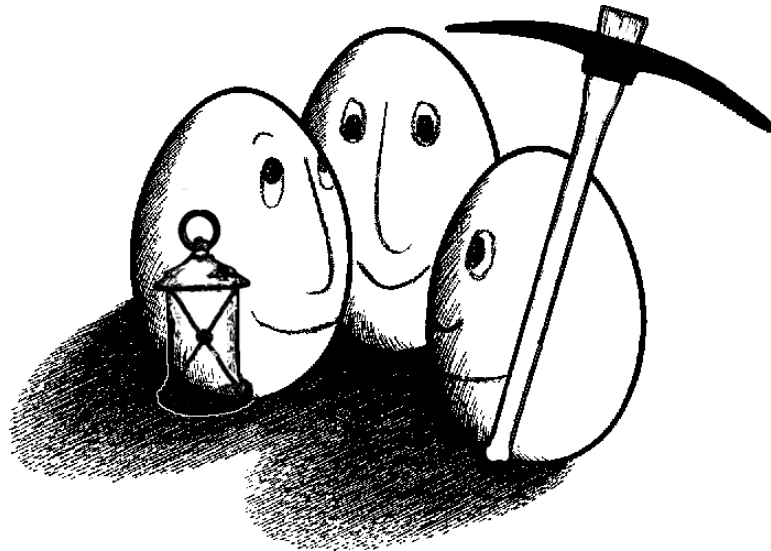

The Yale Distributed Data Mining Simulator

User Guide
Operator Reference
Developer Tutorial



Michael Wurst
wurst@ls8.cs.uni-dortmund.de

University of Dortmund
Department of Computer Science
Chair of Artificial Intelligence
44221 Dortmund, Germany

Contents

1	The Yale Distributed Data Mining Simulator	5
1.1	Introduction	5
1.2	Distributed Data Mining	6
1.3	Network Structures and Communication Patterns	6
1.3.1	Network Structures	6
1.3.2	Low Level Synchronous Communication	7
1.3.3	Higher Level Communication and Cooperation	8
1.3.4	Communication Cost	9
1.4	The Plugin	9
1.4.1	General Structure of an Experiment	9
1.4.2	Sharing Attributes and Examples	12
1.5	Extending the Plugin	16
1.6	Example Applications	17
1.6.1	Distributed Subgroup Discovery	17
2	Operator Reference	19
2.1	Distributed Data Mining	20
2.1.1	AgentIteration	20
2.1.2	ExampleSetConverter	21
2.1.3	GlobalSubgroupAgent	21
2.1.4	NetworkTopologyLoader	22
2.1.5	NetworkTopologySimulator	23
2.1.6	RandomTopologyGenerator	24

2.1.7	RelativeSubgroupAgent	24
2.1.8	SubgroupChain	25
2.1.9	TestAgent	26
2.1.10	TestAgentExampleSharing	27
2.1.11	TestAgentFeatureGeneration	27
2.1.12	TopologyGenerator	28

Chapter 1

The Yale Distributed Data Mining Simulator

1.1 Introduction

Many current data mining tasks can be accomplished successfully only in a distributed setting. The field of distributed data mining has therefore gained increasing importance in the last decade. However, there are still many open questions and challenges. The *YALE* distributed data mining simulator allows to perform distributed data mining experiments in a simple and flexible way. The experiments are not actually executed on distributed network nodes. The plugin only simulate this. Simulation makes it easy to experiment with diverse network structures and communication patterns. Optimal methods and parameters can be identified efficiently before putting the system into use. The network structure can for instance be optimized as part of the general parameter optimization. While this cannot replace testing the system in an actual network, it makes the development stage much more efficient. This follows the general *YALE* philosophy as rapid prototyping tool for large scale data mining applications.

This document gives a brief overview of the simulator, shows how to create experiments and provides points of departure for extending it. Section 1.2 discusses several aspects of distributed data mining. In section 1.3 network structures and communication patterns are briefly introduced. Section 1.4 describes the plugin itself. Finally section 1.5 provides some information and examples of how to extend the plugin.

