the Gini index, i.e. we add an element to the sum, counting the total number of selfloops in the network.
- **Complete edge Gini complement:** This metric considers selfloops. In the previous equation, we would just need to substitute $|\mathcal{U}|(|\mathcal{U}|-1)$ by $|\mathcal{U}|^2$ when it appears.

Configuration file

The configuration for the inter edge Gini complement is:

```
Inter edge Gini complement:
  type: graph
```

For the semi-complete variant is:

```
Semi-complete edge Gini complement:
  type: graph
```

and, finally, the version considering selfloops is:

```
Complete edge Gini complement:
  type: graph
```

14.3.11 Infinite distances

This metric measures the number of node pairs which do not have a path between the first and the second in the network.

Configuration file

```
Infinite distances:
  type: graph
```

14.3.12 Radius

If we consider the eccentricity values of all the users (i.e. the maximum finite distance between a user and the rest of the network), the radius represents its minimum value.

$$\text{Radius}(\mathcal{G}) = \min_{u} \left(\max_{v:\delta(u,v)<\infty} \delta(u,v) \right)$$

where $\delta(u,v)$ represents the distance between two users.

Configuration file

```
Radius:
  type: graph
```

14.3.13 Reciprocal average eccentricity

This metric computes the inverse value of the average eccentricity of the network.

If we consider the eccentricity values of all the users (i.e. the maximum finite distance between a user and the rest of the network), the radius represents its minimum value.

$$\text{RAE}(\mathcal{G}) = \frac{|\mathcal{U}|}{\sum_u \text{Eccentricity}(u)}$$

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Configuration file

```
Reciprocal average eccentricity:
  type: graph
```

14.3.14 Reciprocal diameter

This metric computes the inverse value of the diameter (see *Diameter*), so, when distances are reduced among the users in the network, the value of this metric increases.

If we consider the eccentricity values of all the users (i.e. the maximum finite distance between a user and the rest of the network), the radius represents its minimum value.

$$\text{RD}(\mathcal{G}) = \frac{1}{\text{Diameter}(\mathcal{G})}$$

where $\delta(u, v)$ represents the distance between two users.

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Configuration file

```
Reciprocal diameter:
  type: graph
```

14.3.15 Reciprocity rate

This metric computes the proportion of the edges in the network which are reciprocal.

$$\text{Reciprocity}(\mathcal{G}) = \frac{|\{(u, v) \in E | (v, u) \in E\}|}{|E|}$$

where $\delta(u, v)$ represents the distance between two users.

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Configuration file

```
Reciprocity:
  type: graph
```

14.4 Individual community metrics

Given a community partition, it is possible to compute several metrics considering the different clusters of users. Individual community metrics evaluate the structural properties for each community on its own. RELISON integrates the following metrics:

- *Degree*
- *Size*

14.4.1 Degree

This metric measures the number of external adjacent links to a given community. It is also known as the cut of the community when the out-degree is used.

$$\text{degree}(c) = |\{(u, v) \in E : u \in c \wedge v \notin c\}|$$

where c is the studied community.

Parameters

- `orientation`: selection of the neighborhood of the node we want to use for computing the degrees of the communities. In undirected neighbors, the value of the degree does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: for using the in-degree.
  - `OUT`: for using the out-degree.
  - `UND`: for using the degree $\text{degree}(c) = \text{in-degree}(c) + \text{out-degree}(c)$
  - `MUTUAL`: for using the number of reciprocated links.

Configuration file

```
Degree:
  type: indiv. community
  params:
    orientation:
      type: orientation
      values: [IN,OUT,UND,MUTUAL]
```

14.4.2 Size

This metric measures the number of nodes in a community

$$\text{degree}(c) = |c|$$

where c is the studied community.

Configuration file

```
Size:
  type: indiv. community
```

14.4.3 Volume

This metric measures the sum of the degrees of the nodes in a community.

$$\text{vol}(c) = \sum_{u \in c} |\Gamma(u)|$$

where c is the studied community.

Parameters

- *orientation:* selection of the neighborhood of the node we want to use for computing the degrees of the communities. In undirected neighbors, the value of the degree does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: for using the in-degree.
  - `OUT`: for using the out-degree.
  - `UND`: for using the degree $\text{degree}(c) = \text{in-degree}(c) + \text{out-degree}(c)$
  - `MUTUAL`: for using the number of reciprocated links.

Configuration file

```
Volume:
  type: indiv. community
  params:
    orientation:
      type: orientation
      values: [IN,OUT,UND,MUTUAL]
```

14.5 Global community metrics

Given a community partition, it is possible to compute several metrics for the complete network considering such partition. RELISON integrates the following metrics:

- *Destiny community size*
- *Degree Gini complement*
- *Edge Gini complement*
- *Modularity*
- *Modularity complement*
- *Number of communities*
- *Size Gini complement*
- *Weak ties*

14.5.1 Destiny community size

This metric computes the average size of the destination communities of the links connecting two different communities.

Configuration file

```
Destiny community size:
  type: global community
```

14.5.2 Degree Gini complement

Estimates how balanced the degree distribution for the different communities is:

$$\text{DegreeGiniCompl}(\mathcal{G}|\mathcal{C}) = 1 - \frac{1}{|\mathcal{C}|-1}\sum_{i=1}^{|\mathcal{C}|}(2i - |\mathcal{C}| - 1)\frac{\text{Degree}(c_i)}{\sum_{c'}\text{Degree}(c')}$$

where c_i is i-th community with smaller degree. We differentiate two variants:

- **Inter degree Gini complement**: it does not consider links from a community to itself.
- **Complete degree Gini complement**: it does consider links from a community to itself.

Parameters

- `orientation:` selection of the neighborhood of the node we want to use for computing the degrees of the communities. In undirected neighbors, the value of the degree does not change when this parameter does. This is only useful in directed networks. This allows the following parameters:
  - `IN`: for using the in-degree.
  - `OUT`: for using the out-degree.
  - `UND`: for using the degree $\text{degree}(c) = \text{in-degree}(c) + \text{out-degree}(c)$
  - `MUTUAL`: for using the number of reciprocated links.

Configuration file

The configuration for the metric variant which does not consider the links between node in the same community is:

```
Inter-community degree Gini:
  type: global community
  params:
    orientation:
      type: orientation
      values: [IN,OUT,UND,MUTUAL]
```

while the version considering links between the nodes in the same community is:

```
Complete community degree Gini:
  type: global community
  params:
    orientation:
      type: orientation
      values: [IN,OUT,UND,MUTUAL]
```

14.5.3 Edge Gini complement

The community edge Gini complement computes how balanced the number of links between different pairs of communities is. The metric formulation is:

$$\text{CommEdgeGini}(\mathcal{G}|\mathcal{C}) = 1 - \frac{1}{|\mathcal{C}|(|\mathcal{C}|-1)} \sum_{i=1}^{|\mathcal{C}|(|\mathcal{C}|-1)} (2i - |\mathcal{C}|(|\mathcal{C}|-1) - 1) p((c_1, c_2)_i | \mathcal{G}, \mathcal{C})$$

where $(c_1, c_2)_i$ is the i-th pair of communities with the smaller number of links between them and:

$$p((c_1, c_2)|\mathcal{G}, \mathcal{C}) = \frac{|\{(u,v) \in E | c(u) = c_1 \wedge c(v) = c_2\}|}{|\{(u,v) \in E | c(u) \neq c(v)\}|}$$

where $(u, v)_i$ is the i-th pair of users with an smaller number of links.

We differentiate three variants:

- **Inter edge Gini complement:** This metric does not consider links nodes inside the same community. It takes the previous equation.
- **Semi-complete edge Gini complement:** This metric stores links between nodes in the same community as a different category for the Gini index.
- **Complete edge Gini complement:** This metric considers links inside communities. In the previous equation, we would just need to substitute $|\mathcal{C}|(|\mathcal{C}|-1)$ by $|\mathcal{C}|^2$ when it appears, and $|\{(u,v) \in E | c(u) \neq c(v)\}|$ by E.

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.
- J. Sanz-Cruzado, P. Castells. Enhancing Structural Diversity in Social Networks by Recommending Weak Ties. 12th ACM Conference on Recommender Systems (RecSys 2018), pp. 233-241.

Parameters

For the semi-complete and complete versions, we have a parameter:

- `selfloops`: true if we want to allow selfloops between the nodes, false otherwise.

Configuration file

The configuration for the inter edge Gini complement is:

```
Inter-community edge Gini complement:
  type: global community
```

For the semi-complete variant is:

```
Semi-complete community edge Gini complement:
  type: global community
  params:
    selfloops:
      type: boolean
      values: [true, false]
```

and, finally, the version considering links inside communities is:

```
Complete community edge Gini complement:
  type: global community
  params:
    selfloops:
      type: boolean
      values: [true, false]
```

14.5.4 Modularity

The modularity of a network compares the number of links inside communities to the ones we would have in a random graph keeping the degree distribution. It is correlated to the number of links inside communities. Its formulation is:

$$\text{mod}(\mathcal{G}|\mathcal{C}) = \frac{\sum_{u,v}\left(A_{uv} - \frac{|\Gamma(u)||\Gamma(v)|}{|E|}1_{c(u)=c(v)}\right)}{|E| - \sum_{u,v}\frac{\Gamma(u)||\Gamma(v)|}{|E|}1_{c(u)=c(v)}}$$

where 1_x is equal to 1 when condition x is true, 0 otherwise.

Reference: M.E.J. Newman, M. Girvan. Finding and evaluating community structure in networks. Physical Review E 69(2), pp. 1-16 (2004)

Configuration file

```
Modularity:
  type: global community
```

14.5.5 Modularity complement

This metric is computed as the complement of the modularity, so it measures the number of links between communities in the network. Its formulation is:

$$\text{MC}(\mathcal{G}|\mathcal{C}) = \frac{1 - \text{mod}(\mathcal{G}|\mathcal{C})}{2}$$

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.
- J. Sanz-Cruzado, P. Castells. Enhancing Structural Diversity in Social Networks by Recommending Weak Ties. 12th ACM Conference on Recommender Systems (RecSys 2018), pp. 233-241.

Configuration file

```
Modularity complement:
  type: global community
```

14.5.6 Number of communities

As its name indicates, this metric just takes the number of communities in the partition.

Configuration file

```
Num. communities:
  type: global community
```

14.5.7 Size Gini complement

This metric indicates how balanced the distribution of the community sizes is.

$$\text{SizeGiniCompl}(\mathcal{G}|\mathcal{C}) = 1 - \frac{1}{|\mathcal{C}| - 1} \sum_{i=1}^{|\mathcal{C}|} (2i - |\mathcal{C}| - 1) \frac{\text{Size}(c_i)}{|\mathcal{U}|}$$

where c_i is i-th community with smaller size.

Configuration file

```
Complete size Gini:
  type: global community
```

14.5.8 Weak ties

This metric counts the number of links between different communities.

$$\mathrm{WT}(\mathcal{G}|\mathcal{C}) = |\{(u,v) \in E | c(u) \neq c(v)\}|$$

Reference: E. Ferrara, P. de Meo, G. Fiumara, A. Provetti. On Facebook, most ties are weak. Communications of the ACM 57(11), pp. 78-84 (2012)

Configuration file

```
Weak ties:
  type: global community
```

CHAPTER

FIFTEEN

COMMUNITY DETECTION

In social network environments, it is common to group users. A common way to do this is community detection. Community detection finds set of users tightly connected to each other, but sparsely connected to the rest of the network. In the RELISON framework, we provide a set of community detection algorithms, which can be used for this. We provide a program to easily compute this community partitions, and also, functionality for developing new approaches, or integrating the included ones in Java libraries.

Community detection algorithms are included in the SNA module of the framework, which can be imported into any Java library using Maven as follows:

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-sna</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

SIXTEEN

EXPERIMENTAL CONFIGURATION

In our framework, we include a program that simplifies finding the community division of a network. This program can be executed as follows, once the binary for the library has been generated:

```
java -jar RELISON.jar communities network multigraph directed weighted selfloops␣
↪algorithms output
```

where

- `network`: a file containing the social network graph to analyze.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `algorithms`: a Yaml configuration file for reading the community detection algorithms we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the structural properties.

16.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different properties we want to measure and analyze. This is a Yaml file with the following format:

```
algorithms:
  algorithm_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      value:
      object:
        name_of_the_object:
          param_name1:
            type: int/double/boolean/string/long/orientation/object
            ...
          param_name2:
            type: int/double/boolean/string/long/orientation/object
            ...
  algorithm_name2:
    ...
```

16.2 Output file

Once the community detection algorithm is generated, the community partition is stored into a file. Such file has the following format, where each line is tab-separated:

```
node-id comm-id
```

where *node-id* is the identifier of the user in the network, and *comm-id* is the community number.

CHAPTER

SEVENTEEN

INTEGRATE COMMUNITIES IN A JAVA PROJECT

By importing the SNA package, it is possible to use the community partitions of the network in a Java project, as well as detect these partitions, or define novel community partition approaches. In this section we clarify how this can be done.

- *Communities*
- *Dendograms*
- *Community detection algorithms*

17.1 Communities

The communities in a network represent a partition of the users. A node can only belong to one of these groups. In order to manipulate community partitions, we include different interfaces in the framework.

The most important is the `Communities` class, which stores one of these community partitions. It identifies the different communities with integer numbers, and has the following methods:

```
int getNumCommunities()
```

This method obtains the number of clusters in the partition.

```
IntStream getCommunities()
```

This method obtains an stream with the identifiers of the considered community partitions.

```
int getCommunity(U user)
```

Given a user in the network, it obtains the identifier of the community it belongs to.

Arguments:

- *user*: the identifier of the user.

Returns

- the identifier of the community it belongs to, or -1 if the user does not belong to any community.

```
Stream<U> getUsers(int community)
```

Arguments:

- *community*: the community identifier.

Returns:

- an stream containing the users in the community.

```
void addCommunity()
```

This method adds a community to the partition.

```
boolean add(U user, int comm)
```

This function a user to a community.

Arguments:

- *user*: the user.
- *comm*: the community identifier.

Returns:

- `true` if the user is added, `false` otherwise.

```
int getCommunitySize(int comm)
```

Finally, this method counts the number of users in a given community.

Arguments:

- *comm*: the community identifier.

Returns:

- the number of users in the community.

17.2 Dendograms

Sometimes, community detection algorithms do not only find a community partition, but what we call a dendogram: a tree containing different community partitions, where the leaves represent the nodes in the network, and intermediate nodes consider the union of the communities represented by their children.

We provide the class `Dendogram` for reading and working with these structures, allowing to get community partitions by the number of communities, or by obtaining communities of a given size.

It has the following methods:

```
Tree<U> getTree()
```

This method obtains the underlying tree of the dendogram.

Returns:

- the underlying tree of the dendogram.

```
Communities<U> getCommunitiesByNumber(int n)
```

Obtains a community partition containing, at most, `n` communities.

Arguments:

- *n*: the maximum number of communities.

Returns

- the desired community partition, or `null` if something failed.

```
Map<Integer, Communities<U>> getCommunitiesByNumber()
```

It obtains all the possible community partitions by number.

Returns

- a map, indexed by the number of communities, containing the different community partitions.

```
Communities<U> getCommunitiesBySize(int size)
```

It obtains a partition of the network where the maximum number of users on each community is provided.

Arguments:

- *size*: the maximum number of users on each community.

Returns

- the community partition if everything goes well, `null` otherwise.

```
Map<Integer, Communities<U>> getCommunitiesBySize()
```

It obtains all the possible community partitions by size.

Returns

- a map, indexed by the maximum community size, containing the different community partitions.

17.3 Community detection algorithms

In order to detect communities in a network, we can use many different algorithms. Community detection algorithms inherit the `CommunityDetectionAlgorithm` interface. This interface must be implemented in case we want to develop novel approaches. It has the following methods:

```
Communities<U> detectCommunities(Graph<U> graph)
```

This method, given a network, identifies the community partition according to this algorithm.

Arguments:

- *graph*: the social network graph for detecting communities.

Returns

- the community partition if everything goes well, `null` otherwise.

Then, if the algorithm can find a dendogram, it also inherits the `DendogramCommunityDetectionAlgorithm` interface, which adds the following method:

```
Dendogram<U> detectCommunityDendogram(Graph<U> graph)
```

which finds the dendogram of the network.

Arguments:

- *graph*: the social network graph for detecting communities.

Returns

- the dendogram if everything goes well, `null` otherwise.

CHAPTER

EIGHTEEN

COMMUNITY DETECTION ALGORITHMS SUMMARY

RELISON provides the following algorithms:

18.1 Connectedness community detection algorithms

This family of community detection algorithms considers whether two users are connected in a network or not to determine the community partition of the network. We differentiate two methods:

- *Strongly connected components*
- *Weakly connected components*

18.1.1 Strongly connected components

Under this algorithm, two nodes belong to the same component if there is a finite length path between them in both directions. This method considers the network directionality. In undirected networks, it is equivalent to the *Weakly connected components* algorithm.

Configuration file

```
Strongly connected components:
```

18.1.2 Weakly connected components

Under this algorithm, two nodes belong to the same component if there is a finite length path between them without considering the direction of the edges.

Configuration file

The configuration file for the original method is the following

```
Weakly connected components:
```

18.2 Modularity-based community detection algorithms

This family of community detection algorithms considers network properties to compute the partition. We have implemented the following community detection algorithms.

- *FastGreedy*
- *Girvan-Newman*
- *Infomap*
- *Label propagation*
- *Louvain*
- *Spectral clustering*

18.2.1 FastGreedy

This is an agglomerative clustering method which starts with each node on a different community, and, each step, joins the pair of communities which increases modularity the most (or decreases it the least). From the dendogram, we take the partition maximizing the modularity of the network.

In the framework, we consider three variants of these method in addition to the original version. These additional variants attempt to find not only a set of communities maximizing the modularity of the network, but, at the same time, balancing the number of users on each community. The three additional variants are:

- **Balanced FastGreedy**: fixes a maximum community size, and, it applies again the community detection algorithm over the communities larger than the fixed size.
- **Gini weighted FastGreedy**: joins the pair of communities that, at the same time, increase the modularity and minimizes the Gini index of the community size distribution. Then, it chooses the partition maximizing the modularity (as earlier).
- **Size weighted FastGreedy**: joins the pair of communities that maximize the modularity and minimize the distance between the number of users in the communities to join and the average number of users on each community.

Reference: M.E.J. Newman. Fast Algorithm for detecting community structure in networks. Physical Review E 69(6): 066133 (2004)

Parameters

The original FastGreedy approach does not consider any parameters, but the balanced and Gini weighted variants do.

For the **balanced FastGreedy** algorithm, we have the following parameters:

- `size`: the maximum allowed community size.

whereas for the **Gini weighted FastGreedy**, we take the following parameter:

- `lambda`: the weight given to the Gini index of the size distribution.

Configuration file

The original FastGreedy algorithm has the following configuration file:

```
FastGreedy:
```

whereas the balanced variant has this one:

```
Balanced FastGreedy:
  size:
    type: int
    value: 100
```

the Gini weighted approach has the following:

```
Gini weighted FastGreedy:
  lambda:
    type: double
    value: 0.1
```

and the size weighted variant:

```
Size weighted FastGreedy:
```

18.2.2 Girvan-Newman

The Girvan-Newman algorithm generates the community partition by gradually removing the edge with the largest betweenness in the network. It generates a dendogram, and the partition that maximizes the modularity of the network is selected.

Reference: M. Girvan, M.E.J. Newman. Community structure in social and biological networks, Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)

Configuration file

```
Girvan-Newman:
```

18.2.3 Infomap

The Infomap algorithm computes a community partition of the network by computing the minimum length necessary for describing a random walk in the network. For this it uses a two-level Huffman compressing code: the first one differentiates communities in the network, and the second nodes inside of each community.

To compute this metric, we call to the original implementation of the algorithm, provided by the authors in http://mapequation.org.

Reference: M. Rosvall and C. Bergstrom. Maps of random walks on complex networks reveal community structure. Proceedings of the National Academy of Sciences 105(4), pp. 1118-1123 (2008)

NOTE: As Infomap uses the original implementation (https://github.com/mapequation/infomap), at the moment, this method only works on Linux.

Parameters

- `trials`: the number of iterations of the most external loop of the algorithm.

Configuration file

```
Infomap:
  trials:
    type: int
    value: 1
```

18.2.4 Label propagation

The label propagation algorithm starts with all nodes in different communities. Then, iteratively, each node selects the community of the majority of its neigbors, until everything converges.

