NetKit-SRL Userguide

Sofus A. Macskassy
sofmac@gmail.com

May 4, 2010

Contents

List of Tables

Chapter 1

Copyright

Id

Chapter 2

System Requirements

NetKit requires java 1.5+. For many learning tasks, the optional weka machine learning library (jar-file) is also needed. NetKit needs version 3.4+ of this toolkit.[1] It is available at:

```
http://www.cs.waikato.ac.nz/~ml/weka/index.html
```

You will need to name the library "weka.jar". If you only plan on using the homophily-based methods such as the wieghted-vote Relational Neighbor, then you will not need the weka library.
For more information, please refer to the technical report:

> S. A. Macskassy and Provost, J. (2004) "Classification in Networked Data: A toolkit and a univariate case study." CeDER Working Paper #CeDER-04-08, Stern School of Business, New York University, NY, NY 10012. 2004.

This report describes the various learning and inferencing components in detail. These components are not discussed in this document.

[1] It is known to work with version 3.4.2

Chapter 3

Installation

Installing NetKit is very simple:

1. Obtain NetKit from:

   `http://netkit-srl.sourceforge.net`

2. Uncompress the compressed file. This should result in a directory structure:

   ```
   NetKit/
      examples/
   ```

   The primary directory contains the following files:

   ```
   NetKit-VERSION.jar
   NetKit.properties
   activelearning.properties
   aggregator.properties
   colt.jar
   distance.properties
   fetch-datastructures.jar
   inferencemethod.properties
   lclassifier.properties
   logging.properties
   rclassifier.properties
   readestimate.properties
   weka.jar (optional, depending on the compressed file you download)
   weka.properties
   ```

   The *examples* directory contains a list of example files and the file:

   example-runs.txt

   This file shows a list of possible ways to run NetKit.

3. Download the weka library, if needed, and put it in the NetKit directory.[1]
   Note: you *must* name the weka library: `weka.jar`

4. Install java 1.5+ if you haven't already.

5. You are done.

[1] The library (you need at least version 3.4) can be downloaded at `http://www.cs.waikato.ac.nz/~ml/weka/index.html`

Chapter 4

Running NetKit

NetKit is a commandline tool. It requires a dos-prompt or command-line prompt and is all text based. It takes as input a set of flat files and outputs its predictions to either the screen or a given filename. This section describes in more detail what is needed to run NetKit.

4.1 Data Files

NetKit takes as its primary input a *schema* file, which describes the schema of the data in terms of nodes and edges. The data for each type of node and edge should defined in its own separate file. The filename conventions listed in Table 4.1.

schema.arff	The schema file should always end in *.arff*
nodetype.csv	Files describing nodes and attributes end in *.csv*
edgetype.rn	Files describing edges end in *.rn*

Table 4.1: NetKit filename conventions.

NetKit, for legacy reasons, also supports GDA format from the k-Groups/GDA software released by the Auton Lab at Carnegie Mellon University. We will not specify, or talk about, this format here. For more information about the GDA format, please go to:

`http://www.autonlab.org/autonweb/showSoftware/147/`

(this link was working on May 11, 2005).

4.1.1 Schema file

The name of the schema file should follow the filenaming convention:

dataset.arff

where *dataset* can be any name, such as a descriptive name of the data.
The schema file format specification is shown in Table 4.2.
Note that node-types referred to in an edge-type definition must be defined above the edge-type definition. Other than that, edges and nodes can be defined in any order.

DEFAULT PREDICTION ATTRIBUTE: Note that the last attribute of the first node-type will be the default attribute to build a model for and do estimation on. Also note that the attribute to build a model for currently *must* be of type categorical.

An example schema file is shown in Table 4.3. The schema consists of people, transactions and items. People can be connected to each other directly through marriage and children/parent links. People also have transactions and two people could be linked by following the edges Order–Items–OrderedIn–OrderedBy.

(COMMENT|NODE-TEMPLATE|EDGE-TEMPLATE){+}
COMMENT := # .* *newline*
NODE-TEMPLATE := NODE-TYPE ATTRIBUTE{+} NODE-DATA
EDGE-TEMPLATE := EDGE-TYPE REVERSIBLE? EDGE-DATA

NODE-TYPE := "@nodetype" NODE-NAME *newline*
NODE-DATA := "@nodedata" NODE-FILE *newline*
NODE-NAME := *string*
NODE-FILE := *string*.csv

ATTRIBUTE := "@attribute" ATTRIBUTE-SPECIFICATION *newline*
ATTRIBUTE-SPECIFICATION := *string* ATTRIBUTE-TYPE
ATTRIBUTE-TYPE := key|continuous|real|discrete|int|categorical|ORDINAL-LIST
ORDINAL-LIST := {ORDINALS}
ORDINALS := *string*|*string*,ORDINALS

EDGE-TYPE := "@edgetype" EDGE-NAME EDGE-SOURCE EDGE-DESTINATION *newline*
EDGE-SOURCE := NODE-NAME
EDGE-DESTINATION := NODE-NAME
REVERSIBLE := "@reversible" EDGE-NAME? *newline*
EDGE-DATA := "@edgedata" EDGE-FILE *newline*
EDGE-NAME := *string*
EDGE-FILE := *string*.rn

Table 4.2: NetKit schema specification.

