VFML Reference Manual

Generated by Doxygen 1.3.2

Wed Jul 9 01:59:21 2003

Contents

Chapter 1

VFML

Welcome to the VFML (Very Fast Machine Learning toolkit) for mining high-speed data streams and very large data sets. VFML is made up of three main components. The first is a collection of tools and APIs that help a user develop new learning algorithms. The second component is a collection of implementations of important learning algorithms. The third component is a collection of scalable learning algorithms that were developed by Pedro Domingos and Geoff Hulten (with the help of several other people see **Thanks**). VFML is written in standard C (and a bit of Python), and provide a series of tutorials and examples as well as extensive in source documentation in JavaDoc format. VFML is being distributed under the `LGPL license`.

To quickly begin using the system, you should look at the `Getting Started With VFML` document. If you are going to be developing new tools with the VFML toolkit you should also see `Loading Data`, the `Scan-Dataset` example, and then modify the `implement-learner` example to make your own learner.

1.1 Downloads

- See the `Getting Started` page for links to the software.
- `UCI Datasets`
- `Benchmark Bayesian Networks`

1.2 Topics

These links take you to some tutorials and example code on parts of the VFML system.

- `Getting Started with VFML`
- A tutorial on `loading data`
- A detailed example on `loading data sets`
- An overview of what you will need to do to `create your own learner`
- A detailed example of how to `implement a learner`
- A tutorial on `comparing learning algorithms`

- A detailed example on using **batchtest** to `compare learning algorithms`
- An example on how to `interface with C4.5` in your programs
- A brief description of how to use the learners in VFML to `mine data streams`.

1.3 Moving On

The following sections contain links to the documentation for all of the tools, learners and APIs that you might find useful. You might like to download the reference manual (which contains all this information) in `pdf format`.

- The **Core APIs** describes learning related APIs and ADTs.
- The **Utility APIs** describes generic APIs that may help you.
- The **Tools Section** describes all the data manipulation and generation tools contained in VFML.
- The **Learners Section** describes all of the learners that are included in VFML.
- The **Decision Tree Section** contains documentation for the parts of VFML that are relevant to working with decision trees.
- The **Belief Net Section** contains documentation for the parts of VFML that are relevant to working with belief nets.

1.4 Appendixes

- `C4.5 Format`
- `Bayesian Interchange Format`

1.5 Contact Us

If you have any comments, suggestions, or bug reports, please feel free to send us email: `ghulten@cs.washington.edu` and `pedrod@cs.washington.edu`

Thanks
VFML was made possible by a gift from the Ford Motor Company.

See **Thanks** for a list of additional people that have contributed to VFML.

Wish List
The windows distribution needs to be brought up to date.

1.6 Terms Of Use

Chapter 2

VFML Module Index

2.1 VFML Modules

Here is a list of all modules:

Chapter 3

VFML Data Structure Index

3.1 VFML Data Structures

Here are the data structures with brief descriptions:

Chapter 4

VFML File Index

4.1 VFML File List

Here is a list of all documented files with brief descriptions:

Chapter 5

VFML Page Index

5.1 VFML Related Pages

Here is a list of all related documentation pages:

Chapter 6

VFML Module Documentation

6.1 Core APIs

6.1.1 Detailed Description

These sections should be used as reference for details on the interfaces you can use in your programs. They contain detailed information on all of the APIs that VFML offers.

Files

- file **AttributeTracker.h**

  Keep a record of which attributes are active.

- file **BeliefNet.h**

  A Belief Net Structure with CPT local models.

- file **bnlearn-engine.h**

  Learn the structure of a BeliefNet from data.

- file **C45interface.h**

  Calls the C4.5 decision tree learning system and returns the learned tree.

- file **DecisionTree.h**

  A Decision Tree Structure.

- file **Example.h**

  ADT for training (and testing, etc.) data.

- file **ExampleGenerator.h**

  Generate a random (but reproducible) data set.

- file **ExampleGroupStats.h**

  Sufficient statistics for Entropy and Gini.

- file **ExampleSpec.h**

Schema for training data.

- file **REPrune.h**

  Peforms reduced error pruning on a decision tree.

- file **vfdt-engine.h**

  An API which lets your program learn a DecisionTree from a high-speed data stream.

6.2 Utility APIs

6.2.1 Detailed Description

Some utility code that is used by VFML. You may find it useful or not. If you plan on modifying the internals of VFML you will certainly care about these. (There are a few places where some of these are exposed in an API too.)

Files

- file **bitfield.h**

  Compactly represent a bit field.

- file **Debug.h**

  A set of functions that help your programs produce debugging output in a consistent way.

- file **HashTable.h**

  A hash table.

- file **lists.h**

  Generic list functions.

- file **memory.h**

  Tracks the size of allocations made.

- file **random.h**

  Generates random numbers in a number of ways, and has support for saving and restoring the state of the random number generator.

- file **stats.h**

  Some statistical functions.

6.3 Decision Tree Section

6.3.1 Detailed Description

All of these are related to the use and construction of decision trees.

Files

- file **AttributeTracker.h**

  Keep a record of which attributes are active.

- file **C45interface.h**

  Calls the C4.5 decision tree learning system and returns the learned tree.

- file **c45wrapper**

  Calls C4.5 and tests the learned tree.

- file **c50wrapper**

  Calls C5.0 and tests the learned tree.

- file **cvfdt**

  Learns a DecisionTree from a high-speed time-changing data stream (or very large data set).

- file **decisionstump**

  Learns a decision stump (a DecisionTree with only one split).

- file **DecisionTree.h**

  A Decision Tree Structure.

- file **Example.h**

  ADT for training (and testing, etc.) data.

- file **ExampleGroupStats.h**

  Sufficient statistics for Entropy and Gini.

- file **ExampleSpec.h**

  Schema for training data.

- file **REPrune.h**

  Peforms reduced error pruning on a decision tree.

- file **sprint**

  Learn a decision tree from data sets that do not fit in RAM.

- file **treedata**

  Creates a synthetic data set by sampling from a randomly generated DecisionTree.

- file **vfdt**

  Learns a decision tree from a high-speed data stream or very large data set.

- file **vfdt-engine.h**

  An API which lets your program learn a DecisionTree from a high-speed data stream.

6.4 Belief Net Section

6.4.1 Detailed Description

All of these are related to the use and construction of belief nets.

Files

- file **BeliefNet.h**

  A Belief Net Structure with CPT local models.

- file **beliefnetcorrupt**

  Makes some random changes to a BeliefNet.

- file **beliefnetdata**

  Creates a data set by sampling from a Bayesian Network.

- file **beliefnetscore**

  Tests a BeliefNet in several ways.

- file **bnlearn**

  Learns the structure of a BeliefNet from a data set.

- file **bnlearn-engine.h**

  Learn the structure of a BeliefNet from data.

- file **Example.h**

  ADT for training (and testing, etc.) data.

- file **ExampleSpec.h**

  Schema for training data.

- file **vfbn1**

  Learns the structure of a BeliefNet from a very large data set using sampling.

- file **vfbn2**

  Learns the structure of a BeliefNet from a very large data set using sampling and a new search proceedure.

6.5 Learning Programs

6.5.1 Detailed Description

Learning programs that are included with VFML.

Files

- file **bnlearn**

  Learns the structure of a BeliefNet from a data set.

- file **c45wrapper**

  Calls C4.5 and tests the learned tree.

- file **c50wrapper**

  Calls C5.0 and tests the learned tree.

- file **cvfdt**

  Learns a DecisionTree from a high-speed time-changing data stream (or very large data set).

- file **decisionstump**

  Learns a decision stump (a DecisionTree with only one split).

- file **kmeans**

  Performs k-means clustering.

- file **mostcommonclass**

  Predicts the most common class in the training data.

- file **naivebayes**

  A Naive Bayes learner.

- file **sprint**

  Learn a decision tree from data sets that do not fit in RAM.

- file **vfbn1**

  Learns the structure of a BeliefNet from a very large data set using sampling.

- file **vfbn2**

  Learns the structure of a BeliefNet from a very large data set using sampling and a new search proceedure.

- file **vfdt**

  Learns a decision tree from a high-speed data stream or very large data set.

- file **vfem**

  Performs EM clustering.

- file **vfkm**

  Performs k-means clustering accelerated with sampling.

6.6 Other Tools

6.6.1 Detailed Description

Some tools to help you work with data sets and learners.

Files

- file **batchtest**
  Performs cross validation of a collection of learners on a collection of datasets.
- file **beliefnetcorrupt**
  Makes some random changes to a BeliefNet.
- file **beliefnetdata**
  Creates a data set by sampling from a Bayesian Network.
- file **beliefnetscore**
  Tests a BeliefNet in several ways.
- file **bindata**
  Converts continuous attributes into discrete ones.
- file **cleandata**
  Cleans up a data set in several ways.
- file **clusterdata**
  Creates a synthetic data set from randomly generated clusters.
- file **combinedata**
  Combines a series of data sets into a single large one.
- file **folddata**
  Randomly splits a data set into a collection of train/test pairs.
- file **sampledata**
  Draws a sample from a data set.
- file **shuffledata**
  Randomizes the order of a data set and rewrites it.
- file **treedata**
  Creates a synthetic data set by sampling from a randomly generated DecisionTree.
- file **uRunner**
  Distribute a collection of jobs across a cluster of computers.
- file **xvalidate**
  Performs cross validation of a learner on a data set.

Chapter 7

VFML Data Structure Documentation

7.1 _BeliefNet_ Struct Reference

7.1.1 Detailed Description

Belief network ADT.

See **BeliefNet.h** for more detail.

The documentation for this struct was generated from the following file:

- **BeliefNet.h**

7.2 _BeliefNetNode_ Struct Reference

7.2.1 Detailed Description

Belief net node with full CPTs for local models.

See **BeliefNet.h** for more detail.

The documentation for this struct was generated from the following file:

- **BeliefNet.h**

7.3 _BITFIELD_ Struct Reference

7.3.1 Detailed Description

ADT for compactly representing a bit field.

See **bitfield.h** for more detail.

The documentation for this struct was generated from the following file:

- **bitfield.h**

7.4 _DecisionTree_ Struct Reference

7.4.1 Detailed Description

ADT for working with decision trees.

See **DecisionTree.h** for more detail.

The documentation for this struct was generated from the following file:

- **DecisionTree.h**

7.5 _Example_ Struct Reference

7.5.1 Detailed Description

ADT for working with examples.

See **Example.h** for more detail.

The documentation for this struct was generated from the following file:

- **Example.h**

7.6 _ExampleGenerator_ Struct Reference

7.6.1 Detailed Description

Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** for more detail.

The documentation for this struct was generated from the following file:

- **ExampleGenerator.h**

7.7 _ExampleGroupStats_ Struct Reference

7.7.1 Detailed Description

Sufficient statistics for Entropy and Gini.

The documentation for this struct was generated from the following file:

- **ExampleGroupStats.h**

7.8 _ExampleSpec_ Struct Reference

7.8.1 Detailed Description

Schema for training data.

The documentation for this struct was generated from the following file:

- **ExampleSpec.h**

7.9 _StatTracker_ Struct Reference

7.9.1 Detailed Description

Holds simple summary statistics of a sample.

See **stats.h** for more detail.

The documentation for this struct was generated from the following file:

- **stats.h**

7.10 _VFDT_ Struct Reference

7.10.1 Detailed Description

Holds the information needed to learn decision trees from data streams.

The documentation for this struct was generated from the following file:

- **vfdt-engine.h**

7.11 FloatListPtr Struct Reference

7.11.1 Detailed Description

An ADT for a list that holds floats.

See **lists.h** for more detail.

The documentation for this struct was generated from the following file:

- **lists.h**

7.12 HashTable Struct Reference

7.12.1 Detailed Description

A hash table ADT.

See **HashTable.h** for more detail.

The documentation for this struct was generated from the following file:

- **HashTable.h**

7.13 IntListPtr Struct Reference

7.13.1 Detailed Description

An ADT for a list that holds void ints.

See **lists.h** for more detail.

The documentation for this struct was generated from the following file:

- **lists.h**

7.14 VoidListPtr Struct Reference

7.14.1 Detailed Description

An ADT for a list that holds void pointers.

See **lists.h** for more detail.

The documentation for this struct was generated from the following file:

- **lists.h**

Chapter 8

VFML File Documentation

8.1 AttributeTracker.h File Reference

8.1.1 Detailed Description

Keep a record of which attributes are active.

Sometimes you need to keep track of which attributes you have already considered, and which are still available for consideration. The attribute tracker interface helps you efficiently acomplish this task; you can think of it as a wrapper around a bit field.

For example, when learning a decision tree, each split on a discrete attribute deactivates an attribute from further consideration. In this situation you would deactivate the attribute, and then make one clone for each child.

Functions

- AttributeTrackerPtr **AttributeTrackerNew** (int numAttributes)

  Allocates memory for a new AttributeTracker structure.

- void **AttributeTrackerFree** (AttributeTrackerPtr at)

  Frees the memory associated with the attribute tracker.

- AttributeTrackerPtr **AttributeTrackerInitial** (**ExampleSpecPtr** es)

  Respects 'ignore's in the ExampleSpec.

- AttributeTrackerPtr **AttributeTrackerClone** (AttributeTrackerPtr at)

  Creates a copy of the passed AttributeTracker and returns the copy.

- void **AttributeTrackerMarkActive** (AttributeTrackerPtr at, int attNum)

  Marks the specified attribute as active.

- void **AttributeTrackerMarkInactive** (AttributeTrackerPtr at, int attNum)

  Marks the specified attribute as inactive.

- int **AttributeTrackerIsActive** (AttributeTrackerPtr at, int attNum)

Returns 1 if the specified attribute is active.

- int **AttributeTrackerAreAllInactive** (AttributeTrackerPtr at)
  Returns 1 if none of the attributes are active.
- int **AttributeTrackerNumActive** (AttributeTrackerPtr at)
  Returns a count of at's active attributes.

8.1.2 Function Documentation

8.1.2.1 int AttributeTrackerAreAllInactive (AttributeTrackerPtr *at*)

Returns 1 if none of the attributes are active.

This is a constant time operation (it does not depend on the number of attributes).

8.1.2.2 AttributeTrackerPtr AttributeTrackerClone (AttributeTrackerPtr *at*)

Creates a copy of the passed AttributeTracker and returns the copy.

8.1.2.3 void AttributeTrackerFree (AttributeTrackerPtr *at*)

Frees the memory associated with the attribute tracker.

8.1.2.4 AttributeTrackerPtr AttributeTrackerInitial (ExampleSpecPtr *es*)

Respects 'ignore's in the ExampleSpec.

Creates an attribute tracker to track the attributes in the passed ExampleSpec. Any of es's attributes marked as ignore will be initially set to inactive.

8.1.2.5 int AttributeTrackerIsActive (AttributeTrackerPtr *at*, int *attNum*)

Returns 1 if the specified attribute is active.

8.1.2.6 void AttributeTrackerMarkActive (AttributeTrackerPtr *at*, int *attNum*)

Marks the specified attribute as active.

8.1.2.7 void AttributeTrackerMarkInactive (AttributeTrackerPtr *at*, int *attNum*)

Marks the specified attribute as inactive.

8.1.2.8 AttributeTrackerPtr AttributeTrackerNew (int *numAttributes*)

Allocates memory for a new AttributeTracker structure.

Creates a new attribute tracker with one bit for each of numAttributes. All the attributes are initially marked as active.

8.1.2.9 int AttributeTrackerNumActive (AttributeTrackerPtr *at*)

Returns a count of at's active attributes.

This is a constant time operation (it does not depend on the number of attributes).

8.2 batchtest File Reference

8.2.1 Detailed Description

Performs cross validation of a collection of learners on a collection of datasets.

Requires: **xvalidate** and **folddata** be in your path.

You can use batchtest with large datasets, but you will need enough disk space to hold 'folds' copies of the largest dataset. Of course, the learners you use with batchtest must also be able to work with the large datasets.

Batchtest expects to find two input files: one that contains descriptions of the datasets it should use, and one that contains descriptions of the learners it should run on them. In these files blank lines are ignored and lines beginning with '#' as the first character on the line are ignored as comments; every other line contains a description of a learner or a dataset. In the dataset file, the description lines have the following format:

```
[path to the directory holding the dataset] ::  [file stem of the dataset]
```

In the learners file, the description lines should contain the command to run with the appropriate arguments. When batchtest invokes a learner, it will run the exact line from the learners file with the `<filestem>` of the current cross-validation fold of the current dataset appended. The learner should be prepared to look for input in in C4.5 format in `<filestem>.names` and `<filestem>.data`; it should test on the examples in `<filestem>.test` and print the following to standard out:

```
error-rate  size
```

The learner's error rate on the test set, followed by some whitespace, followed by the size of the learned model (in whatever unit you want), followed by a newline.

Batchtest will collect the output of the runs of the learners, average them and report:

```
mean-error-rate (standard deviation of error rate) mean-size (standard deviation
of size) average-utime (standard deviation of utime) average-stime (standard
deviation of stime)
```

for example:

```
26.111 (5.500) 0.000 (0.000) 0.013 (0.005) 0.010 (0.008)
```

The times are very accurate on UNIX. Under CYGNUS (windows) utime will be a good estimate, but not as accurate and stime will be zero.

Wish List
I think the input format for batchtest is a little brittle and it could use some improvement

Arguments

- -data <dataset file>
  - Set the path to the file containing the dataset descriptions (default datasets)
- -learn <learner file>
  - Set the path to the file containing the learner descriptions (default learners)
- -folds <n>
  - Sets the number of train/test sets to create (default 10)

- -seed <n>
  - Sets the random seed, multiple runs with the same seed will produce the same datasets (defaults to a random seed). If you use a random seed, the value of the randomly selected seed will be printed at the start of the run. You can later use that seed to repeat the experiment. You can pass the same seed to `xvalidate` to focus on a particular learner/dataset combination for debugging. If that doesn't help enough, you can pass the same seed to `folddata` to recreate the exact test/training sets for closer inspection.
- -v
  - Can be used multiple times to increase the debugging output

Example

Contents of datasets file:

```
<p># Here are the datasets<br>
../../datasets/mushroom/ :: mushroom<br>
<br>
../../datasets/voting/ :: voting</p>
```

Contents of the learners file:

```
<p># A simple learner to set a baseline<br>
mostcommonclass -u -f<br>
# My fancy learner with a couple different parameter sets<br>
deep-thought -tc 4.7 -e 1.1 -u -f<br>
deep-thought -tc 2 -e 5 -u -f</p>
```

```
batchtest -data datasets -learn learners -folds 15 -seed 100
```

Does 15-fold cross-validation of the learners in the learners file on the datasets in the datasets file. It will use a seeded random number generator so the exact experiment could be reproduced. The actual calls to the learners will look something like this:

```
<p>mostcommonclass -u -f mushroom0<br>
mostcommonclass -u -f mushroom1<br>
mostcommonclass -u -f mushroom2<br>
...<br>
deep-thought -tc 4.7 -e 1.1 -u -f mushroom0<br>
deep-thought -tc 4.7 -e 1.1 -u -f mushroom1<br>
...<br>
etc...</p>
```

You should see the `using-batchtest` example for a more detailed example complete with sample -data and -learn files.

8.3 BeliefNet.h File Reference

8.3.1 Detailed Description

A Belief Net Structure with CPT local models.

This is the interface for creating, using, printing, & serializing Belief Nets (aka Bayesian Networks). This document will first describe the properties of BeliefNets and then of BeliefNetNodes. A belief net is a compact representation of a joint probability distribution of all of the variables in a domain. For each variable there is a local model (represented by a BeliefNetNode) that models the probability of the various values of that varable given the values of the variables that affect them. A BeliefNet is an acyclic graph of the nodes, where an edge represets (loosely) that the variable at the source affects the varable at the target. The exact semantics of an edge are a bit more complex. For a more detailed discussion see `David Heckerman's Tutorial`

Wish List

A version of this that uses DecisionTree local models instead of full CPTs. This would also need a new structure learning tool (a modification of **bnlearn**).

Data Structures

- struct **_BeliefNet_**

  Belief network ADT.

- struct **_BeliefNetNode_**

  Belief net node with full CPTs for local models.

BeliefNetNode

- typedef **_BeliefNetNode_ BeliefNetNodeStruct**

  Belief net node with full CPTs for local models.

- typedef **_BeliefNetNode_** ∗ **BeliefNetNode**

  Belief net node with full CPTs for local models.

- void **BNNodeAddParent** (**BeliefNetNode** bnn, **BeliefNetNode** parent)

  Adds the specified node as a parent to bnn.

