The DELVE Manual

C. E. Rasmussen, R. M. Neal, G. E. Hinton, D. van Camp,
M. Revow, Z. Ghahramani, R. Kustra, and R. Tibshirani

Version 1.1
December 1996

This manual describes the preliminary release of the DELVE environment. Some features described here have not yet implemented, as noted. Support for regression tasks is presently somewhat more developed than that for classification tasks.

We recommend that you exercise caution when using this version of DELVE for real work, as it is possible that bugs remain in the software. We hope that you will send us reports of any problems you encounter, as well as any other comments you may have on the software or manual, at the e-mail address below. Please mention the version number of the manual and/or the software with any comments you send.

For the latest DELVE news, visit http://www.cs.utoronto.ca/~delve/

Send comments to delve@cs.utoronto.ca

This project was supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Institute for Robotics and Intelligent Systems.

If you publish results obtained using DELVE, please cite this manual, and mention the version number of the software that you used.

CONTENTS

1 INTRODUCTION

DELVE — Data for Evaluating Learning in Valid Experiments — is a collection of datasets from many sources, an environment within which this data can be used to assess the performance of methods for learning relationships from data, and a repository for the results of such assessments.

Many methods for learning relationships from empirical data have been developed by researchers in statistics, pattern recognition, artificial intelligence, neural networks, and other fields. Methods in common use include simple linear models, nearest neighbor methods, decision trees, multilayer perceptron networks, and many others of varying degrees of complexity. Properly comparing the performance of these learning methods in realistic contexts is a surprisingly difficult task, requiring both an extensive collection of real-world data, and a carefully-designed scheme for performing experiments.

The aim of DELVE is to help researchers and potential users to assess learning methods in a way that is relevant to real-world problems and that allows for statistically-valid comparisons of different methods. Improved assessments will make it easier to determine which methods work best for various applications, and will promote the development of better learning methods by allowing researchers to easily determine how the performance of a new method compares to that of existing methods.

This manual describes the DELVE environment in detail. First, however, we provide an overview of DELVE's capabilities, describe briefly how DELVE organizes datasets, methods, and learning tasks, and give an example of how DELVE can be used to assess the performance of a learning method.

1.1 What DELVE can do for you

DELVE can help you assess the performance of learning methods in three major ways:

1. The DELVE archive contains a collection of many datasets that are appropriate for developing and assessing learning methods.

2. The DELVE software helps you use this data to assess learning methods. DELVE also provides guidelines on how such assessments should be done.

3. The DELVE archive also records the results of assessing many other learning methods on the same datasets, performed in the same way, along with detailed descriptions of these methods.

All these components of the DELVE environment are freely available via the Web, at URL

`http://www.cs.utoronto.ca/~delve/`

If you are interested in using a learning method for a particular application, you may find DELVE useful in determining which methods might be appropriate for you to use. To do this, you would look at the results of various learning methods on problems that appear similar to your application. For those methods that seem promising, you could refer to the detailed descriptions recorded in the DELVE archive.

If you are a researcher developing new learning methods, you will no doubt wish to know how well the methods you develop compare in performance to existing methods. DELVE can help you answer this question by providing a large number of datasets to test on, by providing standard conventions for conducting experiments that facilitate comparisons, by providing the results of other methods on the same datasets, and by performing appropriate tests of the statistical significance of the observed differences in performance.

After using DELVE to assess a novel learning method, you can submit your results for inclusion in the DELVE archive. You should provide a detailed description of your method, and include results of applying your method to a selection of datasets. In this way other researchers and users of learning methods will be able to benefit from your work.

1.2 The DELVE hierarchy of data, methods, and results

DELVE organizes data, learning methods, and experimental results in a hierarchical fashion. This section provides an informal description of this hierarchy, sufficient for you to follow the example in the next section.

The DELVE hierarchy is contained within one or more top-level directories, each of which has a name starting with the five letters "`delve`". All such `delve` directories contain two sub-directories, corresponding to the two main divisions of the DELVE hierarchy. The `data` sub-directory contains information on datasets, and on learning tasks defined for these datasets. The `methods` sub-directory contains information on learning methods, and on the results of applying these methods to various learning tasks.

By using more than one top-level `delve` directory, you can keep datasets and results that come from the DELVE archive separate from datasets and results that you are working on yourself, but have not yet submitted to the archive. Some research groups may also find it convenient to maintain a group `delve` directory, in addition to the private `delve` directories of the group members.

When you use DELVE, you will see information on data and methods from all such `delve` directories that are currently active, merged into a single hierarchy. In the rest of this section, we will for simplicity describe this hierarchy as if was contained in a single directory.

The data part of the hierarchy begins with a number of *datasets*, each of which has its own sub-directory within the `data` directory. A dataset is a list of cases, with each case consisting of values for a number of attributes. Some additional information is also specified at the dataset level, such as names and ranges for attributes.

Prototasks are the next lower level in the data hierarchy. A prototask defines which cases in the dataset are relevant to the learning task, which attributes of a case we wish to predict (the target attributes), and which attributes we wish to predict the targets from (the input attributes). There may be several different prototasks for a dataset, each of which has a sub-directory within the dataset's directory.

At the *task* level the size of training set for use in learning is specified, along with whatever prior information is available (which can be used, for example, to select encodings for the attributes). The task level specifies enough information that a learning method will have a well-defined expected performance with respect to any particular loss function. A *task instance* is a particular training set and test set for a task, to which we can actually apply a learning method. The performance of a method on several task-instances is used to estimate its expected performance on the task.

Tasks do not have directories of their own in the data part of the DELVE hierarchy. However, the results of applying a particular method to a particular task are contained in a directory in the methods part of the DELVE hierarchy.

The methods part begins at the `methods` directory of a top-level DELVE directory. Within this directory are sub-directories for the various learning methods that have been assessed, each of which will contain a description of the method, and perhaps the programs implementing it. The directory for a method will also contain sub-directories for every dataset that has been used in assessing the method, within which will be sub-directories for each prototask to which the method has been applied. Inside the directories corresponding to prototasks will be task directories, containing the results of applying the method to the various task instances.

A DELVE hierarchy is illustrated in Figure 1.1. The top-level `delve` directory could reside anywhere on your file system, but it's name must start with `delve`, and it must contain two sub-directories called `data` and `methods`.

In Figure 1.1, the data part of the hierarchy contains two datasets, `demo` and `kin-8nh`, each with its own sub-directory within the `data` directory. Inside each dataset directory there are two files: `Dataset.data` which contains the cases, and `Dataset.spec` which contains information about the data. There is also a sub-directory called `Source`, which contains all the data and information used to build the dataset, as it was originally obtained.

The `demo` dataset in Figure 1.1 has two prototasks, `age` and `income`, which differ in the attribute that is to be predicted. These prototask directories contain the files needed to specify both the prototask itself and the tasks that are defined for it.

Back at the top level, the `methods` directory in Figure 1.1 contains sub-directories for two methods: `lin-1` (a linear regression model), and `knn-cv-1` (a k-nearest-neighbor method). The descriptions and program source for these methods are contained in their `Source` directories. The results of applying the methods to various tasks are contained in directories whose names combine the name of the prior for the task — `std`, for "standard", in these

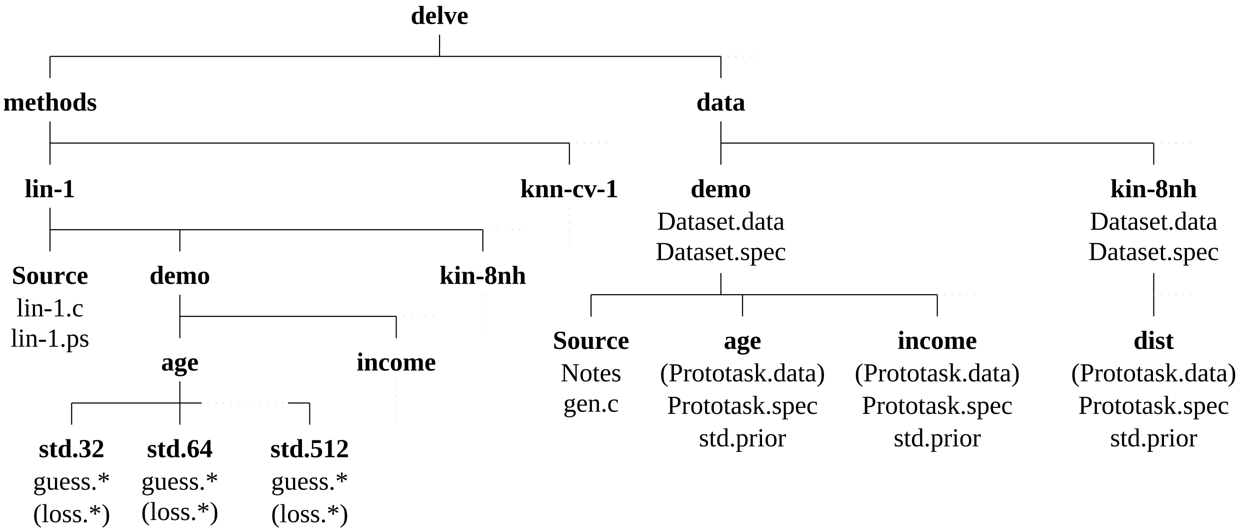

Figure 1.1: Schematic diagram of the structure of a DELVE directory. Names of directories are in bold font, names of files are in normal font. Files in brackets may not be present, as they can be generated by DELVE if needed. A dotted line indicates parts of the directory that are left out.

examples — and the size of the training set. These task directories may contain files with names such as `guess.*`, `prob.*`, and `loss.*` files, which record the final results of learning and prediction (here, "`*`" indicates several possible endings, which specify a particular task-instance and loss function).

1.3 Using DELVE: A tutorial example

This section is a walk-through tutorial, which introduces DELVE by showing how you can test a simple learning method using the DELVE utilities.

This tutorial assumes that you (or someone else) have installed DELVE on your computer. For details on how to do this, see Appendix A. It also assumes that the DELVE utilities are somewhere in your shell's search path. This will likely be true if DELVE has been installed in its usual place in `/usr/local/bin`, but if it has been installed elsewhere, you may have to set your `PATH` environment variable appropriately.

The roles of the DELVE commands used in this tutorial are described in more detail in later chapters. For detailed descriptions of command syntax and options, refer to Appendix C.

Telling DELVE where to look for information — setting your `DELVE_PATH`

First you need to know how DELVE looks for information in one or more active `delve` directories. Formally, a `delve` directory must have a name starting with the five letters "`delve`" and *must* have two sub-directories called `data` and `methods`. You tell DELVE where to look for these directories by setting the `DELVE_PATH` shell environment variable. This path works analogously to the shell search path, except that DELVE looks in all the directories in `DELVE_PATH` rather than stopping as soon as the first match is found. It therefore makes little difference what order the directories in `DELVE_PATH` come in.

You can create your own `delve` directory, and tell DELVE to use both it and a `delve` directory that holds data, methods, and results from the DELVE archive. Assuming that DELVE has been installed on your machine in directory `/usr/local/lib/delve`, you might do this as follows, if you use a shell program like `csh`:

```
unix> cd $HOME
unix> mkdir delve delve/data delve/methods
unix> setenv DELVE_PATH /usr/local/lib/delve:$HOME/delve
```

If you use a shell like `sh`, you would instead say:

```
unix> cd $HOME
unix> mkdir delve delve/data delve/methods
unix> DELVE_PATH=/usr/local/lib/delve:$HOME/delve
unix> export DELVE_PATH
```

In either case, you would probably want to put the commands that set `DELVE_PATH` in your shell start-up file (either `.cshrc` or `.profile`), so that `DELVE_PATH` will be set again when you next log in.

Setting up your `DELVE_PATH` in this way lets you keep the material distributed with DELVE separate from the results of your own experiments. You could also have several `delve` directories of your own, or include other users' `delve` directories in your `DELVE_PATH` in order to access their results. Also, whenever your current directory is within a valid `delve` directory, that directory will be temporarily added to the list of active `delve` directories, in addition to those in `DELVE_PATH`. This lets you easily look in a `delve` directory that you don't usually access.

Listing information — `dls` and `mls`, `dinfo` and `minfo`, `dmore` and `mmore`

Once you have set your `DELVE_PATH` to a list of `delve` directories, you can use various DELVE commands to look at information in the DELVE hierarchy that is contained in these directories. These commands come in two flavours — "`d`" commands that look in the `data` part of the hierarchy, and "`m`" commands that look in the `methods` part. You can, for instance, find out what files are in the directory for a particular dataset using `dls`, or get a formatted display of various information about a dataset using `dinfo`.

You can specify what you want to look at with these commands in two ways. One way is to give a *dpath* or *mpath* that specifies the location of a file or directory in the `data` or `methods` part of the hierarchy. Such dpaths and mpaths start with "/", and are translated by DELVE into one or more Unix path names within the active `delve` directories. The other way is to specify a relative Unix path name (which doesn't start with "/" or "~") of a file or directory in the DELVE hierarchy.

The `dls` and `mls` commands are analogous to the Unix `ls` command. They let you look at what files and sub-directories exist in the `data` and `methods` parts of the DELVE hierarchy. For example, we can use the `dls` command with a dpath of "/" to list the datasets found in all the active `delve` directories:

```
unix> dls /
demo kin-8nh kin-8nm
```

There might be many more than these three datasets, of course, depending on what you have installed, and on how your `DELVE_PATH` is set. Note that the three datasets shown would not necessarily be located in the same Unix directory.

We can list the files and sub-directories for the `demo` dataset as follows:

```
unix> dls /demo
Dataset.data Source       age          income       siblings
Dataset.spec Summary      colour       sex
```

Once again, these files and sub-directories might not all be in the same Unix directory, though in this case, these files and sub-directories are in fact all present in the directory for `demo` from the DELVE archive. To see exactly where things exist, you can use the `-l` option. (By the way, you can find out about options for any DELVE command by executing the command with the `-h` option.)

Here we use `-l` with the corresponding `mls` command, to see what methods are available, in what places:

```
unix> mls -l /
/usr/local/delve/methods/:
knn-cv-1 lin-1
```

We see that there are just two methods, `knn-cv-1` and `lin-1`, and that files for both methods are found only in the directory holding information from the DELVE archive. If we had tested one of these methods on new datasets of our own, however, results for that method could exist in our private `delve` directory as well, in which case directories for the method would exist in both places.

Two DELVE commands similar to the Unix `more` command also exist, called `dmore` and `mmore`. Here we use `dmore` to look at the summary description for the `demo` dataset:

```
unix> dmore /demo/Summary
The "demo" dataset was invented to provide an example for the DELVE
manual, and to test the DELVE software and software that implements
learning methods.  To those ends, it has a variety of numerical and
categorical attributes.  Cases for the "demo" dataset were artificially
generated from a distribution based on simple demographic assumptions
and various stereotypical notions concerning the relationships between
people's sex, age, number of siblings, income, and favourite colour.
Prototasks are defined for predicting each of these attributes given
the others.
```

We could also use `dmore` to look at the specification file for a dataset, but we would not usually do so, since DELVE provides a command `dinfo` for conveniently displaying this and other information about datasets. We can ask about information for the `demo` dataset as follows:

```
unix> dinfo /demo
Dataset: /demo
Origin: artificial
Usage: development
Order: uninformative
Number of attributes: 5
Prototasks:
        age
        colour
        income
        sex
        siblings
```

We would see more details if we used the `-a` option (ie, the command '`dinfo -a /demo`'). Similarly, we can ask for information about the `age` prototask from the `demo` dataset with the command "`dinfo /demo/age`", and so on. There is a corresponding `minfo` command for getting information on learning methods, and on their application to learning tasks.

Applying your learning method to a task — `mgendir` and `mgendata`

Now that you have seen how to obtain information about datasets and methods in DELVE, we will see how we can go about testing a simple learning method, which we will call `mymethod`. First, we need to create a directory for the method, with a structure of sub-directories similar to that for the `lin-1` method depicted in Figure 1.1. These sub-directories will hold the results of applying the method to the `demo/age` prototask. We could create all these sub-directories using the Unix `mkdir` command, but it is more convenient to use the DELVE `mgendir` command:

```
unix> cd delve/methods
unix> mkdir mymethod
unix> cd mymethod
unix> mgendir demo/age
demo
demo/age
demo/age/std.32
demo/age/std.64
demo/age/std.128
demo/age/std.256
demo/age/std.512
```

Now that the `mgendir` command has created the appropriate directories, we can proceed to put files containing training and test data into the sub-directory for one of the tasks (with the standard prior, and 256 training cases) using the `mgendata` command:

```
unix> cd demo/age/std.256
unix> mgendata
  segmenting cases...
  splitting test inputs and targets...
  encoding instance 0 training data...
  encoding instance 0 test inputs...
  encoding instance 0 test targets...
  encoding instance 1 training data...
  encoding instance 1 test inputs...
  encoding instance 1 test targets...
  encoding instance 2 training data...
  encoding instance 2 test inputs...
  encoding instance 2 test targets...
  encoding instance 3 training data...
  encoding instance 3 test inputs...
  encoding instance 3 test targets...
```

This command creates files in the current directory pertaining to four tasks instances. The `train.`n files contain the inputs and targets for the training cases in instance n, the `test.`n files the inputs for the test cases, the `targets.`n files the true targets for the test cases, and the `normalize.`n files the normalization constants used in encoding the data. Files called `Coding-used` and `Test-set-stats` are also created; they hold information used by later commands.

You can get information about the way this method is being applied to this task using the `minfo` command. When called with no arguments, this command will give information about the method and task associated with the current directory, as illustrated below:

```
unix> minfo
Task: /demo/age/std.256
Training set size: 256
Inputs:
 col attr name          type    relevance  coding   options
   1    1  SEX           binary   nlmh      -1/+1         -
   2    3  SIBLINGS      integer  nlmh      nm-abs        -
   3    4  INCOME        real     nlmh      nm-abs        -
   4    5  COLOUR:pink   nominal  nlmh      1-of-n        -
   5    5  COLOUR:blue                   ...
   6    5  COLOUR:red                    ...
   7    5  COLOUR:green                  ...
   8    5  COLOUR:purple                 ...
Targets:
 col attr name          type    relevance  coding   options
   1    2  AGE           real     nlmh      nm-abs        -
```

This shows things such as the way the various attributes have been encoded in the data files to be used by the method (in this case, the default encodings were used). Similar information would be displayed by the `dinfo` command, but `minfo` will show any information specific to how this learning method is being applied to this task, whereas `dinfo` shows only information about a dataset itself, and its associated tasks.

Each of the `train.`n files that were created above contains one line for each training case. With the default encoding used above, the four input attributes are encoded as eight numbers, which appear at the beginning of the line. (The `COLOUR` attribute is encoded in `1-of-n` form, which uses five numbers to represent which of its five possible values the attribute has.) The target is encoded as a ninth number, at the end of the line.

You can now train your model using the data in each of the four `train.`n files. This is to be done separately for each file, as the four training files are for four instances of the task, which are to be handled completely independently of each other. You then use the results of this training to make guesses for the targets in the test cases that go with each task instance, given the inputs for these cases in the `test.`n files. Your method should write its guesses in the files `cguess.S.0` though `cguess.S.3`, one guess per line. Here, the prefix '`c`' indicates that the guess are for the coded form of the attribute, not the original form in which it appears in the dataset file. The suffix `S` indicates that the guesses are designed for use with the squared error loss function.

For this tutorial, we don't want to get into the complexities of writing a realistic learning method, so we'll use as an example a method that simply predicts the constant zero for every test case. Notice that, as seen in the output from `minfo` above, the (default) encoding of the targets used here is `nm-abs`, which means that they are shifted and re-scaled so that the median of the target values in the training cases is zero, and the average absolute deviation from the median in the training cases is one. Because of this, always predicting zero, while not very sophisticated, is at least not wholely unreasonable. This method can be implemented by the following `awk` commands:

```
unix> awk ' { print "0.0" } ' test.0 > cguess.S.0
unix> awk ' { print "0.0" } ' test.1 > cguess.S.1
unix> awk ' { print "0.0" } ' test.2 > cguess.S.2
unix> awk ' { print "0.0" } ' test.3 > cguess.S.3
```

Notice that the training data is ignored here (though it is implicitly used through the use of a normalized encoding), and the test data is looked at only in order to determine how many test cases there are. However, this is certainly not typical behaviour for a learning method!