Reference: U.N. Raghavan, R. Albert, S. Kumara. Near linear time algorithm to detect communities in large-scale networks. Physical Review E 76: 036106 (2007).

Configuration file

```
Label propagation:
```

18.2.5 Louvain

The Louvain algorithm applies a multi-level community detection algorithm. It starts with all the nodes in different communities, and, iteratively, moves a node to another community in the network where the increment in the modularity is maximum.

When the modularity does not vary, it condenses the network, so communities are now the nodes, and applies the algorithm over that condensed network.

Reference: V. Blondel, J. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks. Journal of Statistical Mechanics 10 (2008)

Parameters

- `threshold`: the minimum variance of the modularity. If in an iteration it changes less than this threshold, we end the phase.

Configuration file

```
Louvain:
  threshold:
    type: double
    value: 0.0001
```

Spectral clustering

The spectral clustering algorithm is a community detection technique for finding a balanced set of communities. It uses the max-flow min-cut theory to find a partition such as the number of edges between two sets is minimized, where a cut between two communities is just the number of edges between them.

We consider two variants of this algorithm:

- **Ratio cut spectral clustering**: Minimizes the ratio cut of the partition, which is defined as:

$$\text{RatioCut}(\mathcal{G}|\mathcal{C}) = \frac{1}{|\mathcal{C}|}\sum_{c}\frac{|\{(u,v) \in E | u \in c \wedge v \notin c\}|}{|c|}$$

- **Normalized cut spectral clustering**: Minimizes the normalized cut of the partition, defined as:

$$\text{RatioCut}(\mathcal{G}|\mathcal{C}) = \frac{1}{|\mathcal{C}|}\sum_{c}\frac{|\{(u,v) \in E | u \in c \wedge v \notin c\}|}{\text{vol}(c)}$$

where

$$\text{vol}(c) = \sum_{v \in c}|\Gamma(v)|$$

Reference: R. Zafarani, M.A. Abassi, H. Liu. Social Media Mining: An Introduction. Chapter 6. 2014

Parameters

- `k`: the desired number of communities.

Configuration file

For the ratio cut version, the configuration file would look as:

```
Ratio cut spectral clustering:
  k:
    type: int
    value: 10
```

and, for the normalized cut version:

```
Normalized cut spectral clustering:
  k:
    type: int
    value: 10
```

CHAPTER

NINETEEN

LINK PREDICTION AND RECOMMENDATION

Networks are not static objects: they are evolving over time. One of the main elements that evolves over time is the set of links in the network. Link prediction and link recommendation have studied the creation of new edges in the network. Link prediction just tries to identify which of them would form naturally in the network in the future, whereas link recommendation (also known as contact or people recommendation in the network) has a more active role, by suggesting users to befriend in networks. In the end, they are both similar tasks.

In this framework, we provide functionalities and metrics for both tasks, which we define in this section.

In order to use these functionalities, it is important to import the following package using Maven:

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-linkpred</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

TWENTY

DATA PREPARATION

Before using some of the link prediction / recommendation algorithms and metrics, it is sometimes necessary to prepare some data. For instance, content-based approaches take as input inverted indexes containing the contents generated by each user in the network, and supervised methods need a suitable set of feature vectors for each target-candidate user pairs.

In this section, we provide information about how we can generate these data.

20.1 Content index generator

Some algorithms and metrics use, as input, an index containing the information about the user-generated contents associated to each user. In order to generate these indexes, we provide two programs, taking advantage of the Lucene library (https://lucene.apache.org/). We detail here how to execute these programs.

20.1.1 User index generator

This is the Lucene index used by the Twittomender algorithm (and the novelty and diversity metrics). Each user has associated a single document, built by concatenating the information pieces published either by the user, or by her neighbors. In order to build this index, we have to execute the following program:

```
java -jar RELISON.jar index user graph multigraph directed weighted selfloops␣
↪information-pieces header orientation index-route
```

where

- `graph`: a file containing the social network.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `information-pieces`: a file containing the information pieces (See *Information pieces file* below).
- `header`: true if the file contains a header, false otherwise.
- `orientation`: * `own`: uses the pieces created by each user as their representation. * `IN`: uses the pieces created by the incoming neighbors of the user as her representation. * `OUT`: uses the pieces created by the outgoing neighbors of the user as her representation. * `UND`: uses the pieces created by both the outgoing and incoming neighbors of the user as her representation. * `MUTUAL`: uses the pieces created by the mutual neighbors of the user as her representation.

- `output`: a directory in which to store the index.

20.1.2 Information pieces index generator

This is the Lucene index used by the Centroid CB algorithm. An user-generated content is related to each document in the index. A relation between information pieces and creators is also stored.

```
java -jar RELISON.jar index infopiece graph multigraph directed weighted selfloops␣
↪information-pieces header orientation index-route
```

where

- `information-pieces`: a file containing the information pieces (See *Information pieces file* below).
- `header`: true if the file contains a header, false otherwise.
- `output`: a directory in which to store the index.

20.1.3 Information pieces file

The information pieces (individual user-generated contents) file needs to have the following format (CSV divided by tabs):

```
infoId  userId  text  reprCount  likeCount  created  truncated
```

where

- `infoId`: identifier of the information piece.
- `userId`: identifier of the creator.
- `text`: the content of the information piece.
- `reprCount`: number of times the piece has been repropagated.
- `likeCount`: number of likes the piece has been received.
- `created`: UNIX timestamp indicating the date of creation.
- `truncated`: whether we are taking the complete text, or just a small part.

The text must be in UTF-8 format, and user-generated contents are separated by line skips. Fields (like text) which might have tabs or line skips inside must be properly escaped, and surrounded by “”.

20.2 Feature vector generation

In order to execute the supervised people recommendation / link prediction algorithms, it is first necessary to compute the feature vectors for the different links involved in the recommendation. Here, we provide information about how these features can be computed:

```
java -jar RELISON.jar featuregen train-network test-network multigraph directed weighted␣
↪selfloops readtypes config output rec-length (-users test/all -print true/false -
↪reciprocal true/false -distance max -feat-data file index -comms commfile)
```

where

- `train-instance-network`: a file containing a network for retrieving the feature vectors for the training set.

- `train-class-network`: a file containing a network for retrieving the relevance of each training instance.
- `test-instance-network`: a file containing a network for retrieving the feature vectors for the test set.
- `test-class-network`: a file containing a network for retrieving the relevance of each test instance.
- `directed`: true if the network to study is directed, false otherwise.
- `weighted-sampling`: true if we use edge weights for the sampler selecting a suitable set of target-candidate user pairs.
- `weighted-classes`: true if we use edge weights as class outputs (otherwise, binary classes).
- `weighted-features`: true if we use edge weights when computing the features.
- `config`: a Yaml file containing the information about a) samplers, b) metrics and algorithms for computing features (see *Configuration file* below).
- `train-sampling`: a Yaml configurator containing the information about the sampling method for training target-candidate user pairs.
- `test-sampling`: a Yaml configurator containing the sampling method for the test target-candidate user pairs.
- `train-network` : a file containing the test social network, which is used to evaluate the effectiveness of the recommendation algorithms.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `config`: a Yaml configuration file for reading the people recommendation algorithms and the evaluation metrics we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the structural properties.
- `rec-length`: the maximum number of links to recommend to each user.
- **Optional arguments:**
  - `-users test/all`: indicates whether to generate recommendations for all the users (*all*) or just for those who have links in the test set (*test*). By default: *all*.
  - `-print true/false`: indicates whether we want to print the recommendations or not. By default: *true*.
  - `-reciprocal true/false`: in directed networks, this parameter indicates whether we want to recommend reciprocal edges or not. By default: *false*.
  - `-distance max directed`: *max* indicates the maximum distance between the target and the candidate users. *directed* indicates whether we want to consider link orientation when computing such distance. By default, it does not limit the distance.
  - `-feat-data file index`: in case we want to compute feature-based metrics, *file* specifies the location of a feature file (user-feature-value tab-separated triplets), or an index containing the features. If *index* is equal to true, we consider that *file* points to an index.
  - `-comms commfile`: route to a file containing a community partition of the network.

20.2.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the features we want to use. These features can be either a) the score of a recommendation for the link or b) the result of an structural metric. The configuration file is as it follows:

```
algorithms:
  algorithm_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      values: [value1,value2,...,valueN] / value
      range:
      - start: startingValue
        end: endingValue
        step: stepValue
      - start: <...>
      objects:
        name_of_the_object:
          param_name1:
            type: int/double/boolean/string/long/orientation/object
            <...>
          param_name2:
            type: int/double/boolean/string/long/orientation/object
            <...>
  algorithm_name2:
    ...
metrics:
  metric_name1:
    type: vertex/pair
      params:
        param_name1:
          type: int/double/boolean/string/long/orientation/object
          values: [value1,value2,...,valueN] / value
          range:
          - start: startingValue
            end: endingValue
            step: stepValue
          - start: <...>
          objects:
            name_of_the_object:
              param_name1:
                type: int/double/boolean/string/long/orientation/object
                <...>
              param_name2:
                type: int/double/boolean/string/long/orientation/object
                <...>
  metric_name2:
    <...>
```

where `algorithms` shows the part of the configuration file dedicated to the parameter grid of the people recommendation algorithms, whereas the `metric` tag shows the start of the structural metrics to use. At the moment, this program does only admit metrics working over users (they are applied over both the target and candidate users) and over pairs of users.

Finally, there are two more configuration files to consider, which apply a sampling strategy to select a suitable set of

target-user candidate pairs for generating the features. These configuration files are considered as follows:

```
samplers:
   sampler_algorithm:
     param_name1:
       type: int/double/boolean/string/long/orientation/object
       value: value
       object:
         name_of_the_object:
           name: object_name
           params:
             object_param_name1:
               type: int/double/boolean/string/long/orientation/object
               value: value
             object_param_name2:
               type: int/double/boolean/string/long/orientation/object
               <...>
             <...>
```

20.2.2 Output files

This program produces as outcome the LETOR feature files. For more information about this format, see http://terrier.org/docs/v4.0/learning.html

20.3 Feature vector samplers

In order to select the possible target-candidate user pairs for generating feature vectors for supervised methods, the RELISON library provides several methods. We summarize them below:

- *All*
- *Distance two*
- *Distance two link prediction*
- *Link prediction*
- *Recommender*

20.3.1 All

This sampler selects all the possible target-candidate user pairs (it just removes pairs in the training test).

Configuration file

```
All:
```

20.3.2 Distance two

This sampler just selects all the candidate users who share at least a common neighbor with the target user.

Parameters

- **uSel: the neighborhood selection for the target user.**
  - IN: it considers the incoming neighborhood of the target user.
  - OUT: it considers the outgoing neighborhood of the target user.
  - UND: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - MUTUAL: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- **vSel: the neighborhood selection for the candidate user.**
  - IN: it considers the incoming neighborhood of the candidate user.
  - OUT: it considers the outgoing neighborhood of the candidate user.
  - UND: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - MUTUAL: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Distance two:
  uSel:
    type: orientation
    value:  IN/OUT/UND/MUTUAL
  vSel:
    type: orientation
    value: IN/OUT/UND/MUTUAL
```

20.3.3 Distance two link prediction

This sampler just selects candidate users who share at least a common neighbor with the target user. It selects all the target-candidate user pairs in that collection, along with an equal number of them.

Parameters

- **uSel: the neighborhood selection for the target user.**
  - IN: it considers the incoming neighborhood of the target user.
  - OUT: it considers the outgoing neighborhood of the target user.
  - UND: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - MUTUAL: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- **vSel: the neighborhood selection for the candidate user.**
  - IN: it considers the incoming neighborhood of the candidate user.
  - OUT: it considers the outgoing neighborhood of the candidate user.
  - UND: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - MUTUAL: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Distance two link prediction:
  uSel:
    type: orientation
    value:  IN/OUT/UND/MUTUAL
  vSel:
    type: orientation
    value: IN/OUT/UND/MUTUAL
```

20.3.4 Link prediction

This sampler just selects all the target-candidate user pairs in the test set, along with an equal number of negative links (not in the training set).

Configuration file

```
Link prediction:
```

20.3.5 Recommender

For each target user, it takes the top k recommended people as the sampled individuals.

Parameters

- k: the maximum number of target-candidate user pairs to retrieve for each target user.
- rec: the recommendation algorithm.

Configuration file:

```
Recommender:
  k:
    type: int
    value: 1000
  rec:
    type: object
    object:
      name: recommender_name
      params:
        parameter_name1:
          type: parameter_type
          value: parameter_value
        parameter_name2:
        <...>
```

Any recommendation algorithm can be used here, so, take a look at the algorithm configuration to determine the best option.

CHAPTER

TWENTYONE

EXPERIMENTAL CONFIGURATION

As there are many programs in the framework which we can use, we summarize here how they can be configured and used:

21.1 Link recommendation experiments

The RELISON framework provides a program for recommending people in social network environments. After the binary for the library is generated, we can use the program with the following terminal command:

```
java -jar RELISON.jar recommendation train-network test-network multigraph directed␣
↪weighted selfloops readtypes config output rec-length (-users test/all -print true/
↪false -reciprocal true/false -distance max -feat-data file index -comms commfile)
```

where

- `train-network`: a file containing the training social network, which is taken as input to recommenders.
- `test-network` : a file containing the test social network, which is used to evaluate the effectiveness of the recommendation algorithms.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `config`: a Yaml configuration file for reading the people recommendation algorithms and the evaluation metrics we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the structural properties.
- `rec-length`: the maximum number of links to recommend to each user.
- **Optional arguments:**
  - `-users test/all`: indicates whether to generate recommendations for all the users (*all*) or just for those who have links in the test set (*test*). By default: *all*.
  - `-print true/false`: indicates whether we want to print the recommendations or not. By default: *true*.
  - `-reciprocal true/false`: in directed networks, this parameter indicates whether we want to recommend reciprocal edges or not. By default: *false*.

- `-distance max directed`: *max* indicates the maximum distance between the target and the candidate users. *directed* indicates whether we want to consider link orientation when computing such distance. By default, it does not limit the distance.
- `-feat-data file index`: in case we want to compute feature-based metrics, *file* specifies the location of a feature file (user-feature-value tab-separated triplets), or an index containing the features. If *index* is equal to true, we consider that *file* points to an index.
- `-comms commfile`: route to a file containing a community partition of the network.

21.1.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different people recommendation methods we want to apply and evaluate. This is a Yaml file with the following format:

```
algorithms:
  algorithm_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      values: [value1,value2,...,valueN] / value
      range:
      - start: startingValue
        end: endingValue
        step: stepValue
      - start: <...>
      objects:
        name_of_the_object:
          param_name1:
            type: int/double/boolean/string/long/orientation/object
            <...>
          param_name2:
            type: int/double/boolean/string/long/orientation/object
            <...>
  algorithm_name2:
    ...
metrics:
  metric_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      <...>
    param_name2:
      <...>
  metric_name2:
  <...>
```

where `algorithms` shows the part of the configuration file dedicated to the parameter grid of the people recommendation algorithms, whereas the `metric` tag shows the start of the evaluation metric section of the Yaml file.

21.1.2 Output files

This program produces two outcomes: the evaluation file and the recommendations.

Evaluation file

This file contains the evaluation metrics for each algorithm. The first line contains the header, whereas the rest show the metric values for a single algorithm. Each line has the following (tab-separated) format:

```
Variant Fraction metric1 metric2 <...> metricN
```

where fraction represents the number of the algorithm (divided by the total number of algorithms in the comparison).

For example:

```
Variant Fraction P@10  R@10  nDCG@10
Random  0.5 0.001 0.001 0.0013
Popularity  1.0 0.4 0.23  0.3482
```

Recommendation file

This file contains the recommendations produced for the different users. It does not have a header, and each line has the following (tab-separated) format:

```
TargetUserId  CandidateUserId value
```

where the target-candidate user pairs are sorted by a) the target user and b) the score (in descending order). Order between users might be arbitrary.

Example:

```
883345842 10671602  0.7839427836033016
883345842 242101122 0.7510278151340579
883345842 230377004 0.6487410202793975
883345842 19604744  0.6219403238554378
883345842 398306220 0.6129622813222247
883345842 181561712 0.525116653773563
883345842 176566242 0.525116653773563
883345842 105119490 0.525116653773563
883345842 11254812  0.5196496019742988
883345842 11348282  0.5094869396470944
883609597 430916286 3.08258431711799
883609597 756033804 2.7629745300415265
883609597 11254812  2.629591896712651
<...>
```

21.2 Link prediction experiments

The RELISON framework provides a program for predicting the next links to appear in the social network. After the binary for the library is generated, we can use the program with the following terminal command:

```
java -jar RELISON.jar prediction train-network test-network multigraph directed weighted␣
↪selfloops readtypes config output rec-length (-users test/all -print true/false -
↪reciprocal true/false -distance max -feat-data file index -comms commfile)
```

where

- `train-network`: a file containing the training social network, which is taken as input to recommenders.
- `test-network` : a file containing the test social network, which is used to evaluate the effectiveness of the recommendation algorithms.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `config`: a Yaml configuration file for reading the people recommendation algorithms and the evaluation metrics we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the structural properties.
- **Optional arguments:**
  - `-users test/all`: indicates whether to generate predictions for all the users (*all*) or just for those who have links in the test set (*test*). By default: *all*.
  - `-print true/false`: indicates whether we want to print the prediction or not. By default: *true*.
  - `-reciprocal true/false`: in directed networks, this parameter indicates whether we want to predict reciprocal edges or not. By default: *false*.
  - `-distance max directed`: *max* indicates the maximum distance between the target and the candidate users. *directed* indicates whether we want to consider link orientation when computing such distance. By default, it does not limit the distance.
  - `-feat-data file index`: in case we want to compute feature-based metrics, *file* specifies the location of a feature file (user-feature-value tab-separated triplets), or an index containing the features. If *index* is equal to true, we consider that *file* points to an index.
  - `-comms commfile`: route to a file containing a community partition of the network.

21.2.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different people recommendation methods we want to apply and evaluate. This is a Yaml file with the following format:

```
algorithms:
  algorithm_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      values: [value1,value2,...,valueN] / value
      range:
      - start: startingValue
        end: endingValue
        step: stepValue
      - start: <...>
      objects:
        name_of_the_object:
          param_name1:
            type: int/double/boolean/string/long/orientation/object
            <...>
          param_name2:
            type: int/double/boolean/string/long/orientation/object
            <...>
  algorithm_name2:
    ...
metrics:
  metric_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      <...>
    param_name2:
      <...>
  metric_name2:
  <...>
```

where `algorithms` shows the part of the configuration file dedicated to the parameter grid of the people recommendation algorithms, whereas the `metric` tag shows the start of the evaluation metric section of the Yaml file.

21.2.2 Output files

This program produces two outcomes: the evaluation file and the reranked recommendations.

Evaluation file

This file contains the evaluation metrics for each algorithm. The first line contains the header, whereas the rest show the metric values for a single algorithm. Each line has the following (tab-separated) format:

```
Variant Fraction metric1 metric2 <...> metricN
```

where fraction represents the number of the algorithm (divided by the total number of algorithms in the comparison).

For example:

```
Variant Fraction AUC  F1-score@0.5
Random  0.5 0.5 0.5
Popularity  1.0 0.8 0.78
```

Link prediction file

This file contains the predictions produced by the algorithm. It does not have a header, and each line has the following (tab-separated) format:

```
TargetUserId  CandidateUserId value
```

where the target-candidate user pairs are sorted by b) the score (in descending order).

Example:

```
883345842 10671602  0.7839427836033016
113213211 242101122 0.7510278151340579
342433442 230377004 0.6487410202793975
234324232 19604744  0.6219403238554378
567578745 398306220 0.6129622813222247
234232333 181561712 0.525116653773563
<...>
```

21.3 Reranking experiments

After some recommendations have been generated, we might want to rerank them, so we can enhance some structural property. We show here how we can do this using the framework. Once the binary has been generated, we just have to execute the following command:

```
java -jar RELISON.jar reranking train-network test-network multigraph directed weighted␣
↪selfloops readtypes rec-folder comm-file config output rec-length max-length (-users␣
↪test/all -print true/false -reciprocal true/false -distance max -feat-data file index -
↪comms commfile)
```

where

- `train-network`: a file containing the training social network, which is taken as input to recommenders.
- `test-network` : a file containing the test social network, which is used to evaluate the effectiveness of the recommendation algorithms.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `rec-folder`: a recommendation file / a directory containing recommendation files.
- `comm-file`: route to a file containing a community partition of the network.