- int **BNNodeLookupParentIndex** (**BeliefNetNode** bnn, **BeliefNetNode** parent)

  Returns the index of parent in bnn's parent list.

- int **BNNodeLookupParentIndexByID** (**BeliefNetNode** bnn, int id)

  Looks through the parent list of bnn for a node with node id of 'id'.

- void **BNNodeRemoveParent** (**BeliefNetNode** bnn, int parentIndex)

  Removes the node with index 'parentIndex' from bnn's parent list.

- **BeliefNetNode BNNodeGetParent** (**BeliefNetNode** bnn, int parentIndex)

  Returns the parent at position 'index' in the node's parent list.

- int **BNNodeGetParentID** (**BeliefNetNode** bnn, int parentIndex)

    Returns the node id of the node at position 'index' in the node's parent list.

- int **BNNodeGetNumParents** (**BeliefNetNode** bnn)

    Returns the number of nodes in the target node's parent list.

- int **BNNodeGetNumChildren** (**BeliefNetNode** bnn)

    Returns the number of nodes in the target node's child list.

- int **BNNodeHasParent** (**BeliefNetNode** bnn, **BeliefNetNode** parent)

    Returns 1 if and only if parent is in the node's parent list.

- int **BNNodeHasParentID** (**BeliefNetNode** bnn, int parentID)

    Returns 1 if and only if one of the node's parents has the specified node id.

- int **BNNodeGetNumValues** (**BeliefNetNode** bnn)

    Returns the number of values that the variable represented by the node can take.

- int **BNNodeGetNumParameters** (**BeliefNetNode** bnn)

    Returns the number of parameters in the node's CPT.

- char * **BNNodeGetName** (**BeliefNetNode** bnn)

    Returns the name of the node.

- int **BNNodeStructureEqual** (**BeliefNetNode** bnn, **BeliefNetNode** otherNode)

    Returns 1 if and only if the two nodes have the same parents in the same order.

- void **BNNodeInitCPT** (**BeliefNetNode** bnn)

    Allocates memory for bnn's CPT and zeros the values.

- void **BNNodeZeroCPT** (**BeliefNetNode** bnn)

    Sets the value of all CPT entries to zero.

- void **BNNodeFreeCPT** (**BeliefNetNode** bnn)

    Frees any memory being used by the node's CPTs.

- void **BNNodeAddSample** (**BeliefNetNode** bnn, **ExamplePtr** e)

    Increments the count of the appropriate CPT element by 1.

- void **BNNodeAddSamples** (**BeliefNetNode** bnn, **VoidListPtr** samples)

    Calls BNNodeAddSample for each example in the list.

- void **BNNodeAddFractionalSample** (**BeliefNetNode** bnn, **ExamplePtr** e, float weight)

    Increments the count of the appropriate CPT element by weight.

- void **BNNodeAddFractionalSamples** (**BeliefNetNode** bnn, **VoidListPtr** samples, float weight)

    Calls BNNodeAddFractionalSample for each example in the list.

- float **BNNodeGetCPTRowCount** (**BeliefNetNode** bnn, **ExamplePtr** e)

  Returns the number of samples that have been added to the node with the same parent combination as in e.

- float **BNNodeGetP** (**BeliefNetNode** bnn, int value)

  Returns the marginal probability of the appropriate value of the variable.

- float **BNNodeGetCP** (**BeliefNetNode** bnn, **ExamplePtr** e)

  Get the probability of the value of the target variable given the values of the parent variables.

- void **BNNodeSetCP** (**BeliefNetNode** bnn, **ExamplePtr** e, float probability)

  Sets the probability without affecting the sum of the CPT row for the parent combination.

- float **BNNodeGetNumSamples** (**BeliefNetNode** bnn)

  Returns the number of samples that have been added to the belief net node.

- int **BNNodeGetNumCPTRows** (**BeliefNetNode** bnn)

  Returns the number rows in the node's CPT. This is the number of parent combinations.

BeliefNet

- #define **BNGetUserData**(bn) (((**BeliefNet**)bn) → userData)

  Allows you to store an arbitrary pointer on the BeliefNet.

- typedef **_BeliefNet_ BeliefNetStruct**

  Belief network ADT.

- typedef **_BeliefNet_** ∗ **BeliefNet**

  Belief network ADT.

- **BeliefNet BNNew** (void)

  Creates a new belief net with no nodes.

- void **BNFree** (**BeliefNet** bn)

  Frees the memory associated with the belief net and all nodes.

- **BeliefNet BNClone** (**BeliefNet** bn)

  Makes a copy of the belief net and all nodes.

- **BeliefNet BNCloneNoCPTs** (**BeliefNet** bn)

  Makes a copy of the belief net and all nodes, but does not copy the local models at the nodes.

- **BeliefNet BNNewFromSpec** (**ExampleSpecPtr** es)

  Makes a new belief net from the example spec.

- int **BNGetSimStructureDifference** (**BeliefNet** bn, **BeliefNet** otherNet)

  Returns the symetric difference in the structures.

- void **BNSetName** (**BeliefNet** bn, char ∗name)

  Set's the Belief net's name.

- **ExampleSpec** ∗ **BNGetExampleSpec** (**BeliefNet** bn)

  Returns the ExampleSepc that is associated with the belief net.

- **BeliefNetNode** **BNGetNodeByID** (**BeliefNet** bn, int id)

  Gets the node with the associated index.

- int **BNGetNumNodes** (**BeliefNet** bn)

  Returns the number of nodes in the Belief Net.

- **BeliefNetNode** **BNGetNodeByElimOrder** (**BeliefNet** bn, int index)

  Returns nodes by their order in a topological sort.

- int **BNHasCycle** (**BeliefNet** bn)

  Returns 1 if and only if there is a cycle in the graphical structure of the belief net.

- void **BNFlushStructureCache** (**BeliefNet** bn)

  Needs to be called anytime you change network structure.

- void **BNAddSample** (**BeliefNet** bn, **ExamplePtr** e)

  Modifies all the CPTs in the network by adding a count to the approprite parameters.

- void **BNAddSamples** (**BeliefNet** bn, **VoidListPtr** samples)

  Calls BNAddSample for every example in the list.

- void **BNAddFractionalSample** (**BeliefNet** bn, **ExamplePtr** e, float weight)

  Modifies all the CPTs in the network by adding a weighted count to the approprite parameters.

- void **BNAddFractionalSamples** (**BeliefNet** bn, **VoidListPtr** samples, float weight)

  Calls BNAddFractionalSample for every example in the list.

- long **BNGetNumIndependentParameters** (**BeliefNet** bn)

  Returns the sum over all nodes of the number of independent parameters in the CPTs.

- long **BNGetNumParameters** (**BeliefNet** bn)

  Returns the sum over all nodes of the number of parameters in the CPTs.

- long **BNGetMaxNodeParameters** (**BeliefNet** bn)

  Returns the number of parameters in the node with the most parameters.

- **ExamplePtr** **BNGenerateSample** (**BeliefNet** bn)

  Samples from the distribution represented by bn.

- void **BNSetPriorStrength** (**BeliefNet** bn, double strength)

  Sets prior parameter strength as if some examples have been seen.

- void **BNSmoothProbabilities** (**BeliefNet** bn, double strength)

  Adds a number of samples equal to strength to each CPT entry in the network.

- void **BNSetUserData** (**BeliefNet** bn, void ∗data)
  Allows you to store an arbitrary pointer on the BeliefNet.
- **BeliefNet BNReadBIF** (char ∗fileName)
  Reads a Belief net from the named file.
- **BeliefNet BNReadBIFFILEP** (FILE ∗file)
  Reads a Belief net from a file pointer.
- void **BNWriteBIF** (**BeliefNet** bn, FILE ∗out)
  Writes the belief net to the file.
- void **BNPrintStats** (**BeliefNet** bn)
  Prints some information about the net to stdout.

Inference with Likelihood Sampling

The idea of likelihood sampling is that you have a set of query variables (specified by an example with some of the values unknown) and the system randomly generates samples for the query variables given the values of the non-query variables. After 'enough' samples the distribution of the samples generated for the query variables will be a good match to the 'true' distribution according to the net. The number of samples required can be very large, especially when dealing with events that don't occur often.

- **BeliefNet BNInitLikelihoodSampling** (**BeliefNet** bn, **ExamplePtr** e)
  Set up likelihood sampling and return place holder network.
- void **BNAddLikelihoodSamples** (**BeliefNet** bn, **BeliefNet** newNet, **ExamplePtr** e, int numSamples)
  Adds the requested number of samples to newNet.
- **BeliefNet BNLikelihoodSampleNTimes** (**BeliefNet** bn, **ExamplePtr** e, int numSamples)
  Combines a call to BNInitiLikelihoodSampling with a call to BNAddLikelihoodSamples.

8.3.2 Define Documentation

8.3.2.1 #define BNGetUserData(bn) (((BeliefNet)bn) → userData)

Allows you to store an arbitrary pointer on the BeliefNet.

You are responsible for managing any memory that it points to.

8.3.3 Typedef Documentation

8.3.3.1 typedef struct _BeliefNet_ ∗ BeliefNet

Belief network ADT.

See **BeliefNet.h** for more detail.

8.3.3.2 typedef struct _BeliefNetNode_ * BeliefNetNode

Belief net node with full CPTs for local models.

See **BeliefNet.h** for more detail.

8.3.3.3 typedef struct _BeliefNetNode_ BeliefNetNodeStruct

Belief net node with full CPTs for local models.

See **BeliefNet.h** for more detail.

8.3.3.4 typedef struct _BeliefNet_ BeliefNetStruct

Belief network ADT.

See **BeliefNet.h** for more detail.

8.3.4 Function Documentation

8.3.4.1 void BNAddFractionalSample (BeliefNet *bn*, ExamplePtr *e*, float *weight*)

Modifies all the CPTs in the network by adding a weighted count to the approprite parameters.

See the BeliefNetNode functions for a more detailed description of how the CPTs are represented and handled.

8.3.4.2 void BNAddFractionalSamples (BeliefNet *bn*, VoidListPtr *samples*, float *weight*)

Calls BNAddFractionalSample for every example in the list.

8.3.4.3 void BNAddLikelihoodSamples (BeliefNet *bn*, BeliefNet *newNet*, ExamplePtr *e*, int *numSamples*)

Adds the requested number of samples to newNet.

newNet should have been created by a call to BNInitiLikelihoodSampling. Adds the requested number of samples for the unknown variables in e using the distributions in bn.

8.3.4.4 void BNAddSample (BeliefNet *bn*, ExamplePtr *e*)

Modifies all the CPTs in the network by adding a count to the approprite parameters.

See the BeliefNetNode functions for a more detailed description of how the CPTs are represented and handled.

8.3.4.5 void BNAddSamples (BeliefNet *bn*, VoidListPtr *samples*)

Calls BNAddSample for every example in the list.

8.3.4.6 BeliefNet BNClone (BeliefNet *bn*)

Makes a copy of the belief net and all nodes.

8.3.4.7 BeliefNet BNCloneNoCPTs (BeliefNet *bn*)

Makes a copy of the belief net and all nodes, but does not copy the local models at the nodes.

8.3.4.8 void BNFlushStructureCache (BeliefNet *bn*)

Needs to be called anytime you change network structure.

That is, any time you add nodes, add or remove edges after calling BNHasCycle, or any ElimOrder stuff. The reason is that these two classes of functions cache topological sorts of the network and changing the structure invalidates these caches. In prinicipal, all of the functions that modify structure could be modified to automatically call this. I think things were done this way for efficiency (but I am not sure it actually helps efficiency...

8.3.4.9 void BNFree (BeliefNet *bn*)

Frees the memory associated with the belief net and all nodes.

8.3.4.10 ExamplePtr BNGenerateSample (BeliefNet *bn*)

Samples from the distribution represented by bn.

You are responsible for freeing the returned example (using ExampleFree when you are done with it.

8.3.4.11 ExampleSpec∗ BNGetExampleSpec (BeliefNet *bn*)

Returns the ExampleSepc that is associated with the belief net.

The spec will be automatically created when you read the network from disk or as you add nodes to the net and values to the nodes.

8.3.4.12 long BNGetMaxNodeParameters (BeliefNet *bn*)

Returns the number of parameters in the node with the most parameters.

8.3.4.13 BeliefNetNode BNGetNodeByElimOrder (BeliefNet *bn*, int *index*)

Returns nodes by their order in a topological sort.

If the nodes can not be topologically sorted (perhaps because there is a cycle) this function returns 0.

Note that this function caches (and uses a cache) of the topological sort and that you might want to call BNFlushStructureCache before calling this if you've changed the structure of the net since the cache was filled.

8.3.4.14 BeliefNetNode BNGetNodeByID (BeliefNet *bn*, int *id*)

Gets the node with the associated index.

This is 0 based (like a C array).

8.3.4.15 long BNGetNumIndependentParameters (BeliefNet *bn*)

Returns the sum over all nodes of the number of independent parameters in the CPTs.

This is different from the total number of parameters because one of the parameters in each row can be determined from the values of the others, and so is not independent.

8.3.4.16 int BNGetNumNodes (BeliefNet *bn*)

Returns the number of nodes in the Belief Net.

8.3.4.17 long BNGetNumParameters (BeliefNet *bn*)

Returns the sum over all nodes of the number of parameters in the CPTs.

8.3.4.18 int BNGetSimStructureDifference (BeliefNet *bn*, BeliefNet *otherNet*)

Returns the symetric difference in the structures.

This is defined as the sum for i in nodes of the number of parents that node i of bn has but node i of otherBN does not have plus the number of parents that node i of otherBN has that node i of bn does not have.

8.3.4.19 int BNHasCycle (BeliefNet *bn*)

Returns 1 if and only if there is a cycle in the graphical structure of the belief net.

Note that this function caches (and uses a cache) of the topological sort and that you might want to call BNFlushStructureCache before calling this if you've changed the structure of the net since the cache was filled.

8.3.4.20 BeliefNet BNInitLikelihoodSampling (BeliefNet *bn*, ExamplePtr *e*)

Set up likelihood sampling and return place holder network.

Returns a new belief net with CPT set to start to acumulate samples for the unknown variables in e, you should free this net when you are done with it. Once the net is created you should add as many samples to it as you like using BNAddLikelihoodSamples and then check the CPTs at the appropriate nodes for the generated distributions.

8.3.4.21 BeliefNet BNLikelihoodSampleNTimes (BeliefNet *bn*, ExamplePtr *e*, int *numSamples*)

Combines a call to BNInitiLikelihoodSampling with a call to BNAddLikelihoodSamples.

8.3.4.22 BeliefNet BNNew (void)

Creates a new belief net with no nodes.

8.3.4.23 BeliefNet BNNewFromSpec (ExampleSpecPtr *es*)

Makes a new belief net from the example spec.

All attributes in the spec should be discrete. This adds a node, with the appropriate values, to the net for each variable in the spec. The resulting network has no edges and zeroed CPTs

8.3.4.24 void BNNodeAddFractionalSample (BeliefNetNode *bnn*, ExamplePtr *e*, float *weight*)

Increments the count of the appropriate CPT element by weight.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.25 void BNNodeAddFractionalSamples (BeliefNetNode *bnn*, VoidListPtr *samples*, float *weight*)

Calls BNNodeAddFractionalSample for each example in the list.

8.3.4.26 void BNNodeAddParent (BeliefNetNode *bnn*, BeliefNetNode *parent*)

Adds the specified node as a parent to bnn.

Both nodes should be from the same BeliefNet structure.

8.3.4.27 void BNNodeAddSample (BeliefNetNode *bnn*, ExamplePtr *e*)

Increments the count of the appropriate CPT element by 1.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.28 void BNNodeAddSamples (BeliefNetNode *bnn*, VoidListPtr *samples*)

Calls BNNodeAddSample for each example in the list.

8.3.4.29 void BNNodeFreeCPT (BeliefNetNode *bnn*)

Frees any memory being used by the node's CPTs.

This should be called before changing the node's parents. After a call to this function, you should call BNNodeInitCPT for the node before making any calls that might try to access the CPT (adding samples, doing inference, smoothing probability, comparin networks, etc).

8.3.4.30 float BNNodeGetCP (BeliefNetNode *bnn*, ExamplePtr *e*)

Get the probability of the value of the target variable given the values of the parent variables.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.31 float BNNodeGetCPTRowCount (BeliefNetNode *bnn*, ExamplePtr *e*)

Returns the number of samples that have been added to the node with the same parent combination as in e.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.32 char* BNNodeGetName (BeliefNetNode *bnn*)

Returns the name of the node.

8.3.4.33 int BNNodeGetNumChildren (BeliefNetNode *bnn*)

Returns the number of nodes in the target node's child list.

8.3.4.34 int BNNodeGetNumCPTRows (BeliefNetNode *bnn*)

Returns the number rows in the node's CPT. This is the number of parent combinations.

8.3.4.35 int BNNodeGetNumParameters (BeliefNetNode *bnn*)

Returns the number of parameters in the node's CPT.

8.3.4.36 int BNNodeGetNumParents (BeliefNetNode *bnn*)

Returns the number of nodes in the target node's parent list.

8.3.4.37 float BNNodeGetNumSamples (BeliefNetNode *bnn*)

Returns the number of samples that have been added to the belief net node.

8.3.4.38 int BNNodeGetNumValues (BeliefNetNode *bnn*)

Returns the number of values that the variable represented by the node can take.

8.3.4.39 float BNNodeGetP (BeliefNetNode *bnn*, int *value*)

Returns the marginal probability of the appropriate value of the variable.

That is, the sum over all rows of the number of counts for that value divided by the sum over all rows of the number of counts in the row.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.40 BeliefNetNode BNNodeGetParent (BeliefNetNode *bnn*, int *parentIndex*)

Returns the parent at position 'index' in the node's parent list.

8.3.4.41 int BNNodeGetParentID (BeliefNetNode *bnn*, int *parentIndex*)

Returns the node id of the node at position 'index' in the node's parent list.

8.3.4.42 int BNNodeHasParent (BeliefNetNode *bnn*, BeliefNetNode *parent*)

Returns 1 if and only if parent is in the node's parent list.

8.3.4.43 int BNNodeHasParentID (BeliefNetNode *bnn*, int *parentID*)

Returns 1 if and only if one of the node's parents has the specified node id.

8.3.4.44 void BNNodeInitCPT (BeliefNetNode *bnn*)

Allocates memory for bnn's CPT and zeros the values.

This allocates enough memory to hold one float for each value of the variable associated with the node for each parent combination (an amount of memory that is exponential in the number of parents). This should be called once all parents are in place.

8.3.4.45 int BNNodeLookupParentIndex (BeliefNetNode *bnn*, BeliefNetNode *parent*)

Returns the index of parent in bnn's parent list.

Returns -1 if parent is not one of node's parents.

8.3.4.46 int BNNodeLookupParentIndexByID (BeliefNetNode *bnn*, int *id*)

Looks through the parent list of bnn for a node with node id of 'id'.

See BNNodeGetID for more info.

8.3.4.47 void BNNodeRemoveParent (BeliefNetNode *bnn*, int *parentIndex*)

Removes the node with index 'parentIndex' from bnn's parent list.

To remove the node 'parent' call BNNodeRemoveParent(bnn, BNNodeLookupParentIndex(bnn, parent)).

8.3.4.48 void BNNodeSetCP (BeliefNetNode *bnn*, ExamplePtr *e*, float *probability*)

Sets the probability without affecting the sum of the CPT row for the parent combination.

This means that the probability has the same prior weight before and after a call to this. Put another way, the set probability is equivilant to having seen the same number of samples at the new probability as at the old.

Looks in example to get the values for the parents and the value for the variable. If any of these are unknown changes nothing, prints a low priority warning message, and returns -1 where applicable.

8.3.4.49 int BNNodeStructureEqual (BeliefNetNode *bnn*, BeliefNetNode *otherNode*)

Returns 1 if and only if the two nodes have the same parents in the same order.

Bug

Only returns 1 if the parents are in the same order, but the order probably shouldn't matter.

8.3.4.50 void BNNodeZeroCPT (BeliefNetNode *bnn*)

Sets the value of all CPT entries to zero.

Can be called after InitCPT to reset all the table's values to 0.

8.3.4.51 void BNPrintStats (BeliefNet *bn*)

Prints some information about the net to stdout.

The information includes num nodes, min max avg num parents, etc.

8.3.4.52 BeliefNet BNReadBIF (char ∗ *fileName*)

Reads a Belief net from the named file.

The file should contain a net in `Bayesian Interchange Format` (BIF).

8.3.4.53 BeliefNet BNReadBIFFILEP (FILE ∗ *file*)

Reads a Belief net from a file pointer.