How well did it do? — `mloss` and `mstats`

Once our method has produced `cguess.`n files containing its guesses for targets, we can use the `mloss` command to evaluate the "loss" suffered when using each of these guesses. The loss is based on the difference between the guess and the actual target value. To find the losses as judged by the squared difference between guess and target value, we would use `mloss` with the '`-l S`' option:

```
unix> mloss -l S
  decoding cguess.S.0...
  decoding targets.0...
  creating loss.S.0...
  decoding cguess.S.1...
  decoding targets.1...
  creating loss.S.1...
  decoding cguess.S.2...
  decoding targets.2...
  creating loss.S.2...
  decoding cguess.S.3...
  decoding targets.3...
  creating loss.S.3...
```

The `mloss` command transforms the guesses in the `cguess.S.`n files back to the original domain, storing these transformed guesses in the files `guess.S.`n. It then computes the loss for each test case and writes these losses to the `loss.S.`n files.

We can now use the `mstats` command to get a summary of the predictive performance of our method. Here, we give the '`-l S`' option to `mstats` to say we are only interested in the squared error loss function:

```
unix> mstats -l S
/mymethod/demo/age/std.256
Loss: S (Squared error)
                                                     Raw value   Standardized

                         Estimated expected loss:     520.43       1.06461
                     Standard error for estimate:       41.7     0.0853028

     SD from training sets & stochastic training:    49.1004      0.100441
SD from test cases & stoch. pred. & interactions:    1078.63       2.20648

    Based on 4 disjoint training sets, each containing 256 cases and
             4 disjoint test sets, each containing 256 cases.
```

The first line of this summary gives an estimate for the expected loss when using this method on this task; the next line gives a standard error for this estimate. The lines below these give the standard deviations for the variation in performance due to various causes. For a more detailed discussion of these statistics refer to Section 8. The second column gives the same quantities rescaled to a standardized domain, which makes interpretation easier. In the case of squared error, the losses are standardized by dividing by the sample variance of the targets in all the test cases.

The `mstats` command can also be used to compare the performance of different learning methods. In the `methods` part of the DELVE hierarchy are descriptions and results for a selection of learning methods on some of the DELVE tasks. If you have obtained the results of the linear regression method called `lin-1` from the DELVE archive, you will be able to compare your method to the `lin-1` method as follows (again, with respect to squared error loss):

```
unix> mstats -l S -c lin-1
/mymethod/demo/age/std.256
Loss: S (Squared error)
                                                     Raw value   Standardized

             Estimated expected loss for mymethod:    520.43       1.06461
               Estimated expected loss for /lin-1:    397.82      0.813792
                    Estimated expected difference:    122.61      0.250815
           Standard error for difference estimate:   26.9735     0.0551778

     SD from training sets & stochastic training:    44.5182     0.0910678
SD from test cases & stoch. pred. & interactions:     487.52      0.997285

    Significance of difference (t-test), p = 0.0199425

    Based on 4 disjoint training sets, each containing 256 cases and
             4 disjoint test sets, each containing 256 cases.
```

This shows that the linear method has a smaller expected loss that our more trivial method. Notice that the expected difference between the methods is approximately 4 times greater than the standard error on this estimate. The p-value from the t-test indicates that the difference should be considered significant at the 2% level. The methods used to compute such p-values are described in Section 8.

It could happen that when you tried to compare our method with `lin-1`, as shown above, `mstats` could fail to find `loss` files for the linear method in any of the active `delve` directories. If this were to happen, you could generate the `loss` files needed by `mstats` yourself (assuming that the `guess` for `lin-1` were available). However, you would probably need to generate these files in your own DELVE directory, since you likely don't have permission to write in the directory that holds information from the DELVE archive. To achieve this, you could do the following:

```
unix> cd $HOME/delve/methods
unix> mkdir lin-1
unix> cd lin-1
unix> mgendir demo/age
unix> cd demo/age/std.256
unix> mloss
```

The `mstats` command will now be able to use these `mloss` files, as long as the `delve` directory they are stored within is mentioned in your `DELVE_PATH`, or you are currently inside this `delve` directory.

In similar fashion, you put things in your own `delve` directory that extend what is in the the DELVE archive by adding new datasets, new prototasks for old datasets, new methods and results for new methods, and new results for old methods. However, to avoid confusion, DELVE will not allow you to use names for new things that are the same as the names for things that already exist in the DELVE archive directory.

1.4 What to read next

Section 2 contains a more detailed specification of the scope and aims of the DELVE project; this section may be of general interest. Sections 3 and 4 contain detailed descriptions of how datasets, prototasks, and tasks are specified in DELVE. These sections may be of some interest to all users, but are primarily intended for people who wish to include new datasets in DELVE, or who wish to create new prototasks and tasks based on existing datasets. Section 5 describes the standard loss functions supported by DELVE, and discusses how other loss functions can be incorporated. Section 6 discusses the schemes for learning experiments used in DELVE, and compares these to more traditional schemes such as cross-validation.

Users who want to get straight into using DELVE to test their learning methods may wish to just skim these initial sections, and start serious reading with Sections 7 and 8, which describe the methodology for DELVE assessments, and the DELVE commands required to perform them.

Appendix A tells you how to get software, data, and results from the DELVE archive, while Appendix B tells you how to contribute things to the DELVE archive. Detailed descriptions of DELVE commands are found in Appendix C, and a glossary of DELVE terminology is found in Appendix D.

2 THE SCOPE OF THE DELVE PROJECT

The aim of the DELVE project is to promote the development and use of empirical learning methods by providing a well-designed environment in which the performance of such learning methods can be assessed on data that is relevant to the real world. This is a broad objective, which we can hope only to partially fulfill. This section outlines the scope of the DELVE project at present — the sorts of learning methods that DELVE can handle, the sorts of assessments that DELVE supports for these methods, and the kinds of dataset on which these assessments are performed.

As researchers ourselves, we of course have ideas about which learning methods are most promising, but we have tried to keep such prejudices from affecting the design of DELVE. We have also tried to minimize the extent to which DELVE constrains the sorts of questions that researchers can investigate. Inevitably, however, we have had to use our own judgement in making tradeoffs between different design goals, some of which are mentioned below.

2.1 Learning methods that DELVE can handle

At present, DELVE supports only methods for *supervised learning* — that is, methods that aim to predict one or more *target attributes* using the information provided by some set of *input attributes*. The relationship between the inputs and the targets is learned from a number of *training cases*, in which both the inputs and targets are known. These training cases are modeled as if they were generated more-or-less independently from some source. The goal of learning is to predict the target in a *test case*, generated from the same source as the training cases, but for which only the inputs are known. For some datasets, the cases are not truly independent, but the primary goal is always to learn the relationship of targets to inputs, not to learn the nature of any dependencies between cases.

We distinguish between *regression* tasks, in which the targets (usually one, but sometimes more) are real-valued, and *classification* tasks, in which there is a single target, the *class* of the item in question, which takes on values from a small set. We also provide some limited support for other supervised learning tasks, such as those in which the target is an integer, or an angular value.

The DELVE facilities presently treat the attributes in a case as an unstructured collection of values. In some applications, such as image processing, the attributes (eg, pixel values) are known to have certain relationships to each other (eg, spatial adjacency), which can be of great help in learning. Although data from such application areas could be included in DELVE, assessments using this data may be of limited interest, since DELVE provides no scheme for informing learning methods about such structure in the data.

In future, we hope to also support *unsupervised learning* methods and related statistical methods such as density estimation, in which attributes are not characterized as inputs or targets. As well, we may someday add facilities for assessing *time series* methods, in which the aim is to characterize the sequential dependencies between cases.

2.2 Aspects of performance that can be assessed using DELVE

DELVE is aimed primarily at assessing the *predictive performance* of learning methods — that is, their ability to make predictions in previously unseen cases by generalizing from the information contained in the data used for training. *Computational performance* — the amount of time and space needed for training and subsequent use of the methods — is also of concern. There will often be a tradeoff between predictive performance and computational performance. However, DELVE does not include any datasets where computational considerations appear paramount, as might be the case, for example, when the amount of data is extremely large.

Other characteristics of learning methods are also of interest, such as ease of use by both expert and inexpert users, and the degree to which the results of learning can be interpreted, but DELVE does not support any formal evaluation of such characteristics.

2.3 How DELVE encourages meaningful assessments

The DELVE environment is designed to encourage and assist users to produce meaningful assessments that are *faithful*, *comparable*, and *reproducible*.

To be *faithful*, an assessment of a learning method must be indicative of how well it would perform on an actual task that is of some interest. One must, for example, avoid any inadvertent "cheating", such as would occur if parameters of the learning method were set on the basis of performance on the test cases. Arbitrary restrictions on how learning methods may be used must also be avoided, if better performance might be obtained in a real application by doing things differently.

For assessments of different learning methods to be *comparable*, they must all have been applied in the same context — for instance, with training sets of the same size, and with equivalent attention being paid to prior information. It is perhaps in this respect that a standard environment such as DELVE is most useful.

One requirement for an assessment to be *reproducible* is that the method used be adequately documented. To encourage this, we have provided guidelines for proper documentation, and examples of their use. Reproducibility is most easily achieved if the method is fully automatic. This is not always possible, however, so we suggest ways of improving the reproducibility of methods that involve human decisions.

Furthermore, DELVE is designed to provide assessments that are as *accurate* as is practical, and for which the degree of accuracy is known. DELVE also supports comparisons of learning methods that provide indications of the statistical significance of any observed differences. The power of these comparisons is increased by using the same training and test sets for different methods, which is another advantage of a standard environment.

2.4 Kinds of datasets included in DELVE

Obtaining data is one of the most crucial, and most difficult, parts of building an assessment environment. We have drawn datasets for DELVE from four sources, each of which has its advantages.

Natural datasets come from real-world sources, and were at one time used to address questions of real interest that are similar to those addressed by the supervised learning methods we would like to assess. *Cultivated* datasets also come from the real world, but do not represent real supervised learning problems. Such cultivated data was instead gathered or selected specifically for the purpose of assessing learning methods. We also include real-world datasets that have been altered (eg, by adding noise) in this category.

Simulated datasets are generated by a computer simulation of a real-world phenomenon. To qualify for this category, the simulation should be reasonably realistic, and of a complexity that makes it difficult to see what form the relationships in the data will take. *Artificial* datasets are randomly generated from a distribution defined by a relatively simple mathematical formula.

Natural datasets have the advantage of being arguably representative of the problems we are actually interested in. For example, a statistical consultant might reasonably conclude that it would be worthwhile to learn more about a learning method that has been found to perform better than others on such real-world problems. Relevance to the real world is more doubtful for cultivated, simulated, and artificial datasets. As the datasets become less natural, it also becomes more likely that a researcher may bias the assessment of a learning method by unconsciously selecting problems on which that method can be anticipated to do well.

Why, then, do we include any other than natural datasets? One reason is that the number of readily-available natural datasets is limited, and those that are available are usually not as large as we would like. In the real world, the cost of collecting data is often high, and we must try to obtain the most information possible from a small dataset. To properly assess the performance of a learning method in such a context, however, we need much more data, in order to reduce the uncertainty in our estimate of expected performance. Simulated and artificial datasets can easily be made as large as required (limited only by storage space); this can greatly improve the accuracy of performance estimates.

Another reason for using non-natural datasets is that they can be designed to address certain questions that would otherwise be difficult to answer, such as what the effect is of adding extra noise to the input attributes, or of adding extra irrelevant inputs. In particular, we can design families of tasks that are related in interesting ways — eg, that have more or less noise, or a larger or fewer number of input attributes — and see how these dimensions of variation affect the performance of various learning methods.

When we began collecting datasets for use in assessing supervised learning methods, we had hoped to confine ourselves to datasets where the cases were truly independent, as independence of cases is an assumption behind many existing supervised learning methods. We found, however, that in many otherwise-interesting datasets, there is at least a possibility of dependencies between cases. We therefore decided to include such datasets, both in order to increase the variety of datasets available, and because it seems to us that the possibility of such dependencies is a common feature of real-world problems, which designers of supervised learning methods may be well-advised to accommodate. We have, however, avoided datasets in which the dependencies themselves are the primary focus of interest.

3 DATASET FILES AND SPECIFICATIONS

A dataset is a collection of *cases*. For each case, the values of certain *attributes* are recorded. DELVE stores these attribute values in a file with a standard format that is general enough that a wide variety of datasets can be represented without loss of information. For each dataset, DELVE also keeps a specification file, which records basic information such as the number of attributes and their theoretical ranges. Finally, the original files or programs from which the dataset was derived are retained in the DELVE archive, along with any original documentation.

Files relating to a dataset are kept in a directory with the same name as the dataset, located in the `data` sub-directory of a top-level `delve` directory. Some of the files that may appear in such a dataset directory are listed in Figure 3.1.

3.1 Specifications for datasets: The `dinfo` command

The specifications for a dataset include information about the dataset as a whole, such as its origin and usage within DELVE, plus information about each attribute in the dataset, such as its range of legal values. This information is stored in the dataset's specification file, `Dataset.spec`. However, the only time you will need to directly access this specification file is when you create a new dataset, using the procedure described in Section 3.4.

Usually, it is more convenient to view the specifications for a dataset using the `dinfo` command, as was illustrated in the tutorial in Section 1.3. For instance, to see the specifications (as well as some other information) for the `demo` dataset, you would use the command

```
dinfo /demo
```

Further details on individual attributes of the dataset can be obtained by using the `-a` option

`Summary`	A brief description of the dataset
`Dataset.data`	The actual data, in the format described in Section 3.3
`Dataset.spec`	Specifications for the dataset, usually accessed using the `dinfo` command
`Source`	A sub-directory with files relating to the source of the dataset, such as:
`Notes`	Documentation on the dataset
`original`	The original data file (but sometimes there will be more than one)
`gen.c`	C program for generating dataset (or `gen.f` for a Fortran program, etc.)
Prototask-1 *Prototask-2* *Prototask-3*	Sub-directories for prototasks based on the dataset (see Section 4)

Figure 3.1: Some files and sub-directories that may appear within a DELVE dataset directory.

with `dinfo`, as is illustrated in Figure 3.2.

Note that dataset specifications contain only very basic information, which is not likely to be wrong unless the data has been totally misinterpreted. More debatable prior information may be specified as part of a task description (Section 4.3).

The following characteristics of a dataset as a whole are recorded as part of its specification, and displayed by `dinfo`:

`Origin:` `natural` *or* `cultivated` *or* `simulated` *or* `artificial`

A *natural* dataset was originally gathered for some real-world application; a *cultivated* dataset comes from a real-world source, but was never used to solve a real problem; a *simulated* dataset was generated by a simulator, but is believed to resemble real data — as opposed to an *artificial* which is generated according to some mathematical formula and does not pretend to resemble any real dataset. These distinctions are discussed further in Section 2.4.

`Usage:` `development` *or* `assessment` *or* `historical` *or* `?`

A *development* dataset is recommended for use in developing new learning methods, but to avoid bias, should not be used for formal assessments. An *assessment* dataset is intended for use in formal assessments; use for development should be minimized. A *historical* dataset is included in DELVE because it has been used for assessing learning methods in the past, but is not recommended for general use. A '?' indicates that a recommended usage has not yet been decided on.

`Order:` `informative` *or* `uninformative` *or* `?`

A dataset has an *informative* ordering if the order of cases may convey information that is not already present in the attribute values. The order is recorded as *uninformative* if it is random, or has some basis that is not related to any matter of interest. The order is recorded as '?' if the order appears to be non-arbitrary, but the basis of the ordering cannot be determined from the available documentation.

`Commonality indexes are present`

If this line is displayed by `dinfo`, *commonality indexes* are associated with some or all cases in the dataset. Cases with the same commonality index share something in common, as is described further in Section 3.2. If this line is not displayed, the cases in the dataset do not have commonality indexes.

If the ordering of a dataset is informative, or if commonality indexes are present, the issue of possible dependencies between cases must be addressed, as is discussed in Section 3.2.

Each dataset has a specified number of attributes associated with each case. Datasets in which the number of attributes varies from case to case are not handled by DELVE, though it

```
Dataset: /demo
Origin: artificial
Usage: development
Order: uninformative
Number of attributes: 5
Attributes:
     #  name     c/u range        description
     1  SEX       u  male female  Sex of the person
     2  AGE       u  [0,Inf)      Age of the person in years
     3  SIBLINGS  u  0..Inf       Number of siblings the person has
     4  INCOME    u  [0,Inf)      The person's annual income (dollars)
     5  COLOUR    u  pink blue red green purple
                                  The person's favourite colour
Prototasks:
        age
        colour
        income
        sex
        siblings
```

Figure 3.2: Output of the command: `dinfo -a /demo`.

is possible for the values of some attributes to be missing in some cases (see Section 3.3). The attributes for a dataset are numbered from 1 on up. Attributes can also have short *names*, which can be used in place of numbers to identify them. For the `demo` dataset illustrated in Figure 3.2, the attributes have names of `SEX`, `AGE`, etc.

The dataset specification also records whether each attribute was *controlled* or *uncontrolled* (abbreviated to '`c`' or '`u`' in the output of `dinfo`). The values of a controlled attribute were fixed for each case by the investigator who gathered the data; the values of an uncontrolled attribute were not fixed, though the investigator will often have had some influence on the mechanism by which they were generated. For example, in a dataset concerning the growth of plants under various conditions, the amount of fertilizer applied to a plant would usually be a controlled attribute, whereas the amount of rainfall would be an uncontrolled attribute. This field will be recorded as '?' if it is not clear from the available documentation whether or not the attribute was controlled.

Each attribute in the dataset also has a specified *range*, consisting of a list of items, each of which defines a set of allowed values for the attribute. Such an item can specify a single permitted value (which could be a *missing value*, as discussed in Section 3.3), or a set of permitted numerical values having the form of an open, closed, or half-open interval of real numbers, or a range of integers. The bounds of a real interval can be ordinary numbers, or one of '`Inf`', '`-Inf`', or '`+Inf`', with '`Inf`' representing infinity; these bounds are enclosed by round or square brackets, indicating whether the bound itself is included. For example, `[0,1)` represents the interval from 0 to 1, including 0, but not including 1, and `(0,Inf)` represents the set of positive real numbers. An integer range extending from *low* to *high*, inclusive, is written as *low*..*high* (with no enclosing brackets); *low* and *high* can be infinite, as for real intervals. For example, `1..Inf` represents the positive integers.

Several items can be combined, as in the following range:

```
(-Inf,0)  (0,+Inf)  ?
```

This specifies that the attribute can take on any numerical value other than zero, as well as the missing value indicator, '`?`'.

Note that the range specified for an attribute is the full set of conceivable values, regardless of whether all of these values actually occur. For example, the range `[0,100]` would be appropriate for an attribute that represents the percent by weight of water in a sample of some substance, since it is inconceivable that the value could ever fall outside this range, but any more narrow range would not be appropriate, even if the actual values in the dataset never exceed 10%. Similarly, for an attribute representing a person's birth sign, the appropriate range would be all twelve signs of the zodiac, even if no Scorpios happen to be included in the dataset.

Finally, an attribute may be accompanied by a short *description*, which is ignored by the DELVE software, but may help users keep track of which attribute is which.

3.2 Datasets with dependencies between cases

Dependencies between cases in a dataset are of significance for two reasons. First, a learning method may take account of such dependencies in order to improve learning. For example, a method that adapts its behaviour based on the size of the training set might consider the effective size of the training set to be reduced when training cases are dependent (since the information in one case may largely duplicate the information in other cases). Second, DELVE itself must be aware of possible dependencies in order to avoid assessing learning methods using test cases that are dependent on the cases included in the training set, and in order to properly compute standard errors for performance figures.

Whenever a dataset has an informative ordering, there is the possibility of *sequential dependencies* between the cases. In some circumstances, however, this possibility may be remote enough that it is reasonable to ignore it — for example, if the cases are ordered by the time when their attributes were measured by some machine, it is possible that dependencies are present as a result of temporal variation in the machine's accuracy, but this possibility may be too remote to be worth worrying about.