4.1.2 Node files

The name of a node file should follow the filenaming convention:

nodefile.csv

where *nodefile* can be any name, such as a descriptive name of the nodetype. The file itself consists of lines with values on a line being separated by commas. Each line represents a node of the given nodetype, and the values should be in the exact same order as given in the nodetype block in the schema file.

NOTE: *Unknown values must be identified by a '?'.*

4.1.3 Edge files

The name of an edge file should follow the filenaming convention:

edgefile.rn

where *edgefile* can be any name, such as a descriptive name of the edgetype. The file itself consists of lines with values on a line being separated by commas. Each line represents an edge of the given edgetype, and must follow the format:

srcID,destID,weight

where srcID and destID refers to the IDs of the nodes being connected, and the weight is a positive real value representing the strength of the connection between the two nodes. This weight is generally 1, but could also be larger numbers representing information such as how often two people have met.

```
# this is a comment
@nodetype     Person
@attribute    SocSecNum       KEY
@attribute    FirstName       CATEGORICAL
@attribute    LastName        CATEGORICAL
@attribute    Age             DISCRETE
@attribute    Gender          {Male,Female}
@nodedata     person.csv

@edgetype     Married         Person Person
@reversible
@edgedata     marriage.rn

@edgetype     Child           Person Person
@reversible   Parent
@edgedata     children.rn

@nodetype     Transaction
@attribute    ID              KEY
@attribute    Date            DISCRETE
@attribute    Price           CONTINUOUS
@attribute    ShippingType    CATEGORICAL
@nodedata     transaction.csv

@edgetype     Order           Person Transaction
@reversible   OrderedBy
@edgedata     orders.rn

@nodetype     Item
@attribute    SKU             KEY
@attribute    Name            CATEGORICAL
@attribute    Price           CONTINUOUS
@nodedata     items.csv

@edgetype     Items           Transaction Item
@reversible   OrderedIn
@edgedata     transitems.rn
```

Table 4.3: Example NetKit schema file.

4.2 Command-line options

NetKit can be run in one of two ways:

```
NetworkLearning [OPTIONS] schema-file
NetworkLearning [OPTIONS] -gda class-file edge-file
```

The GDA option takes two GDA files as specified by their software (for more info on GDA support, see §4.1). The available options, which will be explained in greater detail below, are listed in Table 4.4.

-h	a help screen
-activelearning *al*	Use the given active learning strategy
-attribute *NT:A*	which attribute(A) to classify, where NT is the nodetype
-depth *d*	when training, make neighbor class labels visible up to depth *d*
-format '*string*'	output result lines using this line-format
-inferenceMethod *ic*	use the given inference method
-known `FILE`	a file consisting of known labels
-lclassifier *lc*	use the given local (non-relational) classifier
-learnWithTruth	use complete truth for learning a model
-loo	perform a leave-one-out estimation
-maxpicks #	How many iterations of active learning should be done
-numpicks #	How many nodes should be picked per iteration of active learning
-output `FILE`	where to send output estimates
-priorfile `FILE`	read in prior estimates from the given file
-pruneSingletons	remove singleton nodes in the graph
-pruneZeroKnowledge	remove zeroKnowledge nodes in the graph
-rclassifier *rc*	use the given relational classifier
-runs *r*	perform *r* train-test runs
-sample *s*	how much of the graph to sample for training purposes
-saveItPredict	save the predictions after each inference iteration
-saveROC	save ROC curves after final iteration, one per class.
-seed *seed*	use this random seed for sampling
-showAUC	show the Area Under the ROC curve (for each class)
-showAssort	sow assortativity coefficients.
-showItAcc	show the accuracy after each inference iteration
-test `FILE`	a file consisting of the IDs and labels of nodes to test on
-truth `FILE`	a file consisting of true labels
-vprior `FILE`	read class priors from this file
-key value	overrides properties of the same name in the NetKit properties files

Table 4.4: Brief list of command-line options.

4.2.1 Option -activelearning

This option is used to tell NetKit which active learning strategy to use. The list of available strategies are given in the help page (use the '`-h`' option). Also, you can inspect the `activelearning.properties` file (see §5.3).
NOTE: This should only be used with the harmonic relational classifier ('`-rclassifier harmonic`') and no inferencemethod ('`-inferencemethod=null`')

4.2.2 Option -attribute

This option is used to specify which attribute to build a model for and do estimation on. `NT` is the name of the node-type as given in the schema file, and `A` is the name of the attribute.