1.2 Distributed Data Mining

The aim of data mining is to find useful patterns in large data collections. Distributed computing plays an important role in this process for several reasons. First, data mining often requires huge amounts of resources in storage space and computation time. To make systems scalable, it is important to develop mechanisms that distribute the work load among several sites in a flexible way. Second, data is often inherently distributed over several databases, making a centralized processing of this data very inefficient and prone to security risks. Finally, many data mining tasks require connecting heterogeneous resources, as data sources, processing nodes and end user applications.

Typical applications of distributed and grid based data mining include domains, as finance [1] or molecular engineering [2]. The focus of most work in this area has been on scenarios, in which the aim is to find a single, *global* pattern or concept from large, distributed and often heterogeneous data sources [4]. A popular example is distributed mining for association rules [6]. Many big companies collect data about the buying behavior of their costumers from individual branches. A common application is to search for typical buying behavior over all of these branches using distributed data mining. To accomplish this task, usually a data warehouse is build, ensuring the interoperability of all data sources.

There are however also alternative scenarios. One such scenario is collaborative data mining in ad hoc and p2p networks. A large number of loosely coupled nodes applies data mining to different, usually very small and overlapping subsets of the entire data space. The aim is not to learn a general, global pattern to cover all data, but to learn a set of *local* concepts. Each user locally stores a small set of items (e.g. songs). The aim of data mining is to discover interesting structures in each of these local subsets, e.g. groupings of similar items.

A simulation framework for distributed data mining should support several such scenarios, thus being flexible with respect to the data mining tasks, to the data distribution and to the underlying network structure. The YALE distributed data mining simulator is an approach to fulfill these requirements.

1.3 Network Structures and Communication Patterns

1.3.1 Network Structures

A network basically consists of a set of nodes that communicate by exchanging messages. A network is usually modeled as graph (V, E) . Each node $v_i \in V$ can be connected to other nodes in V . The set of nodes that are connected to a node v_i is denoted as $nbrs_i$. If two nodes are connected, they can exchange

messages. If the network is modeled as directed graph, communication can in general be uni-directional. If it is modeled as undirected graph, communication is assumed to be bi-directional.

There are several pattern for network topologies. First, in the client/server model, there is a designated node, the server, that is connected to all nodes via bi-directional communication channels. Second, there are fully connected networks, thus $\forall v, v' \in V : (v, v') \in E$. Finally, nodes can be partially connected. This is a structure found in many peer to peer and ad hoc networks. Often such networks are modeled by making assumptions concerning the distribution of node degrees. Typically, node degrees are assumed to follow a power law distribution. Thus there are few nodes connected to many other nodes and many nodes connected to few nodes. Nodes with high node degree are also referred to as super nodes. They play an important role in the infrastructure, as they enable efficient communication. The simulator allows to create such networks randomly under given distribution parameters.

While the simulation framework allows for uni-directional communication as well, the focus will be bi-directional communication in the rest of this tutorial. Uni-directional communication plays a much less important role in practice, as most network infrastructures allow to establish bi-directional communication channels (e.g. TCP).

1.3.2 Low Level Synchronous Communication

Nodes that are connected by a communication channel may exchange messages. In the following we assume the so called synchronized network model [3]. A communication channel can only hold one message at a time. It is assumed that there is a possibly infinite set of messages M , that can be transferred between nodes connected by a channel. A special message *null* is provided denoting that no message is sent. Each node is represented internally by a state machine with a set of states $states_i$ that is not necessarily finite. Also there are start states $start_i \subseteq states_i$ and terminal states $end_i \subseteq states_i$. Each state is associated with a message generation function $msgs_i : states_i \times neighs_i \rightarrow M \cup \{null\}$ and a state transition function $trans_i : states_i \times M^{|nbrs_i|} \rightarrow states_i$. The model is synchronized as each channel can hold only one message at a time and state changes only occur as messages from all neighbors are received. The model can therefore be executed in the following simple way. At each round, two operations are performed:

1. Apply $msgs_i$ for all nodes and put the messages in the corresponding channels
2. Apply $trans_i$ for each node by reading the messages from the corresponding channels and changing the state of the node

The execution terminates, as all nodes reach a terminal state.