Dependencies between cases may also exist whenever *commonality indexes* are present. Cases with the same commonality index have something in common of a nature that may produce dependencies. For example, suppose the problem is to classify cars by make, given an image of the car. If several cases were obtained by viewing the *same* car from different angles, the whole group of cases should be used either for training or for testing, but not for a mixture of these. Otherwise, a test case might be correctly classified based on some idiosyncratic feature of a training case in the same group (eg, a scratch on the car's bumper). Similarly, in a dataset of spoken words, all the words spoken by one person would share a commonality index.

The presence of commonality indexes or of an informative ordering is merely an indication of the possibility of dependencies, and even if dependencies exist, they may or may not be of significance in the context of a particular learning task. More specific information concerning dependencies may be given in prototask and task specifications. When significant dependencies do exist, they are dealt with in DELVE in one of two ways. One is to properly accommodate the dependencies, as would be necessary in a real-world learning task. The other is to randomly select cases so as to produce an internally-consistent task without dependencies. Such tasks can be useful for assessing learning methods even though they no longer correspond to a real-world situations. These issues are discussed further in Section 4.

Note: Currently, commonality indexes are not really implemented — you can include them in DELVE dataset files, but they will be ignored. Also, the only way of dealing with sequential dependencies at present is to randomize the ordering.

3.3 The DELVE format for dataset files

DELVE datasets are stored in a standard format that is designed to preserve as much relevant information from the original data as possible, even if some of this information is not currently used by DELVE. Users may occasionally wish to look at these dataset files, but programs implementing learning methods do not read these files directly. Instead, a learning method will work with data files that have been appropriately encoded for a given task, as described in Section 7.

A dataset in the DELVE standard format consists of an ordered list of *cases*, each of which consists of values for an ordered list of *attributes*. A case may optionally be accompanied by a *comment*, which may be anything, and by a *commonality index*, a number that identifies several cases as having a common origin. *Note: Commonality indexes aren't implemented yet.*

The number of attributes is a characteristic of the dataset, and all cases have values (of some sort) for all attributes. The value of an attribute may be any of the following:

- A string that represents a number in any of the common forms — that is, with syntax

  $$[\ \texttt{+}\ |\ \texttt{-}\ ]\ [\ \textit{digit}\ldots\ ]\ [\ \texttt{.}\ [\ \textit{digit}\ldots\ ]\ ]\ [\ (\ \texttt{e}\ |\ \texttt{E}\ )\ [\ \texttt{+}\ |\ \texttt{-}\ ]\ \textit{digit}\ldots\ ]$$

  with the restriction that at least one digit must appear, not counting digits after an '`e`' or '`E`'.

- A number as above, preceded or followed by '`:`', representing a *censored value*. If the colon is at the end, the actual value of the attribute is known only to be greater than or equal to the given number; if the colon is at the beginning, the actual value is less than or equal to the given value. *Note: Support of censored values is not yet implemented.*

- The character '?', perhaps followed by other non-space characters. This represents a *missing value*. The other characters may indicate the reason for the value being missing. Just '?' is used for values that are missing due to a random mechanism unrelated to the relationship of inputs to targets. *Note: Missing values are not really implemented yet. About the only thing useful that can be done at present with cases having missing values is to ignore them.*

- Any other string of non-space characters that does not begin with '\', '`@`', '`#`', '(', '`[`', '`+`', '`-`', '`.`', '`:`', or a digit. These strings represent values from a discrete set of categories.

Numerical values are represented in as close to their original form as possible — for example, '5.0' is *not* converted to '5' or to '5.00'. This preserves any information that might be contained in the original choice of the number of significant digits.

A dataset in standard format is encoded as a ASCII file, in which the cases appear in order, with each case being represented by a group of lines. All lines in a group except the last end with a space followed by the character '\'. The whole group of lines for a single case should be thought of in terms of the single line that would result if the '\' and the following newline were removed. Within the line (or group of lines) representing a case, the attribute values appear in order, separated by one or more spaces.

If a case has a commonality index associated with it, it appears after all the attributes. This index consists of the character '@' followed by one or more digits.

If a case has a comment associated with it, it appears at the end of the line, preceded by '`#`'. These comments are ignored by all DELVE programs.

3.4 Preparing a new dataset: The `dcheck` command

When a dataset is obtained, the original data files, documentation, programs, and any other possibly relevant material should be saved in as close to its original form as possible. This archived information may be of interest if, for example, doubts should arise as to whether the original data format was properly interpreted, or questions are raised regarding the real-world relevance of the data. This information goes in the `Source` sub-directory of the dataset's directory.

The dataset should then be converted to the standard DELVE format, and stored in the `Dataset.data` file in the dataset's directory. The aim in doing this should be to retain all information that could be relevant to some use of the data, discarding only fields such as redundant case numbers. Converting a dataset will often be simply a matter of mechanically reformatting it. However, difficulties of interpretation may arise if there are peculiar aspects to the original data, or if it is inadequately documented. In such cases, the rationale for the decisions made should be documented, in the `Notes` file in the `Source` directory for the dataset.

As well as the data file itself, you must create a specification file for the dataset, with the name `Dataset.spec`, which describes how the dataset is to be interpreted and used. The specification file is meant to be machine readable, and, as such, has a very strict format. The file may have zero or more initial comment lines (lines where the first character is a `#`). Immediately after the comments lines there should appear the three lines (in any order):

`Origin:` *origin*
`Usage:` *usage*
`Order:` *order*

These lines specify the information discussed in Section 3.1. Specifically:

origin should be one of the strings `natural`, `cultivated`, `simulated`, or `artificial`.

usage should be one of the strings `development`, `assessment`, `historical`, or ?.

order should be one of the strings `informative`, `uninformative`, or ?.

In addition to the above lines, you may include the optional line:

`Title:` *title*

where *title* is a string describing the dataset. It is not used directly by DELVE, but it is available to users via `dinfo`.

The string `Commonality indexes are present` may appear on the next line. If there are no commonality indexes, this line should be omitted. *Note: Currently, this line must be omitted. You can always include commonality indexes, but they will be ignored.*

Following these lines should be a line contain the single string `Attributes:`. Each remaining line in the file will be interpreted as an attribute description, with the format:

i name control range [`#` *comment*]

The fields above have the following meanings:

i is the integer index for the attribute. Indices should start at one and increment by one for each line.

name is a mnemonic name that can be used in place of the attribute's index. The names must be unique (within a dataset). They may not contain spaces, and may not look like integers.

control is one of the characters `c` or `u`, depending on whether the attributes was controlled or uncontrolled.

```
Origin: artificial
Usage: development
Order: uninformative
Attributes:
 1 SEX      u male female  # Sex of the person
 2 AGE      u [0,Inf)      # Age of the person in years
 3 SIBLINGS u 0..Inf       # Number of siblings the person has
 4 INCOME   u [0,Inf)      # The person's annual income (dollars)
 5 COLOUR   u pink blue red green purple  # The person's favourite colour
```

Figure 3.3: Dataset specification file for the `demo` dataset.

range is the range for the attribute, a list of items of the form described in Section 3.1.

The range for an attribute may optionally be followed by '`#`' and a comment describing the attribute.

The specification file for the `demo` dataset is shown in Figure 3.3.

Once you have created both `Dataset.data` and `Dataset.spec`, you should check that the two are legal and consistent using the `dcheck` command, which will verify that each case has the right number of attributes, and that they are in the specified ranges. Note that missing values are allowed in `Dataset.data` only if they are listed as allowed in `Dataset.spec`. A censored value for an attribute (specified using ':') is allowed only if it includes at least one possible value that is within the attribute's range. *Note: The* `dcheck` *command is not implemented yet.*

4 FROM DATASETS TO TASKS

A dataset does not, by itself, define a problem to be solved. To get a well-defined learning task, we must specify additional information, such as what part of the data we are concerned with, what we hope to predict about this data, and what contextual information is available to assist learning. In the DELVE environment, these specifications have a hierarchical form, in which specificity increases as we go from a *dataset*, to a *prototask*, to a *task*, and finally to a *task instance*.

A *prototask* fixes only the most basic aspects of the learning task — just enough so that it makes sense to compare the performance of various learning methods on the various tasks that derive from the prototask. Specifically, a prototask will define the following:

- The *subset of cases* that a learning method is expected to handle.
- The set of *target attributes* that the method is supposed to predict, and the set of *input attributes* that it may refer to when making these predictions.

A *task* is derived from a prototask by specifying the additional information required so that each learning method will have a well-defined *expected performance* on the task, with respect to some given *loss function* (see Section 5). In particular, to define a task, we must supplement the specifications for the prototask by specifying the following:

- The *number of training cases* in the training set that will be provided to the learning method, and (if applicable) whether this training set will be *stratified* by target value.
- The *prior information* that the method may use to assist the learning.

Note that expected performance is estimated using *task instances*, for which particular training cases are specified, as discussed in Sections 6 and 7.

Specifications and other information relating to a prototask and its tasks are kept in a subdirectory associated with the prototask, located within the directory for the dataset. Some of the files that may appear within such a prototask directory are listed in Figure 4.1.

`Summary`	A brief description of the prototask
`Prototask.data`	Data relevant to the prototask, a subset of that in `Dataset.data`
`Prototask.spec`	Specifications for the prototask and associated tasks, usually accessed using the `dinfo` command
`std.prior`	The "standard" prior information for the prototask
Prior-1`.prior` *Prior-2*`.prior` *Prior-3*`.prior`	Other specifications of prior information

Figure 4.1: Some files that may appear within a DELVE prototask directory.

```
Prototask: /demo/age
Origin: artificial
Cases: all
Order: retain
Test set size: 1024
Training set sizes: 32 64 128 256 512
Test set selection: hierarchical
Maximum number of instances: 8
Inputs:
     #  name     c/u range        description
     1  SEX       u  male female  Sex of the person
     3  SIBLINGS  u  0..Inf       Number of siblings the person has
     4  INCOME    u  [0,Inf)      The person's annual income (dollars)
     5  COLOUR    u  pink blue red green purple
                                  The person's favourite colour
Targets:
     #  name     c/u range        description
     2  AGE       u  [0,Inf)      Age of the person in years
Tasks:
        std.32
        std.64
        std.128
        std.256
        std.512
```

Figure 4.2: Output of the command: `dinfo /demo/age`.

4.1 Specifications for prototasks and tasks: More on `dinfo`

A supervised learning prototask is derived from a dataset by specifying the set of attributes that are available for use as inputs, the set of attributes that constitute the targets to be predicted, and any restrictions on the types of cases for which the learning method is expected to work. It is possible to define many prototasks based on the same dataset, involving different sets of inputs, targets, and cases.

Such prototask specifications are contained in files named `Prototask.spec` in the prototask directories. Usually, users will not look at such files directly, however, but will instead view the information using `dinfo`. For example, the information displayed by `dinfo` for the `age` prototask of the `demo` dataset is shown in Figure 4.2.

The meaning of the prototask specifications displayed by `dinfo` is as follows:

`Origin:` `natural` *or* `cultivated` *or* `simulated` *or* `artificial`

The origin of a prototask and the tasks derived from it is usually the same as that of the dataset on which the prototask is based. For a natural dataset, however, there will generally be only one or a few natural prototasks, those that were of actual interest to the original investigators. Any substantially different prototasks that are based on the same natural dataset are classified as cultivated. In particular, all prototasks based on natural datasets in which the effect of possible sequential dependencies among the cases has been suppressed by random re-ordering are classified as cultivated.

`Cases:` `all` *or* `no missing` *or* filename

This specifies which cases are to be included in the prototask. The special string `all` specifies that all cases are included in the prototask. The string `no missing` specifies that all cases are included except those for which the values of one or more attributes used by the prototask are missing. Otherwise, the cases to include are listed in the given file, as described in Section 4.4.

`Order:` `retain` *or* filename

The order in which cases for the prototask are to be used in constructing training and test sets. The specification may say to `retain` the order in `Dataset.data`. Alternatively, the order may be as specifed in the given file; often this is a file called `Random-order` containing a random re-ordering of cases. Section 4.4 for more details.

`Inputs:` list

A list of indexes or names for attributes of the dataset that the learning method is allowed (but not obliged) to use as inputs.

`Targets:` list

A list of indexes or names for attributes of the dataset that the learning method will attempt to predict.

`Test-Set-Size:` size

The number of cases to be set aside for testing in the standard DELVE set of task instances.

`Training-Set-Sizes:` list

A list of sizes for the training sets for the standard DELVE set of tasks associated with this prototask.

`Test-Set-Selection:` `hierarchical` *or* `common`

Specifies how the test sets should defined for the standard set of task instances. In the `heirarchical` scheme test sets for the different instances are disjoint; in the `common` scheme the same test set is used for all task instances. See Section 6 for further details.

`Maximum-Number-Of-Instances:` number

Specifies the maximum number of task instances used in the standard DELVE scheme. This upper limit is used to prevent a very large number of instances being generated for the tasks with small training sets.

Note that the last four items above are not, strictly speaking, specifications for the prototask, but rather for the standard set of tasks and task instances that DELVE defines for the prototask.

Attributes in the `Inputs:` and `Targets:` list may be identified by number, starting with '1' for the first attribute in the dataset, or by name. An additional attribute, identified by '0', is allowed for datasets with an informative order; its value is the index of the case in the original ordering, starting with one for the first case. (This index attribute is usually not an appropriate input, but provision for its use is included for completeness.) *Note: Attribute '0' is not yet supported by the implementation.*

The ordering of cases in a prototask determines which cases will make up the training and test sets of the various task instances for the standard DELVE set of tasks. Most typically, we will want this ordering to be random, to ensure that cases are effectively independent (even if, in reality, there were dependencies between cases as originally ordered). This can be ensured by using a random re-ordering, though one can also choose to `retain` the ordering if it is certain that the original ordering is random (as will often be the case for simulated or artificial datasets).

When the dataset is in an informative order, one may instead define a *sequential prototask*, in which this order is retained. To avoid certain complications, sequential prototasks are not allowed when the cases also have commonality indexes. In order to allow an appropriate selection of training and test sets, the prototask specification must include a maximum range over which there may be non-negligible sequential dependences that are relevant to the supervised learning task. Note that this may be less than the range over which there are dependencies in the input attributes, as it is only dependences in the noise in the relationship between inputs and targets that are relevant. This maximum range should be set on the high side, to ensure that the performance assessments are not biased. A sequential prototask should not be defined if it is thought that the range of relevant dependencies may be comparable to the number of cases available. *Note: Sequential prototasks are not yet supported by the implementation.*

4.2 The size and nature of the training set for a task

Potentially, a researcher might wish to assess the the performance of learning methods on a task with any number of training cases, up to the maximum that is feasible given the number of cases in the dataset. It is unrealistic, however, to expect all researchers to test their methods on training sets of all possible sizes. DELVE therefore defines a relatively small set of training set sizes for each prototask, which we hope will be adequate for most purposes.

The smallest standard training set size is chosen to be the smallest that the designer of the prototask believes might be sufficient for a learning method to learn something interesting. The larger standard training set sizes are bigger than this smallest size by powers of two, up to a maximum size limited by the need to reserve an adequate test set.

For non-sequential prototasks with a single target taking values from a finite set, DELVE also provides the option of specifying that the training set for a task should be *stratified* by target value — that is, that the training set will contain equal numbers of cases with each target value. The size of a stratified training set must be a multiple of the number of target values. Stratification is natural in applications such as handwritten digit recognition, for which training data would often be collected in a fashion that ensured that there were equal numbers of cases for each digit. The expected performance of a task with a stratified training set will be based on a distribution of test cases in which all values of the target are equally likely. *Note: Support for stratification is not yet implemented.*

4.3 Prior information available for a task

Learning can be (some would say, must be) assisted by the provision of prior information about the relationship to be learned. For real applications, all available prior information should be used to improve performance, to the extent that it can be accommodated by the learning method. But for research into the performance of learning methods, it is not desirable for each researcher to employ whatever prior knowledge they may happen to have about the problem, as the results obtained by different researchers would then not be comparable.

Each DELVE task specification therefore includes an explicit specification of the prior information that is to be regarded as available for use by a learning method. Researchers who happen to know something about the real-world context of the problem beyond what is specified should *not* use such additional information to improve the performance of their learning methods. Indeed, if they happen to know that some of the prior information specified for the task is incorrect, they should still use this information as if they believed it to be true, despite any bad effects this might have on performance. (They could, however, create a new prior specification that reflects their knowledge, and apply their method to tasks based on this new prior.)

Although prior information for real tasks can take many forms, DELVE supports only prior information that is specified in the semi-formal form described below. Most of this prior information is associated with the various input and target attributes for the prototask, and is used to determine the default encodings of attributes, as discussed in Section 7.3. A learning method that uses the default encodings will therefore implicitly be making use of the prior information. A learning method may employ some other way of selecting encodings based on the prior information, however, and may also use prior information in other ways.

```
Task: /demo/age/std.128
Training set size: 128
Inputs:
 col attr name          type    relevance  def coding  options
   1   1  SEX           binary   nlmh       -1/+1        -
   2   3  SIBLINGS      integer  nlmh       nm-abs       -
   3   4  INCOME        real     nlmh       nm-abs       -
   4   5  COLOUR:pink   nominal  nlmh       1-of-n       -
   5   5  COLOUR:blue                 ...
   6   5  COLOUR:red                  ...
   7   5  COLOUR:green                ...
   8   5  COLOUR:purple               ...
Targets:
 col attr name         type    noise-lev def coding  options
   1   2  AGE           real     nlmh       nm-abs       -
```

Figure 4.3: Output of the command: `dinfo /demo/age/std.128`

A prototask will typically come with a "standard" prior specification, stored in the file `std.prior`, which generally will be fairly unspecific (eg, will be vague about how relevant the various inputs are). Other specifications of prior information may also be defined, stored in other files ending in `.prior`. A learning task within a prototask is specified by giving both the name of a prior specification and the number of training cases used, for instance, `std.128`. The prior for a task can be viewed using `dinfo`, as illustrated in Figure 4.3. The output also shows the default encodings derived from this prior information, as explained in Section 7.3.

Note that the explanations of prior specifications given below are meant only as rough guides to their meanings. The precise, quantitative representation of prior knowledge is, after all, a topic for ongoing research in learning. Note also that none of these prior specifications should be taken as indicating absolutely certain knowledge; they mean only that it is considered very likely that the true situation conforms to the specification.

Noise in targets. The amount of inherent noise that is thought to affect the values of a target is specified using one or more of the characters 'N', 'L', 'M', and 'H', representing no, low, medium, or high noise. If more than one character is specified, the amount of noise is uncertain. For example, a specification of 'NLM' indicates that there might be no noise at all, or there might be a low or medium amount of noise, but it is thought that there is not a high amount of noise.

If a target is noise-free, its value will be the same in all cases where the input attributes are the same. This does not imply that the target can be always be predicted with certainty on the basis of information from a finite training set, since there may be no training case with inputs that match a particular test case. It means, rather, that it *would* be possible to predict the target with certainty if we had enough training data. For many prototasks, the inputs will be different for every case that is actually available, so that the characterization is hypothetical in nature (as is the case below as well).

A real-valued target is said to have a low amount of inherent noise if the spread in the distribution of target values over cases where the inputs are all the same is roughly 1% or less of the spread of target values for all cases. For a target with a medium amount of noise, the spread for particular values of the inputs is roughly 10% of the overall spread. For targets with a high amount of noise, the figure is substantially higher, perhaps approaching 100%. Here, the spread is assumed to be measured in a unit such as standard deviation, but the term is left deliberately vague, as it could be, for example, that the standard deviation is not defined for a target that takes on occasional extreme values. The intent is that the rough figures of 1% and 10% should be interpreted with respect to some intuitively appropriate notion of spread.

For discrete targets, a low amount of noise means that the target value differs from that which is most common for the given inputs about 1% or less of the time, with the corresponding figure for medium noise being about 10%, and for high noise something substantially greater than that.

Note: At present, the noise level specified does not affect the default encoding, but this may soon change. For the moment, it is probably best to always specify a noise level prior of 'NLMH', as it is expected that the default coding with this specification will not change in the future.

Dependencies between cases. For a sequential prototask, or a prototask based on a dataset containing cases with commonality indexes, the prior specification for a task must include information on the anticipated strength of any dependencies between cases with the same commonality index, or which are close to each other in sequential order. This specification will consist of one or more of the characters 'N', 'L', 'M', and 'H', representing the possibility of no, low, medium, or high dependencies.