NOTE: The attribute to build a model for currently *must* be of type categorical.

SYNTAX:	-attribute `NT:A`
DEFAULT:	The last attribute of the first node-type in the schema file.

4.2.3 Option -depth

This option is primarily for testing purposes. When estimating the performance of a model, some known labels $\mathbf{v}_{test}$ are hidden while learning the model M. Model M is then applied to estimate the labels in $\mathbf{v}_{test}$.
This option is used to make labels of neighbors up to the given depth visible while training model M. With this, it is possible to get an estimate of how well you could do, where you to be given this information. This can also be thought of as sampling subgraphs of the given depth.

SYNTAX:	-depth `int`
DEFAULT:	0
RANGE:	0+

4.2.4 Option -format

This option specifies the format of the predictions. NetKit will output a prediction for each of the nodes that are given in the test set (see §4.2.6, §4.2.18). Each such prediction is printed on one line, using the given output format.

SYNTAX:	-format `'FORMAT'`
DEFAULT:	see text

The format of the string is defined in Table 4.5

FORMAT := SEGMENT+
SEGMENT := CONSTANT|%TAG
CONSTANT := *string*
TAG := id|class|label|predictlabel|predictscore|prediction!*string*|*string*

Table 4.5: Specification for 'format' string using `-format` option.

The tags will be replaced by node-specific values as listed in Table 4.6.

tag	**value**
id	the id(KEY) of the node whose attribute-value is estimated
class	the true (if known) value of the attribute
label	alias for the 'class' tag
string	the score for the class defined by '*string*' (this class must be one of the possible categorical values for the attribute whose value is being estimated)
predictlabel	the attribute-value with the highest score
predictscore	the highest score
prediction!*string*	this is equivalent to the string: *string*:%*string* *EXCEPT:* if the *string* is the same as the predicted class, in which case this evaluates to an empty string.

Table 4.6: Valid tags and their node-specific values for the 'format' string using the `-format` option.

DEFAULT value is based on the categories of the attribute to be estimated. If we assume that the possible values are {`high,medium,low`}, then the default format string is:

'%id %label %predictlabel:%predictscore %prediction!high %prediction!medium %prediction!low'

EXAMPLE: This option makes it very flexible to customize the output needed. For example, were you to estimate whether a person is a threat, where the attribute could take on either 'threat' or 'nonthreat' you could generate an SQL insertion command for each of the predicted scores:

'insert into persontable (id,threatscore) values (%id,%threat);'

or you could update an existing table:

'update persontable set threatscore = %threat where id = %id;

4.2.5 Option -inferenceMethod

This option is used to tell NetKit which inference method to use. The list of available inference methods are given in the help page (use the '`-h`' option). Also, you can inspect the `inferencemethod.properties` file (see §5.2).

SYNTAX:	-inferencemethod `IC`
DEFAULT:	relaxlabel

Currently valid parameters are listed in Table 4.7. These can be configured and customized in a properties file (see §5.2). Their default configurations can be overridden on the commandline by specifying different values for their configuration parameters (using the "`-key value`" option).

IC	**description**
null	no inference is to take place. In this case, the local classifier is applied, and then the relational classifier is applied only once
relaxlabel	relaxation labeling is used. valid key-value pairs (see §4.2.28): `numit`, how many iterations to run (default: 99) `beta`, what is the initial temperature (default: 1) `decay`, how much to decay `beta` at each iteration (default: 0.99)
gibbs	gibbs sampling valid key-value pairs (see §4.2.28): `numit`, how many iterations to run (default: 2000) `burnin`, what are the number of burnin iterations (default: 200) `numchains`, how many chains to run (default: 10)
iterative	iterative classification valid key-value pairs: `numit`, how many iterations to run (default: 1000)

Table 4.7: Valid parameters for the `-inferenceMethod` option.

4.2.6 Option -known

The `-known` option is used to specify the known attribute information. This can be used in two ways:

1. If the nodetable has more information and you want to evaluate the learning performance based only on the specified information.
2. Using this, you can introduce 'noise' without changing the original table.

NetKit will assume that it is to estimate the attribute values for all nodes not specified in the given known file. This can be overridden by using the `-test` option (see §4.2.25).

SYNTAX:	-known `FILE`
DEFAULT:	none

The file format of the known file is:

id_i,known-value_i
...
id_k,known-value_k

4.2.7 Option -lclassifier

This option is used to tell NetKit which local, or non-relational, classifier to use. The local classifier is used to initialize the priors. Using this by itself is akin to traditional machine learning. The list of available classifiers are given in the help page (use the '`-h`' option). Also, you can inspect the `lclassifier.properties` file (see §5.4).