In this basic model, only one message can be exchanged between two nodes per round. The model can be extended to allow several messages per round by combining them to one message. We assume such an extension in the following, thus agents can send arbitrary many messages to their neighbor nodes in each round.

There are two kinds of failures that may occur in such a network: link errors and node errors. Link errors include the modification or loss of a message as well as permutations of the order of messages send from a node A to a node B. Node failures are stopping failures (a node does not send any messages any more) and Byzantine failures (a node sends arbitrary messages possibly not complying to the protocol the nodes agreed upon). Such errors can be easily incorporated into the model.

1.3.3 Higher Level Communication and Cooperation

Range Limited Broadcast

The basic communication model only allows neighboring nodes to exchange messages. If messages should be exchange between nodes that are not directly connected, they must be routed via other nodes. Often it makes sense to simulate such data exchange as atomar action that can be achieved in one round. This can easily be achieved by extending the set of neighbors to nodes that can be reached from the current node, even if this does not reflect the actual structure of the network. A similar task is broadcast. A broadcast message should reach all nodes within a given distance from the current node. Such a maximal distance is usually defined in terms of the maximal number of hops. A hop denotes the forwarding of a message by a node. A node v' is reachable from v if there is a minimal path between v and v' which length is smaller or equal to the maximal number of hops. Often it makes sense to model broadcast as an atomar operation, that can be achieved in one round. This can again be simulated by extending the neighborhoods to all nodes reachable by a certain number of hops.

Blackboards

While message exchange and broadcast represent rather simple networking techniques, blackboards allow for higher level communication. A blackboard is a shared data space from which all nodes may read information and to which they write information. There are several very advanced models for the structure of such blackboards. In the following, a very simple structure is assumed. Each entry on the blackboard is a triple consisting of a *nodeId*, a *tag* and an

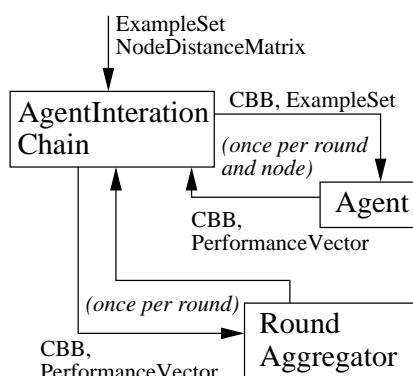


Figure 1.1: The structure of a distributed data mining experiment

object. The *nodeId* represents the node who wrote the item to the blackboard. The *tag* is simply a string describing the object. The *object* is the actual content of the entry. The write and read operation on a blackboard are modeled as atomic communication actions achieved in one round.

1.3.4 Communication Cost

Communication is usually associated with cost. The structure of this cost strongly depends on the application area and the underlying network structure. In general, it is assumed that the cost of a regular message and of a broadcast message depend on the network distance between sender and receiver(s). The cost of the message may also depend on the payload of the message.

Broadcast communication is assumed to be equivalent to exchanging messages with a node that has a network distance of one to each node. The cost is accounted for each object written or received from the blackboard using the same metric as for individual messages. The simulation framework can easily be extended to allow for other forms of communication cost as well.

1.4 The Plugin

1.4.1 General Structure of an Experiment

The plugin simulates distributed data mining assuming a synchronized network model. The simulation is performed in rounds. In each round, each agent is executed exactly once. It may read messages and information from the blackboard, write information to the blackboard and send messages by broadcast or by direct communication to another node. Nodes are identified by integer ids,

ranging from zero to the number of nodes minus one. The number of messages sent per round is not limited (besides it must be finite). Cost is accumulated over all nodes and rounds. After a fixed number of rounds, the simulation stops.