If there is a high degree of dependence between such cases, knowing the true target for one case would, if the true nature of the relationship were known, permit one to predict the target in another case that is nearby, or has the same commonality index, with an accuracy that is better than would be possible without knowing the true target for such another case, by a factor of around 100 or more (in terms of some intuitively appropriate measure of "spread" such as discussed above for noise levels). For a medium degree of dependence, the corresponding factor would be around 10, and for a low degree of dependence, much less (perhaps around 2). If there is "no" dependence, very little or no improvement in predictions would be possible from knowing the true target in another case that is close in sequential order, or that has the same commonality index.

For a sequential prototask, the maximum range over which it is thought that non-negligible dependencies may occur will also be specified as part of the prior information. This maximum range will often be the same as that specified in the prototask specification, but might differ if the effect of changing this aspect of the prior information is being investigated. Note that it is possible for dependencies to persist over a long range even if the magnitude of these dependencies is low. It is usually reasonable, however, to expect that the strength of the dependencies will likely decline at least somewhat with increasing range, even before the maximum is reached.

Note: These prior specifications regarding dependencies between cases have not yet been implemented.

Relevance of inputs. The degree of relevance that an input attribute is thought to possess is specified using one or more of the characters 'N', 'L', 'M', and 'H', representing no, low, medium, or high relevance. If more than one character is specified, this indicates that the degree of relevance is uncertain, except that it is likely to be in one of the categories mentioned.

The meaning of degree of relevance can be explained in terms of the variation in target valuess, after the component of the variation due to inherent noise is eliminated. An input is considered to be of high relevance if as it varies over the range of values that may actually occur in combination with the other input values (which are kept fixed), the target attributes often vary over close to their full range (discounting variation that is due to inherent noise). The effects of some inputs may depend on the values of other inputs. To be considered highly relevant, it is not necessary that the input always have a big effect; only that it does so in many of the cases. Note the mention above of the range of values for the input that actually occur in conjunction with the other inputs. It may sometimes be known that an input would have a big effect if it were to take on an extreme value, but this does not make the input highly relevant unless such extreme values are likely to actually occur.

An input is considered to be of medium relevance if it can have a somewhat smaller effect on the targets — say, changing them by about 10% of their range. Variation in inputs of low relevance might affect the targets to the extent of about 1% of their range. Inputs of "no" relevance have substantially less effect (perhaps none).

Learning methods may use prior information about relevance in various ways. A Bayesian method might use this information to set up a prior distribution for model parameters. A method prone to "overfitting" might reduce the number of model parameters when the training set is small by looking only at inputs thought to be highly relevant .

Note: At present, the relevance specification does not affect the default encoding, but this may soon change. For the moment, it is probably best to always specify a relevance prior of 'NLMH', as it is expected that the default coding with this specification will not change in the future.

Binary attributes. An input or target attribute that takes on only two possible values (not counting missing values) can be specified to be either *symmetric* or *active-passive*.

For a symmetric binary attribute, nothing is known about the two possible values that would justify treating one differently from another. The actual significance of the two values may be quite different, however — we just have no prior knowledge of which way around the effects might go.

For an active-passive attribute, one of the two values is specified to be *passive*; the other is then *active*. Exactly what this means will depend on the problem; the general concept is best defined by an example. In a medical diagnosis task, binary input attributes indicating whether the patient has fever, chest pain, and yellow toenails are active-passive, with the presence of the symptom being the active value. We expect that the presence of such a symptom will have specific diagnostic implications, pointing to a relatively small class of diseases. In contrast, the absence of fever does not in itself suggest a diagnosis. For a binary target, the "passive" value is considered to be the "default", though this does not necessarily mean that it occurs more often than the "active" value.

What, if anything, the distinction between symmetric and active-passive attributes should mean for the proper treatment of binary inputs and targets is a matter for researchers developing learning methods to judge. However, the default DELVE encodings (see section 7.3) do treat symmetric inputs symmetrically, and active-passive inputs asymmetrically.

Categorical attributes. An input or target attribute that takes on a finite number of possible values (three or more, not counting missing values) may be specified to be *nominal* or *ordinal*. This distinction affects the default encoding of the attribute, as discussed in Section 7.3.

The values of a nominal attribute are significant only in that they are distinct from one another, except that one of the values may optionally be singled out as the *passive* value. The meaning of such a passive specification is analogous to that described above for binary attributes.

The values of an ordinal attribute have a defined ordering, which must be specified, if it differs from the order in which the possible values are listed in the dataset specification. The first value in this ordering may optionally be specified to be *passive*. *Note: There is currently no way of overriding the ordering of attribute values in the dataset specification.*

Real-valued attributes. Currently, no specific prior information pertaining to real-valued attributes is recorded, other than the noise level and degree of relevance, as discussed above. Formal specification of prior information regarding promising transformations of real-valued input and target attributes may be allowed in future. The expected degree of smoothness in the relationship between a real-valued input attribute and the targets might also be useful prior information, but this also has not been standardized. In the absence of such information, it is appropriate to assume that relationships are often smooth, or at least continuous, but that discontinuities are not impossible.

Integer attributes. At present, no special special prior specifications are defined for integer attributes. The relevance and noise level priors apply, however.

Angular attributes. Numeric attributes interval can be specified to be *angular*. These attributes are thought to have a circular meaning, for which all that matters is the modulus of the value with respect to some unit. For instance, a attribute giving the time of day could be considered to be angular, with a modulus of 24 hours.

Angular attributes are by default encoded in terms of the sine and cosine of the angle they define (see Section 7.3). This representation respects the assumed continuity as values wrap around.

4.4 Defining prototasks: The `dgenorder` and `dgenproto` commands

Before a new dataset can be used to assess learning methods in DELVE, at least one prototask must be defined for it. Researchers may also wish to define new prototasks for existing datasets. This section describes how to do these things, as well as the approach taken in defining the standard DELVE prototasks.

The purpose of defining a prototask is to support interesting experiments, which say something significant about the learning methods that are assessed. For some datasets, such interesting prototasks may need to have special features. For example, if a potential input attribute is very highly correlated with a target attribute, it may be best to leave it out of the allowed set of input attributes, in order to prevent the prototask from being so easy that it is uninteresting. If the inputs in a few cases differ greatly from those in the other cases, it might be of interest to define a prototask that excludes cases with these extreme inputs, in order to assess learning methods that do not purport to handle such extrapolation well. The documentation for a prototask with unusual features should include a statement of the research questions the prototask is meant to address, and a justification for its specifications in terms of these objectives.

Most standard DELVE prototasks are defined with no specialized objectives in mind, however, and include all attributes and all cases. Complications due to *missing data* arise fairly often, however. Since many of the supervised learning methods we would like to assess do not naturally handle missing data, we hope to obtain a good collection of DELVE prototasks in which the values of input attributes are never missing. We expect that this will require creating some such prototasks by excluding a few input attributes whose values are missing in many cases, or by excluding a few cases for which the values of one or more attributes are missing, or by doing a bit of both.

The designer of a prototask decide how to deal with any dependencies between cases that may be present. We take two approaches to this for the standard DELVE prototasks. For some prototasks, we accommodate the dependencies in a proper fashion (*or rather, we will do so once the required facilities are implemented*). In particular, we ensure that there are no significant dependencies between training and test cases, as this would invalidate the results. Other times, however, we instead circumvent sequential dependencies by randomly reordering the dataset. This second approach allows us to define tasks for which ignoring dependencies gives internally consistent results, although such tasks no longer correspond to real-world situations.

`Prototask.spec`

```
Cases: all
Inputs: 1 3 4 5
Order: retain
Origin: artificial
Targets: 2
Test-Set-Size: 1024
Training-Set-Sizes: 32 64 128 256 512
Test-Set-Selection: hierarchical
Maximum-Number-Of-Instances: 8
```

`std.prior`

```
1 NLMH binary
3 NLMH integer
4 NLMH real
5 NLMH nominal
2 NLMH real
```

Figure 4.4: Prototask specification (`Prototask.spec`) and standard prior specification (`std.prior`) for the `age` prototask of the `demo` dataset.

When a non-sequential prototask is defined it is recommended that the cases always be randomly re-ordered, unless it is known for certain that the existing order is random. Certainly this must be done if the ordering is `informative`, or is sorted by some attribute value. It should also be done even if it is thought that the order is arbitrary, in order to provide greater certainty that assessments based on the assumption of no sequential dependence will be internally valid. When cases have commonality indexes, this random re-ordering must keep cases with the same index grouped together (in random order), while randomly ordering the groups themselves.

To create a prototask, you first must create a directory for the prototask within the DELVE hierarchy. This directory must have the same name as the new prototask, and be located within one of the directories for the dataset in the DELVE hierarchy. Within this prototask directory, you must create a `Prototask.spec` file, containing the specifications for the prototask and the standard set of tasks associated with it, and also one or more files containing prior specifications, usually including `std.prior`, which contains the "standard" prior information.

These files have formats paralleling the output of `dinfo` for a prototask and for a task. A `.prior` file should have one line per attribute, specifying the attribute number, the noise level or relevance prior, the type of the variable, and any additional options. For example, the line for a nominal attribute, numbered 2, thought to be of at least medium relevance, and which has a passive value of `none`, would be be

```
2 MH nominal passive=none
```

An angular attribute must be accompanied by a `unit=`*modulus* specification.

The `Prototask.spec` and `std.prior` files for the `/demo/age` prototask are shown in Figure 4.4.

The `/demo/age` prototask includes `all` cases in the dataset. Another built-in option is `no missing`, which specifies that all cases should be included except those for which one or more of the attributes used in the prototask are missing. One can also give for `Cases` the name of a file that contains an explicit list of case numbers to include, one case per line, with numbers starting at one. The order of lines in this file does not matter. This case file should be located in the DELVE hierarchy, within the prototask directory.

The order for the `/demo/age` prototask is specified as `retain`. This prototask is non-sequential, but the data is artificially generated in a fashion that guarantees that the original data file is in random order. For a natural or cultivated dataset, one would normally randomize the ordering explicitly (assuming that the prototask is not meant to be sequential). One does this by specifying a file that contains such an ordering. This file must be located in the DELVE hierarchy, within the prototask directory. The order file should have one line per case, with each line containing the index of a case. Indexes start at one, and go up to the number of cases that are used in the prototask. Note that if any cases were left out of the prototask, these will *not* be the indexes of the cases in `Dataset.spec`.

Most often, this ordering file will be called `Random-order`, and will be generated automatically using the `dgenorder` command. This command will also take care of the complications involved in handling commonality indexes. *Or at least it will once commonality indexes have been properly implemented*

Another command that you will often wish to use after creating a new prototask is `dgenproto`, which creates the intermediate file `Prototask.data`, containing the portion of `Dataset.data` relevant to this prototask. This intermediate file will be created "on-the-fly" by other commands, as needed, but creating a single permanent copy will save time. You will also want to use the `dcheck` command, in order to check that the prototask specifications are consistent with the dataset specifications.

Here is how you would go about creating a non-sequential prototask for a natural dataset:

```
unix> cd dataset             # Change to a delve directory for the dataset
unix> mkdir prototask        # Create a directory for the new prototask
unix> cd prototask           #   and change into it
unix> edit Prototask.spec   # Create the prototask specification file
unix> edit std.prior        # Create the standard prior specification
unix> dcheck                # Check that it's all consistent
unix> dgenorder             # Generate the Random-order file
unix> dgenproto             # Generate the Prototask.data file
```

The `dgenproto` step is optional, but usually advisable; if it is done, it must be after `dgenorder` has been done. For a simulated or artificial dataset, where the cases are already in random order, the ordering would usually be `retain`, and the `dgenorder` step would be omitted. *Note: The* `dcheck` *command is not implemented yet, so you will have to leave out that step at present.*

5 PREDICTIONS AND LOSS FUNCTIONS

Together, the specifications for a prototask and for one of its tasks determine what is to be learned and what information will be available on which to base learning. To complete the specification of a learning problem, we need to say what form the output of a learning method should take, and how the performance of a method on a task will be judged.

DELVE supports assessments only of the predictive performance of learning methods — the degree to which the relationships learned can be used to predict attributes in previously unseen cases. For this purpose, the relevant output of a supervised learning method is a set of *predictions* for the target attributes in a set of *test cases* for which only the input attributes are known. The accuracy of these predictions is judged by how well they match the actual values of the targets, as measured by some *loss function*.

For some methods, learning, making predictions, and judging the loss from these predictions may be sequential activities, with the nature of the predictions required having no effect on the learning itself, and with the loss function by which these predictions will be judged having no effect on the predictions themselves. In general, however, this need not be so. A learning method may be designed to behave quite differently depending on the predictions that it will be required to produce, or on the loss function by which these predictions will ultimately be judged.

5.1 Types of predictions

DELVE expects learning methods to produce predictions in the form of either *guesses* or *predictive distributions*. A real application might require either type of prediction, and many learning methods will be able to produce predictions of both types.

A *guess* for a target in a test case is a value of the same type as the target itself — that is, if the target is categorical, the guess will be one of the possible target values, and if the target is numerical, so will the guess be (though a guess for an integer target need not be an integer). If there is more than one target attribute, a separate guess is made for each target. One might sometimes wish to allow a learning method to decide to make no guess for a target (at a penalty); provisions for this are described in Section 5.3.

The accuracy of a guess is judged by a loss function that measures how close the guess is to the true value, as described below in Section 5.2.

A *predictive distribution* is a probability distribution for the targets in a test case, conditional on the known values of the inputs for the test case. In theory, a learning method that produces predictions of this form should output a complete representation of the predictive distribution for each test case. Given this distribution and the actual value, a loss could then be computed using one of the loss functions described below (Section 5.2).

However, the predictive distribution for a target produced by a learning method could be arbitrarily complex (at least for real-valued targets). When there is more than one target, the predictive distribution might in general involve dependencies between targets. Due to the difficulty of defining a standard representation for predictive distributions that is both convenient and sufficiently general, DELVE does not require learning methods to actually output their predictive distributions. Instead, the computation of the loss based on the predictive distribution and the actual target values is left for the learning method itself to compute, using its internal representation of the predictive distribution.

Tasks with a single categorical target are an exception to this general procedure. In this case only, a learning method may output an explicit representation of the predictive distribution for each test case, as described in Section 7.5, leaving the computation of losses to DELVE. This is in fact the preferred procedure, since it makes the predictive distributions available for examination, and avoids the possibility that the learning method will compute the losses incorrectly.

5.2 Standard loss functions supported by DELVE

The accuracy of a prediction for a test case is measured by a *loss function*, which takes two arguments: The prediction output by the method for a particular test case, and the true values of the targets for that case. The value of the loss function is a single real number that represents the "loss" suffered when the given prediction is used in a situation where the true values of the targets are as given.

Note that the loss function is defined in terms of a single test case, not a set of test cases. The goal of prediction is to minimize the expected value of this loss on a test case that is randomly drawn from the distribution of cases defined for the prototask. In assessing the performance of a method, we will of course use test sets with many cases, taking the average loss over many test cases as an estimate of the expected loss on a single test case.

The loss function for an actual application might sometimes be quite complex and specialized. DELVE does not attempt to assess methods for producing predictions in such a context, but concentrates instead on a predictions that will be judged using a few simple loss functions. These loss functions have been selected because they are already in common use, and because they emphasize somewhat different aspects of predictive performance. The performance of a learning method with respect to these standard loss functions can be compared to that of the many other methods that will have been assessed with the same loss functions. More specialized loss functions may be of interest for some prototasks, however, and DELVE does provide some support for them, as is described in Section 5.3.

Each of the standard loss functions has a one-letter abbreviation. This abbreviation is used to specify a loss function, and occurs in the standard names for files holding predictions and losses on a task instance, as is described further in Section 7. The standard loss functions are summarized in Figure 5.1.

	abbrev.	*categorical?*	*integer?*	*real?*	*angular?*
For guesses:					
Squared-error loss	S		$\surd$	$\surd$	
Absolute-error loss	A		$\surd$	$\surd$	
0-1 loss	Z	$\surd$	$\surd$		
For predictive distributions:					
Log-probability loss	L	$\surd$	$\surd$	$\surd$	$\surd$
Squared-probability loss	Q	$\surd$			

Figure 5.1: Standard loss functions, their abbreviations, and the types of targets for which they can be used.

For predictions that take the form of guesses, the standard loss functions are all based on a "distance" of some kind between a guess and the true value of a target. For tasks with more than one target, the total loss is simply the sum of the losses based on the distance of each target guess from the true target value.

For guesses of targets that take on integer or real values, DELVE supports two loss functions, based on squared and absolute distance. The *squared-error loss* is the square of the difference between the guess and the true target value. Those who take a probabilistic approach to learning should note that the expected squared-error loss is minimized by guessing the mean of the predictive distribution for the target. The *absolute-error loss* is the absolute value of the difference between the guess and the true target value. The expected absolute-error loss is minimized by guessing the median of the predictive distribution.

For guesses of integer and categorical targets, DELVE supports *0-1 loss*, in which the loss is zero if the guess is correct, and one if it is incorrect. The optimal strategy for minimizing 0-1 loss is to guess the target value with greatest probability (the mode of the predictive distribution).

DELVE does not currently support any loss functions for guesses of targets that take on angular values. There is also no provision for using different loss functions for the various targets in a case.

For predictions that take the form of a predictive distribution, one may use *log probability loss*, which is minus the log (to base e) of the probability or probability density of the true target values under the predictive distribution. Log probability loss may be used with targets of any kind. Note that if all targets are integer or categorical, the predictive distribution will consist of probabilities for the various combinations of target values. If instead the targets are all real or angular, the predictive distribution will take the form of a probability density (which must be finite if log probability loss is to be used). If some targets are integer or categorical and others are real or angular, the log probability loss will be computed from the hybrid probability/density of the true target values.

Squared-probability loss may be used with predictive distributions for a single categorical target. In this case, the prediction takes the form of a list of probabilities, $p_1, \ldots, p_n$, for the possible target values, which are labeled 1 to n, with t being the true target value for the case in question. (As mentioned in Section 5.1, in this case only, the learning method may produce the predictive distribution explicitly.) The squared probability loss is the square of one minus the probability assigned to the true target value, plus the squares of the probabilities assigned to all the other possible target values — that is, $(1-p_t)^2 + \sum_{i \neq t} p_i^2$.

Note that the expected value of both the log probability loss and the squared probability loss is minimized by a distribution matching the true probabilities. The log probability loss will be infinite if the probability or probability density for the true target is zero, but the squared-probability loss is never greater than two.

5.3 Using a specialized loss function

Note: The facilities described in this section have not yet been implemented.

In addition to the standard predictions and loss functions described above, DELVE supports specialized predictions in which guessing is optional, and specialized loss functions defined by an arbitrary loss matrix. These facilities are intended for use with natural prototasks that come from application areas where such specialized predictions and loss functions are appropriate, or with cultivated or synthetic prototasks that are intended to mimic such actual applications. For example, an automatic postal code recognition system may have the option of referring hard-to-recognize postal codes to a human worker, and in a medical testing application, we might wish to treat a false positive as less serious than a false negative.

Guessing can be made optional by specifying a *no-guess penalty*, which is the loss suffered when the learning method decides to make no guess — presumably because the method is so uncertain of the value of the target that it expects the loss produced with its best guess to be greater than the no-guess penalty. This form of prediction and loss function is specified by appending the value of the no-guess penalty followed by "N" to the abbreviation of any of the loss functions for guesses in Figure 5.1. For example, "Z0.2N" specifies 0-1 loss with a penalty of 0.2 for not making a guess.

A non-standard loss function for guesses of a single categorical target can be specified by means of a *loss matrix*, in which the loss for every possible combination of a guess and a true value for the target is explicitly specified, with the restriction that the losses must be non-negative, and be zero when the guess is correct. A loss for making no guess may also be specified, separately for each true target value.

Use of a loss matrix is specified by giving "M" followed by a file name wherever you would otherwise use an abbreviation for a standard loss function. This file should be located in the `data` part of the DELVE hierarchy, in the directory for the corresponding prototask. The file should contain as many lines as there are possible values for the target attribute, plus one additional line if the method is to be allowed to make no guess. The lines correspond to possible guesses, according to the ordering of possible attribute values in the dataset specification. Each such line should contain numerical values for the losses suffered for each possible true value of the target, again in the order given by the dataset specification.