SYNTAX:	-lclassifier `LC`
DEFAULT:	classprior

Currently valid parameters are listed in Table 4.8. These can be configured and customized in a properties file (see §5.4). Their default configurations can be overridden on the commandline by specifying different values for their configuration parameters (using the "`-key value`" option).

null	does nothing
classprior	sets all to-be-estimated attributes to the class prior the class prior can be overridden by using the `-vprior` option (see §4.2.27).
uniform	sets all to-be-estimated attributes to a uniform distribution
external	gets priors from an external file. you *must* use the `-priorfile` option with this (see §4.2.13).
weka	There are various classifiers from WEKA defined. You must refer to the properties file for these definitions (see §5.4). **NOTE:** This requires the WEKA library.

Table 4.8: Valid parameters for the `-lclassifier` option.

4.2.8 Option -learnWithTruth

This option is used for testing purposes. It tells NetKit to learn models based on the truth. It will build models based on the training samples as implied by the `-known` (§4.2.6), `-test` (§4.2.25) or `-sample` (§4.2.18) options, but it will have access to the complete truth when learning the models (e.g., although neighbor nodes might not be in the training set, their attributes will be available when learning the model).

SYNTAX:	-learnWithTruth

4.2.9 Option -loo

This option is used for testing purposes. It runs a leave-one-out estimation: With-hold the attribute value of one node, learn a model based on all other nodes, then apply the model to the with-held node. Do this for all nodes and report the final performance statistics. Because there is always only one node to be tested, no collective inferencing (see §4.2.5) is done—just like using the `null` inference method..

SYNTAX:	-loo

This cannot be used with any of the other options that imply a train/test split on the nodes in the data. These options include:

CANNOT BE USED WITH:
`-known, -runs, -sample, -test`

4.2.10 Option -maxpicks

If an active learning strategy is selected, this options specifies how many nodes should end up being picked throughout the active learning process.

SYNTAX:	-maxpicks #
DEFAULT:	10

4.2.11 Option -numpicks

If an active learning strategy is selected, this options specifies how many nodes should be picked per iteration of the process.

SYNTAX:	-numpicks #
DEFAULT:	1

4.2.12 Option -output

This option specifies a file-stem of where to send predictions. The actual file it will be sent to is:

`FILE`.predict

This option is also indirectly used by the `-saveItPredict` (§4.2.19) and `-saveROC` (§4.2.20) options.

SYNTAX:	-output `FILE`
DEFAULT:	standard output

4.2.13 Option -priorfile

Use this option with the external local classifier (see §4.2.7). The option specifies an external file from which to read initial priors.

SYNTAX:	-priorfile `FILE`
DEFAULT:	none

The current supported file format for this external file is listed in Table 4.9.

`ID truevalue` $\texttt{label}_a\texttt{:score}_a \ldots \texttt{label}_c\texttt{:score}_c$
`...`
`ID truevalue` $\texttt{label}_a\texttt{:score}_a \ldots \texttt{label}_c\texttt{:score}_c$

Table 4.9: File format for `-priorfile` option.

Each line contains a prediction for a given node, where the `ID` identifies the node and the `truevalue` specifies what the truth is. There must be a score for each of the possible labels for the attribute being estimated. The ordering of these scores does not matter.

4.2.14 Option -pruneSingletons

Use this option to prune out nodes in the data that are not connected to any other nodes. This is the setting that is often used for evaluating the relational learners, as these singleton nodes obviously have no relations and therefore do not give any insight into how well the relational learners are performing.

SYNTAX:	-pruneSingletons

4.2.15 Option -pruneZeroKnowledge

This is a stronger pruning than the `-pruneSingletons` (see §4.2.14) option. Given a set of nodes whose labels are known (the training examples), this setting will remove any node in the graph that are not connected (directly or indirectly) to a node in the training set.

SYNTAX:	-pruneZeroKnowledge

4.2.16 Option -rclassifier

This option is used to tell NetKit which relational, classifier to use. The relational classifier is used in the inner loop of the collective inference methodsn. The list of available classifiers are given in the help page (use the '`-h`' option). Also, you can inspect the `rclassifier.properties` file (see §5.3).

SYNTAX:	-rclassifier `LC`
DEFAULT:	wvrn

Currently some of the valid parameters are listed in Table 4.10. These can be configured and customized in a properties file (see §5.3). Their default configurations can be overridden on the commandline by specifying different values for their configuration parameters (using the "`-key value`" option).

wvrn	use the weighted-vote Relational Neighbor classifier (previously referred to as pRN—the probabilistic Relational Neighbor)
cdrn-norm-cos	use the class-distributional RN classifier with normalized values of neighbor-class values and using the cosine distance metric valid key-value pairs (in addition to those specified below): `distance`, which distance function to use (cosine, l1, l2)
no-bayes	use the network-only Bayes classifier (with a markov random field formulation).
nolb-lr-distrib	use the logistic regression link-based classifier where values are normalized. **NOTE:** This requires the WEKA library.
nolb-lr-count	use the logistic regression link-based classifier where values are absolute counts. **NOTE:** This requires the WEKA library.
weka	There are various classifiers from WEKA defined. You must refer to the properties file for these definitions (see §5.3). **NOTE:** This requires the WEKA library.
*	key-value pairs that are valid with any of the above: `aggregation`, which attributes to aggregate on `aggregators`, which aggregators to use `useintrinsic`, include intrinsic variables when learning the model (true, false)

Table 4.10: Valid parameters for the `-rclassifier` option.