The file pointer should be opened for reading and should contain a net in `Bayesian Interchange Format` (BIF).

8.3.4.54 void BNSetName (BeliefNet *bn*, char ∗ *name*)

Set's the Belief net's name.

This doesn't really affect anything (except it is recorded if you write out the belief net), but using it may make you feel better.

8.3.4.55 void BNSetPriorStrength (BeliefNet *bn*, double *strength*)

Sets prior parameter strength as if some examples have been seen.

Multiplies all the counts in all of the network's CPTs so that each parent combination has the equivilant of strength samples, divided acording to that combination's distribution in the network.

Hrm, does that confuse you too?

8.3.4.56 void BNSetUserData (BeliefNet *bn*, void ∗ *data*)

Allows you to store an arbitrary pointer on the BeliefNet.

You are responsible for managing any memory that it points to.

8.3.4.57 void BNSmoothProbabilities (BeliefNet *bn*, double *strength*)

Adds a number of samples equal to strength to each CPT entry in the network.

This effectivly smooths the probabilities towards uniform.

8.3.4.58 void BNWriteBIF (BeliefNet *bn*, FILE ∗ *out*)

Writes the belief net to the file.

Out should be a file open for writing, pass stdout to write to the console. The net it written in `Bayesian Interchange Format` (BIF).

8.4 beliefnetcorrupt File Reference

8.4.1 Detailed Description

Makes some random changes to a BeliefNet.

This program loading an existing belief network (in `BIF format`) and corrupts it in several, user controlable, ways. This can be used to generate prior networks for structure learning, or to create several similar synthetic concepts (with the help of `beliefnetdata`) from a single `benchmark network`.

Multiple runs with the same seed parameter produce the same results.

VFML comes with a collection of benchmark belief nets, and you may want `more information on these`.

Wish List

This tool does not do anything smart with parameters when it changes the structure, and it should.

Arguments

- -f <file name for output>
  - (default DFOut.bif)
- -target <dir>
  - Set the directory to contain the output dataset (default '.')
- -bnf <file containing belief net>
  - (default DF.bif)
- -stdout
  - output the new net to stdout (default to <stem>.data)
- -startFromEmpty
  - Remove all links from net before making any changes (default start from the net the way it is)
- -epsilon <val>
  - Change every parameter in the network by adding or subtracing (with even probability) a number selected randomly in the range 0 – *val*. The CPTs in the network are then renormalized. This step is taken before any structure changes are made. The default is to leave parameters unchanged and to only change structure.
- -maxParentsPerNode <num>
  - Limit each node to <num> parents (default no limit)
- -numChanges <num>

  - Randomly add 2∗num, then remove 2∗num, then reverse 2∗num links (default num defaults to 4). Each of these changes affects the parameters in the network. The current version of this tool does not do anything smart with these, so once this option is invoked do not trust the parameter values. This option will not introduce changes that violate the *maxParentsPerNode* parameter, or that introduce cycles. If it is trying to add or reverse a link and can not without violating these constraints after trying 100 random operations beliefnetcorrupt will give up on the change and move on to the next one. This option is overridden by the -random option if it is used.

- -random <num>
  - Random starting net with <num> links. This option will override the (default) num-Changes behavior. More specifically, this option starts by removing all edges from the net, and then adding random edges (so that no changes add a cycle or violate the maxParents constraint).

- -seed <seed>
  - Sets the random seed, multiple runs with the same options and seed will produce the same network (defaults to a random seed)

- -h
  - Run with this argument to get a list of arguments and their meanings.

- -v
  - Can be used multiple times to increase the debugging output

Example

```
beliefnetcorrupt -f corrupted -bnf alarm.bif -numChanges 2 -seed 111
```

Creates a network based on the alarm network by adding 2 links, then removing 2 links, then reversing 2 links. Reproduce the same network everytime the same arguments are used (thanks to the seed parameter).

8.5 beliefnetdata File Reference

8.5.1 Detailed Description

Creates a data set by sampling from a Bayesian Network.

>This program creates a synthetic data set by loading an existing belief network (in `BIF format`) and sampling from it, possibly introducing noise. This program will also create a .names file for the resulting data set. More specifically, belifnetdata reads the network, generates samples from it with **BNGenerateSample**, and adds noise to them (if requested) with **ExampleAddNoise**.

Multiple runs with the same seed parameter produce the same results. Also note that running this command with one level of -v will output some statistics about the belif net which you might find useful.

VFML comes with a collection of benchmark belief nets, and you may want `more information on these`.

Arguments

- -h
  - Run with this argument to get a list of arguments and their meanings.

Example

```
beliefnetdata -f train -bnf alarm.bif -train 1000 -seed 111 -noise 5
```

Creates 1000 samples from the alarm network, randomly corrupts 5% of their values, write the resulting samples to train.data (and create a file train.names) and reproduce the same data set everytime the same arguments are used (thanks to the seed parameter)

8.6 beliefnetscore File Reference

8.6.1 Detailed Description

Tests a BeliefNet in several ways.

This program determines the log-likelihood of a data set given a belief net. It can also compare the structure of two networks. In log-likelihood mode it loads the belief net, and then scans the data set, accumulating the likelihood of each example in the data set given the network.

beliefnetscore can smooth the parameters in the network before computing this likelihood. (Run beliefnetscore -h for the precice parameters to use.) This smoothing works as follows. Each parameter in the network is multiplied by the desired strength, and then 1 is added to each local model is renormalized. If you do not use this argument, and there is a 0 probability in the network, but that even occurs in the data set, beliefnetscore will crash.

In comparison mode it loads both networks and the outputs the structural difference between the two networks. This is sometimes known as the symetric difference and is measured by iterating over the nodes in each network and counting the number of times that the node has a parent that the coresponding node in the other network does not have.

Wish List

Move the comparision mode from this tool into a new tool, beliefnetcompare, and have that tool do more interesting comparisions (e.g. measure the KL-distance between the distriutions encoded in the networks).

Arguments

- -h
  - Run with this argument to get a list of arguments and their meanings.

8.7 bindata File Reference

8.7.1 Detailed Description

Converts continuous attributes into discrete ones.

Converts all continuous attributes in a data set to categorical ones. Uses two passes over data, one to gather the stats needed to pick bin boundaries, and one to do the conversion (although the first pass can be done on a sample with the -samples argument below).

bindata uses one of two methods to select bin boundaries. The first is to find the range of each attribute (by identifing its highest and lowest value) and then dividing the range into even with bins. This is the default method. The other method assumes that the attribute was generated from a Gaussian, estimates the mean and variance of the Gaussian from data, and sets bin boundaries so that each bin holds an even amount of the Gaussian's probability mass.

Thanks
to Chun-Hsiang Hung for doing the core development work for this tool.

Wish List
that this tool would have more methods for selecting bin boundaries, for example to reduce entropy.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -fout <filestem>
  - Set the name of the output dataset (default DF-out)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -target <dir>
  - Set the directory to contain the output dataset (default '.')
- -test
  - Also converts the test set (but does not use the test set to pick boundaries)
- -samples <n>
  - Use the first n examples to pick bin boundaries (default use the whole train set)
- -bins <n>
  - Sets the number of bins (default 10)
- -gaussian
  - Pick bin boundaries by fitting a Gaussian and making even probability bins
- -h
  - Display usage information and exit.

- -v
  - Can be used multiple times to increase the debugging output

8.8 bitfield.h File Reference

8.8.1 Detailed Description

Compactly represent a bit field.

Data Structures

- struct **_BITFIELD_**

  ADT for compactly representing a bit field.

Typedefs

- typedef **_BITFIELD_ BitFieldStruct**

  ADT for compactly representing a bit field.

- typedef **_BITFIELD_** ∗ **BitFieldPtr**

  ADT for compactly representing a bit field.

Functions

- BitField **BitFieldNew** (int byteSize)

  Create a new BitField with the specified number of bits.

- void **BitFieldFree** (BitField b)

  Frees the memory associated with the BitField.

- int **BitFieldGetNumBytes** (BitField b)

  Returns the number of bytes being used to represent the BitField.

- int **BitFieldGetBit** (BitField b, long offset)

  Returns the value of the specifed bit.

- void **BitFieldSetBit** (BitField b, long offset, int val)

  Sets the value of the specified bit.

8.8.2 Typedef Documentation

8.8.2.1 typedef struct _BITFIELD_ ∗ BitFieldPtr

ADT for compactly representing a bit field.

See **bitfield.h** for more detail.

8.8.2.2 typedef struct _BITFIELD_ BitFieldStruct

ADT for compactly representing a bit field.

See **bitfield.h** for more detail.

8.8.3 Function Documentation

8.8.3.1 void BitFieldFree (BitField *b*)

Frees the memory associated with the BitField.

8.8.3.2 int BitFieldGetBit (BitField *b*, long *offset*)

Returns the value of the specifed bit.

8.8.3.3 int BitFieldGetNumBytes (BitField *b*)

Returns the number of bytes being used to represent the BitField.

8.8.3.4 BitField BitFieldNew (int *byteSize*)

Create a new BitField with the specified number of bits.

8.8.3.5 void BitFieldSetBit (BitField *b*, long *offset*, int *val*)

Sets the value of the specified bit.

8.9 bnlearn File Reference

8.9.1 Detailed Description

Learns the structure of a BeliefNet from a data set.

Learns the structure and parameters of a Bayesian network using the standard method. The structural prior is set by using P(S) = kappa ^ (symetric difference between the net and the prior net). The parameter prior is K2. This is basically a wrapper around the code in bnlearn-engine.h

bnlearn can load training data into RAM, if space is available, or it can scan data repeatedly from disk.

See the documentation for the bnlearn-engine for more information about the parameters and their meanings.

bnlearn takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data`.

Thanks
to Matt Richardson for making substantial optimizations to the bnlearn program.

Wish List
A version of this program that is intelligent about dealing with unobserved (or partially observed) variables.

Arguments

- -h
  - Run bnlearn -h for a list of the arguments and their meanings.

8.10 bnlearn-engine.h File Reference

8.10.1 Detailed Description

Learn the structure of a BeliefNet from data.

An API to VFML's Belief Net structure learning engine. The **bnlearn** program is basically a wrapper around this interface. You can use this interface to avoid making system calls to learn belief nets, or the callbacks in this interface may allow you to modify the learner's behavior enough so that you don't have to edit any code.

Learns the structure and parameters of a Bayesian network using the standard method. The structural prior is set by using P(S) = kappa ^ (symetric difference between the net and the prior net). The parameter prior is K2.

To use this interface you generate a parameters structure using the BNLearn_NewParams method, fill in any parameters you want (to identify prior networks, callbacks, training data, etc), then call the BN_Learn method, extract what you want from the parameters structure, and then free the structure with BNLearn_FreeParams. See the header file for the details of the available parameters.

Thanks
to Matthew Richardson for extracting this interface from the bnlearn learner.

Functions

- BNLearnParams * **BNLearn_NewParams** ()

  Makes a new parameters structure with some default values.

- void **BNLearn_FreeParams** (BNLearnParams *params)

  Frees the parameters structure.

- void **BNLearn** (BNLearnParams *params)

  Do the learning.

8.10.2 Function Documentation

8.10.2.1 void BNLearn (BNLearnParams * *params*)

Do the learning.

8.10.2.2 void BNLearn_FreeParams (BNLearnParams * *params*)

Frees the parameters structure.

You are responsible for freeing any additonal memory that may have been allocated by the run (for example any learned BeliefNet).

8.10.2.3 BNLearnParams* BNLearn_NewParams ()

Makes a new parameters structure with some default values.

8.11 C45interface.h File Reference

8.11.1 Detailed Description

Calls the C4.5 decision tree learning system and returns the learned tree.

An interface that writes the examples to a file, calls C4.5 on them, and returns the tree that C4.5 learns. See the `example program` for more information on how to use this interface. C4.5 must be in your path for this to work, you can grab it from Professor Quinlan's homepage at `http://www.cse.unsw.edu.au/~quinlan/`.

Functions

- **DecisionTreePtr C45Learn** (**ExampleSpecPtr** es, **VoidListPtr** examples)
  Makes the call and returns the tree.

8.11.2 Function Documentation

8.11.2.1 DecisionTreePtr C45Learn (ExampleSpecPtr *es*, VoidListPtr *examples*)

Makes the call and returns the tree.

8.12 c45wrapper File Reference

8.12.1 Detailed Description

Calls C4.5 and tests the learned tree.

A wrapper that calls C4.5 to learn a decision tree, times its execution, and extracts the important bits from its output. C4.5 must be in your path for this learner to work, you can get it from Professor Quinlan's homepage at `http://www.cse.unsw.edu.au/~quinlan/`.

This wrapper runs C4.5, tests the learned tree on the test data (if appropriate) and outputs: `test set error rate <whitespace> size of tree <whitespace> seconds`.

You might also be interested in the **DecisionTreeReadC45** function.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -args <string>
  - Passes these additional arguments to C4.5 (default none)
- -noPrune
  - Reports the results from c4.5's unpurned tree (default reports from pruned tree)
- -noTest
  - Suppress the -u argument, which makes C4.5 output accuracy on the training set (default test accuracy on the <filestem>.test file)
- -v
  - Can be used multiple times to increase the debugging output

8.13 c50wrapper File Reference

8.13.1 Detailed Description

Calls C5.0 and tests the learned tree.

A wrapper that calls C5.0 to learn a decision tree, times its execution, and extracts the important bits from its output. C5.0 must be in your path for this learner to work. It is a commercial product and you can get information about purchasing it from Professor Quinlan's homepage at `http://www.cse.unsw.edu.au/∼quinlan/`.

This wrapper runs C5.0, tests the lerned tree on the test data (if appropriate) and outputs: `test set error rate <whitespace> size of tree <whitespace> seconds`.

Arguments

- -f <filestem> -Set the stem name (default DF)
- -args <string>
  - Passes these additional arguments to C4.5 (default none)
- -v
  - Can be used multiple times to increase the debugging output

8.14 cleandata File Reference

8.14.1 Detailed Description

Cleans up a data set in several ways.

This tool cleans a data set in several ways, and outputs the cleaned data. It will scan the training data and gather some simple stats about it (if needed, see below). It then does a pass over training and testing data, filling in missing categorical attribute values with the most common value given the class, and filling in missing continuous attribute values with the average value given the class.

The tool can also add a new attribute value to each categorical attribe, called 'u' (short for unknown) and rewrite the data set as appropriate.

The tool can also remove every attribute in the data set that is marked ignored.

cleandata accepts an input stem and expects to find a file *stem*.data and *stem*.names and optionally one called *stem*.test. cleandata outputs a file named *stem*-clean.data, *stem*-clean.names, and optionally *stem*-clan.test.

cleandata will work with a single pass over the data set if and only if you use -addValue and there are no continuous attributes. Otherwise it usues an additional pass.

Thanks
to Chun-Hsiang Hung for doing the core development work for this tool.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -addValue
  - Adds a 'u' for unknown value to all categorical attributes (default fill in the most common value)
- -removeIgnore
  - removes every attribute marked 'ignored' in names from the data set (default leave them in)
- -h
  - Display usage information and exit.
- -v
  - Can be used multiple times to increase the debugging output

8.15 clusterdata File Reference

8.15.1 Detailed Description

Creates a synthetic data set from randomly generated clusters.

This program creates a synthetic data set by selecting cluster centroids and generating samples by ranomly picking a centroid and sampling from a spherical gaussian with the centroid as its mean and a user specified standard deviation. This data generator has been used to evaluate the VFKM and VFEM systems.

The centroids are randomly placed uniformly in a N dimensional unit hypercube (where N is the number of continuous dimensions), except that if any centroid is placed closer than: `(sqrt(N) / (num centroids + 1)) * std deviation` to an already placed one its location is resampled. (If any centroid can not be placed after 1000 resamples and error is reported.) Note that 'unit hypercube' means that each dimension ranges from 0 - 1.0.

Finally, training samples are generated by randomly selecting a centroid and sampling from a Gaussian with it as the mean and the specified standard deviation (specified by a parameter to the program) for each dimension. Note that the value of a sample's dimension may fall outside the 0 - 1.0 range.

Arguments

- -h
  - Run with this argument to get a list of arguments and their meanings.

Example

```
clusterdata -continuous 20 -clusters 3 -stdev 0.05 -conceptSeed 21 -seed 1234
-train 1000
```

Creates 1000 samples in 20 dimensions by sampling from a mixture of 3 Gaussians with a standard deviation of 0.05. This same data set could be recreated by using the same *seed* and *conceptSeed* flags.

8.16 combinedata File Reference

8.16.1 Detailed Description

Combines a series of data sets into a single large one.

Sometimes you may need to combine a collection of small data files into one large one. Usually this could be accomplished with the unix cat utility, but sometimes it will not be available and sometimes it may not perfrom as needed (for example, when the source files do not end with newlines). combinedata was created for those situations.

combinedata reads a collection of files as specified by the required -dataFiles argument and writes them to the name specified by the -fout argument.

Arguments

- -names <filename>
  - Set the names file (default DF.names)
- -fout <filename>
  - Set the name of the output dataset (default DF-out)
- -target <dir>
  - Set the output directory (default '.')
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -h
  - Display usage information and exit
- -v
  - Can be used multiple times to increase the debugging output
- -dataFiles <list of files>
  - till the end of the line list the files to combine (REQUIRED)

Example

```
combinedata -names test -fout big-data -dataFiles small-data0 small-data1
small-data2
```

Will create a file called big-data.data which contains the examples from the three specified files. All data files had better share the format specified in test.names.

8.17 cvfdt File Reference

8.17.1 Detailed Description

Learns a DecisionTree from a high-speed time-changing data stream (or very large data set).

Learns a decision tree from a high-speed time-changing data stream or a very large data set as described in `this paper`. cvfdt does not work with continuous attributes.

cvfdt takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data`.

Thanks
to Laurie Spencer for doing the core development work for cvfdt.

Wish List
Modify this learner to work with continuous attributes.
An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

Arguments

- -h
  - Run cvfdt -h for a list of the arguments and their meanings.

8.18 Debug.h File Reference

8.18.1 Detailed Description

A set of functions that help your programs produce debugging output in a consistent way.

Functions

- void **DebugWarn** (int condition, char ∗str)
  Outputs string to the current target if condition is non-zero.
- void **DebugError** (int condidion, char ∗str)
  If condition is non-zero outputs string to the current target and halts the program.
- void **DebugSetMessageLevel** (int level)
  Sets the level of message that will be reported by DebugMessage.
- int **DebugGetMessageLevel** (void)
  Returns the current message level.
- void **DebugMessage** (int condition, int level, char ∗str,...)
  Formats and prints a message if conditions are met.
- void **DebugSetTarget** (FILE ∗target)
  Sets the target for debugging output.
- FILE ∗ **DebugGetTarget** (void)
  Gets the current target for debugging output.

8.18.2 Function Documentation

8.18.2.1 void DebugError (int *condidion*, char ∗ *str*)

If condition is non-zero outputs string to the current target and halts the program.

8.18.2.2 int DebugGetMessageLevel (void)

Returns the current message level.

8.18.2.3 FILE∗ DebugGetTarget (void)

Gets the current target for debugging output.

8.18.2.4 void DebugMessage (int *condition*, int *level*, char ∗ *str*, ...)

Formats and prints a message if conditions are met.

If the condition is non-zero and the current message level is greater than level this calls printf with str as the format string, and any additional arguments as arguments.

8.18.2.5 void DebugSetMessageLevel (int *level*)

Sets the level of message that will be reported by DebugMessage.

Defaults to 0.

8.18.2.6 void DebugSetTarget (FILE ∗ *target*)

Sets the target for debugging output.

Defaults to stdout. All future calls to the Debug functions will sent output to the specified file.

8.18.2.7 void DebugWarn (int *condition*, char ∗ *str*)

Outputs string to the current target if condition is non-zero.

8.19 decisionstump File Reference

8.19.1 Detailed Description

Learns a decision stump (a DecisionTree with only one split).

This is a very simple learner, but it may be useful as a baseline to compare your learner against.

The decisionstump learner works in time proportional to the number of training examples. It also requires memory that is proportional to the number of classes $*$ number attributes $*$ number of values. Note that this can be very large for continuous attributes (which, in the worst case, have a separate value for each training example). The maxThresholds argument can be used to control this.