6 SCHEMES FOR LEARNING EXPERIMENTS

Tasks are sufficiently well-defined that each learning method has a well-defined expected performance on each task, which is the expected value of some specified loss function on a randomly selected test case, when using the predictions produced by the learning method based on the given prior information and a random training set of the specified size. We do not, and never will, know this expected performance exactly, but we can estimate the expected performance by performing experiments in which we apply the learning method to several *task instances*, each of which has a particular set of training cases, and a particular set of test cases.

There are many possible schemes for defining task instances, with different advantages and disadvantages. This section describes the standard schemes used in DELVE, and discusses why we chose this scheme for experiments.

6.1 Issues in designing learning experiments

For research purposes, we are usually interested not so much in the numerical value of the expected loss for a method applied a task, but rather in the relative performance of several learning methods on the same task. Such performance comparisons can be done more accurately if the various performance estimates are all based a common set of task instances, in which the training and test sets contain the same cases.

The statistical benefits of such a common set of task instances are discussed further in Section 8. In this section, we describe the standard scheme used in DELVE to define a common set of task instances for each task. This scheme has been designed not only to allow for good estimates, and an indication of their accuracy, but also to limit the number of task instances, and hence the number of times that a learning method must be applied to a training set in order to obtain a performance estimate for a task. Minimizing the number of applications is important if sophisticated learning methods are to be evaluated, which may, at least in early stages of research, be computationally intensive. It is even more important for learning methods that involve decisions made by a human analyst. In order to achieve these goals, we have been willing to forgo the use tasks based on small datasets, as we believe that any research questions that these datasets might be useful in answering can equally well be addressed using larger datasets.

Two different situations arise depending on whether we are dealing with real or synthetic datasets. For real datasets the number of available cases is often a limiting factor, and it therefore seems best to use large a single common test set for all instances — what is referred to in DELVE as a *common* testing scheme. On the other hand, if we are dealing with synthetic data, it is usually possible to generate an unlimited amount of data for testing, and in this case the limiting factor will be the disk space needed to store the prediction and loss files for all the applications of methods to task instances. In this case it seems more profitable to use disjoint test sets for different instances, allowing a much larger number of test cases in total for a given amount of disk storage. This is what we call the *hierarchical* testing scheme.

The standard DELVE schemes for defining task instances are certainly not the only possible ways of estimating expected performance, however. Some researchers may prefer to use some other scheme, such as leave-one-out cross-validation. One may also wish to evaluate the performance of a new method on exactly the same task instances as were used to evaluate some older method. For these reasons, we allow users to specify non-standard task instances, which will enable them to perform such evaluations using the DELVE facilities described in Section 7. *Note: This facility isn't implemented yet, however.*

6.2 DELVE's standard set of task instances

In the standard set of tasks for each prototask, the training set size is one of a series of numbers that differ by factors of two. The designer of a prototask based on some dataset might, for example, have specified standard tasks with training set sizes of 20, 40, and 80. The same range of training set sizes, and the same actual training and test sets, are used for all specifications of prior information, and for all loss functions. The designer of a prototask also specifies how many cases should be reserved for use in testing.

To obtain the standard set of task instances that go with a task, DELVE first reserves the specified number of cases for use in testing. Training sets of the desired sizes are then obtained by successively dividing the set of remaining cases (whose number will usually have been arranged to be a multiple of the largest standard training set size). In the above example, suppose that the prototask was applicable to 500 cases in the dataset. We could reserve 340 cases testing, leaving 160 cases for inclusion in training sets. For the task with a training set of size 80, this allows for two task instances, obtained by partitioning the 160 cases not in the test set into two subsets. Similarly, four instances can be created of the task with a training set of size 40, obtained by dividing each of the training sets of size 80 in half, and eight instances of the task with 20 training cases, obtained by subdividing the 40-case training sets yet again. (It would also be possible to define a single instance of a task with a training set of size 160, but with only a single training set, no empirical assessment could be made of the variability of performance on this task with respect to random choice of training set.)

In the above example, the generation of test sets for each task instance would depend on the type of `Test-Set-Selection` specified for the prototask, as explained in the previous section. If the `Test-Set-Selection` is `common` then all test cases will be included in a single common test set, used for every instance. If the `Test-Set-Selection` is `hierarchical` then the test cases will also be divided into smaller disjoint subsets, one for each instance of a particular size.

The successive partitioning described above is performed using the order of cases as defined in the prototask specification. For prototasks without any complications, the test set consists of the first so-many cases in this ordering, and the training sets are taken from the later part of the ordering. The training sets of different sizes are obtained by successively dividing the full set of potential training cases into contiguous blocks. Recall that the ordering of cases will be random unless the prototask is intended to be sequential.

The above scheme becomes a bit more complicated if the the prototask has special features. For a sequential prototask, a gap of unused cases will be left between the cases used for testing and those used for training. The size of this gap will be the maximum range of dependencies given in the prototask specification. For data with commonality indexes, the ordering will group cases with the same commonality index together, and a gap will be left if necessary to ensure that no cases used for testing have the same commonality index as a case in some training set. These provisions to eliminate dependencies between the training and test sets are needed for the performance on the test set to be a faithful indication of real performance. Note, however, that for sequential prototasks and for prototasks where there are commonality indexes, there may still be dependencies between the training sets for different instances of a task (though we try to avoid this with commonality indexes). This could reduce the accuracy of the performance estimates, but does not introduce any bias.

The provisions in the above paragraph have not been implemented yet. Special provisions will also be needed to handle tasks whose training sets are specified to be stratified.

6.3 Using non-standard task instances

The facilities in this section have not been implemented yet.

A non-standard task instance (perhaps for a task with a non-standard size of training set) can be specified by giving an explicit list of the indexes of the cases making up the training and test sets. These indexes are with respect to the ordering of the original dataset, but must be among those included in the prototask.

For a sequential prototask, the list for the training set must be a sub-sequence of the prototask ordering, and the test cases must be further from the training cases than the maximum range of dependencies specified for the prototask. It is the user's responsibility to ensure that the manner in which cases were selected for the training and test sets is valid in other respects.

7 ASSESSING A LEARNING METHOD

This section and the following explain the details of how to use DELVE to assess learning methods. We start here with guidelines on documenting your method, and then discuss how you can apply your method to a set of task instances.

The information relating to a method and its application various tasks is organized into files and directories in the `methods` part of the DELVE hierarchy. This organization is illustrated in Figure 1.1, and some of the files involved are listed in Figure 7.1 below.

7.1 Documenting the method to be assessed

An essential part of reporting results for a learning method is to document, as precisely as possible, what the method actually does. These descriptions should be detailed enough to allow someone to implement the method from the description and get results similar to those reported. The description should include a specification of how data should be encoded for use by this method, on the basis of the available prior information. Without such a specification, it is unclear how the method would be applied to a new task. If the method uses DELVE's default encodings, you can just say that. The description for a method should also specify such matters as how to decide when an optimization procedure has converged. You can get an idea of the level of detail required in documentation by looking at the existing documentation in the `methods` directory.

Precise specification of what a learning method does is easiest if the method is fully automatic. However, there may be situations when it is undesirable to formulate fully automatic

`Summary`	A brief description of the method
`Source`	A sub-directory with files that document the method, and perhaps programs that implement it
dataset/prototask/task	A sub-directory holding results for one task, with files such as:
`Test-set-stats`	Statistics from the test data used to standardize losses
`Coding-used`	The attribute encoding that was used to generate the data files
`normalize.`n	Normalization constants from training data for instance n
`train.`n	Training data (inputs and targets) for instance n
`test.`n	Test inputs for instance n
`targets.`n	Test targets for instance n
`cguess.`n	Coded guesses for test targets for instance n
`guess.`n	Uncoded guesses for test targets for instance n
`loss.S.`n	Squared error losses produced using the guesses for instance n

Figure 7.1: Some files and sub-directories that may appear within a DELVE methods directory.

methods. In these cases, careful descriptions of the heuristics used, together with examples of the human choices made on sample tasks may be useful. Since our overall goal is to evaluate how well learning methods can be expected to work on novel tasks, when applied by people who are not necessarily the designers of the learning method, the proper approach to assessing a non-automatic method would be for the developers of the method to get other people to apply the method following their documentation. This method of evaluation may perhaps be too cumbersome in practice, but it is useful to keep in mind while documenting a non-automatic learning method.

In many cases it may be a good idea to supply the source of a computer implementation as a part of the documentation, since the program itself may be able to resolve important details of the methods. One should not consider cryptic computer code to be a substitute for an intelligible description, however.

It is also useful to include some rough estimates of the computational costs associated with applying the learning method. Some learning procedures can use arbitrary amounts of computation time; in this case a fully-specified method must indicate how the time is limited in practice. Different time allowances will define different (albeit closely related) methods.

Learning algorithms often have parameters whose values need to be set using empirical trials. DELVE includes a suite of developmental datasets that are intended for used in such trial runs. However, it is possible that you will discover ways of improving your method as a result of running it on one of DELVE's assessment datasets. This is unfortunate, since modifying the method based on performance on these datasets may introduce bias in the evaluations. If a method was tuned using the assessment datasets, you should therefore include in your documentation a short description of what tests were done, on what datasets, so that people can take account of this tuning when judging the significance of the results obtained.

Documentation and programs relating to a method should be placed in the DELVE hierarchy in the `Source` sub-directory of the method's directory. A brief summary of the method should also be placed in the `Summary` file in the method's directory.

7.2 Creating directories for assessments: The `mgendir` command

For each dataset used to assess a method, a directory with the name of the dataset will exist in the DELVE hierarchy, within the directory for the method. These directories need not all be in the same actual directory, but may instead be in located within various of the active `delve` directories. This allows you to assess existing methods on new tasks without having to write into the directory holding results from the DELVE archive.

You can create such directories manually if you wish, but it is usually easier to create an appropriate directory structure using the `mgendir` command. This command will generate all the directories associated with a given dataset, prototask, or task. If a task is specified, only the directory for this task will be generated (along with the directories needed to contain this task directory, if they do not already exist). If a prototask is specified, then directories for all the tasks associated with this prototask will be generated. Typically there will be many tasks, with different training set sizes, and perhaps with different prior information. Similarly, if a dataset is specified, directories for all prototasks defined for the dataset will be created.

`Mgendir` creates these directories in or below the current directory. If some of the directories already exist, `mgendir` simply makes sure that they are up to date. An example will illustrate the command:

```
unix> cd delve/methods; mkdir mymethod; cd mymethod
unix> mgendir demo/income/std.32
demo/income
demo/income/std.32
unix> mgendir /demo/income
demo/income/std.64
demo/income/std.128
demo/income/std.256
demo/income/std.512
```

In this example we first generated the task named `std.32` of the `demo/income` prototask. The `mgendir` command created the appropriate directories for that dataset, prototask and task. We then asked to have the entire set of tasks for the `income` prototask generated. In this case `mgendir` skips the existing directories and generates the new ones. Notice that the identity of the current directory is important. For example, if your current directory is at the task level, you should not ask `mgendir` to generate directories for a new dataset — this will cause mixing of the different levels. Always issue the `mgendir` command from the correct level (or higher up, as in the above example).

Note that `mgendir` just creates directories; it does not create the data files needed to train and test your learning method. That is done by the `mgendata` command.

The above discussion has focused on the most common usage of generating directories according to existing specifications in the corresponding `data` part of the DELVE hierarchy. You may sometimes want to generate tasks with different specifications. For example, you might want to use an existing prototask, but with a new specification for prior information. In this case, you would create a new prior specification file in your `data` directory, and specify this name to `mgendir` to generate the data.

7.3 Specifying how attributes are to be encoded

Part of the definition of a learning method is the manner in which attributes are encoded in a form suitable for the technique used. For example, inputs to a neural network must be numeric, so a method based on neural networks that handles categorical inputs must include a definition of how a categorical value is represented as one or more numbers.

Some researchers may be interested in developing better encoding methods, in which case they will of course employ whatever methods they think are most promising. DELVE has facilities that support a number of common encoding methods, but it is of course possible that you will have to implement the encoding you want to use yourself.

For researchers who are not especially interested in encoding methods, DELVE supplies *default encodings* for attributes, selected on the basis of the prior information for the task. If you have no reason not to, it is probably best for you to stick with the default encodings, as that will make it easier to isolate the reasons for any differences in performance between your method and other methods that also uses the default encodings.

An encoding specification consists of a name for the encoding, perhaps followed an additional `passive`, `unit`, or `center` argument. The possible encodings are as follows:

`ignore` Ignore the attribute.

`copy` Copy the raw attribute value unmodified from the dataset file.

`0/1` Encode a binary attribute as '0' or '1', with '0' being the passive value. An argument of `passive=`*value* is mandatory.

`-1/+1` Encode a binary attribute using a symmetric encoding of '-1' for the first value and '$+1$' for the second value (as ordered in the dataset specification).

`1-of-n` Encode a categorical attribute as a list of zeros and ones. If the attribute has n possible values, and no `passive` argument is specified, values will be encoded using n numbers, exactly one of which is '1', with the others being '0'. If an argument of `passive=`*value* is given, the n possible values will be encoded as $n-1$ numbers, with the passive value being encoded by all the numbers being '0', and the non-passive values being encoded as before, by setting exactly one of the numbers to '1'.

`therm` Encode a categorical attribute by a thermometer code, using a list of $n-1$ numbers with values of $-x$ or $+x$, where n is the number of categories for the attribute, and x is a scaling factor described below. The lowest value of the attribute (according to the ordering in the dataset specification) will be encoded by setting all numbers to $-x$. For the next higher value, the first number will be $+x$ and the remaining ones $-x$, and so forth. The scaling factor x is determined by the `scale=`*string* option, where *string* is one of `none`, `linear`, or `sqrt`. If it is `none`, then $x = 1.0$. If it is `linear`, then $x = (n-1)^{-1}$. If it is `sqrt`, then $x = (n-1)^{-1/2}$. The default value is `sqrt`.

`nm-sqr` Encode a numerical attribute by shifting and re-scaling its values so that the distribution of these values in the training set has mean zero and variance one. If a `centre=`*c* argument is specified, the values will be shifted to have c as their mean rather than zero.

`nm-abs` Encode a numerical attribute by shifting and re-scaling its values so that the distribution of these values in the training set has median zero and average absolute deviation from the median of one. If a `centre=`c argument is specified, the values will be shifted to have c as their median rather than zero.

`0-up` Encode a categorical attribute as an integer, from zero and up to the number of possible values minus one (using the ordering of values in the dataset specification).

`1-up` Encode a categorical attribute as an integer, from one and up to the number of possible values (using the ordering of values in the dataset specification).

`rectan` Encode a numerical value, x, as two numbers, $\sin(2\pi x/u)$ and $\cos(2\pi x/u)$, where u is the value specified by a mandatory argument of the form `unit=`u.

If you need to use encodings other than these, you will have to specify a coding as close as possible from the list above, and then modify the data files DELVE produces using a program of your own.

When you generate data files using the `mgendata` command (described in the next section), DELVE will by default use encodings from the above list that are selected on the basis of the prior information specified for the task (see Section 4.3). The default encoding for an attribute is based first of all on the type assigned to the attribute in the prior specification, in the following way:

`binary` attributes with a `passive` value are coded as `0/1`; those without a passive value are coded as `-1/+1`.

`nominal` attributes are encoded as `1-of-n`, with a passive option if a passive value is specified in the prior.

`ordinal` attributes are encoded using `therm`, with the default scale option `sqrt`.

`real` attributes are encoded using `nm-abs`.

`integer` attributes are also encoded using `nm-abs`.

`angular` attributes are encoded using the `rectan` code, with the `unit` argument as specified in the prior.

You can override the default encodings by giving the name of a file of alternate encodings (typically called `encoding`) to the `mgendata` command, using the '`-c`' option. For the format of this file, see the documentation for `mgendata` in Appendix C. This is useful if you wish to use other than the default encodings, and also if the software your using has built-in facilities that implement the default encodings, but expects to receive attributes in some other format.

The manner in which choices of encodings are made is logically part of the learning method and should be documented as part of the description of the learning method being assessed. If other than default encodings are being used, you will probably have to manually specify how attributes are to be encoded for a particular task, according to the rules defined for the method. In theory, however, a method's encoding rules could be implemented automatically, using a program that reads the relevant specification files.

7.4 Creating data files for training: The `mgendata` command

Once you have decided on the encodings to be used by a method on some task (which may be just deciding to use the defaults), you can use the `mgendata` command to generate the training and testing data files to be read by the program implementing the method. These files must be placed in the directory for the task within the `methods` part of the DELVE hierarchy, which you will usually have created earlier using `mgendir`. The `mgendata` command can also generate files for all the task instances associated with a prototask or dataset, as described in the detailed documentation for `mgendata` in Appendix C.

For each task, `mgendata` creates files pertaining to all task instances. These files all have the number of the instance (from 0 on up) as a suffix. Four files will be generated for task instance n: `train.`n, `test.`n, `targets.`n and `normalize.`n. The contents of the first three of these files will depend on the encoding used, which can be left to default, or can be specified using the '`-c`' option of `mgendata`, which should be followed by the name of the file containing the alternate encodings. If you type `minfo` (with no arguments) in the task directory for a method after running `mgendata`, you will see a listing of all the numbers involved in encoding the attributes for the present set of data files (as saved in the file `Coding-used`). Typing `dinfo` (with no arguments) will show you what numbers would be produced by the default encodings. These commands can also take explicit task specifications. Figure 7.2 shows the display of the default encodings for the `/demo/age/std.128` task produced by `dinfo`.

The `train` files produced by `mgendata` contain the training cases, one per line. The encoded values of the input attributes for a case appear first on the line, in the order they are mentioned in the prototask specification (and in the output of `dinfo` or `minfo`). The encoded values of the target attributes follow the inputs. All the numbers in a training data file are separated by spaces. Note that there may well be more numbers than attributes, since some attribute encodings produce more than one number — as is the case with the `COLOUR` attribute in Figure 7.2.

The `test` files contain only the input attributes of the test cases. The true targets for the test cases are not supplied, since they should not normally be available to the learning method. An exception is allowed for a method that makes predictions to be evaluated using the log probability loss functions (see Section 7.5), since it is not practical for DELVE to evaluate these losses itself. The true targets are available for this use in the `targets` files.

The `normalize` files contain the offset and scaling constants that may have been used to

```
Task: /demo/age/std.128
Training set size: 128
Inputs:
 col attr name          type    relevance  def coding  options
   1   1  SEX           binary   nlmh        -1/+1        -
   2   3  SIBLINGS      integer  nlmh        nm-abs       -
   3   4  INCOME        real     nlmh        nm-abs       -
   4   5  COLOUR:pink   nominal  nlmh        1-of-n       -
   5   5  COLOUR:blue                ...
   6   5  COLOUR:red                 ...
   7   5  COLOUR:green               ...
   8   5  COLOUR:purple              ...
Targets:
 col attr name          type    noise-lev  def coding  options
   1   2  AGE           real     nlmh        nm-abs       -
```

Figure 7.2: Output of the command: `dinfo /demo/age/std.128`.

encode the data (if `nm-abs` or `nm-sqr` encodings were specified, or were the defaults). You will not normally have to look at the `normalize` files yourself, but they are needed for DELVE to interpret the predictions produced by the method.

Once the training and testing data files for the various task instances have been generated using `mgendata`, you can run your learning method. This should be done completely independently for each task instance, with the run for one instance making no reference to any data files intended for another instance. If your learning method has a stochastic aspect, you should initialize the random seed differently for each instance, for reasons discussed in Section 8.

7.5 Processing predictions on test cases: The `mloss` command

The objective of running your learning method is to produce predictions for the test cases. These predictions will normally be encoded, in the same way as the targets seen by the learning method were encoded. As discussed in Section 5, predictions can take two forms: point predictions or *guesses* for the target values, and *predictive distributions* for the targets. In most circumstances, your method will not read the `targets` files when producing predictions, and the losses with these predictions will be calculated by DELVE, not by the method itself. However, since there is no easy way of representing an arbitrary predictive distribution for a target of real, integer, or angular type, the predictive probability density for the true target must be evaluated by the method itself, if log probability loss is of interest, with reference to the true target values found in the `targets` files.