The simulation cycles are performed within an `AgentIterationChain`. A model of such a chain is depicted in figure 1.1. It contains two inner operators. The first one is invoked once for each agent and each round. It should contain the functionality of the individual agents. The second one is invoked at the end of each round. It can be used for example to evaluate the result of a round.

The `AgentIterationChain` needs as input at least a `NodeDistanceMatrix` `IOObject`, that represents the network topology. This `IOObject` is described in 1.4.1. As most data mining experiments are based on data sets, usually an `ExampleSet` object will be required as well. This `ExampleSet` must contain information on how the individual examples are distributed over the network nodes. This is described in 1.4.1.

Communication is achieved by passing an `CommunicationBlackBoard(CBB)` `IOObject` to each agent. This `IOObject` contains all necessary methods for communication and black board access. It is described in 1.4.1.

The Network Structure

The network structure is defined by an `IOObject` of the type `NodeDistanceMatrix`. This object defines not only whether two nodes are connected, but also how distant they are. This information can be used to derive communication costs or to achieve range limited broadcasts. A concrete network structure must implement the interface `NodeDistanceMatrix` (see figure 1.2).

There are several standard implementations for the network structure. Firstly, a network structure can be read from a file. This is achieved with the `NetworkTopologyLoader`. Files must follow the GraphML standard for graphs.

Second, a topology can be generated with the `TopologyGenerator` operator. This operators currently offers a network structure of fully connected nodes and a client/server structure. In the second case, the server is always the node with id zero.

Finally, network topologies can be created randomly with the `RandomTopologyGenerator` operator. This operator offers different modes and parameters for randomly generated networks.

Examplesets and Data Distribution

Each example in the `ExampleSet` must contain a label attribute for each node, denoting firstly, whether this node should obtain the example and secondly how

```
public interface NodeDistanceMatrix extends IOObject {

    public static final int NOT_CONNECTED = -1;

    /**
     * Return distance between two nodes as number of edges
     * on a shortest path between them.
     *
     * @param id1 id of the first node
     * @param id2 id of the second node
     * @return minimal number of edges to reach the second node from the first
     *         one. If this value is NOT_CONNECTED, the second node is not
     *         reachable from the first one
     */
    public int getDistance(int id1, int id2);

    /**
     * Return the total number of nodes
     *
     * @return number of nodes
     */
    public int getNumberOfNodes();

}
```

Figure 1.2: The node distance interface

the example is labeled. Thus each node may learn a different concept. A node obtains each example for which this label is defined. The corresponding attributes must be named as follows `<prefix><id>`, where `<id>` is an integer id ranging from zero to number of nodes minus one and `<prefix>` is user defined and passed as parameter to the `AgentIterationChain`. At each round, an agent then obtains an example set with exactly the examples for which the corresponding label is defined. Supporting an individual attribute per node allows for overlapping data sets. However, in many applications a partition of examples is sufficient. The operator `ExampleSetConverter` converts an `ExampleSet` that contains only a single, integer attribute (denoting which agent should obtain the example) into one that contains a label attribute for each node. This operator is especially utile in combination with a clustering algorithm. Clustering can be applied to the data set to partition it into several subsets. Then the partition is mapped to a nominal feature, which is used create the binary features for each node. Using random clustering, examples can be randomly distributed among nodes.

Communication and Cooperation

The `CommunicationBlackBoard IOObject` contains several communication and access functions. Figure 1.3 and 1.4 depict the corresponding interfaces. Information cannot be removed explicitly from the blackboard, but it can be overwritten, if an agent writes an object with the same tag twice.

Communication cost is calculated by using a class that must be provided as parameter to the `AgentIterationChain`. Such a class must implement the interface `CommunicationCost`. This interface allows to account communication cost depending on the distance of two nodes and on the payload object that is transferred. Figure 1.5 shows the interface.