- `config`: a Yaml configuration file for reading the reranking algorithms and the evaluation metrics we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the reranked recommendations and the evaluation metrics.
- `rec-length`: the maximum number of links to recommend to each user.
- `max-length`: the maximum number of links on each user recommendation to consider during the reranking phase (min. value should be `rec-length`).
- **Optional arguments:**
  - `-reciprocal true/false`: in directed networks, this parameter indicates whether we allowed recommending reciprocal edges for the original recommendations or not. By default: *false*.
  - `-distance max directed`: *max* indicates the maximum distance between the target and the candidate users. *directed* indicates whether we wanted to consider link orientation when computing such distance. By default, it does not limit the distance.
  - `-feat-data file index`: in case we want to compute feature-based metrics, *file* specifies the location of a feature file (user-feature-value tab-separated triplets), or an index containing the features. If *index* is equal to true, we consider that *file* points to an index.

21.3.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different people recommendation methods we want to apply and evaluate. This is a Yaml file with the following format:

```
rerankers:
  reranker_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      values: [value1,value2,...,valueN] / value
      range:
      - start: startingValue
        end: endingValue
        step: stepValue
      - start: <...>
      objects:
        name_of_the_object:
          param_name1:
            type: int/double/boolean/string/long/orientation/object
            <...>
          param_name2:
            type: int/double/boolean/string/long/orientation/object
            <...>
  reranker_name2:
    <...>
metrics:
  metric_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
      <...>
    param_name2:
      <...>
```

(continues on next page)

(continued from previous page)

```
  metric_name2:
  <...>
```

where `algorithms` shows the part of the configuration file dedicated to the parameter grid of the people recommendation algorithms, whereas the `metric` tag shows the start of the evaluation metric section of the Yaml file.

21.3.2 Output files

This program produces two outcomes: the evaluation file and the reranked recommendations.

Evaluation file

This file contains the evaluation metrics for each algorithm. The first line contains the header, whereas the rest show the metric values for a single algorithm. Each line has the following (tab-separated) format:

```
Variant Fraction metric1 metric2 <...> metricN
```

where fraction represents the number of the algorithm (divided by the total number of algorithms in the comparison).

For example:

```
Variant Fraction P@10  R@10  nDCG@10
Random  0.5 0.001 0.001 0.0013
Popularity  1.0 0.4 0.23  0.3482
```

Reranked recommendation file ^^^^^^^^^^^^^^^^^^^^~^^^^^^^ This file contains the reranked recommendations produced for the different users. It does not have a header, and each line has the following (tab-separated) format:

```
TargetUserId  CandidateUserId value
```

where the target-candidate user pairs are sorted by a) the target user and b) the score (in descending order). Order between users might be arbitrary.

Example:

```
883345842 10671602  0.7839427836033016
883345842 242101122 0.7510278151340579
883345842 230377004 0.6487410202793975
883345842 19604744  0.6219403238554378
883345842 398306220 0.6129622813222247
883345842 181561712 0.525116653773563
883345842 176566242 0.525116653773563
883345842 105119490 0.525116653773563
883345842 11254812  0.5196496019742988
883345842 11348282  0.5094869396470944
883609597 430916286 3.08258431711799
883609597 756033804 2.7629745300415265
883609597 11254812  2.629591896712651
<...>
```

21.4 Effects on the network structure

The RELISON framework provides a program for determining the effect of link prediction and recommendation algorithms might have over the network structure. For this, the program adds the set of recommended links over the network structure, and computes structural metrics over that expanded network. In order to use this program, we use the following command:

```
java -jar RELISON.jar effects train-network test-network multigraph directed weighted␣
↪selfloops readtypes rec-folder comm-files config output rec-length full-graph onlyrel
```

where

- `train-network`: a file containing the training social network, which is taken as input to recommenders.
- `test-network` : a file containing the test social network.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `rec-folder`: a recommendation file / a directory containing recommendation files.
- `config`: a Yaml configuration file for reading the structural metrics we want to apply (see *Configuration file* below).
- `output`: a directory in which to store the metrics.
- `rec-length`: the maximum number of recommended links to each user to consider.
- `full-graph`: true if we use all edges/pairs to compute edge/pair metrics.
- `onlyrel`: true if we only add the relevant edges to the training network.
- **Optional arguments:**
  - `-communities file1,file2,...,fileN`: a comma-separated list of files containing a community partition of the network.
  - `--distances`: true if we want to precompute distances between the users (by default: false). Recommended if we use several distance-based metrics.
  - `--prediction user/global`: we include this option if we want to read the outcome of a link prediction. Then, `user` indicates that we add `rec-length` predicted links per user to the expanded network, whereas `global` indicates that we just add the top `rec-length` of the prediction.

21.4.1 Configuration file

In order to select a suitable set of metrics, the program receives, as input, a configuration file, specifying the different properties we want to measure and analyze. This is a Yaml file with the following format:

```
metrics:
  metric_name1:
    type: vertex/edge/pair/graph/indiv. community/global community
      params:
        param_name1:
          type: int/double/boolean/string/long/orientation/object
```

(continues on next page)

(continued from previous page)

```
        values: [value1,value2,...,valueN] / value
        range:
        - start: startingValue
          end: endingValue
          step: stepValue
        - start: ...
        objects:
          name_of_the_object:
            param_name1:
              type: int/double/boolean/string/long/orientation/object
              ...
            param_name2:
              type: int/double/boolean/string/long/orientation/object
              ...
metric_name2:
  ...
```

In this configuration file, we identify each metric by each name, and, afterwards, we identify its type. We differentiate between six groups of metrics:

- **Vertex metrics:** Properties of individual nodes in the network (e.g. degree, local clustering coefficient).
- **Edge/Pair metrics:** Properties of pairs of users in the network. If they are selected with the "edge" identifier, the metrics are only computed over the set of links in the network.
- **Graph metrics:** Global properties of the network (e.g. global clustering coefficient).
- **Individual community metrics:** Properties of a single community in the partition (e.g. community size, degree).
- **Global community metrics:** Global metrics depending on the community partition (e.g. modularity).

CHAPTER

TWENTYTWO

INTEGRATE LINK RECOMMENDATION / PREDICTION FUNCTIONALITIES IN A JAVA PROJECT

By importing the linkpred package, anyone can use link recommendation and prediction approaches and integrate them in their code. We explore here how to do it, as well as how to define novel methods.

- *Link recommendation*
- *Link prediction*

22.1 Link recommendation

If we want to recommend people to any user of the social network, we can use any implemented people recommendation algorithm. These algorithms implement the `Recommender` interface, originally defined for the RankSys (https://ranksys.github.io) library. This interface has the following methods:

```
Recommendation<U,U> getRecommendation(U user)
```

This method obtains a recommendation. It does not limit the size of the recommendation, so it returns every recommendable candidate user.

Arguments:

- *user*: the identifier of the user.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

```
Recommendation<U,U> getRecommendation(U user, int maxLength)
```

This method obtains a recommendation, containing (at most) a fixed number of candidate users.

Arguments:

- *user:* the identifier of the user.
- *maxLength:* the maximum number of candidate users to recommend.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

```
Recommendation<U,U> getRecommendation(U user, int maxLength, Predicate<U> filter)
```

This method obtains a recommendation, containing (at most) a fixed number of candidate users. Candidate users are only considered if they pass a filter.

Arguments:

- *user:* the identifier of the user.
- *maxLength:* the maximum number of candidate users to recommend.
- *filter:* a filter for selecting the set of candidate users.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

```
Recommendation<U,U> getRecommendation(U user, Predicate<U> filter)
```

This method obtains a recommendation for a single user. It does not limit the size of the recommendation, so it returns every recommendable candidate user. Candidate users are only recommendable if they pass a filter.

Arguments:

- *user:* the identifier of the user.
- *filter:* a filter for selecting the set of candidate users.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

```
Recommendation<U,U> getRecommendation(U user, Stream<U> candidates)
```

This method obtains a recommendation for a single user, and receives the set of candidate users by argument. These set of candidate users is ranked and returned.

Arguments:

- *user:* the identifier of the user.
- *candidates:* an stream containing the different users.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

22.2 Building new recommenders

In case we want to build new recommendation algorithms, we might want to implement all the previous functions. But we can do it much simpler. Considering that we receive a FastGraph, we can create a method that inherits from the `UserFastRankingRecommender` class.

These methods must use a `FastGraph<U>` in the constructor, and then, you only have to implement the following method:

```
public Int2DoubleMap getScoresMap(int uidx)
```

This method generates all the possible scores for the target-candidate user pairs, and returns them in a map, indexed by the identifier of the candidate user.

Arguments:

- *user:* the identifier of the user.
- *candidates:* an stream containing the different users.

Returns

- the recommendation object (containing the target user and a list of sorted candidate user-score pairs).

This would be, for example, the case of the `Popularity` algorithm, which we show below:

```
public class Popularity<U> extends UserFastRankingRecommender<U>
{
    /**
     * Link orientation for selecting the neighbours of the candidate node.
     */
    private final EdgeOrientation vSel;

    /**
     * Constructor for recommendation mode.
     *
     * @param graph graph.
     * @param vSel  link orientation for selecting the neighbours of the candidate node.
     */
    public Popularity(FastGraph<U> graph, EdgeOrientation vSel)
    {
        super(graph);
        this.vSel = vSel;
    }

    /**
     * Constructor for recommendation mode.
     *
     * @param graph graph.
     */
    public Popularity(FastGraph<U> graph)
    {
        super(graph);
        this.vSel = EdgeOrientation.IN;
    }

    @Override
    public Int2DoubleMap getScoresMap(int uidx)
    {
        U u = this.uidx2user(uidx);

        Int2DoubleMap scoresMap = new Int2DoubleOpenHashMap();
        scoresMap.defaultReturnValue(-1.0);

        this.getAllUsers().forEach(v -> scoresMap.put(this.item2iidx(v), this.getGraph().
↪getNeighbourhoodSize(v, vSel) + 0.0));
        return scoresMap;
    }
```

22.3 Link prediction

Finally, if we want to integrate link prediction methods in the code, we need to use the `LinkPrediction` interface, which contains the following methods:

```
Prediction<U> getPrediction()
```

This method just finds ranks all the possible user-user pairs.

Returns

- a prediction: a sorted list of user-user-score triplets.

```
Prediction<U> getPrediction(int maxLength)
```

This method just finds ranks a fixed number of user-user pairs (at most).

Arguments:

- *maxLength*: the maximum number of links to predict.

Returns

- a prediction: a sorted list of user-user-score triplets.

```
Prediction<U> getPrediction(Predicate<Pair<U>> filter)
```

This method just finds ranks the set of possible user-user pairs passing a given filter.

Arguments:

- *filter*: the user-user pair filter.

Returns

- a prediction: a sorted list of user-user-score triplets.

```
Prediction<U> getPrediction(int maxLength, Predicate<Pair<U>> filter)
```

This method just finds ranks the set of possible user-user pairs passing a given filter. It only predicts a fixed number of links.

Arguments:

- *maxLength*: the maximum number of links to predict.
- *filter*: the user-user pair filter.

Returns

- a prediction: a sorted list of user-user-score triplets.

```
Prediction<U> getPrediction(Stream<Pair<U>> candidates)
```

This method ranks a given set of pairs of users.

Arguments:

- *candidates*: a stream containing the pairs of users to consider.

Returns

- a prediction: a sorted list of user-user-score triplets.

```
double getPredictionScore(U u, U v)
```

This method obtains the prediction score for a pair of users.

Arguments:

- *u*: the first user.
- *v*: the second user

Returns

- the prediction score for the given pair of users.

22.4 Using people recommendation algorithms as link predictors

People recommendation and link prediction are closely related (they differ on how the task is carried), so it is possible to use link recommendation algorithms to predict the next links to appear in a network. For this, we include in the framework a class, `RecommendationLinkPredictor`. This class receives in the constructor the recommender we want to use.

CHAPTER

TWENTYTHREE

LINK PREDICTION/RECOMMENDATION ALGORITHMS SUMMARY

The RELISON framework provides a comprehensive collection of link recommendation and prediction algorithms. We summarize them in this section:

RELISON provides the following algorithms:

23.1 Baselines

This family of people recommendation algorithms includes some important sanity-check baselines to include in the experiments.

- *Random recommendation*
- *Popularity-based recommendation*

23.1.1 Random recommendation

This algorithm generates random recommendation scores for each candidate user in the network.

Configuration file

```
Random:
```

23.1.2 Popularity-based recommendation

This algorithm recommends the set of most followed users in the network (i.e. those maximizing the in-degree).

Configuration file

```
Popularity:
```

23.2 Content-based algorithms

This family of people recommendation algorithms considers feature information about the users (user-generated contents, communities, etc. to produce the recommendations).

- *Centroid CB*
- *Twittomender*

23.2.1 Centroid CB

Content-based recommendation algorithm, based on a tf-idf scheme. Each piece of content published by a single user is consdered as its content. Then, using some of these contents, a centroid using tf-idf weights is computed for each user. The recommendation score of a link is just the cosine similarity between the centroids of two separated users.

Reference: J. Sanz-Cruzado, P. Castells. Enhancing Structural Diversity in Social Networks by Recommending Weak Ties. 12th ACM Conference on Recommender Systems (RecSys 2018), 233-241 (2018).

Parameters

- `index`: a directory containing a Lucene index. In order to obtain it, it is necessary to execute the `CBIndexGenerator` program over the set of contents of the users. This index stores the whole set of user-generated contents (each content is considered a separate document), and each piece is identified by its author.
- `orientation`: (*OPTIONAL*) if this parameter is not available, we generate the centroid using just the set of pieces created by the user. If we include this parameter, we create it by using the pieces of her neighbors (selected according to this orientation value). It can take the possible values:
  - `IN`: for using the contents of the incoming neighbors.
  - `OUT`: for using the contents of the outgoing neighbors.
  - `UND`: for using the contents of both the incoming and outgoing neighbors.
  - `MUTUAL`: for using the contents of those networks who share a reciprocal link with the user.

Configuration file

```
centroidCB:
  index:
    type: String
    values: [file1,file2,...,fileN]
  (orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL])
```

23.2.2 Twittomender

Content-based recommendation algorithm, based on a tf-idf scheme. Each user in the network is represented by the concatenation of a set of user-generated contents, stored in an index. Then, the algorithm represents the user as a tf-idf vector, and finds the recommendation score by computing the cosine similarity between the vectors of the target and candidate users.

Reference: J. Hannon, M. Bennet, B. Smyth. Recommending Twitter Users to Follow Using Content and Collaborative Filtering Approaches. 4th Annual International ACM Conference on Recommender Systems (RecSys 2010), 199-206 (2010).

Parameters

- `index`: a directory containing a Lucene index. In order to obtain it, it is necessary to execute the `TwittomenderIndexGenerator` program over the set of contents of the users. This index stores each user as a document.

Configuration file

```
Twittomender:
  index:
    type: String
    values: [file1,...,fileN]
```

23.3 Friends of friends

Friends of friends algorithms represent one of the most well-known and commonly used family of link recommendation and prediction algorithms. They are algorithms which consider the common neighbors between the target and the candidate users to produce the recommendation scores. RELISON includes the following algorithms:

- *Adamic-Adar*
- *Cosine similarity*
- *Hub depressed index*
- *Hub promoted index*
- *Jaccard*
- *Local Leicht-Holme-Newman index*
- *Most common neighbors*
- *Resource allocation*
- *Sørensen*

23.3.1 Adamic-Adar

The Adamic-Adar algorithm promotes users with a high number of common friends, but giving more importance to those friends with low degree (as they are more unique to both friendship circles than popular users).

References:

- L.A. Adamic, E. Adar: Friends and neighbors on the Web. Social Networks, 25(3), 211–230 (2003)
- D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. 12th International Conference on Information and Knowledge Management (CIKM 2003), ACM, 556-559 (2003).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `wSel`: the neighborhood selection for the common neighbor.
  - `IN`: it considers the incoming neighborhood of the common neighbor.
  - `OUT`: it considers the outgoing neighborhood of the common neighbor.
  - `UND`: it considers the all the possible neighbors of the common neighbors ($\Gamma_{out}(w) \cup \Gamma_{in}(w)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the common neighbor ($\Gamma_{out}(w) \cap \Gamma_{in}(w)$)

Configuration file

```
Adamic-Adar:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  wSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.2 Cosine similarity

Also known as the Salton index, this algorithms represents each user as a vector, where each other user in the network is a coordinate. The weights of the edges between the original user and the rest of the network are taken as the coordinate values. Then, the score is computed as the cosine between those vectors.

Reference: L. Lü, T. Zhou. Link Prediction in Complex Networks: A survey. Physica A: Statistical Mechanics and its Applications, 390(6), 1150-1170 (2011).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
Cosine:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.3.3 Hub depressed index

Friends of friends approach for favoring the recommendation nodes with smaller degree than the target user.

References:

- L. Lü, T. Zhou, Y. Zhang. Predicting missing links via local information. European Physical Journal B 71, 623-630 (2009).
- E. Ravasz, A.L. Somera, D.A. Mongru, Z.N. Oltvai, A-L. Barabasi. Hierarchical Organization in Metabolic Networks, Science 297 (2002)

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Hub depressed index:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.4 Hub promoted index

Friends of friends approach for favoring the recommendation nodes with higher degree than the target user.

References:

- L. Lü, T. Zhou, Y. Zhang. Predicting missing links via local information. European Physical Journal B 71, 623-630 (2009).
- E. Ravasz, A.L. Somera, D.A. Mongru, Z.N. Oltvai, A-L. Barabasi. Hierarchical Organization in Metabolic Networks, Science 297 (2002)

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Hub promoted index:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.5 Jaccard

The Jaccard algorithm uses as a recommendation score the probability that any neighbor of the target and candidate user is common to both.

References:

- P. Jaccard. Etude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin de la Societe Vaudoise des Sciences Naturelles 37(142),547–579 (1901)
- D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. 12th International Conference on Information and Knowledge Management (CIKM 2003), ACM, 556-559 (2003).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.

  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Jaccard:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.6 Local Leicht-Holme-Newman index

This algorithm assigns high similarity to node pairs that have many neighbors in common in comparison to the expected number of common neighbors in a configuration model.

Reference: E.A. Leicht, P. Holme, M.E.J. Newman. Vertex Similarity in Networks. Physical Review E 73(2): 026120 (2006).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Local LHN:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.7 Most common neighbors

The most common neighbors algorithm just takes the number of common neighbors between the target and candidate users as the recommendation score.

Reference: D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. 12th International Conference on Information and Knowledge Management (CIKM 2003), ACM, 556-559 (2003).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
MCN:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.8 Resource allocation

Based on the physical resource allocation process, this method finds the amount of a resource that would reach

Reference: L. Lü, T. Zhou, Y. Zhang. Predicting missing links via local information. European Physical Journal B 71, 623-630 (2009).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `wSel`: the neighborhood selection for the common neighbor.
  - `IN`: it considers the incoming neighborhood of the common neighbor.
  - `OUT`: it considers the outgoing neighborhood of the common neighbor.
  - `UND`: it considers the all the possible neighbors of the common neighbors ($\Gamma_{out}(w) \cup \Gamma_{in}(w)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the common neighbor ($\Gamma_{out}(w) \cap \Gamma_{in}(w)$)

Configuration file

```
Resource allocation:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  wSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.3.9 Sørensen

This method is based on an statistic index for comparing how similar to samples are (here, how similar the neighbors of two users are).

References:

- L. Lü, T. Zhou. Link Prediction in Complex Networks: A survey. Physica A: Statistical Mechanics and its Applications, 390(6), 1150-1170 (2011).
- T. Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. Biologiske Skrifter 5(4), pp. 1-34 (1948)

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
Sorensen:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.4 Information retrieval

This group of approaches includes multiple implementations of information retrieval models, initially devised for searching text documents in massive information spaces. All the algorithms included here are collaborative filtering approaches (they do not use any feature information, just the connections in the network). Details about how these models were adapted and their formulations for contact recommendation can be found in the following references:

References:

- J. Sanz-Cruzado, P. Castells, C. Macdonald, I. Ounis. Effective Contact Recommendation in Social Networks by Adaptation of Information Retrieval Models. Information Processing & Management 57(5) (2020).
- J. Sanz-Cruzado, C. Macdonald, I. Ounis, P. Castells. Axiomatic Analysis of Contact Recommendation Methods in Social Networks: an IR perspective. 42th European Conference on Information Retrieval (ECIR 2020), 175-90 (2020).