The actual losses are in all cases evaluated by the `mloss` command, which will refer to files of predictions produced by the method. In general, a method may wish to make different predictions for use with different loss functions. Accordingly, the files to which a method writes predictions may have names incorporating the abbreviation of the loss function for which they are intended. Prediction files have one of three possibile root names, according to the form of prediction: `guess`, for point predictions, `prob`, for predictive distributions, and `ptarg`, for probabilities (or probability densities) of the true target value. Prediction files always have the instance number as a final suffix (e.g. `guess.3`). If a specific loss function is specified, it goes between the root and the instance number (e.g. `guess.S.0`). The name of a `prob` or `ptarg` file can be prefixed by "l" to indicate that it contains the (natural) logs of the probabilities (or densities), rather than the probabilities themselves. Finally, names of prediction files may optionally have a leading '`c`' to indicating that they are for encoded data. For a more extensive discussion of these conventions, refer to the discussion of `mloss` in Appendix C.

The `mloss` command performs two tasks: it decodes predictions (in the typical situation where the method's predictions were encoded), and it evaluates losses. When `mloss` is invoked it looks to see if it can find encoded prediction files. If so, it decodes the encoded predictions in the files and writes these to files with the initial '`c`' removed from their name. For example, `cguess.A.0` would be decoded into `guess.A.0`. After this, `mloss` looks for the decoded prediction files (which it may just have produced itself), computes the losses using them, and writes them to files called `loss.`$l.n$, where l is the loss function, and n is the instance number. Section 8 discusses how these loss files are analysed.

After running `mloss` you can remove the `train.*`, `test.*`, `targets.*`, and `normalize.*` files (they can be regenerated using `mgendata` if you should ever want them again). You should keep the `Coding-used` and `Test-set-stats` files, as they are used by `mstats` and `minfo`. Usual practice is to also remove any encoded prediction files, keeping only the decoded versions (`guess.*`, `prob.*`, etc.). You can remove the `loss.*` files as well, if you need to save disk space, as they can be regenerated from the decoded prediction files using `mloss`, but it is better if possible to keep the loss files around so that performance comparisons between methods can be made conveniently.

7.6 Submitting your results to the DELVE archive

Once you have documented a learning method, and tried it out on a number of tasks, you may submit the method and the results of applying it for inclusion in the DELVE archive. Other people will then be able to examine your method and results, and compare the results they obtain with their methods to those that you obtained.

You submit a method to the archive by sending the complete directory structure for the method, containing the documentation and tests on all the datasets you have tried. This directory will be placed in the `methods` directory of the DELVE archive. It is also possible to submit new results on existing methods, and new datasets and prototask specifications. For details on how to go about submitting material to the DELVE archive, see Appendix B.

It should be understood that submission of a learning method to the DELVE archive constitutes a form of publication. Once your method has been incorporated into the archive, other researchers will start publishing comparisons of their results with yours. For these comparisons to be intelligible to other researchers, it is necessary for methods to remain in the archive once they have been submitted, in their original form, though you will be able to submit new commentary on the method, explaining any new developments. When a bug is found in the program implementing the method, or a substantial improvement has been made to the learning method, a new updated version may be submitted.

8 ANALYSING THE RESULTS

Suppose we have applied several learning methods to one or more tasks, and used the `mloss` command to evaluate the losses for the predictions they produced, as described in the previous section. We can now use the `mstats` command to compute summaries of losses, and perform tests of statistical significance on observed differences. We hope that in this way we will be able to draw interesting conclusions about the relative performance of these learning methods on these tasks.

The `mstats` command addresses two basic questions. First, how can we compute an estimate of the performance of a method on some task, together with an indication of uncertainly in the estimate? Second, how can we judge whether an observed difference in performance between two methods is statistically significant? This section will explain the theory of the statistical procedures used to answer these questions, and the commands that implement these procedures.

A task is the basic unit for which an expected loss can be defined. However we cannot apply our learning methods directly to tasks, since no specific training cases are associated with a task. Instead we apply our methods to a set of task instances and use the observed performance these particular instances to estimate the expected performance on the task. Note, in particular, that we are generally not interested in the performance on a certain set of test cases, nor in the performance when using particular training sets. Rather, we wish to estimate the expected performance on a new test case when the learning method has been trained on a new training set, both of which are randomly drawn from the same distributions as are the available task instances.

In order to form estimates that are appropriate for this context, we use a set of statistical techniques known as ANOVA (for ANalysis Of VAriance). In each experiment, we try several training sets and for each training set we evaluate the losses for many test cases. The appropriate analysis depends on whether a single common test set or a hierarchical design with disjoint test sets was used. The analysis for the hierarchical model is simplest, so we begin with that.

8.1 Analysing the hierarchical loss model

In the hierarchical model, the losses for a particular set of task instances and a particular learning method are modeled by:

$$y_{ij} = \mu + a_i + \varepsilon_{ij}, \tag{1}$$

where y_{ij} is the loss on training set i and test case j (from the i'th test set — remember, in the hierarchical model there is a separate test set for each training set). There are I instances, each of which contain J test cases. The overall mean loss is given by μ. The parameter a_i is a random variable which explains the variation in losses due to individual training sets. The ε_{ij} parameters account for the residual variation in the losses which are unexplained by the model.

The loss model in eq. (1) captures the notion that individual training sets may not be equally well suited to learn the true relationship of the data. As an example, it may be that one particular training set contains an outlier, which can be accounted for by the corresponding a_i taking on a large positive value. The residuals ε_{ij} account for the variability in losses that are unexplained by the contributions from training set factor a. This variability may be due to variation in the "difficulty" of test cases (either in general, or when a particular training set is used). Any stochastic aspects of the learning method can also contribute to the variability in either the a_i or the ε_{ij}, as discussed below.

We propose using simple independent Gaussian assumptions about the model parameters:

$$a_i \sim \mathcal{N}(0, \sigma_a^2) \qquad \varepsilon_{ij} \sim \mathcal{N}(0, \sigma_\varepsilon^2). \tag{2}$$

These assumptions are primarily based on simplicity requirements for the following analysis of the results. For many loss functions the distributions of the above variables may not be well approximated by Gaussians. However, it is generally believed that the t-test which will be used in the following are fairly robust to violations of normality. Finally, it seems that more sophisticated models become very complicated to analyse, which is why we have settled for this simple model as our standard recommendation in DELVE. Note that the loss files are available, so that a more ambitious analysis can be performed if desired.

The parameter in which we are primarily interested is μ, the overall mean performance of the method. An estimate for it, $\hat{\mu}$, can be found as follows:

$$\hat{\mu} \;=\; \frac{1}{IJ}\sum_{i,j} y_{ij} \;=\; \bar{y} \qquad \mathrm{SD}(\hat{\mu}) \;=\; \Big(\frac{\sigma_a^2}{I} + \frac{\sigma_\varepsilon^2}{IJ}\Big)^{1/2}. \tag{3}$$

This above standard error is in terms of the true values of the σ parameters. In practice, we will have to substitute estimates for the σ parameters.

We introduce the following partial means:

$$\bar{y}_i \;=\; \frac{1}{J}\sum_j y_{ij} \tag{4}$$

and the "mean squared error" for a and ε and their expectations

$$\mathrm{MS}_a \;=\; \frac{J}{I-1}\sum_i (\bar{y}_i - \bar{y})^2 \qquad E[\mathrm{MS}_a] \;=\; J\sigma_a^2 + \sigma_\varepsilon^2 \tag{5}$$

$$\mathrm{MS}_\varepsilon \;=\; \frac{1}{I(J-1)}\sum_i \sum_j (y_{ij} - \bar{y}_i)^2 \qquad E[\mathrm{MS}_\varepsilon] \;=\; \sigma_\varepsilon^2. \tag{6}$$

We can now use the following minimum variance unbiased estimators for the σ values

$$\hat{\sigma}_\varepsilon^2 \;=\; \mathrm{MS}_\varepsilon \qquad \hat{\sigma}_a^2 \;=\; \frac{\mathrm{MS}_a - \mathrm{MS}_\varepsilon}{J}. \tag{7}$$

Unfortunately, $\hat{\sigma}_a^2$ may be negative, in which case we set it to zero. The estimated values may be substituted back into eq. (3) to estimate the uncertainty associated with the average loss.

In order to compare two learning methods, the same model can be applied to the *differences* between the losses from two learning methods, identified by k and k'

$$y_{ij} \;=\; y_{ijk} - y_{ijk'} \;=\; \mu + a_i + \varepsilon_{ij}, \tag{8}$$

with similar Gaussian and independence assumptions as in eq. (2). In this case μ is the expected difference in performance and a_i will be the difference effect due to training sets. Since the overall estimate for the mean can be seen as arising from I independent estimates from each of the instances, we can test whether the estimate for μ is should be considered significantly different from zero using a t-test. This is effectively a paired t-test for whether the expected performance of the two methods is different, with the pairing being performed by modeling the differences in losses. The appropriate t statistic to use is

$$t \;=\; \bar{y}\Big(\frac{1}{I(I-1)}\sum_i(\bar{y}_i - \bar{y})^2\Big)^{-1/2} \;=\; \bar{y}\Big(\frac{\mathrm{MS}_a}{J(I-1)}\Big)^{-1/2}, \tag{9}$$

with $I-1$ degrees of freedom.

In cases where the methods to be analysed have stochastic elements, these give rise to variation in the losses that are not explicitly accounted for in the above analysis. There may be stochastic elements in both the training of the method and in the predictions. For example, many neural network methods are initialized with random weights which gives rise to stochasticity in the training phase.

Although these stochastic elements are not explicitly modeled, the additional variability that they lead to will still show up in this model. Some training conventions should be followed so that it always shows up in the same way. If your learning method is stochastic, you should use a different random number seed for every training set. This will result in the variation due to stochastic training being lumped together with the training set effects in the analysis. Similarly, if your prediction procedure is stochastic, you should use random numbers that are independent for each combination of training set and test case, so that the effects of stochastic predictions will be lumped together with the effects of test cases. Following these conventions, the present analysis will take these stochastic effects into account in a consistent way, but you will not be able to separate the stochastic training and prediction effects from the other sources of variability. In future versions of DELVE we may support explicit evaluation of stochastic training effects, since it may often be of interest to know how much performance varies with things such as random initialization of model parameters. However, this extra information will come at the cost of having to run methods multiple times on identical training and test sets, but with different random number seeds.

8.2 Analysis of experiments with common test sets

When using a common test set the nested model described in the previous section is no longer applicable. Instead, we model the losses for a particular set of task instances and a particular learning method by:

$$y_{ij} \;=\; \mu + a_i + b_j + \varepsilon_{ij}, \tag{10}$$

where y_{ij} is the loss on training set i and test case j. The overall mean loss is given by μ. The parameters a_i and b_j are random variables that explain the variation in losses due to individual training sets and test cases respectively. The ε_{ij} parameters are the residual variation in losses, which are unexplained by the model.

The loss model in eq. (10) captures both effects of training sets a_i and test cases b_j. In the case of a common test set, we have computed the loss for each test case using each of the I training sets, and can thus explicitly estimate the effects of the different test cases in general (as opposed to their effect in combination with a particular training set). As before, the residuals ε_{ij} account for the variability of the losses unaccounted for by the model, such as interactions between training sets and test cases. If the methods being tested have stochastic elements, variation due to this will also show up somewhere, as discussed below.

We again propose using simple independent Gaussian assumptions about the model parameters:

$$a_i \sim \mathcal{N}(0, \sigma_a^2) \qquad\qquad b_j \sim \mathcal{N}(0, \sigma_b^2) \qquad\qquad \varepsilon_{ij} \sim \mathcal{N}(0, \sigma_\varepsilon^2). \tag{11}$$

As before, these assumptions are primarily based on simplicity requirements for the following analysis of the results. For many loss functions the distributions of the above variables may not be well approximated by Gaussians. However, it is generally believed that the F-test which will be used in the following is fairly robust to violations of normality.

We wish to find an estimate, $\hat{\mu}$, for the expected loss of the learning method, as well as a standard error associated with this estimate. As our estimate for the expected loss, we can use the average loss. We can estimate the standard deviation for this estimate based on the model defined by eq. (10) and (11):

$$\hat{\mu} \;=\; \bar{y} \qquad\qquad \mathrm{SD}(\hat{\mu}) \;=\; \left(\frac{\sigma_\varepsilon^2}{IJ} + \frac{\sigma_b^2}{J} + \frac{\hat{\sigma}_a^2}{I}\right)^{1/2}, \tag{12}$$

where I is the number of training sets and J the number of test cases. The expected mean is simply the overall average loss. To evaluate the standard error we first need to estimate the values of the σ parameters. Here and in the following we will use the property that when training sets are set up using the DELVE standard scheme (see section 6), the training sets are disjoint subsets of the entire data set. We introduce the overall mean and the marginal means:

$$\bar{y} \;=\; \frac{1}{IJ}\sum_i \sum_j y_{ij} \qquad\qquad \bar{y}_i \;=\; \frac{1}{J}\sum_j y_{ij} \qquad\qquad \bar{y}_j \;=\; \frac{1}{I}\sum_i y_{ij}, \tag{13}$$

and the the "mean squared error" for a, b and ε and their expectations:

$$\mathrm{MS}_a = \frac{J}{I-1}\sum_i (\bar{y}_i - \bar{y})^2 \qquad E[\mathrm{MS}_a] = J\sigma_a^2 + \sigma_\varepsilon^2 \tag{14}$$

$$\mathrm{MS}_b = \frac{I}{J-1}\sum_j (\bar{y}_j - \bar{y})^2 \qquad E[\mathrm{MS}_b] = I\sigma_b^2 + \sigma_\varepsilon^2 \tag{15}$$

$$\mathrm{MS}_\varepsilon = \frac{1}{(I-1)(J-1)}\sum_i \sum_j \left((y_{ij} - \bar{y}) - (\bar{y}_i - \bar{y}) - (\bar{y}_j - \bar{y})\right)^2 \qquad E[\mathrm{MS}_\varepsilon] = \sigma_\varepsilon^2 \tag{16}$$

Now we can use the empirical values of MS_a, MS_b and MS_ε to estimate values for the σ's:

$$\hat{\sigma}_\varepsilon^2 = \mathrm{MS}_\varepsilon \qquad \hat{\sigma}_b^2 = \frac{\mathrm{MS}_b - \mathrm{MS}_\varepsilon}{I} \qquad \hat{\sigma}_a^2 = \frac{\mathrm{MS}_a - \mathrm{MS}_\varepsilon}{J} \tag{17}$$

These estimators are uniform minimum variance unbiased estimators. Unfortunately however, the estimates for σ_a^2 and σ_b^2 are not guaranteed to be positive, so we set them to zero if they are negative. We can then substitute back these variance estimates in eq. 12 to get an estimate for the standard error for the estimated mean performance.

Note that the estimated standard error $\hat{\sigma}$ diverges if we only have a single training set (as is common practise!). This effect is caused by the hopeless task of trying to empirically estimate a variance based on a single observation. At least two training sets must be used, and probably more if accurate estimates of uncertainty are to be achieved.

Another important question is whether we have good evidence that one learning methods is better than another. To settle this question we again use the model from eq. (11), only this time we model the difference between the losses of the two models, k and k':

$$y_{ijk} - y_{ijk'} = \mu + a_i + b_j + \varepsilon_{ij}, \tag{18}$$

under the same assumptions as above. The question now is whether the estimated overall mean difference $\hat{\mu}$ is significantly different from zero. We can test this hypothesis using a quasi-F test [Lindman, Harold R., "Analysis of Variance in Experimental Design", Springer-Verlag, 1992], which uses the F statistic and degrees of freedom:

$$F_{\nu_1,\nu_2} = (\mathrm{SS}_m + \mathrm{MS}_\varepsilon)/(\mathrm{MS}_a + \mathrm{MS}_b), \quad \text{where} \quad \mathrm{SS}_m = IJ\bar{y}^2 \tag{19}$$

$$\nu_1 = (\mathrm{SS}_m + \mathrm{MS}_\varepsilon)^2/(\mathrm{SS}_m^2 + \mathrm{MS}_\varepsilon^2/((I-1)(J-1))) \tag{20}$$

$$\nu_2 = (\mathrm{MS}_a + \mathrm{MS}_b)^2/(\mathrm{MS}_a^2/(I-1) + \mathrm{MS}_b^2/(J-1)). \tag{21}$$

The result of the F-test is a p-value, which is the probability that given the null-hypothesis ($\mu = 0$) is true, we would observed a difference in average performance of this magnitude (positive or negative), or of a more extreme magnitude. In general, a low p-value produces high confidence that the learning method with better performance in this experiment actually has better performance. If the p-value is not low (say, greater than 0.05), it is not implausible that the method whose performance appeared better in this experiment could actually be worse in reality.

As is the case with the hierarchical model, the common test set model will pick up the variability due to stochastic training and stochastic predictions, even though they are not modeled explicitly. Whenever you apply a stochastic method you should initialize it with a different random seed. The uncertainty due to stochastic training will then be lumped together with the training set effects in the model, and the effects of stochastic predictions will by lumped together with the interaction effects. Thus, the present analysis will take these stochastic effects into account, but you will not be able to separate the effects according to their causes. In future versions of DELVE we may support explicit evaluation of the stochastic training set effect, if the method has been run several times on the same training set with different random seeds.

8.3 Obtaining performance statistics: The `mstats` command

The `mstats` command implements the calculations described in the two previous sections. When used to estimate the expected loss for a particular task, `mstats` is called from within the task directory, or is given the path to such a directory in the DELVE hierarchy. The command will look for `loss.`*l.x* files in this directory, and produce the statistics derived from these files. You may specify the desired loss functions with the '`-l`' option. As an example, we can analyse the performance of a linear regression method in the `demo/age/std.128` task, using absolute error loss:

```
unix> mstats -l A /lin-1/demo/age/std.128
/lin-1/demo/age/std.128
Loss: A (Absolute error)
                                                   Raw value   Standardized

                         Estimated expected loss:   15.0988      0.893246
                     Standard error for estimate:  0.667719     0.0395023

      SD from training sets & stochastic training:   1.49368     0.0883662
 SD from test cases & stoch. pred. & interactions:   13.0755      0.773547

     Based on 8 disjoint training sets, each containing 128 cases and
              8 disjoint test sets, each containing 128 cases.
```

Here, the values reported correspond to the parameters of the model in eq. (1); the overall mean performance is followed by the standard error on this estimate. Also the standard deviations for different sources of variability are printed.

In the second column, the values have been standardized, an a manner appropriate for the loss function. The standardized domain is designed such that a simple *baseline* method has a nearly pre-specified performance. This makes the standardized losses easier to interpret than the raw losses. Note however, that these standardizations are obtained by imagining the baseline methods applied to the union of all *test cases*, such that applying the same simple methods to actual training instances will typically yield a standardized error a little larger than might be expected. (We are forced to accept this, since the standardizations must be the same for all instances in a task, whereas the training sets usually differ.)

For the 'S', 'A', 'Z' and 'Q' loss types, we obtain standardized losses by division by the baseline loss. For squared error loss, the baseline method is prediction of the mean. For absolute error loss, the baseline is prediction of the median. For 0/1-loss, the baseline method is to always predict the majority class, yielding a loss of $1 - p^*$, where p^* is the frequency of the majority class. The baseline method for squared probability loss predicts the empirical class probabilities as observed in the test cases, giving a loss of $1-\sum_i p_i^2$, where p_i is the frequency of the i'th class over the test cases. Thus, these simple methods (which don't utilise the inputs) will have a standardized losses of close to 1.0 and better methods will have losses closer to 0.0.

For log probability loss, the baseline method depends on whether the targets are discrete or continuous. For continuous targets the baseline method produces a Gaussian predictive distribution with mean and variance set to the empirical mean and variance of the test cases. Thus, the standardized losses are obtained by subtracting $\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2}$ from the raw values. For discrete targets, the baseline method sets the class probabilities in accordance with the test frequencies; the standardized values are consequently obtained by subtracting $-\sum_i p_i \log(p_i)$ from the raw values, where p_i is the frequencies of class i in the test set. Thus, methods which perform as well as the baseline methods will have a standardized loss of around 0 and better methods will have negative losses. The negative value of the standardized loss can be interpreted as the amount of information (measured in nats) that the method predicts about the targets relative to the baseline method.