The learner takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data`. Depending on command line argument, it will either output the decision stump or test its error rate on `<stem>.test`.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -u
  - Test on the examples in <stem>.test and output in a format appropriate for interface with xvalidate and batchtest (defaults to off)
- -a
  - Outputs the name of the selected attribute
- -outputGains
  - Outputs the information gain (and best threshold) for each attribute
- -outputStump
  - Prints the learned stump. Defaults to output without the -u flag and not output with it
- -maxThresholds <num>
  - Use the first num values from the training set as thresholds for continuous attributes, allows this program to be run on very large data sets (default use all values)
- -v
  - Can be used multiple times to increase the debugging output

Example

```
decisionstump -f banana -source datasets/banana
```

Looks for a dataset named 'banana' in the 'datasets/banana' directory. Outputs the decision stump learned from the data set.

8.20 DecisionTree.h File Reference

8.20.1 Detailed Description

A Decision Tree Structure.

This is the interface for creating, using, printing, & serializing Decision Trees. A decision tree is a recursive structure. Each internal node partitions the data based on the values of an attribute. Each leaf contains a prediction for the distinguished target attribute. For a more detailed discussion see chapter 3 of `Tom Mitchell's book on machine learning`.

Note that all the DecisionTrees created with an ExampleSpec maintain a pointer to the it; you shouldn't free or modify the ExampleSpec until you are done with all the DecisionTrees referencing it.

Wish List

A standard in memroy decision tree induction algorithm. Maybe the best starting point would be the **decisionstump** learner.
This isn't the right place for this wish, but it would be nice to have a RuleSet structure similar to this DecisionTree structure

Data Structures

- struct **_DecisionTree_**

  ADT for working with decision trees.

Typedefs

- typedef **_DecisionTree_ DecisionTree**

  ADT for working with decision trees.

- typedef **_DecisionTree_** ∗ **DecisionTreePtr**

  ADT for working with decision trees.

Functions

- **DecisionTreePtr DecisionTreeNew** (**ExampleSpecPtr** spec)

  Creates a new decision tree node.

- void **DecisionTreeFree** (**DecisionTreePtr** dt)

  Frees the memory associated with the decision tree and all of its children.

- **DecisionTreePtr DecisionTreeClone** (**DecisionTreePtr** dt)

  Recursively creates a copy of dt and returns it.

- int **DecisionTreeIsLeaf** (**DecisionTreePtr** dt)

  Returns 1 if dt is a leaf node and 0 otherwise.

- int **DecisionTreeIsTreeGrowing** (**DecisionTreePtr** dt)

Returns 1 if dt is a growing node has any children which are, and 0 otherwise.

- int **DecisionTreeIsNodeGrowing** (**DecisionTreePtr** dt)

  Returns 1 if dt is a growing node and 0 otherwise.

- int **DecisionTreeGetClass** (**DecisionTreePtr** dt)

  Returns the index of the class which dt predicts.

- void **DecisionTreeSetClass** (**DecisionTreePtr** dt, int theClass)

  Sets dt's class prediction to theClass.

- void **DecisionTreeAddToClassDistribution** (**DecisionTreePtr** dt, **ExamplePtr** e)

  Put the example in the class distribution at the node.

- float **DecisionTreeGetClassProb** (**DecisionTreePtr** dt, int theClass)

  Returns the probability of the class.

- void **DecisionTreeSetClassProb** (**DecisionTreePtr** dt, int theClass, float prob)

  Sets the probability of the class.

- float **DecisionTreeGetClassDistributionSampleCount** (**DecisionTreePtr** dt)

  Returns the number of samples added to the node's distribution.

- void **DecisionTreeZeroClassDistribution** (**DecisionTreePtr** dt)

  Sets the nodes distribution to zeros.

- void **DecisionTreeSetTypeLeaf** (**DecisionTreePtr** dt)

  Changes dt into a leaf node without changing its class prediction.

- void **DecisionTreeSetTypeGrowing** (**DecisionTreePtr** dt)

  Changes dt into a growing node.

- void **DecisionTreeSplitOnDiscreteAttribute** (**DecisionTreePtr** dt, int attNum)

  Changes dt into a discrete split.

- void **DecisionTreeSplitOnContinuousAttribute** (**DecisionTreePtr** dt, int attNum, float threshold)

  Changes dt into a continuous split.

- int **DecisionTreeGetChildCount** (**DecisionTreePtr** dt)

  Returns a count of the direct decendants of dt.

- **DecisionTreePtr DecisionTreeGetChild** (**DecisionTreePtr** dt, int index)

  Returns one of the direct decendants of dt.

- **DecisionTreePtr DecisionTreeOneStepClassify** (**DecisionTreePtr** dt, **ExamplePtr** e)

  Does one step of classifing e with dt.

- int **DecisionTreeClassify** (**DecisionTreePtr** dt, **ExamplePtr** e)

Uses dt to classify e and returns the index of the predicted class.

- void **DecisionTreeGatherGrowingNodes** (**DecisionTreePtr** dt, VoidAListPtr list)
  Searches dt and appends all its growing nodes to the passed list.
- void **DecisionTreeGatherLeaves** (**DecisionTreePtr** dt, VoidAListPtr list)
  Searches dt and appends all its leaf nodes to the passed list.
- int **DecisionTreeCountNodes** (**DecisionTreePtr** dt)
  Returns a count of the number of nodes (of any type) in dt.
- int **DecisionTreeGetMostCommonClass** (**DecisionTreePtr** dt)
  Return the index of the class that is predicted most commonly by leaf nodes in dt.
- void **DecisionTreeSetGrowingData** (**DecisionTreePtr** dt, void ∗data)
  Each decision tree node has a pointer reserved for your use.
- void ∗ **DecisionTreeGetGrowingData** (**DecisionTreePtr** dt)
  Each decision tree node has a pointer reserved for your use.
- void **DecisionTreePrint** (**DecisionTreePtr** dt, FILE ∗out)
  Prints the decision tree to the passed file.
- void **DecisionTreePrintStats** (**DecisionTreePtr** dt, FILE ∗out)
  Prints counts of leaves at each level of the tree.
- **DecisionTreePtr DecisionTreeReadC45** (FILE ∗in, **ExampleSpecPtr** spec)
  Attempts to read a decision tree from the passed file.
- **DecisionTreePtr DecisionTreeRead** (FILE ∗in, **ExampleSpecPtr** spec)
  Attempts to read a decision tree from the passed file.
- void **DecisionTreeWrite** (**DecisionTreePtr** dt, FILE ∗out)
  Writes the decision tree to the passed file.

8.20.2 Typedef Documentation

8.20.2.1 typedef struct _DecisionTree_ DecisionTree

ADT for working with decision trees.

See **DecisionTree.h** for more detail.

8.20.2.2 typedef struct _DecisionTree_ ∗ DecisionTreePtr

ADT for working with decision trees.

See **DecisionTree.h** for more detail.

8.20.3 Function Documentation

8.20.3.1 void DecisionTreeAddToClassDistribution (DecisionTreePtr *dt*, ExamplePtr *e*)

Put the example in the class distribution at the node.

Updates the class distribution at the node with the class of the example. This does not make recursive calls, so you should use DecisionTreeOneStepClassify and add it everywhere until you get to a leaf (if that is what you intend).

8.20.3.2 int DecisionTreeClassify (DecisionTreePtr *dt*, ExamplePtr *e*)

Uses dt to classify e and returns the index of the predicted class.

8.20.3.3 DecisionTreePtr DecisionTreeClone (DecisionTreePtr *dt*)

Recursively creates a copy of dt and returns it.

This function copies the user data pointers, but doesn't copy the data they point to.

8.20.3.4 int DecisionTreeCountNodes (DecisionTreePtr *dt*)

Returns a count of the number of nodes (of any type) in dt.

8.20.3.5 void DecisionTreeFree (DecisionTreePtr *dt*)

Frees the memory associated with the decision tree and all of its children.

This function doesn't do anything with user growing data you may have attached using DecisionTreeSetGrowingData; you must deal with that before calling this function.

8.20.3.6 void DecisionTreeGatherGrowingNodes (DecisionTreePtr *dt*, VoidAListPtr *list*)

Searches dt and appends all its growing nodes to the passed list.

8.20.3.7 void DecisionTreeGatherLeaves (DecisionTreePtr *dt*, VoidAListPtr *list*)

Searches dt and appends all its leaf nodes to the passed list.

8.20.3.8 DecisionTreePtr DecisionTreeGetChild (DecisionTreePtr *dt*, int *index*)

Returns one of the direct decendants of dt.

Index should be between 0 and DecisionTreeGetChildCount(dt) - 1. For nodes that split on continuous attributes use index 0 for the left child (<) and index 1 for the right child (>=).

8.20.3.9 int DecisionTreeGetChildCount (DecisionTreePtr *dt*)

Returns a count of the direct decendants of dt.

That is, return a count of all the nodes that you can reach from dt by taking one step towards the leaves.

8.20.3.10 int DecisionTreeGetClass (DecisionTreePtr *dt*)

Returns the index of the class which dt predicts.

This makes the most sense if dt is a Leaf node, but may be useful at other times as well.

8.20.3.11 float DecisionTreeGetClassDistributionSampleCount (DecisionTreePtr *dt*)

Returns the number of samples added to the node's distribution.

8.20.3.12 float DecisionTreeGetClassProb (DecisionTreePtr *dt*, int *theClass*)

Returns the probability of the class.

Returns what portion of the examples that were added to the class distribution at this node have the associated class.

8.20.3.13 void∗ DecisionTreeGetGrowingData (DecisionTreePtr *dt*)

Each decision tree node has a pointer reserved for your use.

Use the GetGrowingData function to access the value of the pointer.

8.20.3.14 int DecisionTreeGetMostCommonClass (DecisionTreePtr *dt*)

Return the index of the class that is predicted most commonly by leaf nodes in dt.

8.20.3.15 int DecisionTreeIsLeaf (DecisionTreePtr *dt*)

Returns 1 if dt is a leaf node and 0 otherwise.

8.20.3.16 int DecisionTreeIsNodeGrowing (DecisionTreePtr *dt*)

Returns 1 if dt is a growing node and 0 otherwise.

8.20.3.17 int DecisionTreeIsTreeGrowing (DecisionTreePtr *dt*)

Returns 1 if dt is a growing node has any children which are, and 0 otherwise.

8.20.3.18 DecisionTreePtr DecisionTreeNew (ExampleSpecPtr *spec*)

Creates a new decision tree node.

You should use the accessor methods to initialize it and attach it to an existing DecisionTree as needed.

8.20.3.19 DecisionTreePtr DecisionTreeOneStepClassify (DecisionTreePtr *dt*, ExamplePtr *e*)

Does one step of classifing e with dt.

Returns the direct decendant of dt corresponding to the correct value of dt's test attribute. If dt is a leaf or growing node this function will return dt.

8.20.3.20 void DecisionTreePrint (DecisionTreePtr *dt*, FILE ∗ *out*)

Prints the decision tree to the passed file.

FILE ∗ should be opened for writing. The decision tree will be written so as to be understandable by humans. Your mileage may vary.

Note that you could pass STDOUT to the function to write a decision tree to the console.

8.20.3.21 void DecisionTreePrintStats (DecisionTreePtr *dt*, FILE ∗ *out*)

Prints counts of leaves at each level of the tree.

The passed FILE ∗ should be opened for writing. Note that you could pass STDOUT to the function to write the stats to the console.

8.20.3.22 DecisionTreePtr DecisionTreeRead (FILE ∗ *in*, ExampleSpecPtr *spec*)

Attempts to read a decision tree from the passed file.

FILE ∗ should be opened for reading. Attaches the ExampleSpec to the read decision tree.

This function allocates memory which should be freed by calling DecisionTreeFree.

8.20.3.23 DecisionTreePtr DecisionTreeReadC45 (FILE ∗ *in*, ExampleSpecPtr *spec*)

Attempts to read a decision tree from the passed file.

FILE ∗ should be opened for reading. The file, in, should contain a decision tree written in C4.5's binary format, not the pretty-printed text format. A run of C4.5 with its default arguments will produce 2 such files, stem.tree and stem.unpruned.

This function handles leaves, continuous splits, and discrete splits and will not be able to read trees built with C4.5's subsetting options.

8.20.3.24 void DecisionTreeSetClass (DecisionTreePtr *dt*, int *theClass*)

Sets dt's class prediction to theClass.

Does not change dt's type to leaf node. This might be useful for anytime algorithms where a growing node needs to contain a reasonable prediction at all times.

8.20.3.25 void DecisionTreeSetClassProb (DecisionTreePtr *dt*, int *theClass*, float *prob*)

Sets the probability of the class.

Changes the probability of the class without changing the sample count (unless the sample count was zero in which case it is set to 1).

8.20.3.26 void DecisionTreeSetGrowingData (DecisionTreePtr *dt*, void ∗ *data*)

Each decision tree node has a pointer reserved for your use.

Use the SetGrowingData function to change the value of the pointer. You can set the pointer to anything you like (for example, to store sufficient statistics on growing nodes), but remember that you are responsible to manage any memory that it points to.

8.20.3.27 void DecisionTreeSetTypeGrowing (DecisionTreePtr *dt*)

Changes dt into a growing node.

8.20.3.28 void DecisionTreeSetTypeLeaf (DecisionTreePtr *dt*)

Changes dt into a leaf node without changing its class prediction.

If dt is not a growing node this function also frees all of dt's children. Remember that you are responsible for anything stored in any of dt's children's growing pointers and you should clean up these pointers before calling this function.

8.20.3.29 void DecisionTreeSplitOnContinuousAttribute (DecisionTreePtr *dt*, int *attNum*, float *threshold*)

Changes dt into a continuous split.

The new node splits on a threshold on a continuous attribute and adds children to dt for values of attNum < and >= the threshold. The created children start as growing nodes.

8.20.3.30 void DecisionTreeSplitOnDiscreteAttribute (DecisionTreePtr *dt*, int *attNum*)

Changes dt into a discrete split.

The new node splits on the values of a discrete attribute and adds one child to dt for each value of attribute attNum. The created children start as growing nodes.

8.20.3.31 void DecisionTreeWrite (DecisionTreePtr *dt*, FILE ∗ *out*)

Writes the decision tree to the passed file.

FILE ∗ should be opened for writing. The decision tree will be written in a binary format suitable to be read by DecisionTreeRead, but this function ignores any growing data that you've associated with dt – if you need to save growing data you will need to serialize it some other way.

Note that you could pass STDOUT to the function to write an example to the console.

8.20.3.32 void DecisionTreeZeroClassDistribution (DecisionTreePtr *dt*)

Sets the nodes distribution to zeros.

8.21 doxygen.h File Reference

8.21.1 Detailed Description

Used to hold doxygen documentation. Ignore this file.

8.22 Example.h File Reference

8.22.1 Detailed Description

ADT for training (and testing, etc.) data.

This is the interface for working with instances of the Example ADT. Note that all access to attributes is 0 based (just like C arrays).

Note that all the Examples created with an ExampleSpec maintain a pointer to the ExampleSpec, so you shouldn't free it or modify the ExampleSpec until you are done with all the Examples referencing it.

Data Structures

- struct **_Example_**

  ADT for working with examples.

Defines

- #define **ExampleIsAttributeUnknown**(e, attNum) (VALIndex(e → attributes, attNum) == 0)

  Returns 1 if the specified attribute is marked as unknown and 0 otherwise.

- #define **ExampleGetNumAttributes**(e) (VALLength(((**ExamplePtr**)e) → attributes))

  Returns the number of attributes that this example has.

- #define **ExampleGetClass**(e) (((**ExamplePtr**)e) → myclass)

  Returns the value of the example's class.

Typedefs

- typedef **_Example_** **Example**

  ADT for working with examples.

- typedef **_Example_** ∗ **ExamplePtr**

  ADT for working with examples.

Functions

- **ExamplePtr ExampleNew** (**ExampleSpecPtr** es)

  Programmatically creates a new example.

- void **ExampleFree** (**ExamplePtr** e)

  Frees all the memory being used by the passed example.

- **ExamplePtr ExampleClone** (**ExamplePtr** e)

  Allocates memory and copies the example into it.

- void **ExampleSetAttributeUnknown** (**ExamplePtr** e, int attNum)

  Marks the specified attribute value as unknown.

- void **ExampleSetDiscreteAttributeValue** (**ExamplePtr** e, int attNum, int value)

  Considers the specified attribute to be discrete and sets its value to the specified value.

- void **ExampleSetContinuousAttributeValue** (**ExamplePtr** e, int attNum, double value)

  Considers the specified attribute to be continuous and sets its value to the specified value.

- void **ExampleSetClass** (**ExamplePtr** e, int theClass)

  Sets the example's class to the specified class.

- void **ExampleSetClassUnknown** (**ExamplePtr** e)

  Marks the example's class as unknown.

- void **ExampleAddNoise** (**ExamplePtr** e, float p, int doClass, int attrib)

  Randomly corrupts the attributes and class of the example.

- **ExamplePtr ExampleRead** (FILE ∗file, **ExampleSpecPtr** es)

  Attempts to read an example from the passed file pointer.

- **VoidListPtr ExamplesRead** (FILE ∗file, **ExampleSpecPtr** es)

  Reads as many examples as possible from the file pointer.

- int **ExampleIsAttributeDiscrete** (**ExamplePtr** e, int attNum)

  Returns 1 if the specified attribute is discrete and 0 otherwise.

- int **ExampleIsAttributeContinuous** (**ExamplePtr** e, int attNum)

  Returns 1 if the specified attribute is continuous and 0 otherwise.

- int **ExampleGetDiscreteAttributeValue** (**ExamplePtr** e, int attNum)

  Returns the value of the indicated discrete attribute.

- double **ExampleGetContinuousAttributeValue** (**ExamplePtr** e, int attNum)

  Returns the value of the indicated continuous attribute.

- int **ExampleIsClassUnknown** (**ExamplePtr** e)

  Returns 1 if the value of the example's class is known, and 0 otherwise.

- float **ExampleDistance** (**ExamplePtr** e, **ExamplePtr** dst)

  Returns the euclidian distance between the two examples.

- void **ExampleWrite** (**ExamplePtr** e, FILE ∗out)

  Writes the example to the passed file point.er.

8.22.2 Define Documentation

8.22.2.1 #define ExampleGetClass(e) (((ExamplePtr)e) → myclass)

Returns the value of the example's class.

If the value of the example's class is known this returns the value, otherwise this returns -1.

8.22.2.2 #define ExampleGetNumAttributes(e) (VALLength(((ExamplePtr)e) → attributes))

Returns the number of attributes that this example has.

This will be equal to the number of attributes that were in the ExampleSpec used to construct the example.

8.22.2.3 #define ExampleIsAttributeUnknown(e, attNum) (VALIndex(e → attributes, attNum) == 0)

Returns 1 if the specified attribute is marked as unknown and 0 otherwise.

Be sure not to ask for an attNum >= **ExampleGetNumAttributes(e)**.

8.22.3 Typedef Documentation

8.22.3.1 typedef struct _Example_ Example

ADT for working with examples.

See **Example.h** for more detail.

8.22.3.2 typedef struct _Example_ * ExamplePtr

ADT for working with examples.

See **Example.h** for more detail.

8.22.4 Function Documentation

8.22.4.1 void ExampleAddNoise (ExamplePtr *e*, float *p*, int *doClass*, int *attrib*)

Randomly corrupts the attributes and class of the example.

p should be a number between 0-1, which is interpreted as a probability (e.g. a value of .732 would be interpreted as 73.2%). class and attrib are flags which should be 1 if you want noise added to that part of the example and 0 otherwise. Then, for each discrete thing selected by the flags, this function will have the specified probability of changing it, without replacement, to a randomly selected value. This function changes the value of each continuous attribute by adding to it a value drawn from a normal distribution with mean 0 and with standard deviation p.

8.22.4.2 ExamplePtr ExampleClone (ExamplePtr *e*)

Allocates memory and copies the example into it.

8.22.4.3 float ExampleDistance (ExamplePtr *e*, ExamplePtr *dst*)

Returns the euclidian distance between the two examples.

This ignores discrete attributes.

8.22.4.4 void ExampleFree (ExamplePtr *e*)

Frees all the memory being used by the passed example.

8.22.4.5 double ExampleGetContinuousAttributeValue (ExamplePtr *e*, int *attNum*)

Returns the value of the indicated continuous attribute.

If the attNum is valid, and ExampleGetAttributeUnknown(e, attNum) returns 0, and Example-IsAttributeContinuous(e, attNum) returns 1, this function will return the value of the attribute. If the conditions aren't met, there is a good chance that calling this will crash your program.

8.22.4.6 int ExampleGetDiscreteAttributeValue (ExamplePtr *e*, int *attNum*)

Returns the value of the indicated discrete attribute.