The IR models included in this framework are:

- *Binary independent retrieval*
- *BM25*
- *DFRee*
- *DFReeKLIM*
- *DLH*
- *DPH*
- *Extreme BM25*
- *Pivoted normalization vector space model*
- *PL2*
- *Query likelihood*
- *Vector space model*

23.4.1 Binary independent retrieval

Adaptation of the binary independent retrieval (BIR) model.

Reference: K. Sparck Jones, S. Walker, S.E. Robertson. A Probabilistic Model of Information Retrieval: Development and Comparative Experiments. Information Processing and Management 36, 779-808 (part 1), 809-840 (part 2) (2000).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)

Configuration file

```
BIR:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.4.2 BM25

Adaptation of the BM25 probabilistic information retrieval model.

References:

- K. Sparck Jones, S. Walker, S.E. Robertson. A Probabilistic Model of Information Retrieval: Development and Comparative Experiments. Information Processing and Management 36, 779-808 (part 1), 809-840 (part 2) (2000).
- S.E. Robertson, H. Zaragoza. The Probabilistic Relevance Framework: BM25 and Beyond. Foundations and Trends in Information Retrieval 3, 333–389 (2009).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `dlSel`: the neighborhood selection for computing the document length.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `b`: parameter for tuning the effect of the neighborhood size. It takes values between 0 and 1.
- `k`: parameter for tuning the effect of the term frequency in the model. It takes positive values.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
BM25:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  dlSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  b:
    type: double
    range:
      - start: 0.1
        end: 0.99
        step: 0.1
  k:
    type: double
    values: [0.01,0.1,1,10,100]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.3 DFRee

Adaptation of a parameter-free divergence from randomness model using the average of tw information measures.

Reference: G. Amati, G. Amodeo, M. Bianchi, G. Marcone, F.U. Bordoni, C. Gaibisso, G. Gambosi, A. Celi, C.D. Nicola, M. Flammini. FUB, IASI-CNR, UNIVAQ at TREC 2011 Microblog Track. 20th Text REtrieval Confer-ence (TREC 2011) (2011).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
DFRee:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.4 DFReeKLIM

Adaptation of a parameter-free divergence from randomness model using the product of two Kullback-Leibler information measures.

Reference: G. Amati, G. Amodeo, M. Bianchi, G. Marcone, F.U. Bordoni, C. Gaibisso, G. Gambosi, A. Celi, C.D. Nicola, M. Flammini. FUB, IASI-CNR, UNIVAQ at TREC 2011 Microblog Track. 20th Text REtrieval Confer-ence (TREC 2011) (2011).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
DFReeKLIM:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.5 DLH

Adaptation of a parameter-free divergence from randomness model which considers a hypergeometric distribution as a divergence measure, and Laplace normalization.

Reference: G. Amati. Frequentist and Bayesian Approach to Information Retrieval. In: Proceedings of the 28th European Conference on Information Retrieval (ECIR 2006), 13–24 (2006).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
DLH:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

(continues on next page)

(continued from previous page)

```
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.6 DPH

Adaptation of a parameter-free divergence from randomness model which considers a hypergeometric distribution as a divergence measure, and Laplace normalization.

Reference: G. Amati, E. Ambrosi, M. Bianchi, C. Gaibisso, G. Gambosi: FUB, IASI-CNR and University of Tor Vergata at TREC 2007 Blog Track. 16th Text REtrieval Conference (TREC 2007) (2007)

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
DPH:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.7 Extreme BM25

A version of the *BM25* algorithm, when parameter k tends to infinity.

References:

- J. Sanz-Cruzado, P. Castells, C. Macdonald, I. Ounis. Effective Contact Recommendation in Social Networks by Adaptation of Information Retrieval Models. Information Processing & Management 57(5) (2020).
- J. Sanz-Cruzado, C. Macdonald, I. Ounis, P. Castells. Axiomatic Analysis of Contact Recommendation Methods in Social Networks: an IR perspective. 42th European Conference on Information Retrieval (ECIR 2020), 175-90 (2020).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `dlSel`: the neighborhood selection for computing the document length.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `b`: parameter for tuning the effect of the neighborhood size. It takes values between 0 and 1.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
EBM25:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

(continues on next page)

(continued from previous page)

```
dlSel:
  type: orientation
  values: [IN,OUT,UND,MUTUAL]
b:
  type: double
  range:
    - start: 0.1
      end: 0.99
      step: 0.1
k:
  type: double
  values: [0.01,0.1,1,10,100]
(weighted:
  type: boolean
  values: [true,false])
```

23.4.8 Pivoted normalization vector space model

Adaptation of the vector space model information retrieval model with pivoted normalization.

Reference: A. Singhal, J. Choi, D. Hindle, D.D. Lewis, F.C.N. Pereira: AT and T at TREC-7. 7th Text Retrieval Conference (TREC 1998), 186-198 (1998)

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `s`: parameter for tuning the importance of the candidate user length.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
Pivoted normalization VSM:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  s:
    type: double
    values: [0.01,0.1,1,10,100]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.9 PL2

Adaptation of a divergence from randomness model, where the distribution of terms in the document and the collection is measured using a Poisson distribution, a Laplace aftereffect estimation is used as a first normalization, and, term frequency is normalized using Normalisation 2.

References:

- G. Amati, C.J. Van Rijsbergen. Probabilistic Models of Information Retrieval Based on Measuring the Divergence from Randomness. ACM Transactions on Information Systems 20(4), 357–389 (2002).
- G. Amati. Probability Information Models for Retrieval based on Divergence from Randomness. Ph.D. thesis. University of Glasgow. (2003).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `c`: parameter for tuning the importance of the candidate user length.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
PL2:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  c:
    type: double
    values: [0.01,0.1,1,10,100]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.10 Query likelihood

Adaptation of a language model algorithm known as query likelihood. We differentiate three variants, depending on the applied smoothing:

- Dirichlet smoothing (QLD)
- Jelinek-Mercer smoothing (QLJM)
- Laplace additive smoothing (QLL)

References: J.M. Ponte, W.B. Croft. A language modeling approach to information retrieval. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1998), 275-281 (1998)

Parameters

The general parameters are the following:

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Then, each variant has its own parameters. For the **QLD** version:

- `mu`: parameter controlling the trade-off between the regularization term and the original term. It takes values greater than 0.

for the **QLJM** version:

- `lambda`: parameter controlling the trade-off between the regularization term and the original term. It takes values between 0 and 1.

and, for the **QLL** version:

- `phi`: parameter controlling the trade-off between the regularization term and the original term. It takes values greater than 0.

Configuration file

The configuration file for the query likelihood algorithm with Dirichlet smoothing is:

```
QLD:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  mu:
    type: double
    values: [0.01,0.1,1,10,100]
  (weighted:
    type: boolean
    values: [true,false])
```

The configuration file for the query likelihood model with Jelinek-Mercer smoothing is:

```
QLJM:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  lambda:
    type: double
    range:
      - start: 0.1
        end: 0.99
        step: 0.1
  (weighted:
    type: boolean
    values: [true,false])
```

And, finally, for the query likelihood model with Laplace smoothing:

```
QLL:
  uSel:
    type: orientation
```

(continues on next page)

(continued from previous page)

```
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  phi:
    type: double
    values: [0.01,0.1,1,10,100]
  (weighted:
    type: boolean
    values: [true,false])
```

23.4.11 Vector space model

Adaptation of the original vector space model in information retrieval.

Reference: G. Salton, A. Wong, C.S. Yang. A vector space for automatic indexing. Communications of the ACM 18(11), 613-620 (1975).

Parameters

- `uSel`: the neighborhood selection for the target user.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `vSel`: the neighborhood selection for the candidate user.
  - `IN`: it considers the incoming neighborhood of the candidate user.
  - `OUT`: it considers the outgoing neighborhood of the candidate user.
  - `UND`: it considers the all the possible neighbors of the candidate users ($\Gamma_{out}(v) \cup \Gamma_{in}(v)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the candidate user ($\Gamma_{out}(v) \cap \Gamma_{in}(v)$)
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
VSM:
  uSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  vSel:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.5 Matrix factorization

Matrix factorization approaches consider that the adjacency matrix of the social network can be factorized in a group of two or three matrices of smaller dimension. We include the following approaches:

- *Implicit matrix factorization*
- *Fast implicit matrix factorization*

23.5.1 Implicit matrix factorization

Matrix factorization algorithm designed to deal with implicit feedback data in recommender systems.

Reference: Y. Hu, Y. Koren, C. Volinsky. Collaborative filtering for implicit feedback datasets, International Conference on Data Mining (ICDM 2008) (2008).

Parameters

- `lambda`: regulates the importance of the error and the norm of the latent vectors.
- `alpha`: weights the confidence on the weight of the edges.
- `k`: the number of latent factors for each user.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
iMF:
  lambda:
    type: double
    values: [0.1,1,10,100,150]
  alpha:
    type: double
    values: [1,10,40,100]
  k:
    type: int
    range:
      - start: 10
        end: 300
        step: 10
  (weighted:
    type: boolean
    values: [true,false])
```

23.5.2 Fast implicit matrix factorization

Fast matrix factorization algorithm designed to deal with implicit feedback data in recommender systems.

Reference: I. Pilászy, D. Zibriczky and D. Tikk. Fast ALS-based Matrix Factorization for Explicit and Implicit Feedback Datasets. 4th ACM Conference on Recommender Systems (RecSys 2010),71–78 (2010).

Parameters

- `lambda`: regulates the importance of the error and the norm of the latent vectors.
- `alpha`: weights the confidence on the weight of the edges.
- `k`: the number of latent factors for each user.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
Fast iMF:
  lambda:
    type: double
    values: [0.1,1,10,100,150]
  alpha:
    type: double
    values: [1,10,40,100]
  k:
    type: int
    range:
      - start: 10
        end: 300
        step: 10
  (weighted:
    type: boolean
    values: [true,false])
```

23.6 Nearest neighbors algorithms

Nearest neighbors recommendation algorithms select the users in the network that better represent either the target or the candidate users. We include the following collaborative filtering variants of these algorithms:

- *User-based kNN*
- *Item-based kNN*
- *Similarities*

23.6.1 User-based kNN

The user-based kNN algorithm chooses the most similar users to the target user of the recommendation.

Reference: X. Ning, C. Desrosiers, G. Karypis. A Comprehensive Survey of Neighborhood-Based Recommendation Methods. Recommender Systems Handbook, 2nd. Ed., 37-76 (2015).

Parameters

- `k`: the maximum number of neighbors to choose.
- `similarity`: the method to choose (see *Similarities*)
- `q`: the exponent of the similarity.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
UB kNN:
  k:
    type: int
    range:
      - start: 10
        end: 300
        step: 10
  q:
    type: int
    values: 1
  similarity:
    type: object
    objects:
      similarity_name1:
        similarity_param1:
        <...>
  (weighted:
    type: boolean
    values: [true,false])
```

23.6.2 Item-based kNN

The item-based kNN algorithm chooses the most similar users to the candidate user of the recommendation.

Reference: X. Ning, C. Desrosiers, G. Karypis. A Comprehensive Survey of Neighborhood-Based Recommendation Methods. Recommender Systems Handbook, 2nd. Ed., 37-76 (2015).

Parameters

- `k`: the maximum number of neighbors to choose.
- `similarity`: the method to choose (see *Similarities*)
- `q`: the exponent of the similarity.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
IB kNN:
  k:
    type: int
    range:
      - start: 10
        end: 300
        step: 10
  q:
    type: int
    values: 1
  similarity:
    type: object
    objects:
      similarity_name1:
        similarity_param1:
        <...>
  (weighted:
    type: boolean
    values: [true,false])
```

23.6.3 Similarities

The set of all the similarities which can be applied for kNN is large. This number even grows in contact recommendation in social networks, as we can use any possible standalone algorithm as a neighborhood selection method. We allow this in our experiments: any people recommendation algorithm can be selected. The specific grid for these similarities is the same as in the original method.

23.7 Path-based algorithms

These algorithms consider the paths between different nodes in the network to find the recommendation scores. The framework contains the following algorithms within this family.

- *Global Leicht-Holme-Newman index*
- *Katz*
- *Local path index*
- *Matrix forest*
- *Pseudo inverse cosine*

- *Shortest path distance*

23.7.1 Global Leicht-Holme-Newman index

This algorithm considers that two users are similar if their immediate neighbors in the network are themselves similar.

Reference: E.A. Leicht, P. Holme, M.E.J. Newman. Vertex Similarity in Networks. Physical Review E 73(2): 026120 (2006).

Parameters

- `phi`: decay factor for the similarity as we use longer paths between the users.
- `orientation`: the method to choose the adjacency matrix. This parameter does not influence in undirected networks.
  - `IN`: coordinate A_{ij} shows the weight of the (j, i) edge.
  - `OUT`: coordinate A_{ij} shows the weight of the (i, j) edge.
  - `UND`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges.
  - `MUTUAL`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges (but only if both exist).
- `q`: the exponent of the similarity.

Configuration file

```
Global LHN:
  phi:
    type: double
    values: [0.01,0.1,1,10,100]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.7.2 Katz

This algorithm weights the possible paths between the target and candidate users, giving more weight to those at closer distances.

References:

- L. Katz. A new status index derived from sociometric analysis. Psychometrika 18(1), 39-43 (1953)
- D. Liben-Nowell, D., J. Kleinberg. The Link Prediction Problem for Social Networks. Journal of the American Society for Information Science and Technology 58(7) (2007)

Parameters

- b: decay factor for the greater distance paths.
- orientation: the method to choose the adjacency matrix. This parameter does not influence in undirected networks.
  - IN: coordinate A_{ij} shows the weight of the (j, i) edge.
  - OUT: coordinate A_{ij} shows the weight of the (i, j) edge.
  - UND: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges.
  - MUTUAL: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges (but only if both exist).

Configuration file

```
Katz:
  b:
    type: double
    values: [0.01,0.1,1,10,100]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.7.3 Local path index

This algorithm weights the possible paths between the target and candidate users, giving more weight to those at closer distances.

References:

- L. Lü, C. Jin, T. Zhou. Similarity Index Based on Local Paths for Link Prediction of Complex Networks. Physical Review E 80(4): 046122 (2009)
- L. Lü, T. Zhou. Link Prediction in Complex Networks: A survey. Physica A 390(6), 1150-1170 (2011)

Parameters

- b: decay factor for the greater distance paths.
- k: the maximum distance between the target and candidate users (greater or equal than 3).
- orientation: the method to choose the adjacency matrix. This parameter does not influence in undirected networks.
  - IN: coordinate A_{ij} shows the weight of the (j, i) edge.
  - OUT: coordinate A_{ij} shows the weight of the (i, j) edge.
  - UND: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges.
  - MUTUAL: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges (but only if both exist).

Configuration file

```
Local path index:
  b:
    type: double
    values: [0.01,0.1,1,10,100]
  k:
    type: int
    values: [3,4,5,6]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.7.4 Matrix forest

This algorithm takes as score the ratio of the number of spanning divergent forests such that the target and candidate belong to the same tree, rooted in the target user.

References:

- L. Lü, T. Zhou. Link Prediction in Complex Networks: A survey. Physica A 390(6), 1150-1170 (2011)

Parameters

- `alpha`: importance of the Laplacian matrix.
- `orientation`: the method to choose the adjacency matrix. This parameter does not influence in undirected networks.
  - `IN`: coordinate A_{ij} shows the weight of the (j, i) edge.
  - `OUT`: coordinate A_{ij} shows the weight of the (i, j) edge.
  - `UND`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges.
  - `MUTUAL`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges (but only if both exist).

Configuration file

```
Matrix forest:
  alpha:
    type: double
    values: [0.01,0.1,1,10,100]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.7.5 Pseudo inverse cosine

This algorithm represents each user by the u-th row of the pseudo-inverse of the Laplacian matrix of the network. Then, the score is just the cosine similarity between these two vectors.

References:

- F. Fouss, A. Pirotte, J-M. Renders, M. Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. IEEE TKDE 19(3), pp. 355-369 (2007).

Parameters

- `orientation`: the method to choose the adjacency matrix. This parameter does not influence in undirected networks.
  - `IN`: coordinate A_{ij} shows the weight of the (j, i) edge.
  - `OUT`: coordinate A_{ij} shows the weight of the (i, j) edge.
  - `UND`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges.
  - `MUTUAL`: coordinate A_{ij} shows the sum of the weights of the (i, j) and (j, i) edges (but only if both exist).

Configuration file

```
Pseudo-inverse cosine:
  alpha:
    type: double
    values: [0.01,0.1,1,10,100]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
```

23.7.6 Shortest path distance

The shortest path distance recommends people in the network who are close to the target user (the closest, the highest the score shall be)-

Reference: D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. 12th International Conference on Information and Knowledge Management (CIKM 2003), ACM, 556-559 (2003).

Parameters

- `orientation`: the orientation to choose for the edges.
  - `IN`: it considers the distance between the candidate user and the target user.
  - `OUT`: it considers the distance between the target and the candidate user.
  - `UND`: it considers the distance between the target and candidate users if the network was undirected.
  - `MUTUAL`: in this case, we consider just the most natural distance (the `OUT` case).

Configuration file

```
Distance:
  orientation:
    type: orientation
    values: [IN,OUT,UND]
```

23.8 Random walks

These algorithms consider the properties of random walks to determine the recommendation score. We include the following approaches:

- *Commute time*
- *Hitting time*
- *HITS*
- *PageRank*
- *Personalized PageRank*
- *Personalized HITS*
- *Personalized SALSA*
- *PropFlow*
- *SALSA*

23.8.1 Commute time

Average time that a random walker needs to go from the target to the origin user and come back. We consider two variants, depending on the underlying random walk algorithm:

- **PageRank**
- **Personalized PageRank**

Reference: D. Liben-Nowell, D., J. Kleinberg. The Link Prediction Problem for Social Networks. Journal of the American Society for Information Science and Technology 58(7) (2007).

Parameters

- `r`: the PageRank teleport rate.

Configuration file

The non-personalized PageRank version is selected as:

```
Commute time PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

and the personalized one as:

```
Commute time personalized PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.8.2 Hitting time

Average time that a random walker needs to go from the target to the candidate user. We consider two variants, depending on the underlying random walk algorithm:

- **PageRank**
- **Personalized PageRank**

Reference: D. Liben-Nowell, D., J. Kleinberg. The Link Prediction Problem for Social Networks. Journal of the American Society for Information Science and Technology 58(7) (2007).

Parameters

- `r`: the PageRank teleport rate.

Configuration file

The non-personalized PageRank version is selected as:

```
Hitting time PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

and the personalized one as:

```
Hitting time personalized PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.8.3 HITS

Algorithm based on the Hyperlink-Induced Topic Search algorithm.

Reference: J.M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. Journal of the ACM 46(5), 604-642 (1999).

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.

Configuration file

```
HITS:
  mode:
    type: boolean
    values: [true,false]
```

23.8.4 PageRank

This algorithm takes the non-personalized PageRank algorithm (initially designed for estimating the importance of web pages) as a recommendation / prediction method.

Reference: S. Brin, L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. 7th Annual International Conference on World Wide Web (WWW 1998), 107-117 (1998).

Parameters

- `r`: teleport rate of the random walk.

Configuration file

```
PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.8.5 Personalized HITS

Personalized version of the HITS algorithm, where a teleport probability to the target user of the recommendation has been added.

Reference: A. Goel. The Who-To-Follow System at Twitter: Algorithms, Impact and Further Research. 32rd Annual International Conference on World Wide Web (2014), industry track (2014)

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.
- `alpha:` teleport rate.

Configuration file

```
Personalized HITS:
  alpha:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  mode:
    type: boolean
    values: [true, false]
```

23.8.6 Personalized PageRank

Personalized version of the PageRank algorithm, where the random walk always teleports to the target user. Also known as rooted PageRank.

Reference: S. White, P. Smyth. Algorithms for Estimating Relative Importance in Networks. 9th Annual ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003).

Parameters

- `r`: teleport rate of the random walk.

Configuration file

```
Personalized PageRank:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.8.7 Personalized SALSA

Personalized version of the SALSA algorithm, where a teleport probability to the target user of the recommendation has been added.