Specifying the '`-c`' option to `mstats` causes it to compare losses with the method named after the '`-c`' option. For example we can compare the linear method with a k-nearest-neighbor method with respect to absolute-error loss on the `/demo/age/std.128` task as follows:

```
unix> mstats -l A -c knn-cv-1 /lin-1/demo/age/std.128
/lin-1/demo/age/std.128
Loss: A (Absolute error)
                                                   Raw value   Standardized

             Estimated expected loss for lin-1:    15.0988      0.893246
         Estimated expected loss for /knn-cv-1:    13.2854      0.785965
                 Estimated expected difference:     1.8134      0.107281
          Standard error for difference estimate:  0.350707     0.0207478

     SD from training sets & stochastic training:  0.505922     0.0299304
SD from test cases & stoch. pred. & interactions:  9.65323      0.571086

    Significance of difference (t-test), p = 0.00129409

    Based on 8 disjoint training sets, each containing 128 cases and
             8 disjoint test sets, each containing 128 cases.
```

A INSTALLING DELVE ON YOUR COMPUTER

DELVE consists of a set of utility programs for assessing learning methods, a number of datasets that can be used for such assessments, and the results of assessing various learning methods on these datasets. To use DELVE you must at least install the utility programs. You will no doubt wish to install some of the datasets as well (unless you wish to use DELVE only on your own data). If you want to compare your learning methods with others, you will also need to install the relevant results.

Requirements

Currently, DELVE requires you to be running some variant of Unix. It has been tested under IRIX 5.3 and Sun-OS 5.4, but should run under other variants without problems.

The datasets and method results have no requirements beyond a Unix file system. The utilities currently require that you have an ANSI-compliant C compiler and an installed copy of **Tcl** (Tool Command Language). **Tcl** is freely available on the Internet and are extremely portable (i.e. it has almost certainly been ported to whatever variant of Unix you are running). If you do not already have **Tcl** installed, copies of the source are available at the ftp site `ftp.smli.com` in the directory `/pub/tcl`.

Obtaining DELVE

The best way to obtain DELVE is to visit our web site: http://www.cs.utoronto.ca/~delve/. You'll find full instructions on getting and building delve there, as well as the latest news on the software, results, and datasets.

If you don't have access to a web browser, the DELVE distribution is available via anonymous ftp, in multiple compressed `tar` files (Unix tape archive format). Currently the files are available on the machine `ftp.cs.toronto.edu` in the directory `/pub/neuron/delve`.

The files are broken down as follows:

1. The source code for the DELVE utilities and documentation is available in one file: `delve-*.tar.gz`.
2. Each dataset is in its own file, where the name of the file is the same as the dataset (with the appropriate suffix added), e.g. `demo.tar.gz`. The easiest way to obtain datasets is from the Delve web site at `http://www.cs.toronto.edu/~delve` or they can be obtained by ftp from `ftp.cs.toronto.edu` in the directory `/pub/neuron/delve/data/tarfiles`
3. The complete results for each method that has been run on DELVE is in its own file, named in a manner similar to the datasets, but with an `all` appended, e.g.

lin-1-all.tar.gz. Results for a particular method and dataset are stored in files with the -all suffix replaced with the dataset name: lin-1-demo.tar.gz. The source code and description for the methods are stored in another tar file with -all replaced with -Source: lin-1-Source.tar.gz. These files are only available from the Delve web site at http://www.cs.toronto.edu/~delve.

A sample ftp session for obtaining DELVE might be as follows[1]:

```
ftp ftp.cs.toronto.edu
cd /pub/neuron/delve
binary
get software/delve-1.1.tar.gz
get data/tarfiles/demo.tar.gz
quit
```

Installation

Before installing the datasets and method results, you must build and install the DELVE utilities as follows:

1. Obtain the distribution file from our ftp site:

   ```
   ftp ftp.cs.toronto.edu
   get /pub/neuron/delve/software/delve-1.1.tar.gz
   bye
   ```

2. Uncompress and untar the distribution using the gunzip utility:

   ```
   gunzip delve-1.1.tar.gz
   tar xvf delve-1.1.tar
   ```

3. Run the configuration script:

   ```
   cd delve-1.1
   ./configure
   ```

   or, for systems that don't recognize #! in shell scripts:

   ```
   cd delve-1.1
   /bin/sh ./configure
   ```

   By default, the configuration script will set things up to be installed in /usr/local. You can change this by specifying a different prefix in the configure command:

   ```
   ./configure --prefix=/your/install/path
   ```

[1]This example illustrates the process for version 1.1; for other versions replace "1.1" by the version/patch number you wish.

You can also add options for a particular `cc` compiler and compiler flags:

```
./configure --with-cc=gcc --with-cflags=-g
```

For a full list of the options `configure` takes, type:

```
./configure --help
```

The `configure` script generates new Makefiles from their respective templates (Makefile.in). If `configure` can't find something, you can make changes to the intermediate `config.status` script, and invoke this script to reconfigure the Makefiles:

```
vi config.status
./config.status
```

As a last resort, you can edit the Makefiles in the current directory and `doc/` by hand and insert the proper paths.

4. Build the libraries and the executables. From the top-level directory type:

```
make all
```

5. Install the executables, libraries, documentation, and script files. From the top-level directory type:

```
make install
```

If you have problems with the installation, you can use a subset of the commands:

```
make install-binaries
make install-libraries
make install-doc
make install-man
```

Once you've installed the utilities, you can install the datasets. This involves simply extracting the files from their tape archives into the proper directory: the installed top-level DELVE data directory. By default this directory is `/usr/local/lib/delve/data`. If you specified a `--prefix` to the `configure` command, replace the `/usr/local` prefix with the path of that directory.

Each tape archive will create a directory with the same base name as the archive file. This directory will contain all the data and specification files DELVE needs to generate the tasks.

```
mv demo.tar.gz /usr/local/lib/delve/data
cd /usr/local/lib/delve/data
zcat demo.tar.gz | tar xvf -
```

If you want to install a dataset in a private directory, you can do the following

1. Create a directory called `delve` in your home directory (or anywhere else, for that matter).
2. In that directory create two more directories: `data` and `methods`.
3. In the `delve/data` directory, untar the data file as described above.

Once you've done that, you can work in your own private delve directory and you will have access to the datasets you've downloaded, as well the ones installed in `/usr/local/lib/delve/data`.

Once you've extracted the data, you can safely remove the tar file.

Setup

Once the software has been installed you can run any of the DELVE commands without further setup. There are, however, 2 environment variables that make the software more flexible

1. DELVE_PATH - (see also appendix C) allows multiple delve directories to be active. It is similar in flavour to the normal Unix PATH environment variable.
2. DELVE_UNCOMPRESS - Set this environment variable to the name of the Unix utility that will *uncompress* files on the "fly", ie it can read compressed files and uncompress them to stdout. If this environment variable is not set, *zcat* is assumed.

B CONTRIBUTING TO THE DELVE ARCHIVE

The ultimate aim of the DELVE project is to collect datasets, implementations of learning methods, and the results of learning experiments from a wide variety of sources. If you have datasets, methods or results which might be of interest to other users you can submit these to the DELVE archive. To make contributions, you can put files on our ftp-server `ftp.cs.utoronto.ca` in the directory `/pub/incoming`, and notify us by email to `delve@cs.utoronto.ca`. The submitted files should preferably conform to the usual DELVE conventions, and be in the form of a compressed `tar` file.

We welcome contributions of datasets to DELVE. We are particularly seeking large real-world datasets, and realistic simulation programs that can be used to create large datasets. Contributions of datasets should be accompanied by descriptions of the data. For real datasets both the data in its original form and in DELVE format should be supplied, as well as descriptions of the relevant context and the attributes recorded. Also suggestions for prototasks together with specifications of prior information should be included. Naturally, proprietary data cannot be included in DELVE without permissions. For simulated and artificial datasets, programs to generate the data should be supplied (if possible) as well as descriptions of the data attributes and suggestions for prototasks and priors, etc.

You may also contribute new learning methods to the archive. Typically, you would also provide results of running your method on various DELVE datasets. You can conveniently submit the whole methods directory pertaining to your method. Also you need to supply a detailed description of your method. Remember, that the description should be detailed enough that someone else can re-implement the method and get comparable results to the ones you might get for any dataset to which the method is applicable. The easiest way to attain this, is if your method is fully automatic. In particular, you should make sure that your description includes:

- implemetational details allowing someone else to re-implemet your method with similar results
- discussion of the role of all parameters of the method
- discussion of the heuristic rules for setting all parameters of the method on the basis of a particular application, including convergence criteria for iterative methods
- detailed discussion of how attributes should be encoded for the method

Finally, it would be convenient if source code of the program implementing you method can be included in DELVE. This may help clarify details of the implementation, help other researches to easily use the methods and help with identifying possible bugs. Authors should take care not to submit implementations containing any parts whose copyrights prohibit public distribution.

For all contributions it should be considered that submission to DELVE is a form of publication, and once contributions are released with DELVE they cannot in general be retracted, since other people may have used them in their research. Therefore, care should be taken to avoid submissions of erroneous material. If a bug should be discovered in a learning method a new corrected version can be submitted under a different name; eg. a buggy version of `loess-1` could be succeeded by a corrected version named `loess-2` — but the original method and its results would be retained in the archive.

You may also submit experimental results using new combinations of methods and datasets that are already in DELVE. If you repeat experiments for which results are already in the archive, it is of interest whether your results were comparable to the earlier results. Notes of such confirmations can be included in DELVE, but for practical reasons only one set of results can be maintained for each method.

All submitted material will be presented in DELVE with the date, name and address (or email) of the contributor(s) allowing further clarifications and collaboration.

C DESCRIPTIONS OF DELVE COMMANDS

This appendix is a detailed reference for the commands that make up the DELVE working environment. You will not necessarily need to use all these commands every day, as some of them are needed only by people creating new datasets or prototasks.

Introduction to DELVE commands

Before describing the individual DELVE commands, we will describe the common aspects that they all share.

Command syntax

All DELVE utilities have a common calling syntax, along the lines of:

`command` [*option*] *argument* ...

Portions enclosed by "[" and "]" are optional; things before "..." can be repeated several times. A vertical bar, "`|`", separates alternatives, only one of which should be present.

The `command` is the name of the DELVE utility, for example `dinfo`. Commands are named so that those that act on dataset directories have names that begin with the letter '`d`', while those that act on method directories have names beginning with '`m`'.

The options are used to modify the behaviour of the `command`. They take the usual Unix form — a dash followed by a single character, for example: "`-h`". Some options also take a single argument. In this case the argument must immediately follow the option, separated by one or more blank spaces, for example: "`-i foo`". If the argument contains spaces it must be quoted: `-i "this arg has spaces"`.

The arguments differ in number and meaning for each command. However, all commands recognize the two following options:

`-h` This causes the command to print a short help message describing its usage and options, after which it exits normally without doing anything else.

`--` This marks the end of the options for the command. The arguments following this one will be treated as regular command arguments even if they start with a `-`.

Data and method path names

Throughout this appendix, we refer to data path names, or *dpaths* and method path names, or *mpaths*. These paths look just like normal unix path names, but they differ in two important aspects:

- Dpaths and mpaths are defined only for files that exist inside the DELVE directory hierarchy. Dpaths point into the `data` part of the hierarchy; mpaths point into the `methods` part.
- A dpath or mpath may identify a file or directory in any of the active `delve` directories. Dpaths and mpaths for directories can even resolve to several locations within different `delve` directories (though this is not supposed to happen for dpaths and mpaths identifying files).

The DELVE hierarchy is the collection of all the active `delve` directories. A `delve` directory must have a name that starts with the five characters "`delve`", and it must have two subdirectories called `data` and `methods`. DELVE decides on the set of active `delve` directories as follows. First of all, if your current working directory is inside a `DELVE` directory, that `delve` directory will be considered active, for as long as you remain in it. In addition, you may provide a list of `delve` directories in your `DELVE_PATH` environment variable. If you do not have such an environment variable, DELVE will use a default list of directories that was fixed when DELVE was installed.

The `DELVE_PATH` environment variable, if set, should contain a colon separated list of DELVE directories. You can use the command "`dinfo -k delve_path /`" to find out which directories are currently in your `DELVE_PATH`, or what the default list of directories is, if you have not set your `DELVE_PATH`.

All the files relating to DELVE datasets, methods, and the results of applying methods to data are kept in the DELVE hierarchy. Files relating to datasets, but not to any particular method, are stored in the `data` part of the hierarchy, and hence have a dpath. Methods and the results of applying methods are stored in the `methods` part of the hierarchy, and hence have an mpath. An mpath that points to a file or directory relating to the results of a method on a particular dataset, prototask, or task can also be used to identify the corresponding information in the `data` part of the hierarchy.

Dpaths and mpaths may be "absolute", starting with a "/" character, or they may be specified relative to the current directory.

Some examples may clarify these naming conventions. Consider the case where you have a directory called `/usr/local/lib/delve`. Inside this directory are the directories `data` and `methods` (and any number of other files and directories). This is a valid DELVE directory.

Inside the `data` directory there is another directory called `demo`. Its absolute unix path name is: `/usr/local/lib/delve/data/demo`; however, its dpath is simply `/demo`. It does not have a mpath because it is not in the `methods` directory. If your current directory were `demo`, its relative Unix path would be "`.`", as would its relative dpath.

Assume the `methods` directory contained `lin-1/demo`. Its absolute unix path name would be: `/usr/local/lib/delve/methods/lin-1/demo`; however, its mpath would be `/lin-1/demo`.

It would not have a dpath because it is not in the `data` directory. If your current directory were `lin-1/demo`, its relative unix path would be "`.`". This would also be its relative mpath. Similarly, the relative path name of `lin-1` in both schemes would be "`..`".

Finally, note that commands that create files inside a directory need to know the *true* pathname for the directory, *not* just a dpath or mpath, since the latter might resolve to more than Unix directory.

`dcheck` — Validate DELVE data files

Note: This command is not yet implemented.

The `dcheck` command is used to verify that the data and specifications for a dataset and its prototasks are legal and consistent.

Command Summary

> `dcheck` [`-l`] [*dpath* | *mpath*]

The path given `dcheck` must identify a dataset, a prototask for a dataset, or a prior file for a prototask. The default is "`.`", the current directory, which must identify a dataset or prototask. If a dataset is specified, its `Dataset.spec` and `Dataset.data` files are checked for errors. Unless '`-l`' is specified, all the prototasks for the dataset are also checked. A single prototask can be checked by giving a path to that prototask. When a prototask is checked, the `Prototask.spec` file is checked for consistency with the `Dataset.spec`, and, unless '`-l`' is specified, all the `.prior` files for the prototask are also checked for errors. A single `.prior` file can be checked by giving its pathname.

The `dcheck` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-l` This causes `dcheck` to run locally. If a dataset is specified, only information on the dataset itself is checked, not information on its prototasks. If a prototask is specified, only information on the prototask itself is checked, not information in prior files for the prototask.

`dgenorder` — Generate random order for a prototask

The `dgenorder` command is used to set up a random ordering of cases in a prototask. This will usually be necessary only for natural or cultivated datasets, not for simulated or artificial ones, for which the order will presumably already be random.

Command Summary

> `dgenorder` [*path*]

The argument given must be the true path name of a prototask directory (*not* the dpath for a prototask); the default is '.', the current directory. The `dgenorder` command creates a file called `Random-order` within this directory that contains a random ordering of cases in the prototask. This file has one line for each case used by the prototask, with each line containing a number from one up to the total number of cases in the prototask.

This `Random-order` file is meant to be used as the ordering file in the specification for a prototask. Use of a random ordering is advisable whenever the prototask is not sequential (where the ordering is meaningful), unless the ordering is already known for certain to be random (as would often be the case for simulated and artificial data).

Note: For prototasks with commonality indexes, or for which training sets are to be stratified, `dgenorder` *will have to do something cleverer, but such things are not implemented yet.*

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`Random-order`
: this is the output file produced by the command.

dgenproto — Generate prototask data files

Prototasks are composed of a subset of the cases in a dataset, and a subset of the attributes in each case. The prototask data file is an intermediate file between the dataset data file and the task data files. It will be generated "on the fly" if it doesn't exist, but some time will be saved each time it is needed if the creator (or installer) of a prototask creates it once and for all, using the `dgenproto` command. (On the other hand, keeping such prototask data files around permanently takes up disk space.)

Command Summary

`dgenproto` [`-i`] [*path*]

The `dgenproto` command generates prototask data files (called `Prototask.data`) inside prototask directories found within the `data` part of the DELVE hierarchy. The data put in these files is taken from the corresponding dataset data files (called `Dataset.data`), which are also found in the DELVE hierarchy.

The *path* argument is the true path of the directory to generate the data in (*not* its dpath, as the dpath might not specify a unique directory). If *path* is a prototask directory — that is, a subdirectory of a DELVE dataset directory — the prototask data file for that prototask alone is generated. If *path* points to a dataset directory, data for all prototasks in the dataset will be generated. If *path* points to the root of the `data` part of the DELVE hierarchy, data files for all prototasks for all datasets will be generated.

The `dgenproto` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-i` This option causes the command to ignore errors when multiple data files are being generated. The command will continue even if one or more of the files cannot be created.

Example

After obtaining the dataset `/demo` from the DELVE archive and placing it in the archive DELVE directory `/usr/local/lib/delve`, the installer will probably wish to run the following command:

```
unix>dgenproto -i /usr/local/lib/delve/data/demo generating:
/usr/local/lib/delve/data/demo/age/Prototask.data
  extracting cases...
  creating file...
generating: /usr/local/lib/delve/data/demo/colour/Prototask.data
  extracting cases...
  creating file...
generating: /usr/local/lib/delve/data/demo/income/Prototask.data
  extracting cases...
  creating file...
generating: /usr/local/lib/delve/data/demo/sex/Prototask.data
  extracting cases...
  creating file...
generating: /usr/local/lib/delve/data/demo/siblings/Prototask.data
  extracting cases...
  creating file...
```

Files Used

`/`*dataset*`/Dataset.spec`
: contains specifications describing the dataset with dpath `/`*dataset*.

`/`*dataset*`/Dataset.data`
: contains all the data for the dataset with dpath `/`*dataset* in a DELVE standard format.

`/`*dataset*`/`*prototask*`/Prototask.spec`
: contains specifications describing the prototask with dpath `/`*dataset*`/`*prototask*.

`/`*dataset*`/`*prototask*`/Prototask.data`
: this is the output file produced by the command.

`dinfo` — Get information about datasets

Although `dls` and `dmore` can be used to browse through the directories and files that define a dataset, the information the files contain is not presented in a very useful format. The `dinfo` command takes all of the information available and puts it into a more accessible format.

Command Summary

> `dinfo` [`-a` | `-k` *keys*] [`-q`] [`-t`] [*dpath* | *mpath*]

The `dinfo` command prints human readable information summaries about the DELVE dataset, prototask or task whose data path is *dpath*. If `dinfo` is given a method path name instead of a data path name, it converts it to a data path name by removing the method prefix. If not specified, the path defaults to '`.`', which must be in the DELVE heirarchy.

For different types of paths, `dinfo` returns different types of information. The *dpath* argument may specify one of the following:

- The root data directory '`/`'; for which the information available includes: the `DELVE_PATH` and a list of all installed datasets.
- A **dataset**, for which the available information includes: the name of the dataset; its origin; its recommended usage; the order cases occur in it; the number of attributes each case contains; a description of these attributes; and a list of all prototasks in the dataset. An example of a dataset path is '`/demo`'.
- A **prototask**, for which the available information includes: the name of the prototask; its origin; the number of cases it contains; the ordering of these case; the number of cases in each test set; the sizes of the training sets for each task; the scheme used for generating test sets; the maximum number of training instances a task may contain; a list of the attributes to be used as inputs for tasks; a list of the attributes to be used as targets; and a list of the available tasks. An example of a prototask path is '`/demo/age`'.
- A **task**, for which the available information includes: the name of the task, the number of cases in each training set; a list of the attributes to be used as inputs; a list of the attributes to be used as targets; the type, relevance, and default coding method for each attribute. An example of a task path is '`/demo/age/std.128`'.