1.4.2 Sharing Attributes and Examples

One of the most common tasks in many distributed data mining settings is sharing examples among nodes. An example can be transferred by simply adding the corresponding `Example` object as payload to a message. *YALE* provides a powerful management for examples that enables several different views on a single data set. This concept is used by the simulator. If a node receives an example from another agent, it does not actually copy it and adds it to its example set. It merely enables it, thus adds it to its view on the dataset. The `AbstractAgent` class provides a convenience method for this purpose (see figure 1.7).

Another important tasks in distributed data mining is sharing attributes. One node may for example generate a new feature on its local items by supervised

```
public interface CommunicationInterface extends IOObject {

    /**
     * Send a message directly to another agent.
     * If the receiver is not reachable, no cost is accounted.
     * @param type the type of the message
     * @param payload the payload of the message
     * @param receiver the receiver of the agent
     */
    public void sendMessage(String type, Object payload, int receiver);

    /**
     * Broadcast a message to all agents that are reachable.
     * @param type the type of the message
     * @param payload the payload of the message
     */
    public void broadcastMessage(String type, Object payload);

    /**
     * Get all messages of a given type.
     * Note that messages are not deleted from the post box.
     * @param messageType the message type (null for any type)
     * @return an Iterator of Message
     */
    public Iterator<Message> getMessagesByType(String messageType);

}
```

Figure 1.3: The communication interface

```
public interface BlackBoardInterface extends IObject {

    /**
     * Write an object to the black board.
     *
     * @param tag a tag
     * @param obj an object
     */
    public void put(String tag, Object obj);

    /**
     * Obtain objects from the black board
     *
     * @param aid an agent id or -1 for all agents
     * @param tag a tag or null for all tags
     * @param cl a class or null for objects of all types
     * @return an Iterator of Object
     */
    public Iterator get(int aid, String tag, Class cl);

    /**
     * Obtains an object from the local data space.
     *
     * @param key a key identifying the object
     * @return the object
     */
    public Object getFromLocal(String key);

    /**
     * Writer an object to the local data space.
     *
     * @param key a key identifying the object
     * @param obj an object
     */
    public void storeToLocal(String key, Object obj);

}
```

Figure 1.4: The black board interface

```

public interface CommunicationCost {

    /**
     * Get cost for the transfer of a message.
     *
     * @param payload the payload of the message
     * @param numHops the number of edges the messages traverses
     * @return a cost factor
     */
    public double getCost(Object payload, int numHops);

}

```

Figure 1.5: The communication cost interface

Regular Attributes			Agent labels	
f1	f2	f3	a1	a2
0.1	0.3	0.9	?	1
0.3	0.2	0.1	?	0
0.8	0.9	0.9	1	0
0.1	0.3	0.9	1	?
0.9	0.0	0.9	0	?

Figure 1.6: The YALE example management allows for different views on the same data set. The example set of agent 2 (red) contains the first three examples and features f1 and f2. The example set of agent 1 (blue) contains the last three examples and features f2 and f3. Both example sets contain the third example with a different label however.

```

/**
 * Add an example received from another node to the local example set.
 * @param es the local example set
 * @param e the example received from another node
 */
protected void addExample(ExampleSet es, Example e) {...}

/**
 * Add an attribute received from another node to the local example set.
 * @param es the local example set
 * @param att the attribute received from another node
 */
protected void addAttribute(ExampleSet es, Attribute att) {...}
}

```

Figure 1.7: Convenience methods to add attributes and examples received from another agent to the local example set

feature construction. It may now send this attribute to other agents by adding the Attribute object to a message as payload. As for examples, the attribute is not copied but enabled for the receiving agent. There is a convenience method for enabling attributes in an example set as well (see figure 1.7). Note however, that this attribute usually will contain values only for the items that are stored at the node that generated the feature. For all other items, the attribute will contain missing values.