Reference: A. Goel, P. Gupta, J. Sirois, D. Wang, A. Sharma, S. Gurumurthy. The who-to-follow system at Twitter: Strategy, algorithms and revenue impact. Interfaces 45(1), 98-107 (2015)

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.
- `alpha:` teleport rate.

Configuration file

```
Personalized SALSA:
  alpha:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  mode:
    type: boolean
    values: [true, false]
```

23.8.8 PropFlow

This algorithm considers the probability that a random walker starting in the target user reaches the candidate user in less than few steps, using the edge weights as transition probabilities.

Reference: R. Lichtenwalter, J. Lussier, N. Chawla. New perspectives and methods in link prediction. 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2010), 243-252 (2010).

Parameters

- `maxLength`: importance of the Laplacian matrix.
- `orientation`: the method to choose the orientation of the paths.
  - `IN`: the walker advances through incoming edges.
  - `OUT`: the walker advances through outgoing edges.
  - `UND`: the walker advances through any edge.
  - `MUTUAL`: the walker advances through reciprocal edges.
- `weighted`: (*OPTIONAL*) true to use the weights of the edges, false to consider them binary.

Configuration file

```
PropFlow:
  maxLength:
    type: int
    values: [3,4,5,6]
  orientation:
    type: orientation
    values: [IN,OUT,UND,MUTUAL]
  (weighted:
    type: boolean
    values: [true,false])
```

23.8.9 SALSA

Adaptation of the Stochastic Approach for Link-Structure Analysis (SALSA) algorithm.

Reference: R. Lempel, S. Moran. SALSA: The Stochastic Approach for Link-Structure Analysis. ACM TOIS 19(2), 131-160 (2001)

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.

Configuration file

```
SALSA:
  mode:
    type: boolean
    values: [true,false]
```

23.9 Twitter

These algorithms have been reported to be used at some point (at least, in experiments) for the Twitter Who-to-follow service.

- *Closure*
- *Cosine similarity*
- *Love*
- *Money*

23.9.1 Closure

Algorithm that recommends reciprocal edges according to the number of common neighbors between the already existing edge endpoints.

- **PageRank**
- **Personalized PageRank**

Reference: P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, R. Zadeh. WTF: The Who to Follow Service at Twitter. 22nd Annual International Conference on World Wide Web (WWW 2013), 505-514 (2013).

Configuration file

The non-personalized PageRank version is selected as:

```
Closure:
```

23.9.2 Cosine similarity

This considers several variants of the cosine similarity in the experiments: we consider three:

- Average cosine similarity: it takes the average similarity of the candidate user with the authorities the target user is currently following.
- Centroid cosine similarity: for each user, a centroid is built, using the vectors of the followed users. The score is the cosine of two vectors.
- Maximum cosine similarity: it takes the maximum similarity between the authorities that the target user is currently following.

Parameters

- `r`: the PageRank teleport rate, for computing the circles of trust.
- `circlesize`: the size of the circles of trust. If negative or zero, we take the maximum possible size.

Configuration file

The Yaml code for the average cosine algorithm is:

```
Average cosine:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  circlesize:
    type: int
    values: [0,10,100,1000]
```

for the centroid cosine similarity variant, it is:

```
Centroid cosine:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  circlesize:
    type: int
    values: [0,10,100,1000]
```

and, for the maximum cosine similarity:

```
Maximum cosine:
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  circlesize:
    type: int
    values: [0,10,100,1000]
```

23.9.3 Love

Variant of the personalized HITS algorithm, computed over a circle of trust.

Reference: A. Goel. The Who-To-Follow System at Twitter: Algorithms, Impact and Further Research. 32rd Annual International Conference on World Wide Web (2014), industry track (2014).

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.
- `alpha`: teleport rate for the personalized HITS algorithm.
- `r`: the PageRank teleport rate, for computing the circles of trust.

Configuration file

```
Love:
  mode:
    type: boolean
    values: [true,false]
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

(continues on next page)

(continued from previous page)

```
  alpha:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.9.4 Money

Variant of the personalized SALSA algorithm, computed over a circle of trust.

Reference: A. Goel, P. Gupta, J. Sirois, D. Wang, A. Sharma, S. Gurumurthy. The who-to-follow system at Twitter: Strategy, algorithms and revenue impact. Interfaces 45(1), 98-107 (2015).

Parameters

- `mode`: true if we want to use the authorities scores, false if we want to use the hubs scores.
- `alpha`: teleport rate for the personalized HITS algorithm.
- `r`: the PageRank teleport rate, for computing the circles of trust.

Configuration file

```
Money:
  mode:
    type: boolean
    values: [true,false]
  r:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
  alpha:
    type: double
    range:
    - start: 0.1
      end: 0.99
      step: 0.1
```

23.10 Supervised algorithms

In addition to the rest of algorithms included in the comparative, we include some supervised algorithms (either classification approaches or learning to rank algorithms). All these methods receive, as input, two sets of feature vectors: a set of feature vectors for training the machine learning approaches, and a set of feature vectors to compute the definitive scores. We provide information on how to find such feature vectors in SEE HOW TO REFER TO THIS.

We add the following algorithms:

- *LambdaMART*
- *Weka classifiers*

23.10.1 LambdaMART

LambdaMART represents a learning to rank algorithm based on gradient boosted regression trees, with a great success on information retrieval.

Reference: Y. Ganjisaffar, R. Caruana, C. Lopes. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. 34th Annual International ACM SIGIR conference on Research and development in Information Retrieval (SIGIR 2011), 85-94 (2011).

Parameters

- `train`: the file containing the training vectors for the training set.
- `valid`: the file containing the validation vectors for training the algorithm.
- `test` : the file containing the definitive feature vectors.
- `config` : route of the configuration file for the JForests LambdaMART (see https://github.com/yasserg/jforests)
- `tmp`: route to a temporary folder.

Configuration file

The non-personalized PageRank version is selected as:

```
LambdaMART:
  train:
    type: string
    values: [file1,...,fileN]
  valid:
    type: string
    values: [file1,...,fileN]
  test:
    type: string
    values: [file1,...,fileN]
  config:
    type: string
    values: [file1,...,fileN]
  tmp:
    type: string
    values: [dir1,...,dirN]
```

23.10.2 Weka classifiers

As classifiers, we have integrated in our code the use of classifiers from Weka. We can only access some of them in the original program, but all of them can be used if the `MachineLearningWekaRecommender` is used from code.

Reference: E. Frank, M. A. Hall, and I. H. Witten. The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", 4th edition (2016).

Parameters

- `train`: the file containing the training vectors for the training set.
- `test` : the file containing the definitive feature vectors.
- `classifier` : classifier configurations.

Classifiers

From the recommendation program, we allow the use of three different classifiers: logistic regression, naive Bayes, and random forests. The first two do not have any additional parameters. The random forests adds the number of decision stumps.

Configuration file

We configure these approaches as:

```
Weka:
  train:
    type: string
    values: [file1,...,fileN]
  test:
    type: string
    values: [file1,...,fileN]
  classifier:
    type: object
    objects:
      logistic:
      naive-bayes:
      random-forest:
        iterations:
          type: int
          values: [5,10,15,20,25]
```

CHAPTER

TWENTYFOUR

LINK PREDICTION/RECOMMENDATION METRICS SUMMARY

In order to determine the effectiveness of our algorithms, it is necessary to evaluate them. For this, we provide a series of metrics: RELISON provides implementations of several metric types:

24.1 Accuracy metrics

In order to determine how good people recommendation algorithms are, it is common to evaluate them using ranking metrics. This framework allows the use of the following ones (although more of them can be defined, using the metric interfaces provided by the RankSys framework (https://ranksys.github.io)):

- *Average precision*
- *nDCG*
- *Precision*
- *Recall*

24.1.1 Average precision

This ranking metric gives more importance to the correctly recommended users at the top of the ranking. It just represents the area under the precision-recall curve: a plot of the precision of a recomendation as a function of the recall. It is defined as:

$$\text{AP}(u) = \frac{\sum_{k=1}^{|R_u|} P@k(u) \cdot 1_{\text{relevant}(u)}(v_k)}{|\text{relevant}(u)}$$

where $|R_u|$ represents the length of the recommendation ranking for user u, $P@k$ represents the precision at cutoff k (see *Precision*), relevant(u) is the set of links which have u as origin in the evaluation set, and v_k is the k-th recommended user in the ranking.

When it is averaged over the different users in the network, this metric receives the **mean average precision (MAP)** name.

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
MAP:
  cutoff:
    type: int
    values: [1,5,10]
```

24.1.2 nDCG

The cumulative discounted cumulative gain is a metric that considers that a recommendation is better if the relevant links appear at the top of the ranking. For this, it appears a discount term that depends on the position of the user. It also allows considering several degrees of relevance for the recommended links (for example, defined by the weights of the edges).

This metric is defined as:

$$\text{nDCG}(u) = \frac{\text{DCG}(u)}{\text{IDCG(u)}}$$

where the discounted cumulative gain (DCG) for the user is:

$$\text{DCG}(u) = \sum_{k=1}^{|R_u|} \frac{g_u(v_k)}{\log_2(1+k)}$$

where $|R_u|$ represents the length of the recommendation ranking for user u, $g_u(v_k)$ represents the grade of relevance of user v_k for user u and v_k is the k-th recommended user in the ranking.

Then, the IDCG(u) term acts as a normalization, and it's the best possible :math:mbox{DCG}(u) value.

Reference: K. Jarvelin, J. Kekälainen. Cumulated Gain-Based Evaluation of IR Techniques. ACM Transactions on Information Systems, 20, 422–446 (2002).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
nDCG:
  cutoff:
    type: int
    values: [1,5,10]
```

24.1.3 Precision

The precision metric measures the proportion of correctly recommended links in a recommendation:

$$\mathrm{P}(u) = \frac{|R_u \cap \mathrm{relevant}(u)|}{|R_u|}$$

where R_u represents the recommendation ranking for user u, and $\mathrm{relevant}(u)$ is the set of links which have u as origin in the evaluation set.

Reference: R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval: The Concepts and Technology behind Search, 2nd ed. (2011).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Precision:
  cutoff:
    type: int
    values: [1,5,10]
```

24.1.4 Recall

The precision metric measures the proportion of correctly recommended links which have been discovered in a recommendation:

$$\mathrm{R}(u) = \frac{|R_u \cap \mathrm{relevant}(u)|}{|\mathrm{relevant}(u)|}$$

where R_u represents the recommendation ranking for user u, and $\mathrm{relevant}(u)$ is the set of links which have u as origin in the evaluation set.

Reference: R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval: The Concepts and Technology behind Search, 2nd ed. (2011).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Recall:
  cutoff:
    type: int
    values: [1,5,10]
```

24.2 Diversity metrics

Beyond the accuracy of the recommendations, one of the main dimensions which is commonly studied is their diversity (i.e. how different the recommended users are among them). RELISON includes the following diversity measures:

- *Community recall*
- *ERR-IA*
- *Intra-list diversity*
- *Predicted Gini complement*

24.2.1 Community recall

The community recall is a metric based on the subtopic recall diversity metric in information retrieval. It is defined as the proportion of different communities the target user could reach through a direct link if he followed all the recommended users:

$$\text{CommRecall}(u) = \frac{1}{|\mathcal{C}|}\left|\bigcup_{v \in R_u} c(v)\right|$$

References:

- C. Zhai, W. W. Cohen, and J. Lafferty. Beyond Independent Relevance: Methods and Evaluation Metrics for Subtopic Retrieval. 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2003), 10–17 (2003).
- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Input

In order for this metric to work, the recommendation program must receive a community partition.

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Community recall:
  cutoff:
    type: int
    values: [1,5,10]
```

24.2.2 ERR-IA

This metric combines the accuracy of the recommendation with measuring the different aspects retrieved by the recommendation (here, we take communities as aspects). This particular metric, ERR-IA, given a correctly predicted link, it weights down its importance when the community of the recommended candidate has already appeared in previous positions of the ranking. It is defined as:

$$\text{ERR-IA}(u) = \sum_{c \in \mathcal{C}} p(c|u)\text{ERR-IA}(u, c)$$

and

$$\text{ERR-IA}(u, c) = \sum_{k=1}^{|R_u|} \left[\frac{1}{k} p(\text{rel}|v_k, c) \prod_{j=1}^{k-1} (1 - p(\text{rel}|v_j, c)) \right]$$

where v_k is the recommended user in the k-th position in the ranking, and:

$$p(\text{rel}|v, c) = \frac{1}{2} 1_{\{(u,v) \in E_{test} \wedge c(v) = c\}}(u, v)$$

where E_{test} represents the set of test links. The probability $p(c|u)$ represents the probability that the user u is interested on community c, and it is estimated by:

$$p(c|u) = \frac{|\{v \in c | (u, v) \in E \cup E_{test}\}|}{\sum_{c' \in \mathcal{C}} |\{v \in c' | (u, v) \in E \cup E_{test}\}|}$$

Reference: O. Chapelle, S. Ji, C. Liao, E. Velipasaoglu, L. Lai, and S. Wu. Intent-based diversification of web search results: metrics and algorithms. Information Retrieval, 14(6), 572–592 (2011)

Input

In order for this metric to work, the recommendation program must receive a community partition.

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
nDCG:
  cutoff:
    type: int
    values: [1,5,10]
```

24.2.3 Intra-list diversity

The intra-list diversity of a recommendation, measures the average distances between the recommended users. For this, it uses some feature data (for instance, the contents of the tweets published by a user).

$$\text{ILD} = \frac{1}{|\hat{E}|} \sum_{(u,v) \in \hat{E}} \sum_{w \in R_u} \frac{d(v, w)}{|R_u|}$$

where $\hat{E}$ represents the whole set of recommended links, R_u the set of users recommended to user u, and $d(u, v)$ is the distance between the users (in this case, represented by one minus the cosine of the feature vector for each user).

References:

- P. Castells, N. J. Hurley, and Saúl Vargas. Novelty and Diversity in Recommender Systems. Recommender Systems Handbook, 2nd ed., 881–918 (2015).
- S. Vargas and P. Castells. Rank and Relevance in Novelty and Diversity Metrics for Recommender Systems. 5th ACM Conference on Recommender Systems (RecSys 2011), 109-116 (2011)

Input

In order for this metric to work, the recommendation program must receive a set of features representing the users (or an index, generated by the `TwittomenderIndexGenerator` program).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
ILD:
  cutoff:
    type: int
    values: [1,5,10]
```

24.2.4 Predicted Gini complement

The predicted Gini complement how balanced the distribution of recommended users is among all the produced recommendations. It uses for this the Gini complement:

$$\text{PGC} = 1 - \frac{1}{|\mathcal{U}| - 1} \sum_{i=1}^{|} \mathcal{U}|(2i - |\mathcal{U}| - 1) \frac{|\{u|v_i \in R_u\}|}{\sum_u |R_u|}$$

where R_u represents the recommendation ranking for user u, and v_i is the i-th less recommended user.

References:

- S. Vargas and P. Castells. Improving Sales Diversity by Recommending Users to Items. 8th ACM Conference on Recommender Systems (RecSys 2014), 145–152 (2014).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Predicted Gini complement:
  cutoff:
    type: int
    values: [1,5,10]
```

24.3 Novelty metrics

The novelty of a recommendation measures to which extent the user was unaware of the recommended people in the network. RELISON includes the following novelty measures:

- *Long tail novelty*
- *Mean prediction distance*
- *Unexpectedness*

24.3.1 Long tail novelty

This metric (also known as expected popularity complement) represents the prior probability that a random person in the network did not know about the recommended people:

$$\text{LTN} = \frac{1}{|\hat{E}|} \sum_{(u,v)\in\hat{E}} \left(1 - \frac{|\Gamma_{in}(v)|}{|\mathcal{U}|}\right)$$

References:

- P. Castells, N. J. Hurley, and Saúl Vargas. Novelty and Diversity in Recommender Systems. Recommender Systems Handbook, 2nd ed., 881–918 (2015).
- S. Vargas and P. Castells. Rank and Relevance in Novelty and Diversity Metrics for Recommender Systems. 5th ACM Conference on Recommender Systems (RecSys 2011), 109-116 (2011)

Input

In order for this metric to work, the recommendation program must receive a community partition.

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
LTN:
  cutoff:
    type: int
    values: [1,5,10]
```

24.3.2 Mean prediction distance

This metric measures how far from the target user the recommended people are. It is computed as the harmonic mean of the reciprocal distances between the target and recommended users:

$$\text{MPD} = \frac{|\hat{E}|}{\sum_{(u,v)\in\hat{E}} \frac{1}{\delta(u,v)}} - 2$$

References:

- J. Sanz-Cruzado, S.M. Pepa, P. Castells. Structural novelty and diversity in link prediction. 9th International Workshop on Modeling Social Media (MSM 2018) at The Web Conference (WWW 2018). The Web Conference Companion, pp. 1347–1351.
- J. Sanz-Cruzado, P. Castells. Beyond Accuracy in Link Prediction. BIAS 2020: Bias and Social Aspects in Search and Recommendation, pp 79-94.

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Mean prediction distance:
  cutoff:
    type: int
    values: [1,5,10]
```

24.3.3 Unexpectedness

This metric measures how different the recommended users are to the contacts the target user already has:

$$\text{Unexp} = \frac{1}{\hat{E}} \sum_{(u,v)\in\hat{E}} \frac{1}{|\Gamma_{out}(u)|} \sum_{w\in\Gamma_{out}(u)} d(v,w)$$

where $\hat{E}$ represents the whole set of recommended links, and $d(u,v)$ is the distance between the users (in this case, represented by one minus the cosine of the feature vector for each user).

$$p(c|u) = \frac{|\{v \in c | (u,v) \in E \cup E_{test}\}|}{\sum_{c' \in \mathcal{C}} |\{v \in c' | (u,v) \in E \cup E_{test}\}|}$$

References:

- P. Castells, N. J. Hurley, and Saúl Vargas. Novelty and Diversity in Recommender Systems. Recommender Systems Handbook, 2nd ed., 881–918 (2015).
- S. Vargas and P. Castells. Rank and Relevance in Novelty and Diversity Metrics for Recommender Systems. 5th ACM Conference on Recommender Systems (RecSys 2011), 109-116 (2011)

Input

In order for this metric to work, the recommendation program must receive a set of features representing the users (or an index, generated by the `TwittomenderIndexGenerator` program).

Parameters

- `cutoff`: the (maximum) number of recommended users to consider in the computation of the metric.

Configuration file

```
Unexpectedness:
  cutoff:
    type: int
    values: [1,5,10]
```

24.4 Metrics

In order to evaluate link prediction approaches, it is common to use some machine learning and classification metrics. We include the following ones:

- *Accuracy*
- *Area under the ROC curve*
- *F1 Score*
- *Precision*
- *Recall*

24.4.1 Accuracy

The accuracy measures the number of correctly classified links.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

where TP is the number of true positives, FP is the number of false positives, TN is the number of true negatives, and FN is the number of false negatives.

Parameters

We have two options here (mutually exclusive):

- `cutoff`: the (maximum) number of predicted links to consider (all the remaining links shall be considered as negatively predicted links).
- `threshold`: the minimum score to consider as positive (all the remaining links shall be considered as negatively predicted links).

When both appear in the configuration file, they will be considered separately.

Configuration file

```
Accuracy:
  cutoff:
    type: int
    values: [1,5,10]
  threshold:
    type: double
    values: [0.2,0.5,1.0]
```

24.4.2 Area under the ROC curve

The area under the receiver operating characteristic curve (AUC), as its name indicates, measures the area under a curve. Such curve shows the rate of true positives as a function of the rate of false positives.

Reference: T. Fawcett. An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874 (2006).

Configuration file

```
AUC:
```

24.4.3 F1 Score

The F1 score combines precision and recall (see *Precision* and *Recall*, respectively) in a single value. It is the harmonic mean of the two measures:

$$\text{F1-score} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FN} + \text{FP}}$$

where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives.

Parameters

We have two options here (mutually exclusive):

- `cutoff`: the (maximum) number of predicted links to consider (all the remaining links shall be considered as negatively predicted links).
- `threshold`: the minimum score to consider as positive (all the remaining links shall be considered as negatively predicted links).

When both appear in the configuration file, they will be considered separately.

Configuration file

```
F1-score:
  cutoff:
    type: int
    values: [1,5,10]
  threshold:
    type: double
    values: [0.2,0.5,1.0]
```

24.4.4 Precision

The precision measures the proportion of correctly predicted links among those the algorithm has label as positive.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

where TP is the number of true positives and FP is the number of false positives.