If the attNum is valid, and ExampleGetAttributeUnknown(e, attNum) returns 0, and Example-IsAttributeDiscrete(e, attNum) returns 1, this function will return the value of the attribute. If the conditions aren't met, there is a good chance that calling this will crash your program.

8.22.4.7 int ExampleIsAttributeContinuous (ExamplePtr *e*, int *attNum*)

Returns 1 if the specified attribute is continuous and 0 otherwise.

Be sure not to ask for an attNum >= **ExampleGetNumAttributes(e)**.

8.22.4.8 int ExampleIsAttributeDiscrete (ExamplePtr *e*, int *attNum*)

Returns 1 if the specified attribute is discrete and 0 otherwise.

Be sure not to ask for an attNum >= **ExampleGetNumAttributes(e)**.

8.22.4.9 int ExampleIsClassUnknown (ExamplePtr *e*)

Returns 1 if the value of the example's class is known, and 0 otherwise.

8.22.4.10 ExamplePtr ExampleNew (ExampleSpecPtr *es*)

Programmatically creates a new example.

Allocates enough memory to hold all the attributes mentioned in the ExampleSpec passed. Use the ExampleSetFoo functions (see below) to set the values of the attributes and class.

This function allocates memory which should be freed by calling ExampleFree.

8.22.4.11 ExamplePtr ExampleRead (FILE ∗ *file*, ExampleSpecPtr *es*)

Attempts to read an example from the passed file pointer.

FILE ∗ should be opened for reading. The file should contain examples in C4.5 format. Uses the ExampleSpec to determine how many, what types, and how to interpret the attributes it needs to read.

This function will return 0 (NULL) if it is unable to read an example from the file (bad input or EOF). If the input is badly formed, the function will also output an error to the console.

Note that you could pass STDIN to the function to read an example from the console.

This function allocates memory which should be freed by calling ExampleFree.

8.22.4.12 void ExampleSetAttributeUnknown (ExamplePtr *e*, int *attNum*)

Marks the specified attribute value as unknown.

Future calls to ExampleGetAttributeUnknown for that attribute will return 1.

8.22.4.13 void ExampleSetClass (ExamplePtr *e*, int *theClass*)

Sets the example's class to the specified class.

The function doesn't do much error checking so be sure that you call it consistent with ExampleSpecGetNumClasses. If you don't, there is a chance the example could cause your program to crash.

8.22.4.14 void ExampleSetClassUnknown (ExamplePtr *e*)

Marks the example's class as unknown.

8.22.4.15 void ExampleSetContinuousAttributeValue (ExamplePtr *e*, int *attNum*, double *value*)

Considers the specified attribute to be continuous and sets its value to the specified value.

This function doesn't do much error checking so be sure that you call it consistent with ExampleIsAttributeDiscrete, and ExampleIsAttributeContinuous. If you don't, there is a chance the example could cause your program to crash.

8.22.4.16 void ExampleSetDiscreteAttributeValue (ExamplePtr *e*, int *attNum*, int *value*)

Considers the specified attribute to be discrete and sets its value to the specified value.

This function doesn't do much error checking so be sure that you call it consistent with ExampleIsAttributeDiscrete, ExampleIsAttributeContinuous and ExampleSpecGetAttributeValueCount. If you don't, there is a chance the example could cause your program to crash.

8.22.4.17 VoidListPtr ExamplesRead (FILE ∗ *file*, ExampleSpecPtr *es*)

Reads as many examples as possible from the file pointer.

Calls ExampleRead until it gets a 0, allocates a list and adds each example to it. You are responsible for freeing the examples and the list.

8.22.4.18 void ExampleWrite (ExamplePtr *e*, FILE ∗ *out*)

Writes the example to the passed file point.er.

FILE ∗ should be opened for writing. The example will be written in C4.5 format, and could later be read in using ExampleRead.

Note that you could pass stdout to the function to write an example to the console.

8.23 ExampleGenerator.h File Reference

8.23.1 Detailed Description

Generate a random (but reproducible) data set.

Randomly, but reproducably, create a series of examples. These examples could then be classified with some known model and used as a synthetic dataset to test a learner. This uses the **RandomSetState()** functions so that it will produce the same series of examples for the same seed no matter what the rest of your program does with the random number generators.

Data Structures

- struct **_ExampleGenerator_**

  Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** *for more detail.*

Typedefs

- typedef **_ExampleGenerator_ ExampleGenerator**

  Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** *for more detail.*

- typedef **_ExampleGenerator_** ∗ **ExampleGeneratorPtr**

  Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** *for more detail.*

Functions

- **ExampleGeneratorPtr ExampleGeneratorNew** (**ExampleSpecPtr** es, int seed)

  Creates a new example generator.

- void **ExampleGeneratorFree** (**ExampleGeneratorPtr** eg)

  Frees the memory associated with the example generator.

- **ExamplePtr ExampleGeneratorGenerate** (**ExampleGeneratorPtr** eg)

  Makes a random example.

8.23.2 Typedef Documentation

8.23.2.1 typedef struct _ExampleGenerator_ ExampleGenerator

Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** for more detail.

8.23.2.2 typedef struct _ExampleGenerator_ * ExampleGeneratorPtr

Holds the information needed to reproducibly make a random data set. See **ExampleGenerator.h** for more detail.

8.23.3 Function Documentation

8.23.3.1 void ExampleGeneratorFree (ExampleGeneratorPtr *eg*)

Frees the memory associated with the example generator.

8.23.3.2 ExamplePtr ExampleGeneratorGenerate (ExampleGeneratorPtr *eg*)

Makes a random example.

Allocates an example, randomly sets the values of its attributes, and returns it. Uses uniform distributions for all of its decisions. For continuous attributes it uniformly generates a value between 0 and 1.0; you might like to scale this value to fit your needs.

You are must free the ExamplePtr when you are finished with it by calling ExampleFree.

8.23.3.3 ExampleGeneratorPtr ExampleGeneratorNew (ExampleSpecPtr *es*, int *seed*)

Creates a new example generator.

The generator will generate examples conforming to es using a seeded random number generator. If the value of seed is -1 this will select a random initial seed (But you'll need to initialize the random number generator on your system for this to work; The function **RandomInit()** will do the job).

8.24 ExampleGroupStats.h File Reference

8.24.1 Detailed Description

Sufficient statistics for Entropy and Gini.

Tracks and maintains the sufficient statistics needed to calculate Entropy and Gini of discrete and continuous attributes, as well as make some queries about the probability of events in the data.

Data Structures

- struct **_ExampleGroupStats_**

  Sufficient statistics for Entropy and Gini.

Typedefs

- typedef **_ExampleGroupStats_ ExampleGroupStats**

  Sufficient statistics for Entropy and Gini.

- typedef **_ExampleGroupStats_** ∗ **ExampleGroupStatsPtr**

  Sufficient statistics for Entropy and Gini.

Functions

- **ExampleGroupStatsPtr ExampleGroupStatsNew** (**ExampleSpecPtr** es, AttributeTrackerPtr at)

  Creates a structure to track sufficient statistics.

- void **ExampleGroupStatsFree** (**ExampleGroupStatsPtr** egs)

  Frees all the memory that was being used by the structure.

- void **ExampleGroupStatsDeactivate** (**ExampleGroupStatsPtr** egs)

  Temporarily frees the memory being used to hold statistics.

- void **ExampleGroupStatsReactivate** (**ExampleGroupStatsPtr** egs)

  Reallocates the memory that is freed by a call to ExampleGroupStatsDeactivate.

- void **ExampleGroupStatsAddExample** (**ExampleGroupStatsPtr** egs, **ExamplePtr** e)

  Adds the information from the example to the statistics structure.

- void **ExampleGroupStatsWrite** (**ExampleGroupStatsPtr** egs, FILE ∗out)

  A debugging function that prints a representation of the stats structure to specified file.

- long **ExampleGroupStatsNumExamplesSeen** (**ExampleGroupStatsPtr** egs)

  Number of examples being tracked by the structure.

- AttributeTrackerPtr **ExampleGroupStatsGetAttributeTracker** (**ExampleGroupStatsPtr** egs)

  Returns the attribute tracker associated with the structure.

- int **ExampleGroupStatsIsAttributeActive** (**ExampleGroupStatsPtr** egs, int num)

  Tests if the attribute is active.

- void **ExampleGroupStatsIgnoreAttribute** (**ExampleGroupStatsPtr** egs, int num)

  Frees the memory being used by the attribute and stops tracking it.

- int **ExampleGroupStatsGetMostCommonClassLaplace** (**ExampleGroupStatsPtr** egs, int addClass, int addCount)

  Returns the index of the most common class, but adds addCount samples to addClass.

- int **ExampleGroupStatsGetMostCommonClass** (**ExampleGroupStatsPtr** egs)

  Returns the index of the most common class.

- long **ExampleGroupStatsGetMostCommonClassCount** (**ExampleGroupStatsPtr** egs)

  Returns the number of examples with the most common class that were seen by the structure.

- int **ExampleGroupStatsGetMostCommonClassForAttVal** (**ExampleGroupStatsPtr** egs, int att, int val)

  Returns the most common class among examples where the specified attribute has the specified value.

- int **ExampleGroupStatsIsPure** (**ExampleGroupStatsPtr** egs)

  Returns 1 if all the examples shown to the structure have the same class.

- float **ExampleGroupStatsGetValuePercent** (**ExampleGroupStatsPtr** egs, int attNum, int valNum)

  Returns the fraction of examples that have the specified value for the specified attribute.

- double **ExampleGroupStatsGetValueGivenClassMEstimate** (**ExampleGroupStatsPtr** egs, int attNum, int valNum, int classNum)

  Returns P(att = value | class).

- float **ExampleGroupStatsGetClassPercent** (**ExampleGroupStatsPtr** egs, int classNum)

  Returns P(class).

- float **ExampleGroupStatsGetPercentBelowThreshold** (**ExampleGroupStatsPtr** egs, int attNum, float thresh)

  Returns the fraction of examples with a value below the specified threshold.

- double **ExampleGroupStatsGetValueGivenClassMEstimateLogP** (**ExampleGroupStatsPtr** egs, int attNum, int valNum, int classNum)

  Returns a smoothed P(att = value | class).

- double **ExampleGroupStatsGetClassLogP** (**ExampleGroupStatsPtr** egs, int classNum)

Returns the log of the fraction of examples that have the specified class.

- float **ExampleGroupStatsEntropyTotal** (**ExampleGroupStatsPtr** egs)

  Returns the entropy of the class attribute of all examples seen so far.

- float **ExampleGroupStatsEntropyDiscreteAttributeSplit** (**ExampleGroupStatsPtr** egs, int attNum)

  Returns the weighted entropy of the class attribute after partitoning the data by the values of the specified attribute.

- void **ExampleGroupStatsEntropyContinuousAttributeSplit** (**ExampleGroupStatsPtr** egs, int attNum, float ∗firstIndex, float ∗firstThresh, float ∗secondIndex, float ∗secondThresh)

  Finds the entropy of the best split thresholds.

- float **ExampleGroupStatsGiniTotal** (**ExampleGroupStatsPtr** egs)

  Returns the gini index of the class attribute of all examples seen so far.

- float **ExampleGroupStatsGiniDiscreteAttributeSplit** (**ExampleGroupStatsPtr** egs, int attNum)

  Returns the weighted gini of the class attribute after partitoning the data by the values of the specified attribute.

- void **ExampleGroupStatsGiniContinuousAttributeSplit** (**ExampleGroupStatsPtr** egs, int attNum, float ∗firstIndex, float ∗firstThresh, float ∗secondIndex, float ∗secondThresh)

  Finds the Gini index of the best split thresholds.

- void **ExampleGroupStatsIgnoreSplitsWorseThanEntropy** (**ExampleGroupStatsPtr** egs, int attNum, float entropyThresh)

  Stop monitoring some thresholds.

- void **ExampleGroupStatsIgnoreSplitsWorseThanGini** (**ExampleGroupStatsPtr** egs, int attNum, float giniThresh)

  Stop monitoring some thresholds.

- int **ExampleGroupStatsLimitSplitsEntropy** (**ExampleGroupStatsPtr** egs, int attNum, int maxSplits, int pruneDownTo)

  Reduce the number of thresholds being considered if above the max.

- void **ExampleGroupStatsStopAddingSplits** (**ExampleGroupStatsPtr** egs, int attNum)

  Stop adding new split thresholds, but continue to use future examples to evaluate the existing ones.

- int **ExampleGroupStatsNumSplitThresholds** (**ExampleGroupStatsPtr** egs, int attNum)

  Returns the number of thresholds that are being monitored for the specified attribute.

- int **ExampleGroupStatsGetMostCommonClassAboveThreshold** (**ExampleGroupStatsPtr** egs, int attNum, float threshold)

Returns the most common class above the specified value.

- int **ExampleGroupStatsGetMostCommonClassBelowThreshold** (**ExampleGroupStatsPtr** egs, int attNum, float threshold)

  Returns the most common class below the specified value.

8.24.2 Typedef Documentation

8.24.2.1 typedef struct _ExampleGroupStats_ ExampleGroupStats

Sufficient statistics for Entropy and Gini.

8.24.2.2 typedef struct _ExampleGroupStats_ ∗ ExampleGroupStatsPtr

Sufficient statistics for Entropy and Gini.

8.24.3 Function Documentation

8.24.3.1 void ExampleGroupStatsAddExample (ExampleGroupStatsPtr *egs*, ExamplePtr *e*)

Adds the information from the example to the statistics structure.

8.24.3.2 void ExampleGroupStatsDeactivate (ExampleGroupStatsPtr *egs*)

Temporarily frees the memory being used to hold statistics.

Does not free the whole structure. A later call to ExampleGroupStatsReactivate will restore the memory (but not the counts that used to be there). This is a convienient way to focus RAM usage (and learning) in one part of the instance space while keeping the book keeping around to quickly resume learning in another. You shouldn't try to add examples to a deactiveated structure.

8.24.3.3 void ExampleGroupStatsEntropyContinuousAttributeSplit (ExampleGroupStatsPtr *egs*, int *attNum*, float ∗ *firstIndex*, float ∗ *firstThresh*, float ∗ *secondIndex*, float ∗ *secondThresh*)

Finds the entropy of the best split thresholds.

Calculates the entropy of splitting the specified attribute by every threshold under consideration (values are sorted and then a threshold is considered between each pair of adjacent values that have different class). The remaining arguments return the entropy of the best and second best thresholds, along with the thresholds themselves.

This function adds an MDL penelty similar to the one Quinlan uses in C4.5.

Should only be called for continuous attributes.

8.24.3.4 float ExampleGroupStatsEntropyDiscreteAttributeSplit (ExampleGroupStatsPtr *egs*, int *attNum*)

Returns the weighted entropy of the class attribute after partitoning the data by the values of the specified attribute.

Should only be called for discrete attributes.

8.24.3.5 float ExampleGroupStatsEntropyTotal (ExampleGroupStatsPtr *egs*)

Returns the entropy of the class attribute of all examples seen so far.

8.24.3.6 void ExampleGroupStatsFree (ExampleGroupStatsPtr *egs*)

Frees all the memory that was being used by the structure.

8.24.3.7 AttributeTrackerPtr ExampleGroupStatsGetAttributeTracker (ExampleGroupStatsPtr *egs*)

Returns the attribute tracker associated with the structure.

8.24.3.8 double ExampleGroupStatsGetClassLogP (ExampleGroupStatsPtr *egs*, int *classNum*)

Returns the log of the fraction of examples that have the specified class.

8.24.3.9 float ExampleGroupStatsGetClassPercent (ExampleGroupStatsPtr *egs*, int *classNum*)

Returns P(class).

8.24.3.10 int ExampleGroupStatsGetMostCommonClass (ExampleGroupStatsPtr *egs*)

Returns the index of the most common class.

8.24.3.11 int ExampleGroupStatsGetMostCommonClassAboveThreshold (ExampleGroupStatsPtr *egs*, int *attNum*, float *threshold*)

Returns the most common class above the specified value.

Should only be called for continuous attributes.

8.24.3.12 int ExampleGroupStatsGetMostCommonClassBelowThreshold (ExampleGroupStatsPtr *egs*, int *attNum*, float *threshold*)

Returns the most common class below the specified value.

Should only be called for continuous attributes.

8.24.3.13 long ExampleGroupStatsGetMostCommonClassCount (ExampleGroupStatsPtr *egs*)

Returns the number of examples with the most common class that were seen by the structure.

8.24.3.14 int ExampleGroupStatsGetMostCommonClassForAttVal (ExampleGroupStatsPtr *egs*, int *att*, int *val*)

Returns the most common class among examples where the specified attribute has the specified value.

Should only be called for discrete attributes.

8.24.3.15 int ExampleGroupStatsGetMostCommonClassLaplace (ExampleGroupStatsPtr *egs*, int *addClass*, int *addCount*)

Returns the index of the most common class, but adds addCount samples to addClass.

Use addClass of -1 for no addition (or just call ExampleGroupStatsGetMostCommonClass). This adding allows you to, for example, smooth the class towards the parent class during decision tree induction.

8.24.3.16 float ExampleGroupStatsGetPercentBelowThreshold (ExampleGroupStatsPtr *egs*, int *attNum*, float *thresh*)

Returns the fraction of examples with a value below the specified threshold.

Should only be called for continuous attributes.

8.24.3.17 double ExampleGroupStatsGetValueGivenClassMEstimate (ExampleGroupStatsPtr *egs*, int *attNum*, int *valNum*, int *classNum*)

Returns P(att = value | class).

Returns the fraction of examples among those that have the specified class that have the specified value for the specified attribute, but smooths the return value by adding a small amount (that decreases with the number of samples seen) to each class count first.

Should only be called for discrete attributes.

8.24.3.18 double ExampleGroupStatsGetValueGivenClassMEstimateLogP (ExampleGroupStatsPtr *egs*, int *attNum*, int *valNum*, int *classNum*)

Returns a smoothed P(att = value | class).

Returns the log of the fraction of examples among those that have the specified class that have the specified value for the specified attribute, but smooths the return value by adding a small amount (that decreases with the number of samples seen) to each class count first.

Should only be called for discrete attributes.

8.24.3.19 float ExampleGroupStatsGetValuePercent (ExampleGroupStatsPtr *egs*, int *attNum*, int *valNum*)

Returns the fraction of examples that have the specified value for the specified attribute.

Should only be called for discrete attributes.

8.24.3.20 void ExampleGroupStatsGiniContinuousAttributeSplit (ExampleGroupStatsPtr *egs*, int *attNum*, float ∗ *firstIndex*, float ∗ *firstThresh*, float ∗ *secondIndex*, float ∗ *secondThresh*)

Finds the Gini index of the best split thresholds.

Calculates the gini of splitting the specified attribute by every threshold under consideration (values are sorted and then a threshold is considered between each pair of adjacent values that have different class). The remaining arguments return the gini of the best and second best thresholds, along with the thresholds themselves.

Should only be called for continuous attributes.

8.24.3.21 float ExampleGroupStatsGiniDiscreteAttributeSplit (ExampleGroupStatsPtr *egs*, int *attNum*)

Returns the weighted gini of the class attribute after partitoning the data by the values of the specified attribute.

Should only be called for discrete attributes.

8.24.3.22 float ExampleGroupStatsGiniTotal (ExampleGroupStatsPtr *egs*)

Returns the gini index of the class attribute of all examples seen so far.

8.24.3.23 void ExampleGroupStatsIgnoreAttribute (ExampleGroupStatsPtr *egs*, int *num*)

Frees the memory being used by the attribute and stops tracking it.

This is useful if you decide that some attribute will not be used (perhaps using some statistical tests) and would like to use the memory elsewhere.

8.24.3.24 void ExampleGroupStatsIgnoreSplitsWorseThanEntropy (ExampleGroupStatsPtr *egs*, int *attNum*, float *entropyThresh*)

Stop monitoring some thresholds.

Stop monitoring every threshold with an entropy worse than the specified value. This frees some memory, but adding future values to the egs may require some interpolation to estimate the position of the new value in the array of all values for the attribute (and so this introduces some error into future calls for the Entropy or Gini of the attribute).

Should only be called for continuous attributes.

8.24.3.25 void ExampleGroupStatsIgnoreSplitsWorseThanGini (ExampleGroupStatsPtr *egs*, int *attNum*, float *giniThresh*)

Stop monitoring some thresholds.

Stop monitoring every threshold with an Gini worse than the specified value. This frees some memory, but adding future values to the egs may require some interpolation to estimate the position of the new value in the array of all values for the attribute (and so this introduces some error into future calls for the Entropy or Gini of the attribute).

Should only be called for continuous attributes.