Figure 1.6 shows the YALE data management system. Not duplicating any features or examples makes it possible to work even with several hundreds learning nodes, which would be prohibitively expensive if the data set would be copied for each node. This is a key advantage of the simulation approach compared to performing experiments in an actually distributed environment.

1.5 Extending the Plugin

The plugin is extended by creating operators for individual nodes. Such operators can inherit from AbstractAgent. In all cases they must read a CommunicationBlackBoard object from the input and return this object as output. They may also access the ExampleSet and return a PerformanceVector. All IOObjects returned by the agent, beside the CommunicationBlackBoard and the PerformanceVector are deleted. The AgentIterationChain reads the main performance

criterion from the performance vector and averages it with the performances of the other nodes. The accumulated performance is then passed to the second subchain.

The plugin can also be extended by implementing classes that represent the network topology or communication cost. The interfaces that must be implemented for this purposes are described above.

1.6 Example Applications

1.6.1 Distributed Subgroup Discovery

In the following we describe a prototypical application created with framework. The task is to discover rules that describe a data set that is distributed over several nodes. In particular, the algorithm should in a distributed way find the same rules that would be discovered if the data was collected a single node. The approach is described in detail in [5].

The experiment can be loaded from the sample file available at the project sourceforge homepage.

First, the original data set is loaded. Then it is clustered and the example set is annota transformed into a data set that contains an attribute for each agent. There is a parameter 'random fraction' that allows to adjust the skew in the data distribution by choosing between two extremes: all data points are distributed randomly and all data points are distributed according to their cluster.

In a next step, a network topology is created. In this case, all nodes should be connected with each other.

Then, the actual simulation is invoked. At each round, all agents are called exactly once. The 'GlobalSubgroupAgent' contains the implementation of the distributed subgroup discovery algorithm. You may take a look at the corresponding classes in the source directory¹. The algorithm is also described in detail in [5]. The SubgroupChain takes a class that is used to calculate the communication cost. In the given case, communication cost is determined by the class 'RuleCountCost'. At the end of the simulation, the total communication cost is returned as YALE performance vector.

¹src/edu/udo/cs/yale/dlearning/agents/subgroup

Chapter 2

Operator Reference

2.1 Distributed Data Mining

Distributed Data Mining Simulator.

2.1.1 AgentIteration

Group: Distributed.Core

Required input:

- ExampleSet
- NodeDistanceMatrix

Generated output:

- CommunicationBlackBoard

Parameters

- **num_agents:** Number of agents participating in distributed learning. This should match with the agent attribute in the input example set (integer; $1-\infty$)
- **num_cycles:** the number of cycles that the agent team goes through (integer; $1-\infty$; default: 1)
- **agent_attribute_prefix:** the prefix of the attributes that contain information about which agent gets which part of the examples. Attribute names must follow the scheme `jagent_attribute_prefixjagent_idj` (string)
- **cost_class:** Class that implements a communication cost matrix
- **log_file:** name of the log file (filename)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Inner operators: The inner operators must deliver [PerformanceVector, CommunicationBlackBoard].

Short description: Iterates through a set of agents several times passing a blackboard around and providing each agent with an example set.

Description: This class represents an iteration chain that invokes agents in rounds. In each round, every agent is called exactly once.

2.1.2 ExampleSetConverter

Group: Distributed.Core

Required input:

- ExampleSet

Generated output:

- ExampleSet

Parameters

- **agent_attribute_prefix:** the prefix of the attributes that contain information about which agent gets which part of the examples. Attribute names follow the scheme `agent_attribute_prefixiagent_idi` (string)
- **input_attribute_name:** the name of the nominal attribute that defines which agents gets which example (string)
- **random_fraction:** probability that the node to which an example is assigned is chosen randomly (real; 0.0-1.0)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Converts an example to the agent example set format.