Valid values for the `aggregation` key are listed in Table 4.11.

All	Aggregate on all attributes
None	Do not aggregate any attributes (same as traditional machine learning)
ClassOnly	Only aggregate on the class attribute
ExcludeClass	Aggregate on all but the class attribute (similar to traditional relational learning)

Table 4.11: Valid values for `aggregation` key.

Valid values for the `aggregators` are listed in Table 4.12. You can specify a list of aggregators thorugh a comma-separated list. See §5.5 for more information about aggregators.

aggregator	description	valid attributetypes
count	count how often an attribute-value was seen	CATEGORICAL, DISCRETE
exist	was a given attribute-value seen	CATEGORICAL, DISCRETE
max	what is the maximum seen value	CONTINUOUS, DISCRETE
mean	what is the mean seen value	CONTINUOUS, DISCRETE
min	what is the minimum seen value	CONTINUOUS, DISCRETE
mode	what is the most often seen attribute-value	CATEGORICAL, DISCRETE
ratio	the normalized count	CATEGORICAL, DISCRETE

Table 4.12: Valid values for `aggregators` key.

4.2.17 Option -runs

This option is used for testing and evaluation of the system. It tells the system how many test runs to perform. The system will output performance statistics on each run as well as a summary statistic across all runs.
Each test run is based on a new training set and test set split. These splits can be defined in two ways. See §4.2.18 for details.

SYNTAX:	-runs `int`
DEFAULT:	10

This cannot be used with leave-one-out or any of the options that imply only one train/test split on the nodes in the data. These options include:

CANNOT BE USED WITH:
`-loo, -known, -test`

4.2.18 Option -sample

This option is used for evaluation and testing of the learning system. It is used to specify how much of the graph to use as the training set (the remaining part is then used to test the system). The amount of the graph to use for training can be specified in three ways:

1. *absolute* — if the value given is 1 or larger, then it is assumed to be an integer and tells the system how many nodes (randomly sampled) to use for learning a model.
2. *relative* — if the value given lies between 0 and 1, then it is used as the *ratio* of the nodes (e.g., 0.2 means to randomly sample 20% of the nodes for training).
3. *cross-validation* — if the value give is 0, then the system will perform a cross-validation based on the number of runs specified by the `-runs` option (see §4.2.17). This is equivalent to specifying a *ratio* of $\frac{1}{\#runs}$.

SYNTAX:	-sample `(int|real)`
DEFAULT:	0

4.2.19 Option -saveItPredict

This option tells the system to save for each iteration of the inference step the predictions at that iteration.
The system will output the predictions to the files:

`FILE`.*iteration#*.predict

where `FILE` is the name specified in the `-output` option (see §4.2.12).

SYNTAX:	-saveItPredict

4.2.20 Option -saveROC

This option tells the system to save the ROC curves that can be used to visualize system performance.
The system will output the ROC curves to the files:

`FILE`-*class-value-run#*.roc

where `FILE` is the name specified in the `-output` option (see §4.2.12). If no name has been specified, then FILE is set to "roc".

SYNTAX:	-saveROC

4.2.21 Option -seed

Use this option to set the random seed that is used by the system for any stochastic processes.

SYNTAX:	-seed `long`
DEFAULT:	random value based on current time

4.2.22 Option -showAUC

This is an evaluation option which tells the system to output the Area Under the ROC Curve (AUC) as part of the evaluation metrics. There will be one AUC for each possible value of the class attribute, which reflects how well the prediction-scores for that class-value are separated from all other classes.

SYNTAX:	-showAUC

4.2.23 Option -showAssort

This is an option to report graph statistics. Using this option, NetKit will report the assortativity value for the graph, both based on the 'truth' as well as for each of the 'training' sets.

SYNTAX:	-showAssort

4.2.24 Option -showItAcc

This is an option to report inference statistics. Using this option, NetKit will report the accuracy of the predictions after every inference iteration if the number of iterations to run is < 250, otherwise it will show the accuracies after every 10 iterations.

SYNTAX:	-showItAcc

4.2.25 Option -test

This option is used to specify which nodes in the graph to estimate predictions for, where the test file specifies node ids as well as an attribute value against which to evaluate the predictions. NetKit will assume that it is to learn a model based on all nodes *not* specified in the given test file. This can be overridden by using the `-known` option (see §4.2.6).