8.24.3.26 int ExampleGroupStatsIsAttributeActive (ExampleGroupStatsPtr *egs*, int *num*)

Tests if the attribute is active.

Returns 1 if the attribute was active in the initial attribute tracker and has not been ignored by a call to ExampleGroupStatsIgnoreAttribute since then.

8.24.3.27 int ExampleGroupStatsIsPure (ExampleGroupStatsPtr *egs*)

Returns 1 if all the examples shown to the structure have the same class.

8.24.3.28 int ExampleGroupStatsLimitSplitsEntropy (ExampleGroupStatsPtr *egs*, int *attNum*, int *maxSplits*, int *pruneDownTo*)

Reduce the number of thresholds being considered if above the max.

If the attribute is monitoring more than 'maxSplits' split thresholds this function will find the best 'pruneDownTo' based on entropy and start ignoring all the rest. This frees some memory, but adding future values to the egs may require some interpolation to estimate the position of the new value in the array of all values for the attribute (and so this introduces some error into future calls for the Entropy or Gini of the attribute).

Returns the number of thresholds that were pruned.

Should only be called for continuous attributes.

8.24.3.29 ExampleGroupStatsPtr ExampleGroupStatsNew (ExampleSpecPtr *es*, AttributeTrackerPtr *at*)

Creates a structure to track sufficient statistics.

Creates a structure to track the statistics needed to cacluate several common machine metrics for the attributes that are active in the AttributeTracker. This function takes over the memory for the AttributeTracker and will free it when ExampleGroupStatsFree For categorical attributes this uses memory proportional to the number of classes $*$ the number of values of the attribute. For continuous attributes this uses constant memory at first, but as examples are added with ExampleGroupStatsAddExample the memory grows proportionally with the number of unique values of the attribute.

8.24.3.30 long ExampleGroupStatsNumExamplesSeen (ExampleGroupStatsPtr *egs*)

Number of examples being tracked by the structure.

Outputs the number of examples added to the structure with ExampleGroupStatsAddExample since the last call to ExampleGroupStatsReactivate.

8.24.3.31 int ExampleGroupStatsNumSplitThresholds (ExampleGroupStatsPtr *egs*, int *attNum*)

Returns the number of thresholds that are being monitored for the specified attribute.

Should only be called for continuous attributes.

8.24.3.32 void ExampleGroupStatsReactivate (ExampleGroupStatsPtr *egs*)

Reallocates the memory that is freed by a call to ExampleGroupStatsDeactivate.

8.24.3.33 void ExampleGroupStatsStopAddingSplits (ExampleGroupStatsPtr *egs*, int *attNum*)

Stop adding new split thresholds, but continue to use future examples to evaluate the existing ones.

Should only be called for continuous attributes.

8.24.3.34 void ExampleGroupStatsWrite (ExampleGroupStatsPtr *egs*, FILE ∗ *out*)

A debugging function that prints a representation of the stats structure to specified file.

8.25 ExampleSpec.h File Reference

8.25.1 Detailed Description

Schema for training data.

This is based off of Ross Quinlan's `C4.5 format` and can read and write that format to disk.

Bug

Under Cygnus (and windows) lex doesn't seem to do the right thing with EOF rules, so you need to put an extra return at the end of your .names files.

Data Structures

- struct **_ExampleSpec_**

  Schema for training data.

Defines

- #define **ExampleSpecGetAttributeType**(es, num) (AttributeSpecGetType(((AttributeSpecPtr)VALIndex(es → attributes, num))))

  Returns the type of the specified attribute.

- #define **ExampleSpecGetAttributeValueCount**(es, attNum) (AttributeSpecGetNumValues(((AttributeSpecPtr)VALIndex(es → attributes, attNum))))

  Returns the number of values of the attribute.

- #define **ExampleSpecGetAttributeValueName**(es, attNum, valNum) ((char ∗)AttributeSpecGetValueName(((AttributeSpecPtr)VALIndex(es → attributes, attNum)), valNum))

  Return the name of the specified value of the specified attribute.

Typedefs

- typedef **_ExampleSpec_ ExampleSpec**

  Schema for training data.

- typedef **_ExampleSpec_** ∗ **ExampleSpecPtr**

  Schema for training data.

Functions

- **ExampleSpecPtr ExampleSpecNew** (void)

  Programmatically creates a new ExampleSpec.

- void **ExampleSpecFree** (**ExampleSpecPtr** es)

Frees all the memory being used by the ExampleSpec.

- void **ExampleSpecAddClass** (**ExampleSpecPtr** es, char ∗className)

  Adds a new class to the ExampleSpec and gives it the specified name.

- int **ExampleSpecAddDiscreteAttribute** (**ExampleSpecPtr** es, char ∗name)

  Adds a new discrete attribute to the ExampleSpec and gives it the specified name.

- int **ExampleSpecAddContinuousAttribute** (**ExampleSpecPtr** es, char ∗name)

  Adds a new continuous attribute to the ExampleSpec and gives it the specified name.

- void **ExampleSpecAddAttributeValue** (**ExampleSpecPtr** es, int attNum, char ∗name)

  Adds a new value to the specified attribute and gives it the specified name.

- **ExampleSpecPtr ExampleSpecRead** (char ∗fileName)

  Reads a .names formated file and build an ExampleSpec.

- int **ExampleSpecGetNumAttributes** (**ExampleSpecPtr** es)

  Returns the number of attributes that the example spec contains.

- int **ExampleSpecIsAttributeDiscrete** (**ExampleSpecPtr** es, int num)

  Returns 1 if the specified attribute is discrete and 0 otherwise.

- int **ExampleSpecIsAttributeContinuous** (**ExampleSpecPtr** es, int num)

  Returns 1 if the specified attribute is continuous and 0 otherwise.

- int **ExampleSpecIsAttributeIgnored** (**ExampleSpecPtr** es, int num)

  Returns 1 if the specified attribute should be ignored and 0 otherwise.

- char ∗ **ExampleSpecGetAttributeName** (**ExampleSpecPtr** es, int attNum)

  If the attNum is valid, this returns the name of the associated attribute.

- int **ExampleSpecLookupAttributeName** (**ExampleSpecPtr** es, char ∗valName)

  Returns the index of the named attribute.

- int **ExampleSpecLookupAttributeValueName** (**ExampleSpecPtr** es, int attNum, char ∗valName)

  Returns the index of the named value.

- int **ExampleSpecLookupClassName** (**ExampleSpecPtr** es, char ∗name)

  If name is a valid class name, this returns the index associated with the class.

- char ∗ **ExampleSpecGetClassValueName** (**ExampleSpecPtr** es, int classNum)

  If the classNum is valid, this returns the name of the associated class.

- int **ExampleSpecGetNumClasses** (**ExampleSpecPtr** es)

  Returns the number of classes in the ExampleSpec.

- void **ExampleSpecWrite** (**ExampleSpecPtr** es, FILE ∗out)

  Outputs the ExampleSpec in .names format.

8.25.2 Define Documentation

8.25.2.1 #define ExampleSpecGetAttributeType(es, num) (AttributeSpecGetType(((AttributeSpecPtr)VALIndex(es → attributes, num))))

Returns the type of the specified attribute.

There are currently four supported types:

- asIgnore
- asContinuous
- asDiscreteNamed
- asDiscreteNoName

The programmatic construction interface only supports asIgnore, asContinuous, and asDiscreteNamed, but the other is needed to fully support the C4.5 format.

8.25.2.2 #define ExampleSpecGetAttributeValueCount(es, attNum) (AttributeSpecGetNumValues(((AttributeSpecPtr)VALIndex(es → attributes, attNum))))

Returns the number of values of the attribute.

If the attNum is valid and ExampleSpecIsAttributeDiscrete(es, attNum) returns 1, this function will return the number of values that attribute has. Remember that these values will be 0 indexed.

8.25.2.3 #define ExampleSpecGetAttributeValueName(es, attNum, valNum) ((char ∗)AttributeSpecGetValueName(((AttributeSpecPtr)VALIndex(es → attributes, attNum)), valNum))

Return the name of the specified value of the specified attribute.

If attNum is valid, and ExampleSpecIsAttributeDiscrete(es, attNum) returns 1, and valNum is valid, this returns the name of the specified value of the specified attribute.

8.25.3 Typedef Documentation

8.25.3.1 typedef struct _ExampleSpec_ ExampleSpec

Schema for training data.

8.25.3.2 typedef struct _ExampleSpec_ ∗ ExampleSpecPtr

Schema for training data.

8.25.4 Function Documentation

8.25.4.1 void ExampleSpecAddAttributeValue (ExampleSpecPtr *es*, int *attNum*, char ∗ *name*)

Adds a new value to the specified attribute and gives it the specified name.

The specified attribute had better be a discrete attribute. The main use of the name is to read/write Examples and ExampleSpecs in human readable format.

Note that this function takes over the memory associated with the name argument and will free it later. This means that you shouldn't pass in static strings, or strings that were allocated on the stack.

8.25.4.2 void ExampleSpecAddClass (ExampleSpecPtr *es*, char ∗ *className*)

Adds a new class to the ExampleSpec and gives it the specified name.

The main use of the name is to read/write Examples and ExampleSpecs in human readable format. The new class is assigned a value which you can retrieve by calling: ExampleSpecLookupClassName(es, className).

Note that this function takes over the memory associated with the className argument and will free it later. This means that you shouldn't pass in static strings, or strings that were allocated on the stack.

8.25.4.3 int ExampleSpecAddContinuousAttribute (ExampleSpecPtr *es*, char ∗ *name*)

Adds a new continuous attribute to the ExampleSpec and gives it the specified name.

The main use of the name is to read/write ExampleSpecs in human readable format. The function returns the index of the new attribute.

Note that this function takes over the memory associated with the name argument and will free it later. This means that you shouldn't pass in static strings, or strings that were allocated on the stack.

8.25.4.4 int ExampleSpecAddDiscreteAttribute (ExampleSpecPtr *es*, char ∗ *name*)

Adds a new discrete attribute to the ExampleSpec and gives it the specified name.

The main use of the name is to read/write ExampleSpecs in human readable format. The function returns the index of the new attribute, you can use the index to add values to the attribute using ExampleSpecAddAttributeValue.

Note that this function takes over the memory associated with the name argument and will free it later. This means that you shouldn't pass in static strings, or strings that were allocated on the stack.

8.25.4.5 void ExampleSpecFree (ExampleSpecPtr *es*)

Frees all the memory being used by the ExampleSpec.

Note that all the Examples created with an ExampleSpec maintain a pointer to the ExampleSpec, so you shouldn't free it or modify the ExampleSpec until you are done with all the Examples referencing it.

8.25.4.6 char∗ ExampleSpecGetAttributeName (ExampleSpecPtr *es*, int *attNum*)

If the attNum is valid, this returns the name of the associated attribute.

8.25.4.7 char∗ ExampleSpecGetClassValueName (ExampleSpecPtr *es*, int *classNum*)

If the classNum is valid, this returns the name of the associated class.

8.25.4.8 int ExampleSpecGetNumAttributes (ExampleSpecPtr *es*)

Returns the number of attributes that the example spec contains.

Remember that the attributes are indexed in a 0-based fashion (like C arrays) so the actual valid index for the attributes will be from 0 to ExampleSpecGetNumAttributes(es) - 1.

8.25.4.9 int ExampleSpecGetNumClasses (ExampleSpecPtr *es*)

Returns the number of classes in the ExampleSpec.

Remember that the classes will have indexes 0 - ExampleSpecGetNumClasses(es) - 1.

8.25.4.10 int ExampleSpecIsAttributeContinuous (ExampleSpecPtr *es*, int *num*)

Returns 1 if the specified attribute is continuous and 0 otherwise.

Be sure not to ask for an attribute numbered >= ExampleSpecGetNumAttributes(es).

8.25.4.11 int ExampleSpecIsAttributeDiscrete (ExampleSpecPtr *es*, int *num*)

Returns 1 if the specified attribute is discrete and 0 otherwise.

Be sure not to ask for an attribute numbered >= ExampleSpecGetNumAttributes(es).

8.25.4.12 int ExampleSpecIsAttributeIgnored (ExampleSpecPtr *es*, int *num*)

Returns 1 if the specified attribute should be ignored and 0 otherwise.

Be sure not to ask for an attribute numbered >= ExampleSpecGetNumAttributes(es).

8.25.4.13 int ExampleSpecLookupAttributeName (ExampleSpecPtr *es*, char ∗ *valName*)

Returns the index of the named attribute.

Does a linear search through the ExampleSpec's attributes looking for the first one named attributeName. Returns the index of the first matching attribute or -1 if there is no match.

8.25.4.14 int ExampleSpecLookupAttributeValueName (ExampleSpecPtr *es*, int *attNum*, char ∗ *valName*)

Returns the index of the named value.

If attribNum is a valid attribute index this does a linear search through the associated attribute's values for one named attributeName. Returns the index or -1 if there is no match.

8.25.4.15 int ExampleSpecLookupClassName (ExampleSpecPtr *es*, char ∗ *name*)

If name is a valid class name, this returns the index associated with the class.

8.25.4.16 ExampleSpecPtr ExampleSpecNew (void)

Programmatically creates a new ExampleSpec.

Use the ExampleSpecAddFOO functions to add classes, attributes, and their values to the spec.

This function allocates memory which should be freed by calling ExampleSpecFree.

8.25.4.17 ExampleSpecPtr ExampleSpecRead (char ∗ *fileName*)

Reads a .names formated file and build an ExampleSpec.

Attempts to read an example from the passed FILE ∗, which should be opened for reading. The file should contain an ExampleSpec in C4.5 format, that is the file should be a C4.5 names file.

This function will return 0 (NULL) if it is unable to read an ExampleSpec from the file (bad input or the file does not exist). If the input is badly formed, the function will also output an error to the console.

Note that you could pass stdin to the function to read an ExampleSpec from the console.

This function allocates memory which should be freed by calling ExampleSpecFree.

8.25.4.18 void ExampleSpecWrite (ExampleSpecPtr *es*, FILE ∗ *out*)

Outputs the ExampleSpec in .names format.

Writes the example to the passed FILE ∗, which should be opened for writing. The example will be written in C4.5 names format, and could later be read in using ExampleSpecRead.

Note that you could pass stdout to the function to write an ExampleSpec to the console.

8.26 folddata File Reference

8.26.1 Detailed Description

Randomly splits a data set into a collection of train/test pairs.

The folddata tool

This tool is used by the **xvalidate** tool; you may find it useful. For example, your algorithm might perform very poorly on a specific cross validation run, you could use this tool to reproduce the datasets and try to track down the problem.

Folddata splits a dataset into a number of testing and training sets as needed for doing cross-validation. It takes each example in the original dataset and randomly assigns it to one of the 'folds' is it creating (note that this randomness means that the folds won't be exactly evenly sized). Folddata then outputs one dataset for each fold with the examples from the fold as the test set and the examples in all the other folds as the training set.

Folddata works efficiently on large datasets, but will require enough disk space to hold 'folds' copies of the dataset.

Folddata takes input and does output in `c4.5 format`. It expects to find the files <`stem`>`.names` and <`stem`>`.data` and outputs <`stem`>`[0 - n].[names, data, test]`.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -target <dir>
  - Set the output directory (default '.')
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -folds <n>
  - Sets the number of train/test sets to create (default 10)
- -seed <n>
  - Sets the random seed, multiple runs with the same seed will produce the same datasets (defaults to a random seed)
- -h
  - Display usage information and exit
- -v
  - Can be used multiple times to increase the debugging output

Example

```
folddata -f test -target output -folds 15 -seed 10
```

Will create 15 folds from test.names and test.data and put them in the directory named output as test[0-14].names, test[0-14].data, and test[0-14].test. It will use a seeded random generator, so that the exact same dataset could be reproduced.

8.27 HashTable.h File Reference

8.27.1 Detailed Description

A hash table.

Thanks
to Chun-Hsiang Hung for implementing the **HashTable** ADT.

Wish List
A cleaner hash table interface (without the compare function). I think that the sprint learner is the only thing that uses this currently so there isn't much to change to fix this.

Data Structures

- struct **HashTable**

  A hash table ADT.

Functions

- **HashTable** ∗ **HashTableNew** (int size)

  Creates a new hash table with the specified number of entries.

- void **HashTableInsert** (**HashTable** ∗table, int index, void ∗element)

  Inserts the element into the has table at the appropriate place.

- void ∗ **HashTableFind** (**HashTable** ∗table, int index, int(∗cmp)(const void ∗, const int))

  Looks up index in the hash table.

- void **HashTableFree** (**HashTable** ∗table)

  Frees the memory being used by the hash table.

- void ∗ **HashTableRemove** (**HashTable** ∗table, int index, int(∗cmp)(const void ∗, const int))

  Removes the element from the hash table.

8.27.2 Function Documentation

8.27.2.1 void∗ HashTableFind (HashTable ∗ *table*, int *index*, int(∗ *cmp*)(const void ∗, const int))

Looks up index in the hash table.

Not only must you use the same index, but you must also supply a function that returns non-zero when passed the element you want to find and the index (which you passed to the function in the first place...)

8.27.2.2 void HashTableFree (HashTable ∗ *table*)

Frees the memory being used by the hash table.

But doesn't touch the memory being used by the elements in the table. It is your responsibility to free these if you like.

8.27.2.3 void HashTableInsert (HashTable ∗ *table*, int *index*, void ∗ *element*)

Inserts the element into the has table at the appropriate place.

8.27.2.4 HashTable∗ HashTableNew (int *size*)

Creates a new hash table with the specified number of entries.

8.27.2.5 void∗ HashTableRemove (HashTable ∗ *table*, int *index*, int(∗ *cmp*)(const void ∗, const int))

Removes the element from the hash table.

Not only must you use the same index, but you must also supply a function that returns non-zero when passed the element you want to find and the index (which you passed to the function in the first place...)

8.28 kmeans File Reference

8.28.1 Detailed Description

Performs k-means clustering.

Performs k-mean clustering on the continuous attributes in a data set (ignoring any discrete attributes).

The learner takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data.` and outputs the learned centers to a file called `<stem>.centers`.

Evaluates the learned centers by comparing to the centers found in <stem>.test as follows. Learned centers are greedily matched to the closest of the test centers until each center has a match, and then the evaluation is the sum of the squared distance between each test center and its matched learned center.

You can find a more full-featured kmeans clustering algorithm by running **vfkm** with the -batch argument (for example you can set initial centroid locations, etc.

Wish List

Modify this learner to work with discrete attributes.
Move the features from **vfkm** into this learner because this learner will be much easier to modify than that one for new users.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -clusters <dir>
  - Sets the number of clusters to look for, this argument is required.
- -threshold <threshold>
  - Iterate until every centroid moves less than this threshold.
- -u
  - Test by comparing to the centroids in <threshold>.test
- -v
  - Can be used multiple times to increase the debugging output

8.29 lists.h File Reference

8.29.1 Detailed Description

Generic list functions.

Void lists store untyped objects (void pointers) and Int and Float lists are special cased to work with appropriately typed data. The interfaces for the three are the same except all functions for VoidLists are prefaced with 'VL', IntList with 'IL', and FloatList with 'FL'. **Important** The documentation says 'VAL' instead of just 'VL' as a prefix to all functions, macros do the expansion to the array version of lists by default so these are the same in practice.

And since these list data structures are basically arrays that are dynamically sized, inserting or removing elements is O(N) but accessing elements is O(1).

This module also has linked lists which are used very very rarely in VFML. You can access them by adding an additional L to the name, for example **VLNew()** -> **VLLNew()**

Examples

To iterate over the values in a VoidList:

```
VoidListPtr list = (some existing list);
void *e;
int i;

for(i = 0 ; i < VLLength(list) ; i++) {
   e = VLIndex(list, i);
}
```

The same thing with a FloatList:

```
FloatListPtr list = (some existing list);
float f;
int i;

for(i = 0 ; i < FLLength(list) ; i++) {
   f = FLIndex(list, i);
}
```

Defines

- #define **VALLength**(list) (((VoidAListPtr)(list)) → size)

  Returns the number of elements in the list.

- #define **VALIndex**(list, index) ((index < ((VoidAListPtr)list) → size) ? (((VoidAListPtr)list) → array[index]) : (0))

  Returns the element at the index (zero based).

Functions

- VoidAListPtr **VALNew** (void)

  Creates a new list of the specified type.

- void **VALAppend** (VoidAListPtr list, void ∗element)

Append the element to the end of the list.

- void **VALPush** (VoidAListPtr list, void ∗element)

  Push element on the head of the list.

- void **VALSort** (VoidAListPtr list, int(∗cmp)(const void ∗, const void ∗))

  Uses quick sort to sort the list in place with the passed comparision function.