Description: Operator that converts an example set with a single agent attribute to one with an attribute for each agent, as expected by the AgentIterationChain.

2.1.3 GlobalSubgroupAgent

Group: Distributed.Subgroup

Required input:

- CommunicationBlackBoard
- ExampleSet

Generated output:

- CommunicationBlackBoard
- PerformanceVector

Parameters

- **keep_example_set:** Indicates if this input object should also be returned as output. (boolean; default: true)
- **max_depth:** An upper bound for the number of literals. (integer; 1- $+\infty$; default: 2)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Global subgroup discovery agent

Description: Agent that searches for global subgroups in a distributed way.

2.1.4 NetworkTopologyLoader

Group: Distributed.Topology

Generated output:

- NodeDistanceMatrix

Parameters

- **file:** name of the file in graphML format (filename)
- **max_hops:** the time to live for a query message (integer; 1- $+\infty$; default: 1)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Loads a network topology from file. The file must be GraphML format.

Description: Loads a network topology from file. The file must be in GraphML format. Node labels are ignored in the current implementation.

2.1.5 NetworkTopologySimulator

Group: Distributed.Topology

Generated output:

- PerformanceVector

Parameters

- **num_cycles:** Number of random cycles used in the simulation (integer; 1- $+\infty$)
- **num_nodes:** the total number of nodes in the network (integer; 1- $+\infty$; default: 1)
- **num_super_nodes:** the total number of super nodes in the network (integer; 1- $+\infty$; default: 1)
- **num_connects:** the max. number of nodes a node joining the network connects to (integer; 1- $+\infty$; default: 1)
- **max_hops:** the time to live for a query message (integer; 1- $+\infty$; default: 1)
- **prob_super_node_edge:** the probability that two supernodes are connected (real; 0.0-1.0)
- **prob_connect_super_node:** the probability that a node joining the network connects a super node (real; 0.0-1.0)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **reachability:** Avg. number of nodes reachable from a given node
- **time:** The time elapsed since this operator started.
- **traffic:** Avg. number of messages per query

Short description: Simulates a network.

Description: Operator that simulates different topologies.

2.1.6 RandomTopologyGenerator

Group: Distributed.Topology

Generated output:

- NodeDistanceMatrix

Parameters

- **num_cycles:** Number of random cycles used in the simulation (integer; 1- $+\infty$)
- **num_nodes:** the total number of nodes in the network (integer; 1- $+\infty$; default: 1)
- **num_super_nodes:** the total number of super nodes in the network (integer; 0- $+\infty$; default: 1)
- **num_connects:** the max. number of nodes a node joining the network connects to (integer; 1- $+\infty$; default: 1)
- **max_hops:** the time to live for a query message (integer; 1- $+\infty$; default: 1)
- **prob_super_node_edge:** the probability that two supernodes are connected (real; 0.0-1.0)
- **prob_connect_super_node:** the probability that a node joining the network connects a super node (real; 0.0-1.0)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Creates random topologies.

Description: Generates a random network topology.

2.1.7 RelativeSubgroupAgent

Group: Distributed.Subgroup

Required input:

- CommunicationBlackBoard
- ExampleSet

Generated output:

- CommunicationBlackBoard
- PerformanceVector

Parameters

- **keep_example_set:** Indicates if this input object should also be returned as output. (boolean; default: true)
- **max_depth:** An upper bound for the number of literals. (integer; 1- $+\infty$; default: 2)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Local subgroup mining agent

Description: Agent that searches for relative subgroups.