SYNTAX:	-test FILE
DEFAULT:	none

The file format of the test file is:

id$_i$,test-value$_i$
...
id$_k$,test-value$_k$

This cannot be used with leave-one-out or any of the options that imply more than one train/test split on the nodes in the data. These options include:

CANNOT BE USED WITH:
`-loo, -sample, -runs`

4.2.26 Option -truth

This option is used to specify the ground truth against which all evaluations are based. This will override any values seen in the node-table.

SYNTAX:	-truth FILE
DEFAULT:	none

The file format of the truth file is:

id_i,true-value_i ... id_k,true-value_k

4.2.27 Option -vprior

Use this option to tell NetKit the probabilty of seeing each of the possible attribute values for the attribute NetKit is to predict. If this is not specified, then it will be estimated from the training set. This can be used in various manners, but is especially important if the training set does not contain at least one instance of each possible class (this can happen if we are only given positive instances but not negative instances — for example, we only know a few entities to be malicious but know of no entities that are known to be benign).

SYNTAX:	-vprior FILE
DEFAULT:	none

The file format of the vprior file is:

value_1,*probability* ... value_k,*probability*

The *probability* values will be normalized to add to 1 if they don't already. Also, all possible attribute values must be listed in the file.
EXAMPLE: If NetKit is to predict whether the threat-level of an entity, where the possible values are {`high, medium, low`}, then the vprior file might look like this:

high,0.01 medium,0.05 low,0.94

In this case, the *a priori* probability of being a high threat is 1%, being a medium level threat is 5% and being a low threat is 94%.

4.2.28 Option -*key*

This last option in NetKit is used to override any configuration parameters in the property files. For example, if a property in a file is:

cdrn.distance=cosine

then the *key* is "distance" and the value is "cosine". To override the value of this parameter to be "l2" for this particular run of NetKit, you would use the option

```
-distance l2
```

SYNTAX:	-*key* VALUE
DEFAULT:	N/A

To get a list of various keys, see any of the tables above, or see the sections describing the various property files in §5.

4.3 Example runs

There are more example runs described in the examples directory that should have come with the NetKit package. The example runs described in this section will work using the example files provided.
The example runs assume the following directory structure:

some-path`/NetKit/`{*all netkit jar files and property files*} *some-path*`/NetKit/examples/`{*example files*}

Change directory to:

some-path`/NetKit/examples`

> **NOTE:** NetKit by default sends logging output to STDOUT.
> Unless you use the (-output) parameter, predictions will also go to STDOUT.

The example runs show how to use simple combinations of parameters. These can be combined to your own specifications.

4.3.1 Getting help screen

You can run NetKit with no parameters to get help:

```
java -jar ../NetKit.jar
```

4.3.2 Basic run

The basic run takes only a schema file as its input:

```
java -jar ../NetKit.jar goodbad-schema.arff
```

This will perform a 10-fold cross validation run (or whichever is the max number of nodes). By default, NetKit will use the class prior as th local classifier, the weighted vote Relational Neighbor classifier as the relational classifier and relaxation labeling as the inference method. These defaults can be changed by editing the `NetKit.properties` file. This run is to be used for evaluation only as it assumes that everything is known in the graph.

4.3.3 Evaluation using k-fold cross-validation

To use a k-fold cross validation, you need to use the `-runs` (§4.2.17) option. This run uses a 5-fold cross validation and sends the output predictions to `goodbad-out.predict` (`-output`):

```
java -jar ../NetKit.jar -runs 5 -output goodbad-out goodbad-schema.arff
```

This run assumes that the graph is completely labeled. It splits the nodes into 5 equal-sized sets and performs 5 evaluations runs where it holds out one of the sets and trains on the rest. It then predicts the holdout set and finally reports summary performance statistics.

4.3.4 Evaluation using explicit sampling sizes

If you want to do sensitivity analysis on how much data is labeled, then you need to use the `-sample` (§4.2.18) option. This example performs 20 runs where it samples 10% at every run to train on (and test on the remaining 90%). It sends the output to `goodbad-out.predict` and uses the priors specified in the `goodbad-classprior` file to set the initial priors on the unknowns[1]:

```
java -jar ../NetKit.jar -runs 20 -sample 0.1 -vprior
goodbad-classprior -output goodbad-out goodbad-schema.arff
```

4.3.5 Applying to a partially/unlabeled labeled graphs

If the graph is only partially labeled, or even unlabeled, then you can specify a set of nodes and their labels for NetKit to use as the training set. It will then estimate the labels for the remaining nodes. Note that NetKit will not at this time identify the known labels in a partially labeled graph—these must still be specified in an external file.
An example of this type of run is:

[1] Remember that by default NetKit uses the classprior local classifier—we use the file to specify what that class prior should be.

```
java -jar ../NetKit.jar -known goodbad-known.csv -rclassifier
cdrn-norm-cos -output goodbad-out goodbad-schema.arff
```

The run specifies that the known labels are in the file `goodbad-known.csv`, to use the relational classifier `cdrn-norm-cos` and to redirect output predictions to `goodbad-out.predict`. The file format for the known files is given in §4.2.6 (pg. 9).