Parameters

We have two options here (mutually exclusive):

- `cutoff`: the (maximum) number of predicted links to consider (all the remaining links shall be considered as negatively predicted links).
- `threshold`: the minimum score to consider as positive (all the remaining links shall be considered as negatively predicted links).

When both appear in the configuration file, they will be considered separately.

Configuration file

```
Precision:
  cutoff:
    type: int
    values: [1,5,10]
  threshold:
    type: double
    values: [0.2,0.5,1.0]
```

24.4.5 Recall

The recall measures the proportion of correctly predicted links which have been labeled as positive

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

where TP is the number of true positives and FN is the number of false negatives.

Parameters

We have two options here (mutually exclusive):

- `cutoff`: the (maximum) number of predicted links to consider (all the remaining links shall be considered as negatively predicted links).
- `threshold`: the minimum score to consider as positive (all the remaining links shall be considered as negatively predicted links).

When both appear in the configuration file, they will be considered separately.

Configuration file

```
Recall:
  cutoff:
    type: int
    values: [1,5,10]
  threshold:
    type: double
    values: [0.2,0.5,1.0]
```

CHAPTER

TWENTYFIVE

INFORMATION DIFFUSION

Online social networks are dynamic environments where using are constantly creating and sharing new information: that is the case of tweets on Twitter, posts on Facebook and Tumblr, etc. The diffusion of these user-generated contents has been object of study for many decades. RELISON provides a diffusion simulator which can be used to perform diffusion experiments.

We provide information about how to perform these simulations in this section.

Everything needed for the simulation of information diffusion is included in the diffusion module of the framework, which can be imported using Maven as:

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-diffusion</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

TWENTYSIX

SIMULATION DESCRIPTION

In order to use the information diffusion simulations, it is important to understand how they work, so they can be properly configured. Our simulations consider that time is discrete: they are divided in time steps, where information is exchanged between the users (no information is spread in the network between two consecutive iterations of the process). Then, in this section, we describe a single information diffusion step.

26.1 Information spread

We illustrate in the following feature the process for spreading information pieces to different people in the network:

In the figure, the woman in the left represents the user who is going to propagate some information in the network. At each time step, she has three different lists of contents she can propagate:

- **Own contents:** These are contents originally created by the user.
- **Received contents:** These contents have been received by the user (and never spread before).
- **Propagated contents:** These contents have already been propagated in past iterations.

At every point of time, the user might spread a selection of the contents included in her three lists. To determine which contents to spread, we use what we have named a **selection mechanism**. For instance, we might choose one item from each list randomly, all of them, etc. The spread contents are moved to the *propagated contents* list.

Once the elements have been selected, it is necessary to choose the users who shall receive these information pieces. This is done by a **propagation mechanism**, who chooses which users in the network might receive each of the information pieces, and propagates the information to them. We shall later explain how users manage to receive and read these user-generated contents.

But, once the user has propagated some information, we pass all the non-propagated information pieces coming from the *received* list, and we establish if we are going to be able to propagate them in future iterations. For this, we use what we call an **expiration mechanism**. This expiration mechanism determines (using some properties of the piece, like the time it has been in the received list, the author of the piece, etc.) whether the user might be able to share the information in the future (and, as such, it returns to the *received* list), or not (and the piece is removed from that list).

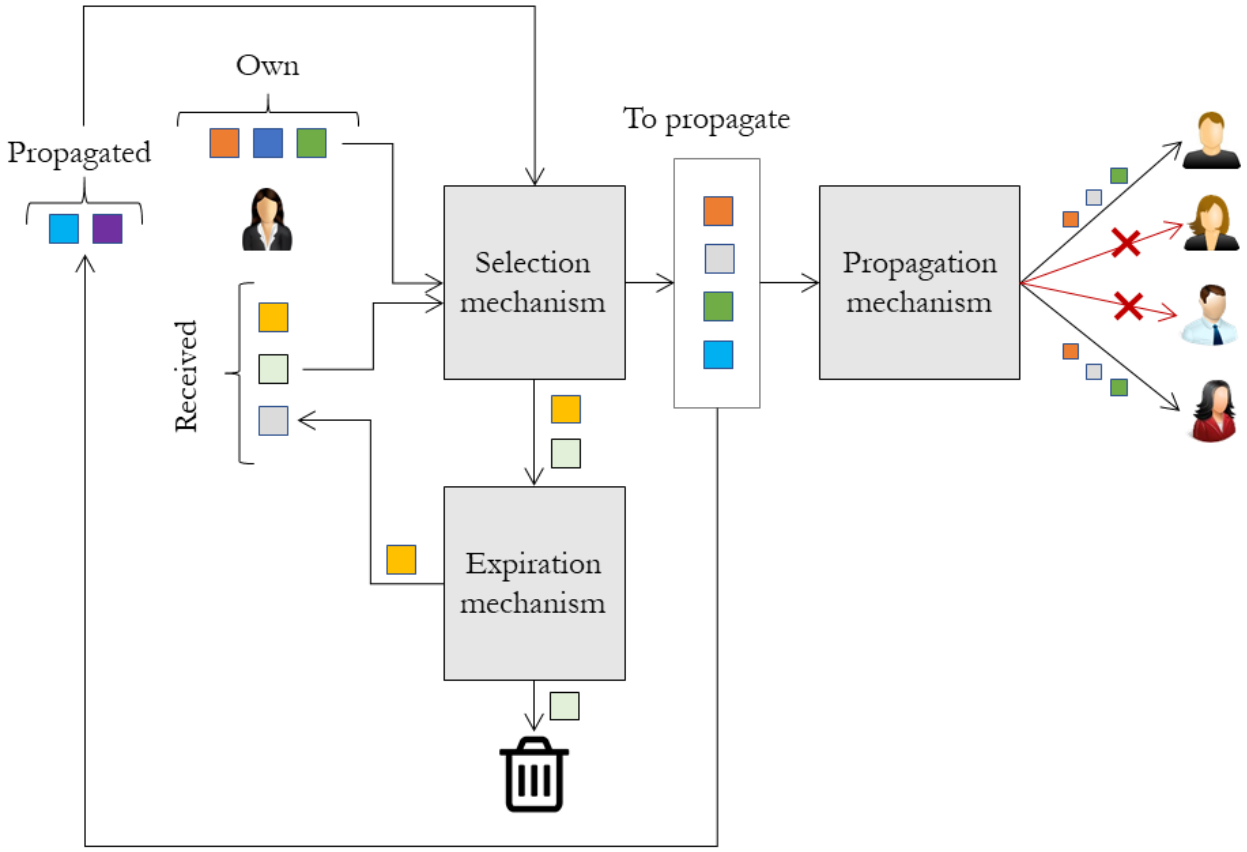

Fig. 1: Example of the spreading information process.

26.2 Information reception

Users store all the received information in a user-generated content list, named the *received list*. We need to describe how pieces manage to reach the *received* list. We illustrate this in the following figure:

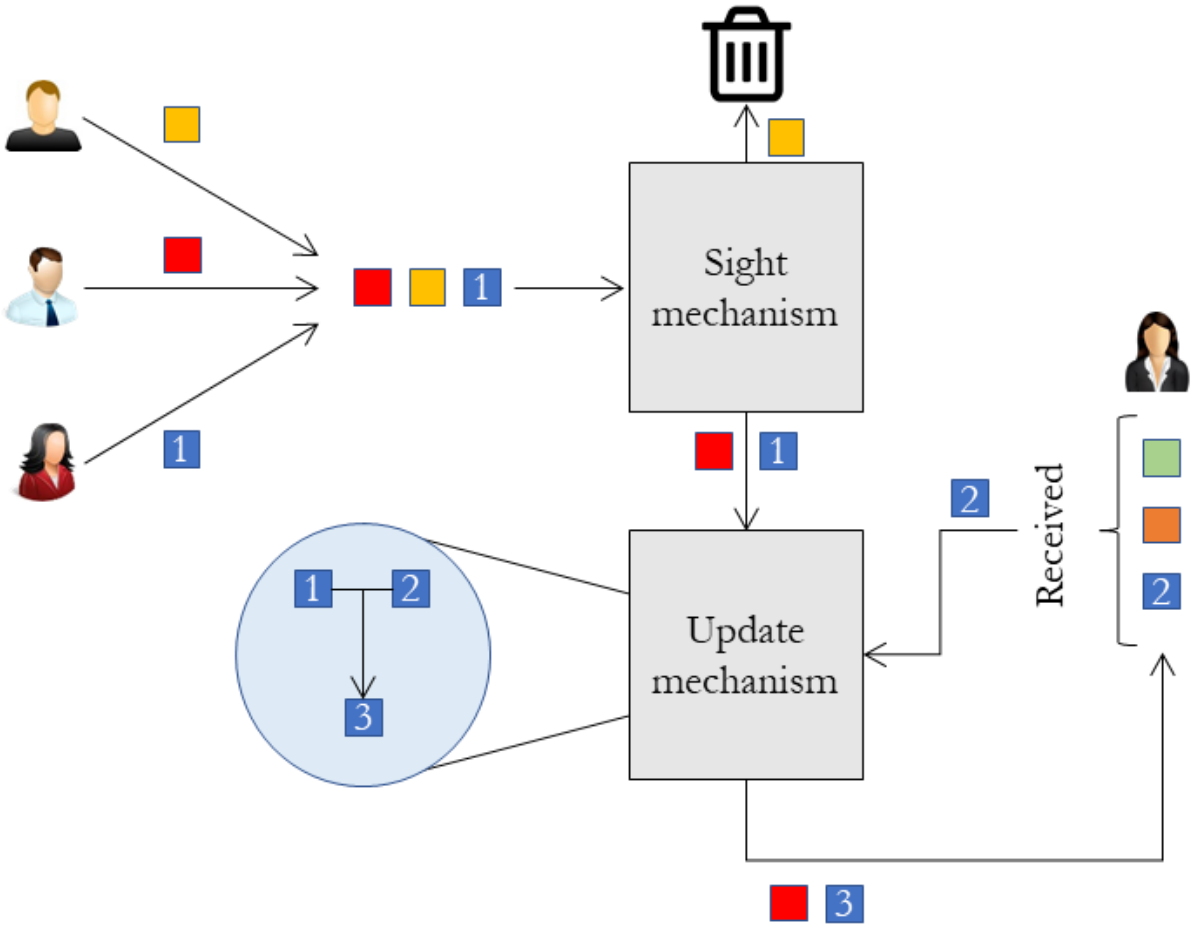

Fig. 2: Example of the receiving information process.

Each iteration, users receive a variable amount of information pieces (for instance, the three pieces in the figure). However, it is uncommon for the people in the network to read all the information is received. Because of this, an attention mechanism is provided, named **sight mechanism** which determines the subset of the received contents which are actually read by the user. Any other piece is discarded, and never reaches the *received list*.

All the information pieces passing this filter are later included in the *received* list. However, it might happen that we received any of the contents in the past (as it is the case of the blue content in the figure). Some of the available meta-information of these user-generated contents might vary (for instance, we might have received it from different users). So, it is interesting to somehow combine this information in order to store it in the *received* list. This is done by what is called an **update mechanism**. The outcome of the update mechanism is then included in the *received list*.

26.3 Use of timestamps

When they are introduced in the simulator, each user-generated content has an specific timestamp. If we also have information about which contents have been forwarded by users different than their creator (e.g. retweets on Twitter, shared posts on Facebook), each forwarded post also comes with a timestamp.

The simulator takes, in the initial iteration, the smallest of these timestamps. Then, each iteration, the simulation increments this timestamp by taking the following available value, until no timestamp is available. These timestamps can be used in the simulation protocol, to define a stop condition, etc.

CHAPTER

TWENTYSEVEN

EXPERIMENTAL CONFIGURATION

RELISON provides a program for executing information diffusion simulations.

```
java -jar RELISON.jar diffusion configuration output numreps network multigraph directed␣
↪weighted selfloops readtypes user-index info-index info (-rec rec-file -n n -test-
↪graph test -userfeats file1,file2,...,fileN -infofeats file1,...,fileN -realprop file -
↪previous file)
```

where:

- `configuration`: a YAML file containing the simulation parameters (See *Configuration file* below).
- `output`: the directory for storing the outcomes of the simulation.
- `numreps`: the number of executions of each simulation.
- `network`: path to a file containing the graph.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `user-index`: a path containing the identifiers of the users (one on each line).
- `info-index`: a path containing the identifiers of the information pieces / user-generated contents (one on each line)
- `info`: a path containing the user-generated contents.
- **Optional arguments:**
  - `-rec rec-file`: path to a recommendation file, whose edges will be added to the network.
  - `-n` n: the number of links (per user) to add from the recommendation (if any). By default: 10.
  - `-test-graph file`: route to a network file containing additional edges (and shall be used for filtering the recommended edges to add).
  - `-userfeats file1,file2,...,fileN`: a comma-separated list of files containing the features for the users in the network (e.g. communities).
  - `-infofeats file1,file2,...,fileN`: a comma-separated list of files containing the features for the information pieces (e.g. hashtags).
  - `-realprop file`: a file indicating which information pieces have been repropagated by users in another information diffusion process.

- `-previous file`: file containing the result of a previous diffusion procedure.

27.1 Configuration file

The configuration file has the following format:

```
simulations:
- protocol:
    name: protocol_name
    type: preconfigured
    params:
      param_name1:
        type: int/double/boolean/string/long/orientation/object
        value: value
        object:
          name: object_name
          params:
            param_name1: <...>
      param_name2:
      <...>
  stop:
    name: stop_condition_name
    params:
      param_name1:
      <...>
  filters:
    filter_name:
      param_name1:
      <...>
- protocol:
    name: protocol_name
    type: custom
    selection:
      name: selection_name
      params:
        param_name1:
          type: <...>
    expiration:
      name: selection_name
      params:
        param_name1:
          type: <...>
    update:
      name: update_name
      params:
        param_name1:
          type: <...>
    propagation:
      name: propagation_name
      params:
        param_name1:
          type: <...>
```

(continues on next page)

(continued from previous page)

```
    sight:
      name: sight_name
      params:
        param_name1:
          type: <...>
- protocol:  <...>
```

As we can see in the previous code, each element in the list corresponds to a different simulation. Each simulation consists on three different elements:

- **filter:** modifies the input data. For instance, it just considers information pieces created before a given timestamp.
- **stop:** the stop condition of the simulation (after no information is propagated, after a given timestamp is reached...)
- **protocol:** the simulation protocol. Indicates how the information travels through the network. We differentiate two types of protocol:
  - *preconfigured*: the protocol is fully implemented in the library
  - *custom*: we build a new protocol by combining its different elements.

27.2 Input files

27.2.1 Information pieces file

The information pieces (individual user-generated contents) file needs to have the following format (CSV divided by tabs):

```
infoId  userId  text  reprCount  likeCount  created  truncated
```

where

- `infoId`: identifier of the information piece.
- `userId`: identifier of the creator.
- `text`: the content of the information piece.
- `reprCount`: number of times the piece has been repropagated.
- `likeCount`: number of likes the piece has been received.
- `created`: UNIX timestamp indicating the date of creation.
- `truncated`: whether we are taking the complete text, or just a small part.

The text must be in UTF-8 format, and user-generated contents are separated by line skips. Fields (like text) which might have tabs or line skips inside must be properly escaped, and surrounded by "".

27.2.2 Real propagated information file

The file indicating which information pieces where repropagated in another diffusion procedure has the following format (divided by tabs)

```
userId  infoId  timestamp
```

where

- `userId`: identifier of the user who repropagated a piece.
- `infoId`: identifier of the repropagated content.
- `timestamp`: UNIX timestamp of the propagation.

27.2.3 Feature files

The files containing information about user / information features have the following format. Each line, they include one user/piece - feature pair, separated by tab.

```
userId/infoId  featureId
```

27.3 Output files

For each simulation, this program generates an output file. However, this output file is binary (and therefore, it cannot be easily read with a text editor). However, it can be read using the provided code.

CHAPTER

TWENTYEIGHT

EVALUATION OF THE INFORMATION DIFFUSION PROCESS

Once the information diffusion has been run, we can evaluate the different properties of the simulation. For this, RELISON provides a program. In order to execute this program, the following command line has to be used:

```
java -jar RELISON.jar diffusion-eval configuration network multigraph directed weighted␣
↪selfloops readtypes user-index info-index input-folder output-folder info (-rec rec-
↪file -n n -test-graph test -userfeats file1,file2,...,fileN -infofeats file1,...,fileN␣
↪-realprop file)
```

where:

- `configuration`: a YAML file containing the parameters for the evaluation (See *Configuration file* below).
- `network`: path to a file containing the graph.
- `multigraph`: true if the network allows multiple edges between each pair of users, false otherwise.
- `directed`: true if the network is directed, false otherwise.
- `weighted`: true if we want to use the weights of the links, false otherwise (weights will be binary).
- `selfloops`: true if we allow links between a node and itself, false otherwise.
- `readtypes`: true if we want to read the types of the edges, false otherwise.
- `user-index`: a path containing the identifiers of the users (one on each line).
- `info-index`: a path containing the identifiers of the information pieces / user-generated contents (one on each line)
- `input-folder`: a directory containing the simulations to evaluate.
- `output-folder`: a directory for storing the evaluations.
- `info`: a path containing the user-generated contents.
- **Optional arguments:**
  - `-rec rec-file`: path to a recommendation file, whose edges will be added to the network. Only used if identifying the recommendation edges is important for evaluating the diffusion.
  - `-n` n: the number of links (per user) to add from the recommendation (if any). By default: 10.
  - `-test-graph file`: route to a network file containing additional edges (and shall be used for filtering the recommended edges to add).
  - `-userfeats file1,file2,...,fileN`: a comma-separated list of files containing the features for the users in the network (e.g. communities).
  - `-infofeats file1,file2,...,fileN`: a comma-separated list of files containing the features for the information pieces (e.g. hashtags).

  - `-realprop file`: a file indicating which information pieces have been repropagated by users in another information diffusion process.

28.1 Configuration file

The configuration file has the following format:

```
metrics:
  metric_name1:
    param_name1:
      type: int/double/boolean/string/long/orientation/object
        value: value
        object:
          name: object_name
          params:
            param_name1: <...>
      param_name2:
      <...>
  metric_name2:
  <...>
distributions:
  distribution_name:
    params:
      param_name1:
      <...>
    times: [iter1,...,iterN]
filters:
  filter_name:
    param_name1:
    <...>
```

As we can see in the previous code, we have to read three different elements:

- **metrics:** contains the metrics to compute in the simulation. The program computes them every iteration, and then outputs them in a file (see *Output files*).
- **distributions:** in addition to metrics, we can find distributions (e.g. the number of times each piece has been received). Differently from metrics, the user has to indicate for which iterations the program should find these distributions. This is indicated in the `times` field.
- **filter:** modifies the input data. For instance, it just considers information pieces created before a given timestamp. It should be the same as in the diffusion procedure.

28.2 Input files

28.2.1 Information pieces file

The information pieces (individual user-generated contents) file needs to have the following format (CSV divided by tabs):

```
infoId  userId  text  reprCount  likeCount  created  truncated
```

where

- `infoId`: identifier of the information piece.
- `userId`: identifier of the creator.
- `text`: the content of the information piece.
- `reprCount`: number of times the piece has been repropagated.
- `likeCount`: number of likes the piece has been received.
- `created`: UNIX timestamp indicating the date of creation.
- `truncated`: whether we are taking the complete text, or just a small part.

The text must be in UTF-8 format, and user-generated contents are separated by line skips. Fields (like text) which might have tabs or line skips inside must be properly escaped, and surrounded by "".

28.2.2 Real propagated information file

The file indicating which information pieces where repropagated in another diffusion procedure has the following format (divided by tabs)

```
userId  infoId  timestamp
```

where

- `userId`: identifier of the user who repropagated a piece.
- `infoId`: identifier of the repropagated content.
- `timestamp`: UNIX timestamp of the propagation.

28.2.3 Feature files

The files containing information about user / information features have the following format. Each line, they include one user/piece - feature pair, separated by tab.

```
userId/infoId  featureId
```

28.3 Output files

We differentiate two types of files: metrics and diffusion files.

28.3.1 Metric files

For a simulation, the format of each line of these files is the following (tab separated): All the metrics for the simulation are displayed in the same file. A file per simulation is generated.

```
numIter metric1 metric2 metric3 ... metricN
```

In the first line, a header will be displayed, indicating which value corresponds to which metric.