- void ∗ **VALRemove** (VoidAListPtr list, long index)

  Removes the item at the specified index and returns it.

- void **VALInsert** (VoidAListPtr list, void ∗element, long index)

  Inserts the element at the specified index.

- void **VALSet** (VoidAListPtr list, long index, void ∗newVal)

  Sets the value at the specified index.

- void **VALFree** (VoidAListPtr list)

  Frees the memory associated with the list.

8.29.2 Define Documentation

8.29.2.1 #define VALIndex(list, index) ((index < ((VoidAListPtr)list) → size) ? (((VoidAListPtr)list) → array[index]) : (0))

Returns the element at the index (zero based).

Note that the indexing is 0 based (like a C array).

8.29.2.2 #define VALLength(list) (((VoidAListPtr)(list)) → size)

Returns the number of elements in the list.

8.29.3 Function Documentation

8.29.3.1 void VALAppend (VoidAListPtr *list*, void ∗ *element*)

Append the element to the end of the list.

8.29.3.2 void VALFree (VoidAListPtr *list*)

Frees the memory associated with the list.

You are responsible for any memory used by the elements of the list.

8.29.3.3 void VALInsert (VoidAListPtr *list*, void ∗ *element*, long *index*)

Inserts the element at the specified index.

The element that used to have that index will have an index one higher (that is, it shifts later elements over). Note that the indexing is 0 based (like a C array).

8.29.3.4 VoidAListPtr VALNew (void)

Creates a new list of the specified type.

8.29.3.5 void VALPush (VoidAListPtr *list*, void ∗ *element*)

Push element on the head of the list.

8.29.3.6 void∗ VALRemove (VoidAListPtr *list*, long *index*)

Removes the item at the specified index and returns it.

Shifts later elements over. Note that the indexing is 0 based (like a C array).

8.29.3.7 void VALSet (VoidAListPtr *list*, long *index*, void ∗ *newVal*)

Sets the value at the specified index.

The previous value at the index is overwritten. Note that the indexing is 0 based (like a C array).

8.29.3.8 void VALSort (VoidAListPtr *list*, int(∗ *cmp*)(const void ∗, const void ∗))

Uses quick sort to sort the list in place with the passed comparision function.

Note this is not implemented for FloatLists.

8.30 memory.h File Reference

8.30.1 Detailed Description

Tracks the size of allocations made.

A wrapper for the standard memory manager that tracks the size of the allocations made using it. This requires an extra 8 bytes per allocation. You can turn this off by editing the **memory.h** file and commenting out the definition of DEBUGMEMORY, but this will cause some of the features of some of the research learners to fail in unpredictable ways.

Defines

- #define **MMemMove**(dst, src, bytes) __SystemMoveMemory(dst, src, bytes)

  Wrapper for memmove that works with pointers allocated by this memory module.

Functions

- void * **MNewPtr** (int size)

  A wrapper around Malloc that tracks the size of the allocation.

- void **MFreePtr** (void *ptr)

  Frees the memory held by the pointer and tracks the change in the module's records.

- void **MSetAllocFailFunction** (void(*AllocFail)(int allocationSize))

  Sets a function that is called if an allocation fails.

- long **MGetTotalAllocation** (void)

  Returns the number of bytes that are currently allocated by the module.

- void **MSetActivePool** (int poolID)

  Set the pool to track future memory allocations.

- int **MGetActivePool** (void)

  Find which pool is being used to track memory.

- long **MGetPoolAllocation** (int poolID)

  Returns the number of bytes that are currently allocated in the specified pool.

- void **MMovePtrToPool** (void *ptr, int poolID)

  Moves the memory from one pool to another.

8.30.2 Define Documentation

8.30.2.1 #define MMemMove(dst, src, bytes) __SystemMoveMemory(dst, src, bytes)

Wrapper for memmove that works with pointers allocated by this memory module.

8.30.3 Function Documentation

8.30.3.1 void MFreePtr (void ∗ *ptr*)

Frees the memory held by the pointer and tracks the change in the module's records.

8.30.3.2 int MGetActivePool (void)

Find which pool is being used to track memory.

The module supports up to 9 pools of memory, with id 1-9. You can get more fine grained tracking by setting the pool and using MGetPoolAllocation. There is also a special pool, with id 0, and any allocations made when that pool are set are tracked in the pool but will not show up in MGetTotalAllocation. The default pool id is 1.

8.30.3.3 long MGetPoolAllocation (int *poolID*)

Returns the number of bytes that are currently allocated in the specified pool.

The module supports up to 9 pools of memory, with id 1-9. You can get more fine grained tracking by setting the pool and using MGetPoolAllocation. There is also a special pool, with id 0, and any allocations made when that pool are set are tracked in the pool but will not show up in MGetTotalAllocation. The default pool id is 1.

8.30.3.4 long MGetTotalAllocation (void)

Returns the number of bytes that are currently allocated by the module.

8.30.3.5 void MMovePtrToPool (void ∗ *ptr*, int *poolID*)

Moves the memory from one pool to another.

Changes (if needed) the tracking of the ptr so that it is now counted against the new pool instead of the pool it was allocated into.

8.30.3.6 void∗ MNewPtr (int *size*)

A wrapper around Malloc that tracks the size of the allocation.

Make sure you use MFreePtr to free any memory allocated by this call or your program will probably crash.

8.30.3.7 void MSetActivePool (int *poolID*)

Set the pool to track future memory allocations.

The module supports up to 9 pools of memory, with id 1-9. You can get more fine grained tracking by setting the pool and using MGetPoolAllocation. There is also a special pool, with id 0, and any allocations made when that pool are set are tracked in the pool but will not show up in MGetTotalAllocation. The default pool id is 1.

8.30.3.8 void MSetAllocFailFunction (void(* *AllocFail*)(int allocationSize))

Sets a function that is called if an allocation fails.

If you call this, then your AllocFail function will be called if an allocation fails. After your function returns the memory module will try the allocation agian, if the allocation fails again, the memory module returns a NULL pointer. You could use AllocFail to flush caches, or clean up the program and quit.

8.31 mostcommonclass File Reference

8.31.1 Detailed Description

Predicts the most common class in the training data.

This is a very simple 'learner', but it may be useful as a baseline to compare your learner against; predicting with 99% accuracy isn't impressive if 98% of the examples have the same class.

The mostcommonclass learner works in time proportional to the number of training examples and uses space proportional to the number of classes. It should be able to work on large datasets.

The learner takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data`. Depending on command line argument, it will either output the most common class or test its error rate on `<stem>.test`.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -u
  - Test on the examples in <stem>.test and output in a format appropriate for interface with xvalidate and batchtest (defaults to off)
- -v
  - Can be used multiple times to increase the debugging output

Example

```
mostcommonclass -f banana -source datasets/banana
```

Looks for a dataset named 'banana' in the 'datasets/banana' directory. Outputs the name of the most common class in the dataset.

8.32 naivebayes File Reference

8.32.1 Detailed Description

A Naive Bayes learner.

An implementation of Naive Bayes that works with categorical attributes. This learner works in time proportional to the number of training examples. It also requires memory that is proportional to the number of classes ∗ number attributes ∗ number of values.

The learner takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data.` Depending on command line argument, it will either output the model if you use the -v argument enough or test its error rate on `<stem>.test`.

Wish List

Modify this learner to work with continuous attributes.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -u
  - Test on the examples in <stem>.test and output in a format appropriate for interface with xvalidate and batchtest (defaults to off)
- -v
  - Can be used multiple times to increase the debugging output

8.33 random.h File Reference

8.33.1 Detailed Description

Generates random numbers in a number of ways, and has support for saving and restoring the state of the random number generator.

Functions

- void **RandomInit** (void)
  Should be called before calling other random functions.
- int **RandomRange** (int min, int max)
  Returns a number between the min and max (inclusive).
- long **RandomLong** (void)
  Returns a random long integer.
- double **RandomDouble** (void)
  Returns a random double from 0 - 1.
- double **RandomGaussian** (double mean, double stdev)
  Samples from a Gaussian with the specified parameters.
- double **RandomStandardNormal** (void)
  Samples from the standard normal distribution.
- void **RandomSeed** (unsigned int seed)
  Seeds the random number generator.
- void ∗ **RandomNewState** (unsigned int seed)
  Creates a new random state to use with RandomSetState.
- void ∗ **RandomSetState** (void ∗state)
  Sets the current random state.
- void **RandomFreeState** (void ∗state)
  Use this to free any state made with RandomNewState.

8.33.2 Function Documentation

8.33.2.1 double RandomDouble (void)

Returns a random double from 0 - 1.

8.33.2.2 void RandomFreeState (void ∗ *state*)

Use this to free any state made with RandomNewState.

8.33.2.3 double RandomGaussian (double *mean*, double *stdev*)

Samples from a Gaussian with the specified parameters.

8.33.2.4 void RandomInit (void)

Should be called before calling other random functions.

8.33.2.5 long RandomLong (void)

Returns a random long integer.

8.33.2.6 void* RandomNewState (unsigned int *seed*)

Creates a new random state to use with RandomSetState.

Each state represents a separate stream of repeatable random numbers. This is implemented with initstate and setstate on Unix and is currently not supported on windows.

8.33.2.7 int RandomRange (int *min*, int *max*)

Returns a number between the min and max (inclusive).

8.33.2.8 void RandomSeed (unsigned int *seed*)

Seeds the random number generator.

8.33.2.9 void* RandomSetState (void * *state*)

Sets the current random state.

The state parameter should have been made by RandomNewState. This allows you to have multiple repeatable random number sequences at once. Returns the previous state.

8.33.2.10 double RandomStandardNormal (void)

Samples from the standard normal distribution.

8.34 REPrune.h File Reference

8.34.1 Detailed Description

Peforms reduced error pruning on a decision tree.

Takes a decision tree and a collection of examples and prunes every portion of the tree that does not contribute to an increase of accuracy on the data set. This requires a single pass over the prune set, and then on order of one pass over the tree for each node that is pruned.

REPrune uses the growing data fields of the decision tree, so it will overwrite anything that you may have there. If you want to trying pruning a tree and then continue growing it you should make a clone of the tree and pass the clone to this interface so that the origional will remain untouched.

Functions

- void **REPruneBatch** (**DecisionTreePtr** dt, VoidAListPtr examples)

  Get the prune set from the in-memory list.

- void **REPruneBatchFile** (**DecisionTreePtr** dt, FILE ∗exampleIn, long pruneMax)

  Get the prune set from the file.

8.34.2 Function Documentation

8.34.2.1 void REPruneBatch (DecisionTreePtr *dt*, VoidAListPtr *examples*)

Get the prune set from the in-memory list.

8.34.2.2 void REPruneBatchFile (DecisionTreePtr *dt*, FILE ∗ *exampleIn*, long *pruneMax*)

Get the prune set from the file.

Do not use more than pruneMax of the examples from the file for pruning; setting pruneMax to 0 means 'no max' (and uses all examples from the file for pruning).

8.35 sampledata File Reference

8.35.1 Detailed Description

Draws a sample from a data set.

sampledata does a scan over a data set and draws a sample from it by including each example in the sample with a user specified probability.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -fout <filestem>
  - Set the name of the output dataset (default DF-out)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -target <dir>
  - Set the directory to contain the output dataset (default '.')
- -seed <seed>
  - Sets the random seed, multiple runs with the same seed will produce the same datasets (defaults to a random seed)
- -sample <n>
  - Sets the percentage of the data to sample (default .8 – that's 80%)
- -stdin
  - Read examples from standard in (default read file)
- -stdout
  - Writes examples to standard out (default write to file)
- -h
  - Display usage information and exit.
- -v
  - Can be used multiple times to increase the debugging output

8.36 shuffledata File Reference

8.36.1 Detailed Description

Randomizes the order of a data set and rewrites it.

shuffledata reads a data set into RAM and outputs it in random order. Data sets that do not fit in RAM are difficult to randomize and VFML does not have a tool to deal with such data sets.

Wish List
for a version of this tool that works on disk and does not need to load data into RAM.

Arguments

- -f <filestem>
  - Set the stem name (default DF)
- -fout <filestem>
  - Set the name of the output dataset (default shuffled)
- -source <dir>
  - Set the directory that contains the dataset (default '.')
- -target <dir>
  - Set the directory to contain the output dataset (default '.')
- -seed <seed>
  - Sets the random seed, multiple runs with the same seed will produce the same datasets (defaults to a random seed)
- -h
  - Display usage information and exit.
- -v
  - Can be used multiple times to increase the debugging output

8.37 sprint File Reference

8.37.1 Detailed Description

Learn a decision tree from data sets that do not fit in RAM.

Learns a decision tree from a data set that is larger than RAM as described in `this paper`.

sprint takes input and does output in `c4.5 format`. It expects to find the files <`stem`>`.names` and <`stem`>`.data`.

Thanks
to Chun-Hsiang Hung for doing the core development work for **sprint**.

Arguments

- -h
  - Run sprint -h for a list of the arguments and their meanings.

8.38 stats.h File Reference

8.38.1 Detailed Description

Some statistical functions.

The stat module has some basic statistical functions and the StatTracker type, which watches a series of samples, keeps some simple summary statistics (sum and sumSquare in the current implementation), and can report on the mean, variance, stdev, etc. of the sample.

Wish List
The StatTracker would track more interesting things.

Data Structures

- struct **_StatTracker_**

  Holds simple summary statistics of a sample.

Typedefs

- typedef **_StatTracker_ StatTrackerStruct**

  Holds simple summary statistics of a sample.

- typedef **_StatTracker_** * **StatTracker**

  Holds simple summary statistics of a sample.

Functions

- **StatTracker StatTrackerNew** (void)

  Creates a new stat tracker that is ready to have samples added to it.

- void **StatTrackerFree** (**StatTracker** st)

  Frees the memory associated with the StatTracker.

- void **StatTrackerAddSample** (**StatTracker** st, double x)

  Records the sample in the tracker.

- double **StatTrackerGetMean** (**StatTracker** st)

  Returns the mean of the samples that have been shown to the tracker.

- double **StatTrackerGetVariance** (**StatTracker** st)

  Returns the variance of the samples that have been shown to the tracker.

- double **StatTrackerGetStdev** (**StatTracker** st)

  Returns the standard deviation of the samples that have been shown to the tracker.

- long **StatTrackerGetNumSamples** (**StatTracker** st)

Returns the number of samples shown to the stat tracker.

- double **StatTrackerGetNormalBound** (**StatTracker** st, double delta)

  Assume Gaussian and return a high confidence bound of the sample .

- double **StatGetNormalBound** (double variance, long n, double delta)

  Return a high confidence bound from the normal distribution.

- double **StatHoeffdingBoundOne** (double range, double delta, long n)

  Returns the one sided Hoeffding bound.

- double **StatHoeffdingBoundTwo** (double range, double delta, long n)

  Returns the two sided Hoeffding bound.

- float **StatLogGamma** (float xx)

  Returns the log of the gamma function of xx.

8.38.2 Typedef Documentation

8.38.2.1 typedef struct _StatTracker_ ∗ StatTracker

Holds simple summary statistics of a sample.

See **stats.h** for more detail.

8.38.2.2 typedef struct _StatTracker_ StatTrackerStruct

Holds simple summary statistics of a sample.

See **stats.h** for more detail.

8.38.3 Function Documentation

8.38.3.1 double StatGetNormalBound (double *variance*, long *n*, double *delta*)

Return a high confidence bound from the normal distribution.

Assumes a Gaussian distribution and returns the distance from the mean within which 1 - delta of the mass lies (esentially the same function as StatTrackerGetNormalBound).

8.38.3.2 double StatHoeffdingBoundOne (double *range*, double *delta*, long *n*)

Returns the one sided Hoeffding bound.

With n observations of a variable with the specified range. That is, the true value is less than the observed mean + this function's result with probability 1 - delta.

8.38.3.3 double StatHoeffdingBoundTwo (double *range*, double *delta*, long *n*)

Returns the two sided Hoeffding bound.

With n observations of a variable with the specified range. That is, the true value is within this function's result of the obseved mean with probability 1 - delta.

8.38.3.4 float StatLogGamma (float *xx*)

Returns the log of the gamma function of xx.

This function is lifted from Numerical Recipes in C.

8.38.3.5 void StatTrackerAddSample (StatTracker *st*, double *x*)

Records the sample in the tracker.

8.38.3.6 void StatTrackerFree (StatTracker *st*)

Frees the memory associated with the StatTracker.

8.38.3.7 double StatTrackerGetMean (StatTracker *st*)

Returns the mean of the samples that have been shown to the tracker.

If you have not shown the tracker any samples this will probably crash.

8.38.3.8 double StatTrackerGetNormalBound (StatTracker *st*, double *delta*)

Assume Gaussian and return a high confidence bound of the sample .

Assumes a Gaussian distribution and returns the distance from the mean within which 1 - delta of the mass lies. This calls the GetStdev function internally and will crash if that crashes.

8.38.3.9 long StatTrackerGetNumSamples (StatTracker *st*)

Returns the number of samples shown to the stat tracker.

8.38.3.10 double StatTrackerGetStdev (StatTracker *st*)

Returns the standard deviation of the samples that have been shown to the tracker.

If you have not shown the tracker at least two samples this will probably crash.

8.38.3.11 double StatTrackerGetVariance (StatTracker *st*)

Returns the variance of the samples that have been shown to the tracker.

If you have not shown the tracker at least two samples this will probably crash.

8.38.3.12 StatTracker StatTrackerNew (void)

Creates a new stat tracker that is ready to have samples added to it.

8.39 treedata File Reference

8.39.1 Detailed Description

Creates a synthetic data set by sampling from a randomly generated DecisionTree.

This program creates a synthetic binary tree and then uses it to label data which can then be used to evaluate learning algorithms. It has been used to evaluate the **vfdt** system.

The synthetic tree is generated starting from a single node as follows. A leaf is selected and is either split on one of the active attributes (each discrete attribute is used at most once on a path from the root to a leaf) or it is pruned. The probability of pruning is set by the *prunePercent* parameter but is 0 if the leaf is not below *firstPruneLevel* and is 1 if the leaf is below *maxPruneLevel*. If the attribute selected for a split is continuous a threshold is generated uniformly in the range 0-1 except that Tree Data ensures that the chosen threshold is not redundant with an earlier split.

Once the tree's structure is created (all leaves have been pruned) a class label is randomly assigned to each leaf and then redundant subtrees (where every leaf has the same classification) are pruned.

Data (training, testing, and pruning) is generated by creating an example and setting its attributes with uniform probability. The tree is used to label the data, and then the class and discrete lables are resampled with uniform probability (without replacement) as specified by the *noise* parameter and continuous attributes are changed by sampling from a gaussian with mean of their current value and standard deviation of *noise*.

Using the same *conceptSeed* (along with the same other parameter) results in the same concept being created. The same *seed* results in data being generated the same (so experiments are easily repeatable).

This program also outputs some additional statistics into the *stem*.stats file.

Arguments

- -h
  - Run with this argument to get a list of arguments and their meanings.

Example

```
treedata -discrete 10 -continuous 0 -noise 15 -conceptSeed 21 -seed 1234 -prune-
Percent 15 -train 100 -test 100 -prune 100
```

Creates 100 training, 100 testing, and 100 pruning examples from a concept tree made with 15% chance of pruning each node past level 3 and 100% chance of pruning past level 18. 15% noise is added to the data. Finally, the same data set will be produced by multiple calls to the function because of the *seed* arguments.

8.40 uRunner File Reference

8.40.1 Detailed Description

Distribute a collection of jobs across a cluster of computers.

This tool is used to take a collection of jobs and easily execute them across a cluster of computers. The tool is actually two programs: the server program, uRunner and the client program, uRunnerClient. You provide a config file and a job queue, then execute the server. The server starts the clients on the selected hosts and they begin taking and executing jobs from the queue (using a safe locking mechanism).

The Config File

```
[GLOBAL]

experimentname = cluster-runs

scratchdir = /scratch/%(experimentname)s/%(host)s/
jobsdir    = .
lockdir    = %(jobsdir)s/
queuefile  = %(jobsdir)s/queue
resultdir  = %(jobsdir)s/results/

clients    = calypso poseidon helios rhea apollo
```

The `[GLOBAL]` statement is required but has no semantics (it comes from the python config parser that uRunner uses). Other than that, the required fields are `scratchdir`, `lockdir`, `queuefile`, `resultdir`, and `clients`. Each of these may contain text and substitutions of previous variables. The syntax for a substitution is (variablename)s (the s let it know you want to sub in a string, another artifact of the python config parser). You can also use the `host` variable which is set to the host that the client is running on (different for each client).