4.3.6 Customizing the output format for scores

You can customize the estimated score output format. For example, if you have an existing database (which was exported into NetKit format), and you want to insert the new scores, then you can have NetKit specifically output SQL commands to update the given table in the database.
An example run might look like this:

```
java -jar ../NetKit.jar -known goodbad-known.csv -format ''insert
into persontable (id,threatscore) values (%id,%Bad)'' -output
goodbad-out goodbad-schema.arff
```

This command takes a list of specified labels (`goodbad-known.csv`), estimates the good/bad scores for the remaining nodes, and outputs lines of the form:

```
insert into persontable (id,threatscore) values (PersonK, 0.039)
...
insert into persontable (id,threatscore) values (PersonM, 0.591)
```

Chapter 5

Customizing NetKit

This chapter describes ways to customize the various components of NetKit. These are the files that need to be edited if you develop new components as well (this is not a developer's guide and we will not discuss developing new components here).
NetKit has 9 configuration files, which are listed in Table 5.1. We will discuss 6 of these below.

`NetKit.properties`	sets default commandline options
`aggregator.properties`	defines aggregators and their behaviors (see §5.5)
`distance.properties`	defines vector-distance functions
`inferencemethod.properties`	defines inference methods and their default parameters (see §5.2)
`lclassifier.properties`	defines non-relational classifiers and their default parameters (see §5.4)
`logging.properties`	defines logging parameters (see §5.1)
`rclassifier.properties`	defines relational classifiers and their default parameters (see §5.3)
`readestimate.properties`	lists the classes that are available to read in the priorfile (see §4.2.13)
`weka.properties`	namees the classifiers that are available through WEKA (see §5.6)

Table 5.1: Property files used by NetKit

5.1 Logging

NetKit outputs a lot of logging information to the screen by default. The file that controls this logging is:

logging.properties

This property file can be used to fine-tune how much information to log. NetKit uses the Java logging utility classes and this file configures this utility.
Logging happens in two parts:

1. setting and configuring a logging *handler*.
2. setting the logging levels for each class.

The possible logging levels are listed in Table 5.2.

5.1.1 Defining the log handler

The logging handler specifies where to send logs. It also defines the format of the logging as well as the global most detailed level it will log. This overrides any more detailed logging requested for various classes (see below).
By default, NetKit uses a ConsoleHandler at the finest logging level as listed in Table 5.3:
The java logging utility classes also support logging to a file using the `FileHandler`. The configuration of a FileHandler is a little more complex. Table 5.4 shows the configuration properties for a FileHandler.
The `pattern` consists of a string that includes the following special components that will be replaced at runtime:

OFF	no logging
SEVERE	only most severe errors
WARNING	
INFO	
CONFIG	
FINE	
FINER	
FINEST	the most detailed logging
ALL	log all message

Table 5.2: Logging levels.

```
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=FINEST
```

Table 5.3: Default NetKit logging handler.

"/"	the local pathname separator
"%t"	the system temporary directory
"%h"	the value of the "user.home" system property
"%g"	the generation number to distinguish rotated logs
"%u"	a unique number to resolve conflicts
"%%"	translates to a single percent sign "%"

Table 5.5 shows an example FileHandler setup, with 5 rotating logs, each log file a maximum of 1Mb. The log files will be written to the user's home directory and will take the names `netkit-1.log` ... `netkit-5.log`.

5.1.2 Defining custom log-levels

The logging facility allows for setting the logging levels for various classes. These can be as fine-grained as needed. Note that although you can set the log-levels to as detailed as you want here, the handler will not output more details than its own level.
NetKit defines log-levels for some of the standard components which is listed in Table 5.6. Each line specifies a log level at a specific component level. These logging names are hierarchical. For example, a class which logs to `netkit.classifiers.relational`, can get its log-level from one of the log-definition lines:

```
netkit.classifiers.relational.level
netkit.classifiers.level
netkit.level
```

The logging utility takes the most specific line when deciding which level to log at.

5.2 Inference Methods

The inference methods are named, defined and configured in the `inferencemethod.properties` file. The configuration of an inference method uses the syntax shown in Table 5.7.
For example, the default configuration for relaxation labeling is shown in Table 5.8.
NetKit comes with four inference methods: null method (no inference), gibbs sampling, iterative classification and relaxation labeling. Read the properties file for their default configuration and explanation of their configuration keys.