28.3.2 Distribution files

The format for the distribution files is (tab-separated):

```
element value
```

CHAPTER

TWENTYNINE

DIFFUSION PROTOCOL SUMMARY

The RELISON framework allows the execution of many different information diffusion protocols. Some of them are pre-defined (following some previous research works). You can find the list of protocols in the following link:

29.1 Protocols

RELISON provides some pre-configured diffusion protocols which can be straightforwardly applied. We list them here and provide a brief description of them:

- *Independent cascade model*
- *Pull*
- *Push*
- *Rumor spreading*
- *Simple*
- *Temporal*
- *Threshold*

29.1.1 Independent cascade model

When diffusion works under this protocol, each time a user receives an information piece, he/she propagates this content to their followers according to a given probability. This probability depends only on the previous user who propagated the information. If the information is not repropagated, then it is discarded.

Reference: J. Goldenberg, B. Libai, and E. Muller. Talk of the Network: A Complex Systems Look at the Underlying Process of Word-of-Mouth, Marketing Letters, 12(3), pp. 211–223 (2001).

Parameters

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `prob`: the probability of sharing a received piece of information (the same for all the users).

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
Independent cascade model:
  prob:
    type: double
    value: value
  numOwn:
    type: int
    value: value
```

29.1.2 Pull

Under this protocol, each iteration, the user selects one of his followees, and obtains the information from that user. The neighbor just propagates a fixed number of their created contents and a fixed number of the received contents.

Reference: A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson. Epidemic algorithms for replicated database maintenance. ACM PODC 1987, pp. 1-12 (1987).

Parameters

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `numRec`: the number of received contents to propagate each iteration.
- `numWait`: number of iterations that the user has to wait before selecting the same neighbor again.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
Pull:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  numWait:
    type: int
    value: value
```

29.1.3 Push

Under this protocol, each iteration, the user selects one of his followers, and propagates him/her information. He/she just propagates a fixed number of their created contents and a fixed number of the received contents.

Reference: A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson. Epidemic algorithms for replicated database maintenance. ACM PODC 1987, pp. 1-12 (1987).

Parameters

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `numRec`: the number of received contents to propagate each iteration.
- `numWait`: number of iterations that the user has to wait before selecting the same neighbor again.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
Push:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  numWait:
    type: int
    value: value
```

29.1.4 Rumor spreading

Also called push-pull protocol, this is a combination of the *Pull* and *Push* protocols, where each user selects a neighbors, and he/she just propagates a fixed number of their created contents and a fixed number of the received contents to the neighbor, and the neighbor sends it to him.

We consider a second version, named *bidirectional rumor-spreading model*, which does not consider the orientation of the edges for selecting the nodes from which receive information and towards whom propagate information.

Reference: A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson. Epidemic algorithms for replicated database maintenance. ACM PODC 1987, pp. 1-12 (1987).

Parameters

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `numRec`: the number of received contents to propagate each iteration.
- `numWait`: number of iterations that the user has to wait before selecting the same neighbor again.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

For the basic rumor spreading model, the configuration is the following:

```
Rumor spreading model:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  numWait:
    type: int
    value: value
```

whereas, for the bidirectional version, it is:

```
Bidirectional rumor spreading model:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  numWait:
    type: int
    value: value
```

29.1.5 Simple

In the simple protocol, each iteration, the user propagates some of his own contents and (at most) a fixed number of the received contents (selected at random) to his followers.

Parameters

- numOwn: the number of contents created by the user to propagate each iteration.
- numRec: the number of received contents to propagate each iteration.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
Simple:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
```

29.1.6 Temporal

The temporal protocol just runs, step by step, how information was propagated in an earlier diffusion: it only propagates a piece at the timestamp it was propagated during the previous diffusion and it only repropagates a piece if the user originally repropagated it.

Parameters

- pure: if true, the user-generated contents are only repropagated if they user received them before the date he repropagated it in the previous diffusion. If he receives it later, he does not propagate it.

Configuration file

```
Temporal:
  pure:
    type: boolean
    value: true/false
```

29.1.7 Threshold

In the threshold model, the users decide to propagate a received piece of content only if a certain threshold of users has sent it to them. We differentiate two variants:

- **Proportion threshold:** the user decides to propagate a piece after more than a given proportion of the users have sent him the content.
- **Count threshold:** the user decides to propagate a piece after more than a given number of the users have sent him the content.

Reference: D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network, KDD 2003, pp. 137–146 (2003).

Parameters

For the probability threshold version, the arguments are the following ones:

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `threshold`: the minimum proportion of users who must send a user-generated content to the user before it can be propagated.

and, for the count threshold version:

- `numOwn`: the number of contents created by the user to propagate each iteration.
- `threshold`: the minimum number of users who must send a user-generated content to the user before it can be propagated.

Configuration file

For the proportion threshold version, the configuration is the following:

```
Proportion threshold:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  threshold:
    type: double
    value: value
```

whereas, for the other version, it is:

```
Count threshold:
  numOwn:
    type: int
    value: value
  numRec:
    type: int
    value: value
  threshold:
    type: int
    value: value
```

However, in order to make the diffusion module configurable, it is also possible to define custom diffusion protocols, built from their fundamental elements, as described in the Simulation description section. The framework already defines some of this parts, which can be found below:

29.2 Expiration mechanisms

During the simulations, users receive many information pieces. However, they do not share them all with their contacts in the same moment they receive them: sometimes, they are never propagated; other times, they are propagated many iterations later (according to other factors, like the amount of people who has sent them the information). Expiration mechanisms decide whether a piece which has been received but not propagated can still be shared in the upcoming iterations.

RELISON has implemented many expiration mechanisms, which can be used to build custom protocols. We provide them a brief explanation below.

- *All not propagated*
- *All not real propagated*
- *All not real propagated with timestamp*
- *Exponential decay*
- *Infinite time*
- *Timed*

29.2.1 All not propagated

Under this expiration mechanism, every user-generated content which has been received but not repropagated is discarded.

Configuration file

```
expiration:
  name: All not propagated
```

29.2.2 All not real propagated

Under this expiration mechanism, the only information pieces that can still be shared in the future are those which were actually shared during a previous diffusion process.

Configuration file

```
expiration:
  name: All not real propagated
```

29.2.3 All not real propagated with timestamp

This expiration mechanism follows the *All not real propagated* one, but with a twist. Under this expiration mechanism, the only information pieces that can still be shared in the future are those which were actually shared during a previous diffusion process if and only if the timestamp of the repropagation has not yet passed.

Configuration file

```
expiration:
  name: All not real propagated timestamp
```

29.2.4 Exponential decay

In this expiration mechanism, each information piece has a probability of staying on the list over time, following and exponential distribution: The probability that an information piece can still be propagated after t iterations is:

$$p(i) = e^{-\lambda t}$$

where λ is what we call the "half-life" of the content, i.e. the time required for the piece to have a 50% probability of being removed.

Parameters

- `half-life`: the time required for the piece to have a 50% probability of being discarded.

Configuration file

```
expiration:
  name: Exponential decay
  params:
    half-life:
      type: double
      value: value
```

29.2.5 Infinite time

In this expiration mechanism, information pieces are never discarded: once the user has received them, they remain in the received list until the user decides to share it.

Configuration file

```
expiration:
  name: Infinite time
```

29.2.6 Timed

When this expiration mechanism is used, a received information piece has to be shared before a fixed time has passed. Otherwise, it is discarded.

Parameters

- `max-time`: the number of iterations before an information piece is discarded.

Configuration file

```
expiration:
  name: Timed
  params:
    max-time:
      type: int
      value: value
```

29.3 Propagation mechanisms

The propagation mechanism selects the people in the network who shall receive each separate information piece. RELISON provides the following propagation mechanisms:

- *All neighbors*
- *All recommended neighbors*
- *Pull*
- *Pull-push*
- *Pull-push pure recommended*
- *Pull-push recommended*
- *Push*

29.3.1 All neighbors

This mechanism sends the information pieces to all the neighbors of the user.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

Configuration file

```
propagation:
  name: All neighbors
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
```

29.3.2 All recommended neighbors

This mechanism sends the information pieces to all the recommended neighbors of the user (i.e. it only propagates information through links created by a link recommendation).

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

Configuration file

```
propagation:
  name: All recommended neighbors
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
```

29.3.3 Pull

In this mechanism, each user selects one of his/her neighbors. Then, the selected neighbor sends him/her the information. After a neighbor is selected, some time needs to pass before it can be selected again.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `waitTime`: the amount of time that needs to pass before a neighbor can be selected again.

Configuration file

```
expiration:
  name: Pull
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    waitTime:
      type: int
      value: num_iter
```

29.3.4 Pull-push

In this mechanism, we combine the *Push* and the *Pull* mechanisms: here, each user selects a neighbor. Then, he/she sends the information pieces to that neighbor, and the neighbor sends the information to him/her. After a neighbor is selected, some time needs to pass before it can be selected again.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `waitTime`: the amount of time that needs to pass before a neighbor can be selected again.

Configuration file

```
expiration:
  name: Push-pull
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    waitTime:
      type: int
      value: num_iter
```

29.3.5 Pull-push pure recommended

This mechanism is just a version of the *Pull-push* propagation mechanism, that limits the neighbor selection to the links created by a recommendation algorithm.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `waitTime`: the amount of time that needs to pass before a neighbor can be selected again.

Configuration file

```
expiration:
  name: Pull-push pure recommended
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    waitTime:
      type: int
      value: num_iter
```

29.3.6 Pull-push recommended

This mechanism is just a version of the *Pull-push* propagation mechanism, that, with a given probability p, selects a neighbors between the connections by a recommendation algorithm, and, with probability $1 - p$, selects one of the original neighbors of the user.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `waitTime`: the amount of time that needs to pass before a neighbor can be selected again.
- `prob`: the probability of selecting a recommended neighbor.

Configuration file

```
propagation:
  name: Pull-push recommended
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    waitTime:
      type: int
      value: num_iter
    prob:
      type: double
      value: probability
```

29.3.7 Push

In this mechanism, each user selects one of his/her neighbors. Then, he/she sends all the information pieces to that neighbor. After a neighbor is selected, some time needs to pass before it can be selected again.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `waitTime`: the amount of time that needs to pass before a neighbor can be selected again.

Configuration file

```
propagation:
  name: Push
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    waitTime:
      type: int
      value: num_iter
```

29.4 Selection mechanisms

The selection mechanism identifies the information pieces which the users will propagate each iteration. RELISON provides the following mechanisms:

- *Count*
- *Independent cascade model*
- *Only own*
- *Pull-push*
- *Real propagated*
- *Recommender*
- *Threshold*
- *Timestamp-based*
- *Timestamp-ordered*

29.4.1 Count

This mechanism selects, at a given time, a fixed number of pieces created by the user, a fixed number of the received pieces and a fixed number of the already propagated pieces to send to other users. Pieces are randomly selected.

Parameters

- `numOwn`: the number of user-created information pieces to propagate.
- `numRec`: the number of received pieces to propagate.
- `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

Configuration file

```
selection:
  name: Count
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    numRec:
      type: int
      value: number_of_received_pieces
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.4.2 Independent cascade model

This mechanism selects, at a given time, for each user, a fixed number of pieces created by the user and a fixed number of the already propagated pieces to send to other users. Then, propagates received pieces with a probability that depends only on the user who sent him the conent.

Parameters

- `numOwn`: the number of user-created information pieces to propagate.
- `prob`: the probability of propagating an information piece.
- `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
selection:
  name: Independent cascade model
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    prob:
      type: double
      value: probability
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.4.3 Only own

This mechanism only propagates information pieces created by the user. It selects a fixed number of them (chosen randomly).

Parameters

- `numOwn`: the number of user-created information pieces to propagate.
- `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

Configuration file

```
selection:
  name: Only own
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.4.4 Pull-push

This mechanism makes a user share all the information he knows (i.e. all the received information and all the information he has propagated in the past). Along with this, it selects a fixed number of their own (not already propagated) contents.

Parameters

- numOwn: the number of user-created information pieces to propagate.

Configuration file

```
selection:
  name: Push-pull
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
```

29.4.5 Real propagated

This mechanism selects, at a given time, a fixed number of the pieces created by the user, a fixed number of already propagated pieces, and the user repropagates a selection of the received information pieces that he/she forwarded in a previous diffusion process (for example, real retweets extracted from Twitter).

Depending on how many of the received information pieces that the user propagates, we differentiate two variants: * **all**: each time the user receives a piece he / she forwarded in a previous diffusion process, he / she propagates it. * **count**: it just selects up to a maximum number of those pieces.

Parameters

Both variants share the following parameters * `numOwn`: the number of user-created information pieces to propagate. * `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

The **count** version has an additional parameter: * `numRec`: the number of received pieces to propagate.

Configuration file

The version that propagates all the received pieces passing the filter has the following configuration file: .. code:: yaml

```
selection:
  name: All real propagated
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

while the version that propagates only some of them:

```
selection:
  name: Count real propagated
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.4.6 Recommender

This mechanism selects, at a given time, a fixed number of the pieces created by the user. At the moment of selecting which pieces to propagate from the received ones, it also selects (at most) a fixed number, but considers the origin of those pieces. With a given probability p, it chooses a piece received from a link created via a recommendation. With $1 - p$, it selects a piece from one of the original links.

Depending on how the selection is done, we differentiate three variants:

- **basic**: before choosing an information piece, a coin is tossed to determine from which list the information piece to propagate will be selected.
- **batch**: each iteration, for each user, it is selected whether all the pieces are selected from the set of recommended links or from the set of original links.
- **pure**: here, we take $p =$, i.e. pieces always come from the recommended links.

Parameters

All variants share the following parameters: * `numOwn`: the number of user-created information pieces to propagate. * `orientation`: the neighborhood to consider.

- `IN`: it considers the incoming neighborhood of the target user.
- `OUT`: it considers the outgoing neighborhood of the target user.
- `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
- `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

And the basic and batch variants also have the following one: * `prob`: probability of choosing pieces which have been sent through recommended links.

Configuration file

The basic version has the following configuration file:

```
selection:
  name: Recommender
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    numRec:
      type: int
      value: number_of_received_pieces
    prob:
      type: double
      value: probability
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

while the batch version has:

```
selection:
  name: Batch recommender
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    numRec:
      type: int
      value: number_of_received_pieces
    prob:
      type: double
      value: probability
    orientation:
      type: orientation
```

(continues on next page)

(continued from previous page)

```
      value: IN/OUT/UND/MUTUAL
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

and the pure one:

```
selection:
  name: Pure recommender
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    numRec:
      type: int
      value: number_of_received_pieces
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.4.7 Threshold

In the threshold selection mechanism, the users decide to propagate a received piece of content only if a certain threshold of users has sent it to them. We differentiate two variants:

- **Probability threshold:** the user decides to propagate a piece after more than a given proportion of the users have sent him the content.
- **Count threshold:** the user decides to propagate a piece after more than a given number of the users have sent him the content.

Then, for each of them, we consider two possibilities, depending on how many pieces that overcome the threshold are shared: * **all**: we propagate all the received pieces that pass the filter. * **limited**: we propagate (at most) a fixed number of them.

Reference: D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network, KDD 2003, pp. 137–146 (2003).

Parameters

All versions receive the following parameters: * `numOwn`: the number of contents created by the user to propagate each iteration. * `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

In the probability thresold version, we have this additional parameters * `threshold`: the minimum proportion of users who must send a user-generated content to the user before it can be propagated. * `orientation`: the neighbor selection.

- `IN`: it considers the incoming neighborhood of the target user.
- `OUT`: it considers the outgoing neighborhood of the target user.
- `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)

- `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

whereas we have the following one for the count version: * `threshold`: the minimum number of users who must send a user-generated content to the user before it can be propagated.

Finally, both limited versions have this parameter: * `numRec`: the maximum number of received pieces to propagate.

Configuration file

For the probability threshold version, the configuration is the following:

```
selection:
  name: Proportion threshold
  params:
    numOwn:
      type: int
      value: value
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
    threshold:
      type: double
      value: value
    (numRepr:
      type: int
      value: value)
```

whereas, for the count version, it is:

```
selection:
  name: Count threshold
  params:
    numOwn:
      type: int
      value: value
    threshold:
      type: int
      value: value
    (numRepr:
      type: int
      value: value)
```

Then, the limited version of the probability threshold is:

```
selection:
  name: Limited proportion threshold
  params:
    numOwn:
      type: int
      value: value
    numRec:
      type: int
      value: value
    orientation:
```

(continues on next page)

(continued from previous page)

```
      type: orientation
      value: IN/OUT/UND/MUTUAL
    threshold:
      type: double
      value: value
    (numRepr:
      type: int
      value: value)
```

and the limited count threshold:

```
selection:
  name: Limited count threshold
  params:
    numOwn:
      type: int
      value: value
    numRec:
      type: int
      value: value
    threshold:
      type: int
      value: value
    (numRepr:
      type: int
      value: value)
```

29.4.8 Timestamp-based

In these selection mechanisms, a user does only propagate his own content when the timestamp of the simulation corresponds to the timestamp on a previous diffusion process.

Depending on how the received pieces are shared with other users, we differentiate two cases: * **Loose**: in this case, a received information piece is shared a) only if the user shared it in a previous process and b) only if the current timestamp is smaller or equal to the timestamp at which the user shared it in that process. * **Pure**: in this case, a received information piece is shared a) only if the user shared it in a previous process and b) only if the current timestamp is smaller or equal to the timestamp at which the user shared it in that process.

Configuration file

For the loose version, the configuration is the following:

```
selection
  name: Loose timestamp-based
```

whereas, for the pure version, it is:

```
selection
  name: Pure timestamp-based
```

29.4.9 Timestamp-ordered

This selection mechanism is equivalent to *Count*, but, pieces are selected according to their creation / reception timestamps.

Parameters

- `numOwn`: the number of user-created information pieces to propagate.
- `numRec`: the number of received pieces to propagate.
- `numRepr`: (*OPTIONAL*) the number of already propagated pieces to repropagate again. By default: 0.

Configuration file

```
selection:
  name: Timestamp-ordered
  params:
    numOwn:
      type: int
      value: number_of_own_pieces
    numRec:
      type: int
      value: number_of_received_pieces
    (numRepr:
      type: int
      value: number_of_pieces_to_repropagate_again)
```

29.5 Sight mechanisms

Once some information pieces have been received, users do not pay the same attention to all of them: they just read some of them, who they might be interested in. This mechanism models this: it allows to select which information pieces are observed by the user. We provide the following sight mechanisms in RELISON:

- *All*
- *All not discarded*
- *All not propagated*
- *All not discarded nor propagated*
- *All recommended*
- *All train*
- *Count*
- *Recommended*

29.5.1 All

This mechanism makes users see all the information pieces they have received.

Configuration file

```
sight:
  name: All
```

29.5.2 All not discarded

The users read all the received user-generated contents which have not been discarded earlier by the expiration mechanism.

Configuration file

```
sight:
  name: All not discarded
```

29.5.3 All not propagated

The users read all the received user-generated contents which have not been propagated earlier by the user.

Configuration file

```
sight:
  name: All not propagated
```

29.5.4 All not discarded nor propagated

A combination of the *All not discarded* and *All not propagated* mechanisms: a user sees all the information pieces who he has never propagated or discarded in earlier iterations.

Configuration file

```
sight:
  name: All not propagated nor discarded
```

29.5.5 All recommended

When this mechanism is used, each user sees all the information pieces coming from recommended users, and which have not previously propagated by the user.

Parameters

- `orientation`: the neighborhood to consider.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

Configuration file

```
sight:
  name: All recommended
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
```

29.5.6 All train

When this mechanism is used, each user sees all the information pieces not coming from recommended users, and which have not previously propagated by the user.

- `IN`: it considers the incoming neighborhood of the target user.
- `OUT`: it considers the outgoing neighborhood of the target user.
- `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
- `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)

Configuration file

```
sight:
  name: All train
  params:
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
```

29.5.7 Count

Each user sees, at most, a pre-determined number of the received information pieces.

Parameters

- `numSight`: the maximum number of pieces each user sees.

Configuration file

```
sight:
  name: Count
  params:
    nuSight:
      type: int
      value: number_of_pieces
```

29.5.8 Recommended

This mechanism defines two probabilities: a probability for seeing pieces coming from recommended links and a probability for seeing pieces coming from the rest of the users.

Parameters

- `probRec`: the probability of observing coming from links created via a recommendation
- `orientation`: the neighborhood from which information pieces come. It is used to determine which of the users come from a recommendation and which of them do not.
  - `IN`: it considers the incoming neighborhood of the target user.
  - `OUT`: it considers the outgoing neighborhood of the target user.
  - `UND`: it considers the all the possible neighbors of the target users ($\Gamma_{out}(u) \cup \Gamma_{in}(u)$)
  - `MUTUAL`: it considers as neighbors those who share a reciprocal link with the target user ($\Gamma_{out}(u) \cap \Gamma_{in}(u)$)
- `probTrain`: the probability of observing an information piece from the original links in the network.

Note: when used in code, there is another option, which takes as input a graph containing the probabilities. However, this option is not (at the moment) available from a configuration file.

Configuration file

```
selection:
  name: Recommended
  params:
    probRec:
      type: double
      value: probability
    probTrain:
      type: double
```

(continues on next page)

(continued from previous page)

```
      value: probability
    orientation:
      type: orientation
      value: IN/OUT/UND/MUTUAL
```

29.6 Update mechanisms

During the diffusion process, it is likely for a user to receive the same information several times. Each iteration, the received information piece contains the following information: the identifiers of the users who have sent him the information in the corresponding iteration and the timestamp of the interaction.

However, when the user has already received the information piece in previous iterations, it is necessary to combine them into just one piece. This is the task of the update mechanism: select which information is actually used in the diffusion process.

RELISON provides the following mechanisms:

- *Merger*
- *Newest*
- *Oldest*

29.6.1 Merger

In this case, all the information pieces are merged into one: the user can access all the people who sent him the information piece during the diffusion process. The timestamp of the reception is the timestamp of the last iteration he/she received the piece.