Scratch Directory The uRunnerClient makes the directory `scratchdir` (and any directories it needs to leading up to it) and frees it (and them) after each job. This means that if you specify a scratch dir that doesn't exist the uRunner cleans it after each run, if it does exist uRunner doesn't change it (or its contents). uRunner changes directory to this scratch directory before running any jobs. You might want to make the scratch directory on local disk so your program will have faster access to it. One common idiom is to copy (or generate) a dataset to a localy mounted disk and run several similar algorithms on the data in a single job. It might also be convienient because multiple copies of the same program won't have conflicts with temporary files.

Lock Directory This must be a network mounted directory and all the clients must be able to access it. During the run clients create lockfiles in this directory so they don't corrupt the queuefile by accessing it simultaneously.

Result Directory This directory must be a network mounted directory and all the clients must be able to access it. If it doesn't exist it will be created. Jobs should copy the results of the programs they execute to this directory so the results won't be lost when the Scratch Directory is cleaned up.

List of Clients A copy of uRunnerClient will be executed on each of these clients via a call to rsh.

The Queue File

```
---
Experiment12-18-0.0300
nice -5 $rundir/clusterdata -continuous 12 -clusters 18 -stdev 0.0300 -train 10000000 -seed 9177 -conceptSeed 18354
nice -5 $rundir/vfkm -progressive -allowBadConverge -clusters 18 -dbSize 10000000 -converge 0.021600 -seed 9178 > outpu
echo "d12 k18 0.0300 " "`cat output`" >> $resultdir/results-vfkm
rm output
nice -5 $rundir/vfkm -progressive -allowBadConverge -clusters 18 -dbSize 10000000 -normalApprox -converge 0.021600 -see
echo "d12 k18 0.0300 " "`cat output`" >> $resultdir/results-pvfkm
rm output
nice -5 $rundir/vfkm -batch -clusters 18 -dbSize 10000000 -normalApprox -converge 0.021600 -seed 9178 > output
echo "d12 k18 0.0300 " "`cat output`" >> $resultdir/results-km
rm output
---
```

The queue file is a series of jobs, see the example above. First note that each job is surrounded by `---` (which means `---` will appear twice between jobs). The first line of each job is the job's name, which can be accessed via the $jobname environment variable. The $rundir environment variable is also available, and points to the directory from which uRunner was executed. The other available environment variables are $resultdir, $scratchdir, and $host. Note, additional variables you declare in the config file are not available.

The example above shows several useful idioms. First, before starting the run, copies of the binaries involved are made in $rundir. This allows development to continue while the experiment is executing. The example first runs 'clusterdata' to create a very large dataset (remember this will be in the scratch directory). It then calls a learning algorithm 'vfkm' with three different but related parameter settings. It concatinates the output of each learner onto a file in the result directory, and cleans up after itself. Notice that the scratch directory mechinism will clean up after the entire run is complete, but within the run some cleaning is needed as well (removing the output file). Also notice that there are some kinda odd constructs which have to do with the shell that python executes, I think you need to make sure your commands are in sh (check this).

Notes

On the UW computers you need some kind of kerberos tickits to rsh. I'm not 100% sure how this works, but you seem to get a bunch of tickets when you first log in, so if you get errors about permission denied try logging out and back in or run the kinit command to get some more.

8.41 vfbn1 File Reference

8.41.1 Detailed Description

Learns the structure of a BeliefNet from a very large data set using sampling.

Learns the structure and parameters of a Bayesian network, accelerated with sampling as described in `this paper`. All variables must be categorical.

vfbn1 takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data`.

Wish List
An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

Arguments

- -h
  - Run vfbn1 -h for a list of the arguments and their meanings.

8.42 vfbn2 File Reference

8.42.1 Detailed Description

Learns the structure of a BeliefNet from a very large data set using sampling and a new search proceedure.

Learns the structure and parameters of a Bayesian network, accelerated with sampling as described in `this paper`. All variables must be categorical.

vfbn2 takes input and does output in `c4.5 format`. It expects to find the files <`stem`>`.names` and <`stem`>`.data`.

Wish List

An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

Arguments

- -h
  - Run vfbn2 -h for a list of the arguments and their meanings.

8.43 vfdt File Reference

8.43.1 Detailed Description

Learns a decision tree from a high-speed data stream or very large data set.

vfdt is described in `this paper`. (This version of VFDT has many extensions since that paper was written, including the ability to learn from domains with continuous attributes, we hope to have a more up-to-date paper to cite here soon).

vfdt takes input and does output in `c4.5 format`. It expects to find the files <`stem`>`.names` and <`stem`>`.data`.

You might also be interested in the **vfdt-engine::h** interface, which provides an API to the learning engine used in this program.

Arguments

- -h
  - Run vfdt -h for a list of the arguments and their meanings.

8.44 vfdt-engine.h File Reference

8.44.1 Detailed Description

An API which lets your program learn a DecisionTree from a high-speed data stream.

An API to VFML's engine for learning decision trees from high-speed data streams. The **vfdt** program is basically a wrapper around this interface. You can use this interface to learn decision trees from a stream of data as it arrives, and use the tree in a parallel to make predictions.

To use this interface you generate a VFDTPtr using the VFDTNew method, set any parameters you want to use to control the learning using the functions that are described below, and then repeatedly feed examples to the VFDTPtr using the VFDTProcessExample method. You can call VFDTGetLearnedTree at anytime to get a copy of the current learned tree. Note that vfdt-engine will take over the memory of any examples feed to it, and you should not free them or your program will crash.

Wish List

A method that checkpoints the learning procedure to disk so that it can be restored at a later time. I think the hard part of this would be checkpointing the ExampleGroupStats structure.

Data Structures

- struct **_VFDT_**

  Holds the information needed to learn decision trees from data streams.

Typedefs

- typedef **_VFDT_ VFDT**

  Holds the information needed to learn decision trees from data streams.

- typedef **_VFDT_** ∗ **VFDTPtr**

  Holds the information needed to learn decision trees from data streams.

Functions

- **VFDTPtr VFDTNew** (**ExampleSpecPtr** spec, float splitConfidence, float tieConfidence)

  Allocate memory to learn a decision tree from a data stream.

- void **VFDTFree** (**VFDTPtr** vfdt)

  frees all the memory being used by the learning process.

- void **VFDTSetMessageLevel** (**VFDTPtr** vfdt, int value)

  Higher message levels print more output to the console.

- void **VFDTSetMaxAllocationMegs** (**VFDTPtr** vfdt, int value)

  Put a limit on the dynamic memory used by the program.

- void **VFDTSetProcessChunkSize** (**VFDTPtr** vfdt, int value)

  Sets the number of examples before checks for tree growth.

- void **VFDTSetUseGini** (**VFDTPtr** vfdt, int value)

  Set the evaluation function to Gini (default is Entropy).

- void **VFDTSetRestartLeaves** (**VFDTPtr** vfdt, int value)

  Consider reactivating leaves where growing was stopped to save RAM.

- void **VFDTSetCacheTrainingExamples** (**VFDTPtr** vfdt, int value)

  Use extra RAM to cache training examples.

- void **VFDTSetPrePruneTau** (**VFDTPtr** vfdt, float value)

  Set the pre-prune parameter.

- void **VFDTSetLaplace** (**VFDTPtr** vfdt, int value)

  Set how many examples worth of smoothing to do in class probability estimates.

- void **VFDTProcessExamples** (**VFDTPtr** vfdt, FILE ∗input)

  Read as many examples as possible from the file and learn from them.

- void **VFDTProcessExamplesBatch** (**VFDTPtr** vfdt, FILE ∗input)

  Learn from the examples in batch mode.

- void **VFDTProcessExampleBatch** (**VFDTPtr** vfdt, **ExamplePtr** e)

  Add the example to the learner without checking for splits.

- void **VFDTBatchExamplesDone** (**VFDTPtr** vfdt)

  Forces vfdt-engine to make as many splits as possible using traditional methods.

- void **VFDTProcessExample** (**VFDTPtr** vfdt, **ExamplePtr** e)

  Adds another example to the learning process.

- int **VFDTIsDoneLearning** (**VFDTPtr** vfdt)

  Returns 1 if no nodes in the tree are still active.

- long **VFDTGetNumGrowing** (**VFDTPtr** vfdt)

  Returns the number of nodes that are growing.

- long **VFDTGetNumBoundsUsed** (**VFDTPtr** vfdt)

  Returns the number of statistical tests made.

- void **VFDTPrintStats** (**VFDTPtr** vfdt, FILE ∗out)

  Prints some information about the growing nodes to the file.

- **DecisionTreePtr VFDTGetLearnedTree** (**VFDTPtr** vfdt)

  Returns the current tree.

8.44.2 Typedef Documentation

8.44.2.1 typedef struct _VFDT_ VFDT

Holds the information needed to learn decision trees from data streams.

8.44.2.2 typedef struct _VFDT_ ∗ VFDTPtr

Holds the information needed to learn decision trees from data streams.

8.44.3 Function Documentation

8.44.3.1 void VFDTBatchExamplesDone (VFDTPtr *vfdt*)

Forces vfdt-engine to make as many splits as possible using traditional methods.

8.44.3.2 void VFDTFree (VFDTPtr *vfdt*)

frees all the memory being used by the learning process.

8.44.3.3 DecisionTreePtr VFDTGetLearnedTree (VFDTPtr *vfdt*)

Returns the current tree.

This will guess on which class to predict at the growing leaves (smoothing with strength of the lapace paramterer towards the distribution seen at the parent). Returns a copy of the internally growing decision tree, and so vfdt-engine can continue to learn on the internal tree unaffected.

8.44.3.4 long VFDTGetNumBoundsUsed (VFDTPtr *vfdt*)

Returns the number of statistical tests made.

8.44.3.5 long VFDTGetNumGrowing (VFDTPtr *vfdt*)

Returns the number of nodes that are growing.

8.44.3.6 int VFDTIsDoneLearning (VFDTPtr *vfdt*)

Returns 1 if no nodes in the tree are still active.

This may happen because RAM runs out, or because everything was pre-pruned.

8.44.3.7 VFDTPtr VFDTNew (ExampleSpecPtr *spec*, float *splitConfidence*, float *tieConfidence*)

Allocate memory to learn a decision tree from a data stream.

Sets up the learning, makes an initial tree with a single node. splitConfidence is the delta parameter from our paper (the probability of making a mistake with the sampling we use) and the tie-Confidence is tau (the minimum difference in gain that you care about). splitConfidence should be a small non-zero number, maybe 10^-7. And tieConfidence should be something in the range of 0 - .1 (although slightly bigger values may be useful).

8.44.3.8 void VFDTPrintStats (VFDTPtr *vfdt*, FILE ∗ *out*)

Prints some information about the growing nodes to the file.

8.44.3.9 void VFDTProcessExample (VFDTPtr *vfdt*, ExamplePtr *e*)

Adds another example to the learning process.

Check for splits as needed (according to the chunk size).

8.44.3.10 void VFDTProcessExampleBatch (VFDTPtr *vfdt*, ExamplePtr *e*)

Add the example to the learner without checking for splits.

When you have added all the examples you want call VFDTBatchExamplesDone to tell vfdt-engine it is time to make splits.

8.44.3.11 void VFDTProcessExamples (VFDTPtr *vfdt*, FILE ∗ *input*)

Read as many examples as possible from the file and learn from them.

This will repeatedly read and learn from examples in the file until it can not read any more. Note that this function will block until that time.

8.44.3.12 void VFDTProcessExamplesBatch (VFDTPtr *vfdt*, FILE ∗ *input*)

Learn from the examples in batch mode.

That is, read them all into RAM and use every example to make every learning decision.

8.44.3.13 void VFDTSetCacheTrainingExamples (VFDTPtr *vfdt*, int *value*)

Use extra RAM to cache training examples.

Default is to cache. This keeps examples in memory to possibly use them to help make several decisions, speeding up the induction. When RAM is full vfdt-engine starts deactivating example caches at the least promising leaves. All caches will be deactivated before any leaves are deactived.

8.44.3.14 void VFDTSetLaplace (VFDTPtr *vfdt*, int *value*)

Set how many examples worth of smoothing to do in class probability estimates.

8.44.3.15 void VFDTSetMaxAllocationMegs (VFDTPtr *vfdt*, int *value*)

Put a limit on the dynamic memory used by the program.

This requires that DEBUGMEMORY is defined in **memory.h** (which is the default). By setting this you limit the amount of memory allocated with calls to MemNewPtr by either your program and by vfdt-engine. This means any other calls you make to VFML functions (e.g. reading examples from disk) will be counted against vfdt's total. (you can use MSetActivePool with pool id 0 to get around this).

If this memory threshold is crossed vfdt-engine starts purging its allocations by first throwing away cahed examples and then disabling learning at the least promising leaves.

8.44.3.16 void VFDTSetMessageLevel (VFDTPtr *vfdt*, int *value*)

Higher message levels print more output to the console.

Levels above 2 print a lot of output. More than you want. I promise.

8.44.3.17 void VFDTSetPrePruneTau (VFDTPtr *vfdt*, float *value*)

Set the pre-prune parameter.

The default is 0.0, which means no pre-pruning. If the gain of all attributes is less than this value then pre-prune. Also, do not call a tie unless an attribute beats another by at least this much.

8.44.3.18 void VFDTSetProcessChunkSize (VFDTPtr *vfdt*, int *value*)

Sets the number of examples before checks for tree growth.

Check for growth at a leaf once every time it accumulates this many examples. Default is 300.

8.44.3.19 void VFDTSetRestartLeaves (VFDTPtr *vfdt*, int *value*)

Consider reactivating leaves where growing was stopped to save RAM.

The default value for this is true, so periodically vfdt-engine looks at all the deactivated leaves to see if any of them are more promising than any of the currently active ones, and makes adjustments. If set to false any leaf that is deactivated is effectively pre-pruned.

8.44.3.20 void VFDTSetUseGini (VFDTPtr *vfdt*, int *value*)

Set the evaluation function to Gini (default is Entropy).

8.45 vfem File Reference

8.45.1 Detailed Description

Performs EM clustering.

Performs EM clustering for sphirical gaussians accelerated with sampling as described in `this paper`. This learner ignores categorical attributes.

VFEM takes input and does output in `c4.5 format`. It expects to find the files <`stem`>`.names` and <`stem`>`.data.` and outputs the learned centers to a file called <`stem`>`.centers`.

It evaluates the learned centers by comparing to the centers found in <stem>.test as follows. Learned centers are greedily matched to the closest of the test centers until each center has a match, and then the evaluation is the sum of the squared distance between each test center and its matched learned center.

Arguments

- -h
  - Run vfem -h for a list of the arguments and their meanings.

8.46 vfkm File Reference

8.46.1 Detailed Description

Performs k-means clustering accelerated with sampling.

Performs k-mean clustering accelerated with sampling as described in `this paper`. This learner ignores categorical attributes.

The learner takes input and does output in `c4.5 format`. It expects to find the files `<stem>.names` and `<stem>.data.` and outputs the learned centers to a file called `<stem>.centers`.

It evaluates the learned centers by comparing to the centers found in <stem>.test as follows. Learned centers are greedily matched to the closest of the test centers until each center has a match, and then the evaluation is the sum of the squared distance between each test center and its matched learned center.

Arguments

- -h
  - Run vfkm -h for a list of the arguments and their meanings.

8.47 xvalidate File Reference

8.47.1 Detailed Description

Performs cross validation of a learner on a data set.

You will probably want to use the **batchtest** tool for large experiments; xvalidate will help you to quickly test things, perhaps as a debugging aid.

You can use xvalidate with large datasets, but you will need enough disk space to hold 'folds' copies of the data. The learner you use with xvalidate must also be able to work with large datasets.

Xvalidate takes input in `C4.5 format` and uses **folddata**, which must be in your path. You use the -c option to tell xvalidate how to run the learner. Xvalidate will append the names of the folds of the datasets to the end of the -c string, the learner must accept the name and read input appropriately.

Xvalidate expects the learner to output results in the following format:

```
error-rate size
```

The learner's error rate on the test set, followed by some whitespace, followed by the size of the learned model (in whatever unit you want), followed by a newline.

Xvalidate will collect the output of the runs of the learner, average them, and report:

```
mean-error-rate (standard deviation of error rate) mean-size (standard deviation
of size) average-utime (standard deviation of utime) average-stime (standard
deviation of stime)
```

for example:

```
26.111 (5.500) 0.000 (0.000) 0.013 (0.005) 0.010 (0.008)
```

The times are very accurate on UNIX. Under CYGNUS (windows) utime will be slightly overestimated and stime will be zero.

Arguments

- -f <filestem>
  - Set the name of the dataset (default DF)
- -source <dir>
  - Set the source data directory (default '.')
- -c <command>
  - Set the learner command. The name of the dataset will be appended to the end of this string and used to invoke the learner (**This is a required argument**)
- -folds <n>
  - Sets the number of train/test sets to create (default 10)
- -seed <n>
  - Sets the random seed, multiple runs with the same seed will produce the same datasets (defaults to a random seed). If you use a random seed, the value of the randomly selected seed will be printed at the start of the run. You can later use that seed to repeat the experiment. You can also pass the same seed to **folddata** to recreate the exact test/training sets for closer inspection.

- -v
  - Can be used multiple times to increase the debugging output

Example

```
xvalidate -source datasets/mushroom -f mushroom -folds 15 -seed 100 -c
''mostcommonclass -u -f''
```

Does 15-fold cross-validation of the 'mostcommonclass' learner on the dataset called 'mushroom' in the 'datasets/mushroom' directory. The mostcommonclass learner will be invoked as: `mostcommonclass -u -f <constructed-dataset-name>` for each of the 15 constructed datasets. It will use a seeded random number generator so the exact experiment could be reproduced.

Chapter 9

VFML Page Documentation

9.1 Thanks

page VFML VFML was made possible by a gift from the Ford Motor Company.

File bindata to Chun-Hsiang Hung for doing the core development work for this tool.

File bnlearn to Matt Richardson for making substantial optimizations to the bnlearn program.

File bnlearn-engine.h to Matthew Richardson for extracting this interface from the bnlearn learner.

File cleandata to Chun-Hsiang Hung for doing the core development work for this tool.

File cvfdt to Laurie Spencer for doing the core development work for cvfdt.

File HashTable.h to Chun-Hsiang Hung for implementing the **HashTable** ADT.

File sprint to Chun-Hsiang Hung for doing the core development work for **sprint**.

9.2 Wish List

page VFML The windows distribution needs to be brought up to date.

File batchtest I think the input format for batchtest is a little brittle and it could use some improvement

File BeliefNet.h A version of this that uses DecisionTree local models instead of full CPTs. This would also need a new structure learning tool (a modification of **bnlearn**).

File beliefnetcorrupt This tool does not do anything smart with parameters when it changes the structure, and it should.

File beliefnetscore Move the comparision mode from this tool into a new tool, beliefnetcompare, and have that tool do more interesting comparisions (e.g. measure the KL-distance between the distriutions encoded in the networks).

File bindata that this tool would have more methods for selecting bin boundaries, for example to reduce entropy.

File bnlearn A version of this program that is intelligent about dealing with unobserved (or partially observed) variables.

File cvfdt Modify this learner to work with continuous attributes.

An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

File DecisionTree.h A standard in memroy decision tree induction algorithm. Maybe the best starting point would be the **decisionstump** learner.

This isn't the right place for this wish, but it would be nice to have a RuleSet structure similar to this DecisionTree structure

File HashTable.h A cleaner hash table interface (without the compare function). I think that the sprint learner is the only thing that uses this currently so there isn't much to change to fix this.

File kmeans Modify this learner to work with discrete attributes.

Move the features from **vfkm** into this learner because this learner will be much easier to modify than that one for new users.

File naivebayes Modify this learner to work with continuous attributes.

File shuffledata for a version of this tool that works on disk and does not need to load data into RAM.

File stats.h The StatTracker would track more interesting things.

File vfbn1 An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

File vfbn2 An API to this learner like the one to learning BeliefNet structure in beliefnet-engine.h

File vfdt-engine.h A method that checkpoints the learning procedure to disk so that it can be restored at a later time. I think the hard part of this would be checkpointing the ExampleGroupStats structure.

9.3 Bug List

Global BNNodeStructureEqual(BeliefNetNode bnn, BeliefNetNode otherNode)
Only returns 1 if the parents are in the same order, but the order probably shouldn't matter.

File ExampleSpec.h Under Cygnus (and windows) lex doesn't seem to do the right thing with EOF rules, so you need to put an extra return at the end of your .names files.

Index