2.1.8 SubgroupChain

Group: Distributed.Subgroup

Required input:

- ExampleSet
- NodeDistanceMatrix

Generated output:

- CommunicationBlackBoard

Parameters

- **num_agents:** Number of agents participating in distributed learning. This should match with the agent attribute in the input example set (integer; 1- $+\infty$)
- **num_cycles:** the number of cycles that the agent team goes through (integer; 1- $+\infty$; default: 1)
- **agent_attribute_prefix:** the prefix of the attributes that contain information about which agent gets which part of the examples. Attribute names must follow the scheme `jagent_attribute_prefixijagent_idi` (string)

- **cost_class**: Class that implements a communication cost matrix
- **log_file**: name of the log file (filename)
- **max_cache**: Bounds the number of rules considered per depth to avoid high memory consumption, but leads to incomplete search. (integer; 1- $+\infty$; default: 10000)
- **k**: Number of rules (integer; 1- $+\infty$; default: 1)
- **relative**: search for relative subgroups (boolean; default: false)

Values

- **applycount**: The number of times the operator was applied.
- **looptime**: The time elapsed since the current loop started.
- **time**: The time elapsed since this operator started.

Inner operators: The inner operators must deliver [PerformanceVector, CommunicationBlackBoard].

Short description: Subgroup Discovery Chain

Description: Operator chain for distributed subgroup discovery.

2.1.9 TestAgent

Group: Distributed.Test

Required input:

- CommunicationBlackBoard

Values

- **applycount**: The number of times the operator was applied.
- **looptime**: The time elapsed since the current loop started.
- **time**: The time elapsed since this operator started.

Short description: Tests the system by exchanging some standard messages.

Description: Agent that sends queries to other agents. Queries received from other agents are sent simply back.

2.1.10 TestAgentExampleSharing

Group: Distributed.Test

Required input:

- CommunicationBlackBoard
- ExampleSet

Generated output:

- CommunicationBlackBoard
- PerformanceVector

Parameters

- **keep_example_set:** Indicates if this input object should also be returned as output. (boolean; default: true)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Tests the system by sharing examples with other agents.

Description: Agent used to test the example sharing functionality of the framework.

2.1.11 TestAgentFeatureGeneration

Group: Distributed.Test

Required input:

- CommunicationBlackBoard
- ExampleSet

Generated output:

- CommunicationBlackBoard
- PerformanceVector

Parameters

- **keep_example_set:** Indicates if this input object should also be returned as output. (boolean; default: true)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Generates random features and propagates them.

Description: Agent used to test the feature sharing functionality of the framework.

2.1.12 TopologyGenerator

Group: Distributed.Topology

Generated output:

- NodeDistanceMatrix

Parameters

- **topology_class:** Class that implements a network topology
- **num_nodes:** The number of network nodes (integer; 1- $+\infty$)

Values

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Creates a static standard topologies.

Description: Generates several standard network topologies, like client/server.

Bibliography

- [1] Philip K. Chan, Wei Fan, Andreas L. Prodromidis, and Salvatore J. Stolfo. Distributed data mining in credit card fraud detection. *IEEE Intelligent Systems*, 14(6):67–74, 1999.
- [2] Werner Dubitzky, Damian McCourt, Mykola Galushka, Mathilde Romberg, and Bernd Schuller. Grid-enabled data warehousing for molecular engineering. *Parallel Computing*, 30(9-10):1019–1035, 2004.
- [3] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [4] B. Park and H. Kargupta. Distributed Data Mining: Algorithms, Systems, and Applications. In Nong Ye, editor, *Data Mining Handbook*, pages 341–358. IEA, 2002.
- [5] Michael Wurst and Martin Scholz. Distributed subgroup discovery. In *Proceedings of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD-06)*, 2006.
- [6] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining*, 7(4), 1999.

Index

AGENTITERATION, 20
EXAMPLESETCONVERTER, 21
GLOBALSUBGROUPAGENT, 21
NETWORKTOPOLOGYLOADER, 22
NETWORKTOPOLOGYSIMULATOR, 23
RANDOMTOPOLOGYGENERATOR, 24
RELATIVESUBGROUPAGENT, 24
SUBGROUPCHAIN, 25
TESTAGENT, 26
TESTAGENTEXAMPLESHARING, 27
TESTAGENTFEATUREGENERATION, 27
TOPOLOGYGENERATOR, 28