5.3 Relational Classifiers

The relational classifiers are named, defined and configured in the `rclassifier.properties` file. The configuration of a relational classifier uses the same syntax as the inference methods and is shown in Table 5.7.
For example, the default configuration for the network-only logistic regression link-based method (see the tech report referred to in §2) is shown in Table 5.9.
NetKit comes with twelve configured relational classifiers: weighted vote relational neighbor, probablistic relational neighbor classifier, two configurations of a class-distributional relational neighbor , network-only bayes classifier, four

`level`	what is the logging level (default: ALL)
`limit`	what is the maximum file size (default: 0 — limitless)
`count`	what is the number of rotating logs (default: 1 — no rotating)
`append`	should it append to an existing logfile (default: false)
`pattern`	what is the output log-file name. (default: "%h/java%u.log") (described below).

Table 5.4: Configuration parameters for a FileHandler.

```
handlers=java.util.logging.FieHandler
java.util.logging.FileHandler.level=FINEST
java.util.logging.FileHandler.limit=1000000
java.util.logging.FileHandler.count=5
java.util.logging.FileHandler.append=false
java.util.logging.FileHandler.pattern=%h/netkit-%g.log
```

Table 5.5: Example FileHandler definition.

```
netkit.level=INFO
netkit.graph.level=CONFIG
netkit.graph.io.level=OFF
netkit.classifiers.level=INFO
netkit.util.level=INFO
netkit.inference.level=CONFIG
```

Table 5.6: Default logging levels for NetKit components.

name.class=`fully.specified.class.name`
name.configuration_key=`value`

Table 5.7: Syntax for defining and configuring an inference method.

relaxlabel.class=`netkit.inference.RelaxationLabeling`
relaxlabel.numit=`99`
relaxlabel.beta=`1.00`
relaxlabel.decay=`0.99`

Table 5.8: Default relaxation labeling configuration.

nolb-lr-distrib.class=`netkit.classifiers.relational.NetworkWeka`
nolb-lr-distrib.classifier=`logistic`
nolb-lr-distrib.useintrinsic=`false`
nolb-lr-distrib.aggregators=`ratio`
nolb-lr-distrib.aggregation=`ClassOnly`

Table 5.9: Default network-only logistic-regression configuration.

configurations for network-only logistic regression, logistic regression, naive bayes, and J48. The last three use the WEKA machine learning toolkit.
Read the properties file for their default configuration and explanation of their configuration keys.

5.4 NonRelational Classifiers

The non-relational (local) classifiers are named, defined and configured in the `lclassifier.properties` file. The configuration of a non-relational classifier uses the same syntax as the inference methods and is shown in Table 5.7.
NetKit comes with seven configured non-relational classifiers: null (does no estimation), uniform prior, class prior, external prior, logistic regression, naive bayes and J48. The last three use the WEKA machine learning toolkit.
Read the properties file for their default configuration and explanation of their configuration keys.

5.5 Aggregators

The available aggregators are named and defined in the `aggregator.properties` file. An aggregator is defined by its name, class and which types of attributes it can aggregate.

name.class=`fully.specified.class.name`
name.accept=`ATTRIBUTE_TYPE[,ATTRIBUTE_TYPE...]`

Table 5.10: Syntax for defining an aggregator.

NetKit comes with seven aggregators: count, exist, max, mean, min, mode and ratio.
Read the properties file for their default configuration and explanation of their configuration keys.

5.6 WEKA

In order to make classifiers from WEKA available to NetKit, you need to let NetKit know about them and give them a name for lookup purposes. The `weka.properties` file does this. In fact, you can list *any* classifier that uses the WEKA API—therefore thirdparty classifiers that uses the WEKA API could also be listed here. In order to make a classifier available, you need to give it a name and spcify the fully specified class name.
The default weka properties file is shown in Table 5.11.

```
logistic.class=weka.classifiers.functions.Logistic
leastsquares.class=weka.classifiers.functions.LeastMedSq
naivebayes.class=weka.classifiers.bayes.NaiveBayesMultinomial
j48.class=weka.classifiers.trees.J48
```

Table 5.11: Default WEKA configuration file.

Chapter 6

Known issues and limitations

NetKit runs everything in memory. It is therefore constrained by the amount of memory available. On a 32-bit machine, this means that it can only allocate about 1.8Gb worth of memory at most. NetKit has been successfully run on such a machine on a data set of 1 million entities and 10 million links, where the entities only had 2 attributes.
NetKit has been very successful on many domains when using the vanilla weighted-vote Relational Neighbor (wvRN) classifier with relaxation labeling. These include domains such as people (terrorists), companies, web-pages, and bibliometrics.
We have found that wvRN is sensitive to the sparsity of links—if a node in the graph is not connected (Even indirectly) to a node whose label is 'known', then wvRN will not be able to make any reasonable estimation of the attribute-value. This should not be surprising.
One last potential sensistivity issue for wvRN is the noise level (ratio of mislabeled entities whose labels are 'known'). For terrorist domains, our studies have shown that wvRN is quite robust up until a noise level of 25%, after which it starts to break down.