```
update:
  name: Merger
```

29.6.2 Newest

In this case, the user only knows information about the newest received information piece (i.e. it can access the people who have sent this information piece in the last iteration, and the timestamp of reception is the one of this last iteration).

Configuration file

```
update:
  name: Newest
```

29.6.3 Oldest

In this case, the user only knows information about the oldest received information piece (i.e. it can access the people who have sent this information piece first, and the timestamp of reception is the one of the first time he/she received it).

Configuration file

```
update:
  name: Oldest
```

In addition, it is necessary to provide, for each simulation, two additional elements: a set of filters to preprocess the input data for the simulations, and a stop condition, which determines when the simulation ends. We list them below:

29.7 Stop conditions

The stop conditions determine when the simulation ends, according to the properties of the diffusion. We consider different options for the RELISON framework:

- *Max. timestamp*
- *No more new*
- *No more propagated*
- *No more timestamps*
- *No more timestamps nor propagated information*
- *Num. iter*
- *Total propagated*

29.7.1 Max. timestamp

The simulation stops a) when a certain timestamp is reached or b) when no more timestamps are available.

Parameters

- `max`: the maximum timestamp.

Configuration file

```
stop:
  Max. timestamp:
    max:
      type: long
      value: maximum_timestamp
```

29.7.2 No more new

The simulation stops when no more new information is propagated (i.e. people might still propagate information, but the recipients already know that information).

Configuration file

```
stop:
  No more new:
```

29.7.3 No more propagated

The simulation stops when no information has been propagated during the last iteration.

Configuration file

```
stop:
  No more propagated:
```

29.7.4 No more timestamps

The simulation stops when no more timestamps are available.

Configuration file

```
stop:
  No more timestamps:
```

29.7.5 No more timestamps nor propagated information

The simulation stops when no more timestamps are available or when users are not propagating information to others.

Configuration file

```
stop:
  No more timestamps nor propagated:
```

29.7.6 Num. Iter

The simulation stops when a fixed number of iterations have passed.

Parameters

- `numIter`: the number of simulation iterations to run.

Configuration file

```
stop:
  Num. iter:
    numIter:
      type: int
      value: max_iter
```

29.7.7 Total propagated

The simulation stops when a fixed amount of pieces have been propagated.

Parameters

- `limit`: the number of pieces which need to be propagated.

Configuration file

```
stop:
  Num. iter:
    limit:
      type: long
      value: num_pieces
```

29.8 Information filters

The information filters allow the preprocessing of the information used in the diffusion simulations. RELISON provides the following ones:

- *Basic*
- *Creator*
- *Empty feature*
- *Information feature*
- *Information feature selection*
- *Minimum information feature frequency*
- *Num. information pieces*

- *Only repropagated pieces*
- *Relevant edges*

29.8.1 Basic

This filter does not modify the data. It returns the same, original data.

Configuration file

```
Basic:
```

29.8.2 Creator

This filter only keeps information pieces with information about the user who created them.

Configuration file

```
Creator:
```

29.8.3 Empty feature

This filter adds an empty feature for each information piece without features.

Configuration file

```
Empty feature:
```

29.8.4 Information feature

This filter only leaves those information pieces containing a given feature.

Parameters

- `feature`: the name of the feature

Configuration file

```
Information feature:
  feature:
    type: string
    value: feature_name
```

29.8.5 Information feature selection

This filter only keeps some information feature values, and the information pieces containing those values.

Parameters

- `feature`: the name of the feature
- `file`: a file containing the feature values to keep (one each line)

Configuration file

```
Information feature selection:
  feature:
    type: string
    value: feature_name
  file:
    type: string
    value: file_route
```

29.8.6 Minimum information feature frequency

This filter only keeps the information features which appear, at least, in a fixed number of information pieces. It only keeps information pieces containing those features.

Parameters

- `feature`: the name of the feature
- `minValue`: a file containing the feature values to keep (one each line)

Configuration file

```
Minimum information feature frequency:
  feature:
    type: string
    value: feature_name
  minValue:
    type: int
    value: value
```

29.8.7 Num. information pieces

This filter only keeps a limited number of information pieces for each user. In case a user has more user-generated contents, it selects the given amount randomly.

Parameters

- `numPieces`: the maximum number of information pieces to keep for each user.

Configuration file

```
Num. information pieces:
  numPieces:
    type: int
    value: value
```

29.8.8 Only repropagated pieces

This filter keeps only those information pieces which have been propagated by users different than their creator in a previous diffusion process.

Configuration file

```
Only repropagated:
```

29.8.9 Relevant edges

If recommendation links have been added to the original graph, we only keep those recommended edges which were relevant.

Configuration file

```
Relevant edges:
```

CHAPTER

THIRTY

INFORMATION DIFFUSION METRICS SUMMARY

RELISON provides many tools which can be used to analyze the properties of the information diffusion properties. We differentiate two types:

- **Metrics**: provide a single value for all the iteration. We summarize the available metrics below:

30.1 Creator metrics

This group of metrics measures properties of the information diffusion by considering the creators of the contents which are propagated through the network. RELISON includes the following metrics:

- *Creator entropy*
- *Creator Gini complement*
- *Creator recall*

30.1.1 Creator entropy

The creator entropy measures the entropy of the distribution that counts the times that information created by each user has been propagated. We differentiate two variants:

- **Individual:** we measure the number of times each user has received information from each creator, and find the entropy of that distribution. The final result is the average over all the users in the network.
- **Global:** for each creator, we count the number of times that one of his/her created contents has reached a user in the network. Then, the entropy is computed over that distribution.

Parameters

- `unique`: true if we only count the first time an information piece has been received by a user, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual creator entropy:
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global creator entropy:
  unique:
    type: boolean
    value: true/false
```

30.1.2 Creator Gini complement

The creator Gini complement measures how balanced is the number of times that the information created by each user has been propagated. We differentiate two variants:

- **Individual:** we measure the number of times each user has received information from each creator, and find the Gini complement of that distribution. The final result is the average over all the users in the network.
- **Global:** for each creator, we count the number of times that one of his/her created contents has reached a user in the network. Then, the Gini complement is computed over that distribution.

Parameters

- `unique`: true if we only count the first time an information piece has been received by a user, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual creator Gini complement:
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global creator Gini complement:
  unique:
    type: boolean
    value: true/false
```

30.1.3 Creator recall

For each user in the network, this metric computes the proportion of users in the network who have created an information piece which has been received by the user. This metric indicates the fraction of people in the network that each user has discovered through user-generated contents. The value is then averaged over all the user.

Configuration file

```
Creator recall:
```

30.2 Information pieces metrics

This group of metrics measures properties of the information diffusion by considering the user-generated contents (information pieces) which are propagated through the network.

- *Information count*
- *Information Gini complement*
- *Real propagated recall*
- *Speed*

30.2.1 Information count

This metric measures the average number of information pieces which the users have received over time.

Configuration file

```
Information count:
```

30.2.2 Information Gini complement

This metric measures the number of times each information piece has been received, and it finds how balanced the distribution is, by computing the Gini complement.

Configuration file

```
Information Gini complement:
```

30.2.3 Real propagated recall

This metric measures the proportion of information pieces which have shared by other users in a previous diffusion process which have been received. We differentiate two variants:

- **Individual:** we measure the proportion of the information pieces propagated by each user in the previous process which have already been received by the user.
- **Global:** we measure the global proportion of the information pieces propagated in the previous process which have already been received by the corresponding users.

Parameters

- `unique`: true if we only count the first time an information piece has been received by a user, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual real propagated recall:
```

whereas, for the global version, it is:

```
Global real propagated recall:
```

30.2.4 Creator Gini complement

The creator Gini complement measures how balanced is the number of times that the information created by each user has been propagated. We differentiate two variants:

- **Individual:** we measure the number of times each user has received information from each creator, and find the Gini complement of that distribution. The final result is the average over all the users in the network.
- **Global:** for each creator, we count the number of times that one of his/her created contents has reached a user in the network. Then, the Gini complement is computed over that distribution.

Parameters

- `unique`: true if we only count the first time an information piece has been received by a user, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual creator Gini:
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global creator Gini:
  unique:
    type: boolean
    value: true/false
```

30.2.5 Speed

The speed of a simulation measures the number of contents which have been propagated and seen during the simulation (considering that each information piece is received only once by a user).

Configuration file

```
Speed:
```

30.3 Feature metrics summary

This group of metrics measures properties of the information diffusion in terms of the features of the users and the information pieces.

- *External feature Gini complement*
- *External feature rate*
- *External feature recall*
- *Feature entropy*
- *Feature Gini complement*
- *Feature KL divergence*
- *Feature recall*

30.3.1 External feature rate

This metric considers the proportion of the received features which are unknown to the users. We differentiate two cases:

- **Individual**: we consider the proportion of the received features which are unknown to the each user, and then, we average over the set of people in the network.
- **Global**: we globally count the number of received feature unknown to the users.

In order to determine which features are unknown to the users, we do the following: * **Information features**: we consider that a feature is unknown to the user if it does not appear in any of the information pieces created by the user. * **User features**: we consider that a feature is unknown to the user if the user does not have that feature.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

For the individual feature rate, we have the following configuration file:

```
Individual external feature rate:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

and, for the global version:

```
Global external feature rate:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

30.3.2 External feature recall

For each user in the network, this metric computes the proportion of unknown features in the network which have been received. Then, this values are averaged to obtain the metric.

In order to determine which features are unknown to the users, we do the following: * **Information features**: we consider that a feature is unknown to the user if it does not appear in any of the information pieces created by the user. * **User features**: we consider that a feature is unknown to the user if the user does not have that feature.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.

Configuration file

```
External feature recall:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
```

30.3.3 External feature Gini complement

The external feature Gini complement measures how balanced is the number of times that each feature has been propagated. We also count the cases where the feature was unknown to the receiving user. We differentiate two variants:

- **Individual:** we measure the number of times each user has received each feature, and find the Gini complement of that distribution. The final result is the average over all the users in the network.
- **Global:** for each feature, we count the number of times that it has reached a user in the network. Then, the Gini complement is computed over that distribution.

In order to determine which features are unknown to the users, we do the following: * **Information features**: we consider that a feature is unknown to the user if it does not appear in any of the information pieces created by the user. * **User features**: we consider that a feature is unknown to the user if the user does not have that feature.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual external feature Gini complement:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

Global external feature Gini complement:

feature:
type: string value: feature_name

userFeature:
type: boolean value: true/false

unique:
type: boolean value: true/false

30.3.4 Feature entropy

The feature entropy measures the entropy of the distribution that counts the times that each feature has been propagated. We differentiate two variants:

- **Individual:** we measure the number of times each user has received each feature, and find the entropy of that distribution. The final result is the average over all the users in the network.
- **Global:** for each creator, we count the number of times that each feature has reached a user in the network. Then, the entropy is computed over that distribution.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual feature entropy:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global feature entropy:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

30.3.5 Feature Gini complement

The feature Gini complement measures how balanced is the number of times that each feature has been propagated. We differentiate two variants:

- **Individual:** we measure the number of times each user has received each feature, and find the Gini complement of that distribution (independently for each user). The final result is the average over all the users in the network.
- **Global:** for each feature, we count the number of times that it has reached a user in the network. Then, the Gini complement is computed over that distribution.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual feature Gini complement:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global feature Gini complement:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

30.3.6 Feature KL divergence

This metric uses the Kullback-Leibler divergence to compare a) the distribution of times each feature has been propagated during the simulation (the P distribution) and b) an approximation of the actual distribution of the features, estimated by counting the number of pieces created by each user and containing the corresponding features (the Q distribution):

$$KLD = -\sum_{f \in F} P(f) \log \left(\frac{P(f)}{Q(f)} \right)$$

We provide two different options:

- **Individual:** we measure the KL-divergence for each user separately, and then we average.
- **Global:** we measure the distributions for the whole network.

The metric measures the information gain achieved by the estimated distribution.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

For the individual version, the configuration file is the following:

```
Individual feature KLD:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

whereas, for the global version, it is:

```
Global feature KLD:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

30.3.7 Feature recall

For each user in the network, this metric computes the proportion of features in the network which have been received. Then, this values are averaged to obtain the metric.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.

Configuration file

```
Feature recall:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
```

30.3.8 Feature user entropy

For each possible value of a feature, this metric counts the number of users who have received it, and computes the entropy of the distribution.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.

Configuration file

```
Feature user entropy:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
```

30.3.9 Feature user Gini complement

For each possible value of a feature, this metric counts the number of users who have received it, and computes the Gini complement of the distribution, to determine how balanced the distribution is.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.

Configuration file

```
Feature user Gini complement:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
```

30.3.10 User-feature count

This metric measures the number of different (user,feature) pairs which are observed during the diffusion. The (user, feature) pair is only observed if the user receives a content with the corresponding feature.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.

Configuration file

```
User-feature count:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
```

30.3.11 User-feature Gini complement

This metric measures the number of times that different (user,feature) pairs which are observed during the diffusion. The (user, feature) pair is only observed if the user receives a content with the corresponding feature. Then, this metric determines how balanced the distribution over those pairs is.

Parameters

Both metrics receive the same parameters:

- `feature`: the name of the features we are going to use.
- `userFeature`: true if the studied features are user features, false if they are information pieces features.
- `unique`: true if we just count each time a user receives a piece once, false otherwise.

Configuration file

```
User-feature Gini complement:
  feature:
    type: string
    value: feature_name
  userFeature:
    type: boolean
    value: true/false
  unique:
    type: boolean
    value: true/false
```

- **Distribution**: analyze individual properties of the different elements in the diffusion (users, contents, user/content features). An individual value for each of these elements is provided each iteration.

30.4 Distributions

For a group of elements, distribution measure their individual properties in a given moment of the simulation. This framework includes the following distributions:

- *Features*
- *Information*

30.4.1 Features

This distributions measure properties of the different features. We differentiate three distributions here:

- **Information feature**: this distribution measures the amount of times that information pieces containing this feature have been received by the users in the simulation (or, in different words, how many times each feature has been received).
- **User feature**: this distribution measures times pieces created by people with these features have been received.
- **Mixed feature**: this distribution considers how many times information pieces containing a certain information feature and created by a user with a certain user feature have been received.

Parameters

In the case of the individual feature cases (both the information and the user cases), the configuration receives just one parameter:

- `feature`: the name of the user/information feature to use.

In the case of the mixed feature case, we take two:

- `userFeature`: the name of the user feature we are going to use.
- `infoFeature`: the name of the information pieces feature we are going to use.

Configuration file

For the information feature distribution, we have the following configuration file:

```
Information features:
  feature:
    type: string
    value: feature_name
```

for the user feature distribution, it is:

```
User features:
  feature:
    type: string
    value: feature_name
```

and, for the mixed case:

```
Mixed features:
  userFeature:
    type: string
    value: feature_name
  infoFeature:
    type: string
    value: feature_name
```

30.4.2 Information

For each information piece, this distribution measures how many times it has been received.

Configuration file

```
Information:
```

30.4.3 Users

For each user in the network, this distribution measures how many information pieces they have received.

Configuration file

For the individual version, the configuration file is the following:

```
Users:
```

CHAPTER

THIRTYONE

GRAPH GENERATION

RELISON provides algorithms for generating graphs. In this section, we provide some guide on how to do this.

The graph generators are included in the core module of the framework, which can be imported into any Java library using Maven as follows:

```
<dependency>
  <groupId>es.uam.eps.ir</groupId>
  <artifactId>RELISON-core</artifactId>
  <version>1.0.0</version>
</dependency>
```

CHAPTER

THIRTYTWO

GRAPH GENERATION ALGORITHMS

The RELISON framework contains the following graph generators:

- *Complete*
- *Empty*
- *No links*
- *Random*
- *Preferential attachment*
- *Watts-Strogatz*

All the graph generators are configured in the following way:

```
GraphGenerator<U> gen = new GraphGenerator<>() // Substitute here the corresponding␣
↪generator
gen.configure(arg1,arg2,...,argN)
```

Then, the graphs are generated with:

```
Graph<U> graph = gen.generate()
```

32.1 Complete

The complete graph generator creates a network where every pair of nodes is connected. It is defined in the `CompleteGraphGenerator` class.

32.1.1 Arguments

The arguments for configuring a complete graph are:

- `directed`: if the graph to generate is directed.
- `numNodes`: the number of nodes in the network.
- `selfloops`: true if we also add links between a node and itself.
- `generator`: a node generator.

32.2 Empty

The empty graph generator creates a network without links. It is defined in the `EmptyGraphGenerator` class (and the `EmptyMultiGraphGenerator` for multigraphs).

32.2.1 Arguments

The arguments for configuring an empty graph are:

- `directed`: if the graph to generate is directed.
- `weighted`: the number of nodes in the network.

32.3 No links

The empty graph generator creates a network without links. It is defined in the `NoLinkGraphGenerator` class.

32.3.1 Arguments

The arguments for configuring a graph without links are:

- `directed`: if the graph to generate is directed.
- `numNodes`: the number of nodes in the network.
- `generator`: a node generator.

32.4 Preferential attachment

This generator starts with a fully connected set of nodes. Then, iteratively, a new node appears in the network, and creates a fixed amount of links to the nodes already in the network. The link destinations are selected with a probability proportional to their in-degree.

Reference: A-L. Barabási, R. Albert. Emergence of Scaling in Random Networks. Science 286(5439), pp. 509-512 (1999)

32.4.1 Arguments

The arguments for configuring a preferential attachment network are:

- `directed`: true if the graph we want to generate is directed, false otherwise.
- `initialNodes`: the number of initial nodes.
- `numIter`: the number of iterations, i.e. the number of additional nodes to add.
- `numEdges`: the number of edges to add each iteration.
- `generator`: a node generator.

32.5 Random

This generator creates a network where the nodes are connected with a certain probability.

Reference: P. Erdös, A. Rényi. On Random Graphs. I, Publicationes Mathematicae Debrecen 6(1), pp. 290-297 (1959)

32.5.1 Arguments

The arguments for configuring a random Erdös-Renyi network are:

- `directed`: true if the graph we want to generate is directed, false otherwise.
- `numNodes`: the number of nodes in the network.
- `prob`: the probability that two nodes are joined with a link.
- `generator`: a node generator.

32.6 Watts-Strogatz

This generator first creates a ringed network, where each user is connected to other n nodes. Then, with certain probability, each link is randomly reconnected to other node.

Reference: D.J. Watts, S.H. Strogatz. Collective dynamics of 'small-world' networks. Nature 393(6684), pp. 440-442 (1998)

32.6.1 Arguments

The arguments for configuring a random Erdös-Renyi network are:

- `directed`: true if the graph we want to generate is directed, false otherwise.
- `numNodes`: the number of nodes in the network.
- `meanDegree`: the average degree of each node in the initial network (only int values).
- `beta`: the probability of rewiring an edge.
- `generator`: a node generator.