The `dinfo` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-a` Causes the command to print out all the information it knows about the path you are querying. By default, it only prints "interesting" information.

`-k` *keys*
Print only information about fields in the `keys` list. Keys can be obtained with the `-q` option. This is useful, for example, if you are only interested in what prototasks a dataset contains.

`-q` Print, instead of the information normally printed, the keys for the information. For example the command `dinfo -q /demo` would print:

```
dataset origin usage order number-of-attributes prototasks
```

These keys can be used as arguments for the `-k` option. Note that the `-k` and `-a` options affect the behaviour of this option, i.e. `-q` causes the command to print the keys for the information it would print given all other options.

`-t` Print information in a terse format: no headings are printed, and the format is more suitable as input to another program than to a human. The `-t` and `-k` options can conveniently be used together in scripts.

`-v` Print the software version number.

Example

An example of a command to obtain information about the `demo` dataset is:

```
unix> dinfo /demo
Dataset: /demo
Origin: artificial
Usage: development
Order: uninformative
Number of attributes: 5
Prototasks:
        age
        colour
        income
        sex
        siblings
```

Similar results would be obtained if you were in a directory with dpath `/demo` and you typed '`dinfo .`' or '`dinfo`'.

If you only wanted to know what prototasks the dataset contained, and you wanted the output to be machine readable, you could use the command:

```
unix> dinfo -t -k prototasks /demo
age colour income sex siblings
```

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format. The contents of the file are not used, but its existence may be checked.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`*.prior`
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

dls — List contents of DELVE data directories

In the DELVE environment, if a given dpath refers to a directory, it could resolve to multiple true directories. This can be inconvenient if you want to list the files contained in the dpath. To help in this situation, DELVE supplies the `dls` utility for listing all files that reside in directories with a common dpath.

Command Summary

`dls` [`-l`] [*dpath* | *mpath*]

The `dls` command lists the merged contents of all directories with the common data path name *dpath*, or if *dpath* refers to a file, it repeats its name. If `dls` is given a method path name (an mpath) as an argument instead of a data path name, it converts it to a data path name by removing the method prefix.

The output of the command is sorted alphabetically. If no path is given on the command line, it defaults to '`.`', which must be a dpath or an mpath.

The `dls` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-l` Print a long listing, where files are grouped by the directory they are contained in, and the true path name of each directory is printed.

Example

If you wished to list all files in the data directories with dpath `/demo/age` you could use `dls` as follows:

```
unix> dls /demo/age
Prototask.data  Prototask.spec  std.prior
```

Since `dls` allows you to give either a dpath or a mpath as an argument, you could obtain the same results using the command '`dls /lin-1/demo/age`'.

`dmore` — Browse or page through DELVE data files

In a manner similar to the `dls` and `mls` commands, DELVE provides a utility called `dmore` for viewing text files given their dpath. This allows you to look at a file without knowing its true path.

Command Summary

> `dmore` *dpath* | *mpath* ...

The `dmore` command displays the contents of text files that reside in DELVE data directories. The *dpath* arguments are the data path names of the files to be displayed. If `dmore` is given method path names instead of data path names, it converts them to data path names by removing the method prefix. Files are displayed on the terminal, one screenful at a time.

To view the files, `dmore` passes its output through a pager. The default pager is `more`, but it can be changed by setting the environment variable `PAGER` to the name of the command you wish to use.

The `dmore` command recognizes only the `-h` "help" option described in the introduction to this section.

Example

To view the file containing the standard prior information for the `demo/age` prototask, you could use the command:

```
unix> dmore /demo/age/std.prior
1 NLMH binary
3 NLMH integer
4 NLMH real
5 NLMH nominal
2 NLMH real
```

mgendata — Generate task data files

Once you have created the directory hierarchy that will contain the data to train and test your method on, you have to populate it with the actual data. To do this you use the command `mgendata`.

Command Summary

`mgendata` [`-c` *file*] [`-q`] [*path*]

The `mgendata` command generates task data files inside a DELVE method directory from dataset or prototask data files inside a DELVE data directory. The *path* argument is the *true* path of the directory to generate the data in: *not* its mpath (since the mpath could easily resolve to multiple directories). The *path* argument must be a subdirectory of a DELVE methods directory.

If *path* points to a task directory, only data for that task will be generated. If it points to a prototask, data for all tasks in the prototask will be generated. If it points to a dataset, data for all tasks in all prototasks will be generated.

In each task directory, four sets of files are generated. Each set contains the same number of files as there are training instances in the task. Each file in a set has a unique extension '.n', where n is the integer index of the training instance which the file corresponds to.

- Each instance in the task has a **training file** called `train.`n. This file contains cases that are to be used for training your learning method. Each line in the file contains the data for one case. A case contains the encoded representation of all attributes to be used for the task (see Section 7.3 for a description of encoding schemes) printed to the file such that all values are separated by white space. In each case, input attributes come first, followed by target attributes (i.e. each line contains both input values and target values).

- As well as a training file, each instance in the task has a **testing file** called `test.`n. These files contain encoded input attributes for all testing cases (one case per line, all values separated by white space). Testing files do *not* contain target values; they contain only input values.

- For each testing file, there is a corresponding **target file**. This file contains the encoded target attributes for the testing cases, one case per line. Target files are called `targets.`n.

- Data attributes can be encoded using various forms of normalization. To keep track of the normalization constants, a **normalization file** `normalize.`n is created for each instance. This file contains the mean, variance, median, and average absolute deviation from the median for each attribute (one attribute per line).

As well as the above sets of files, two single files are generated: `Coding-used` which will contain a description of the method used to encode each of the attributes (in the form described below), and `Test-set-stats` which will contain statistics derived from the testing data. These files are needed to calculate the losses and evaluate the method performance after it has been run.

The `mgendata` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-c` *file* This option allows you to override the default encoding of attributes. The file should contain one encoding specification per line, containing first an identifier of an attribute (either number or name) followed by the desired encoding. If options are to be given for the encoding, they should appear on the same line, in the form `option=value`, where `option` is the option's name (for example `passive`), and `value` is the value it is to be set to. Multiple option/value pairs may appear on the same line, separated by spaces. No spaces may appear between the option's name and the equal sign, or between the equal sign and the value. All attributes which are not mentioned in the encoding file retain their default encodings.

The valid encodings and their options are described in section 7.3.

`-q` Command should run quietly. Normally `mgendata` prints the names of the files that it is working on.

Example

Suppose that you are in a directory whose mpath is `/lin-1`, and that you have previously run `mgendata`. If you now want to generate training and testing files for the task of the `/demo/age` prototask based on standard prior information and using 256 cases in each training set, you would use the command:

```
unix> mgendata ./demo/age/std.256
./demo/age/std.256
  segmenting cases...
  splitting test inputs and targets...
  encoding instance 0 training data...
  encoding instance 0 test inputs...
  encoding instance 0 test targets...
  encoding instance 1 training data...
  encoding instance 1 test inputs...
  encoding instance 1 test targets...
  encoding instance 2 training data...
  encoding instance 2 test inputs...
  encoding instance 2 test targets...
  encoding instance 3 training data...
  encoding instance 3 test inputs...
  encoding instance 3 test targets...
```

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`Prototask.data`
: contains all the data for the prototask with dpath /*dataset*/*prototask* in a DELVE standard format.

/*dataset*/*prototask*/`*.prior`
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`Random-Order`
: contains the ordering to use when extracting cases from the `Dataset` file and generating the `Prototask` file for the prototask with dpath /*dataset*/*prototask*.

/*method*/*dataset*/*prototask*/*task*/`train.`*n*
: created to hold the encoded inputs and targets for training cases.

/*method*/*dataset*/*prototask*/*task*/`test.`*n*
: created to hold the encoded inputs for test cases.

/*method*/*dataset*/*prototask*/*task*/`targets.`*n*
: created to hold the encoded targets for test cases.

/*method*/*dataset*/*prototask*/*task*/`normalize.`*n*
: created to hold the normalization constants used in the encoding.

/*method*/*dataset*/*prototask*/*task*/`Coding-used`
: created to hold coding actually used in creating the data files for /*dataset*/*prototask*/*task*.

/*method*/*dataset*/*prototask*/*task*/`Test-set-stats`
: created to hold statistics of the testing data for the task with dpath /*dataset*/*prototask*/*task*.

`mgendir` — Generate task directories

When you first want to run a new method on a dataset, you must build the directory tree that will contain all of the training and testing data. You could use a normal Unix command such as `mkdir`, but that would be quite tedious, or you could use the DELVE command `mgendir`.

Command Summary

`mgendir` [`-l`] [`-q`] [*path*]

The `mgendir` command generates directory trees for DELVE datasets, prototasks, or tasks inside a method directory. The *path* argument is the *true* path of the root of the tree to create: *not* its mpath (since the mpath could easily resolve to multiple directories). The *path* must be a subdirectory of a DELVE methods directory.

If path specifies a method, directories for all available datasets are created in the method directory. If it specifies a dataset, directories for all prototasks and tasks of that dataset are generated. If it specifies a prototask or a task, only directories associated with them are generated.

`mgendir` will not complain if parts of the directory tree already exist.

The `mgendir` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-l` This causes `mgendir` to run locally. This means that sub-directories are not created. If you specify a method name, no dataset directories are generated. If you specify a dataset name, no prototask directories are generated. If you specify a prototask name, no task directories are generated.

`-q` Command should run quietly. Normally `mgendir` prints the names of the subdirectories as they are created.

Example

Assuming that you were in a directory with mpath `lin-1`, and you wanted to generate the directory tree files for the entire `demo` dataset, you could use the command:

```
unix> mgendir ./demo
./demo
./demo/age
./demo/age/std.32
./demo/age/std.64
./demo/age/std.128
./demo/age/std.256
./demo/age/std.512
./demo/colour
./demo/colour/std.32

...

./demo/siblings/std.512
```

Similarly, you could generate the directories for just the `age` prototask using the command:

```
unix> mgendir ./demo/age
./demo
./demo/age
./demo/age/std.32
./demo/age/std.64
./demo/age/std.128
./demo/age/std.256
./demo/age/std.512
```

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format. The contents of the file are not used by `dinfo` but its existence is checked.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`*.prior`
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

`minfo` — Get information about learning methods

In a manner similar to `dinfo` the `minfo` command can be used to obtain information about DELVE methods.

Command Summary

`minfo` [`-a` | `-k` *keys*] [`-q`] [`-t`] [*mpath*]

The `minfo` command prints human readable information summaries about the DELVE method, dataset, prototask or task whose DELVE method path name is given by the *mpath* argument. If no path is specified, it defaults to '`.`', which must be a DELVE methods directory. The `minfo` command returns information about datasets, prototasks, or tasks *as they were used by the method*, not as they appear in the `data` directory. For example, when the mpath argument specifies a dataset, the list of prototasks returned by `minfo` will contain only those the method was run on, not all of the ones available to be run on.

For different types of paths, `minfo` returns different types of information. The *mpath* argument may specify one of the following:

- Information available for the root data directory `/` includes: the `DELVE_PATH` and a list of the methods that have been run on DELVE datasets.
- For a `method` directory, the information available includes a list of all datasets the method has been run on. An example of a method path is '`/lin-1`'.
- For a `dataset`, the available information includes all the information returned by `dinfo` for datasets, with the exception that the list of prototasks includes only those that the method has been run on. An example of a dataset path is '`/lin-1/demo`'.
- For a `prototask`, the available information includes all the information returned by `dinfo` for prototasks, with the exception that the list of tasks includes only those that the method has been run on. An example of a prototask path is '`/lin-1/demo/age`'.
- For a `task`, the available information includes all the information returned by `dinfo` for a task, with the exception that the actual coding method used for the data is printed, *not* the default method. An example of a task path is '`/lin-1/demo/age/std.128`'.

The `minfo` command recognizes the same options as `dinfo`.

Example

An example of a command line that could be used to obtain information about the `demo` dataset as it was used by lin-1 would be:

```
unix> minfo /lin-1/demo
Dataset: /demo
Origin: artificial
Usage: development
Order: uninformative
Number of attributes: 5
Prototasks:
        age
        income
```

Similar results would be obtained if your current working directory had the mpath `/lin-1/demo`, and you typed '`minfo .`' or '`minfo`'.

If you only wanted to know what `demo` prototasks the `lin-1` method was run on, and you wanted the output to be machine readable, you could use the command:

```
unix> minfo -t -k prototasks /lin-1/demo
age income
```

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format. The contents of the file are not used, but its existence may be checked.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`*.prior`
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

/*method*/`*`
: used to get the list of datasets the method with mpath /*method* has been run on.

/*method*/*dataset*/`*`
: used to get the list of prototasks from the dataset with dpath /*dataset* that the method with mpath /*method* has been run on.

/*method*/*dataset*/*prototask*/`*`
: used to get the list of tasks from the prototask with dpath /*dataset*/*prototask* that the method with mpath /*method* has been run on.

/*method*/*dataset*/*prototask*/*task*/`Coding-used`
: contains the coding scheme used to generate the task data files for the task with mpath /*method*/*dataset*/*prototask*/*task*.

`mloss` — Generate task loss files

Once you have run a method on a task and produced predictions, you will need to calculate the loss from the true targets and your predictions. Loss functions are discussed in detail in Section 5.

Command Summary

`mloss` [`-i` *instances*] [`-l` *losses*] [`-q`] [*path*]

The `mloss` command decodes prediction files and generates loss files. The *path* argument for `mloss` is the *true* path of the directory to generate the decoded prediction and loss files in (*not* its mpath, since the mpath could easily resolve to multiple directories). The prediction files used to generate the losses can reside in any directory with the same mpath. If not specified, *path* defaults to the current directory.

If *path* points to a task directory, only loss files for that task will be generated. If it points to a prototask, loss files for all tasks in the prototask will be generated. If it points to a dataset, loss files for all tasks in all prototasks will be generated. Finally, if it points to a method, loss files will be generated for all tasks that the method has been run on.

The `mloss` command can generate losses using using any of the five following measures. Each measure has a single-character code associated with it:

`A` Absolute error loss.

`S` Squared error loss.

`Z` Zero-one loss.

`L` Negative log-probability loss.

`Q` Squared-probability loss.

You must write your predictions to files with special names in order to get them evaluated with the intended loss measure. Depending on the type of the prediction, the file may have one of three *root* names: `guess` for files that contain guesses for the targets, `prob` for files containing class probabilities and `ptarg` for files containing the probabilities (or densities) of the true targets under the method's predictive distribution. In general the methods do not need to read the `targets` files, with the exception of the situations where the method produces a predictive distribution and the targets are real, integer or angular. In these cases there seems to be no general convenient way of conveying the predictive distribution; instead the method must itself evaluate the probability (or density) of the true target under the predictive distribution and write this to a prediction file with the `ptarg` root name.

A number of prefixes and extensions may be added to these root names. The instance number is always added as an extension, e.g. `guess.3`. Optionally, the name of a spcific loss function can also be specified as an extension, e.g. `guess.S.3`. If no loss function is specified then the predictions can be applied to any loss function for which that particular root is meaningful (although, loss specific predictions always take precedence over generic ones). Some prediction files may have a '`c`' prefixed to their name, indicating that the predictions are in the coded domain. This will normaly be the case for the files your method writes, since it only sees that training and test files which have been encoded. The '`c`' prefix can be applied to files with the `guess` or `ptarg` root names. Files which contain probabilities (or densities) may have an '`l`' prefixed their name indicating that the predictions are made in the (natural) log domain. Some examples of names of prediction files are `cguess.A.0`, `prob.3` and `clptarg.L.7`. Note, that the prefixes and extensions must follow the order given in these examples.

The first task `mloss` performs is decoding the predictions. It places the decoded predictions in files with the same names as those containing the coded ones with the initial '`c`' removed. For example the decoded predictions for `cguess.A.0` would be placed in `guess.A.0`. Similarly, the decoded predictions for `clptarg.7` would be placed in `lptarg.7`.

Once the predictions have been decoded, `mloss` generates loss files based on those predictions and the target values. The losses are placed in files named `loss.`*l*.*n*, where where the '*l*' and '*n*' characters have the same meanings as above. For loss files the '.*l*' extension is not optional (as the values in the file are defined by the loss function).

The prediction files used to generate the losses for a particular measure are found by first looking for all prediction files specific to that loss (i.e. files that have the appropriate '.*l*' extension). If even one such file exists for a given measure, then only files with that extension are used to generate the losses. If no such files exist for the given loss, `mloss` looks for prediction files where the loss was not specified (i.e. files with the appropriate root name, but without the '.*l*' extension). It then uses these to calculate the loss. If none of these files exist, a warning message is printed, and no loss files for that measure are generated.

A table of the allowed combinations of target types and loss functions is given in section 5.2. Whenever predictions are made in files with the `prob` root, `mloss` automatically normalises the probabilites to sum to unity. However, this is not possible for predictions with the `ptarg` root, so users should be careful to ensure that their method's predictive distribution is correctly normalised when using these predictions.

The `mloss` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-i` *instances* This allows you to specify which training instances you want to evaluate the loss for. It should be a list of integer values or the string '`all`'. It's default value is '`all`'.

-l *losses* This allows you to specify the loss functions mloss attempts to evaluate. You can specify any combination of A, S, L, Q, and Z. By default, mloss attempts to evaluate all approriate types.

-q Command should run quietly. Normally mloss prints the names of the files that it is working on.

Note: The mloss *command does not yet support the specialised loss functions discussed in section 5.3.*

To generate the loss files, mloss must temporarily decode the target files. Because of this, the target files *must* be present in the mpath of the task.

Files Used

/*dataset*/Dataset.spec
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/Dataset.data
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format. The contents of the file are not used by dinfo but its existence is checked.

/*dataset*/*prototask*/Prototask.spec
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/*.prior
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

/*method*/*dataset*/*prototask*/*task*/targets.*n*
: contains the coded targets for the n'th training instance of the task with dpath /*dataset*/*prototask*/*task*/, as made by the method with mpath /*method*.

/*method*/*dataset*/*prototask*/*task*/[c]guess[.*l*].*n*
: contains the guesses for the (optianally coded) targets of the n'th training instance of the task with dpath /*dataset*/*prototask*/*task*/, as made by the method with mpath /*method*, with an optionally specified loss function (A, S or Z).

/*method*/*dataset*/*prototask*/*task*/[c][l]ptarg[.L].*n*
: contains the (optionally log) probabilities (or densities) for the (optionally coded) targets of the n'th training instance of the task with dpath /*dataset*/*prototask*/*task*/, as made by the method with mpath /*method*. The L loss function may optionally be specified.

/*method*/*dataset*/*prototask*/*task*/[l]prob[.*l*].*n*
: contains the (optionally log) probabilities of the targets for the n'th training instance of the task with dpath /*dataset*/*prototask*/*task*/, as made by the method with mpath /*method*, with an optionally specified loss function (L or Q).

/*method*/*dataset*/*prototask*/*task*/`Coding-used`
contains the coding scheme used to encode the data for the method with mpath /*method*/*dataset*/*prototask*/*task*.

/*method*/*dataset*/*prototask*/*task*/`normalize.`*n*
contains the normalizing constants used to encode the data for the method with mpath /*method*/*dataset*/*prototask*/*task*/

`mls` — List contents of DELVE method directories

As with dpaths, if a given mpath refers to a directory, it could resolve to multiple true directories. To list all files in directories with a common mpath, DELVE supplies the `mls` utility.

Command Summary

`mls` [`-l`] [*mpath*]

The `mls` command lists the merged contents of all directories with the common method path name `mpath`, or if `mpath` refers to a file, it repeats its name.

The output of the command is sorted alphabetically. If no path is given on the command line, it defaults to '`.`', which must be a DELVE method directory.

The `mls` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-l` Print a long listing, where files are grouped by the directory they are contained in, and the true path name of each directory is printed.

Example

If you wished to list all files in the method directories with mpath `/lin-1/demo` you could use the `mls` command as follows:

```
unix> mls /lin-1/demo
age income
```

mmore — Browse or page through DELVE method files

The command corresponding to `dmore` for viewing DELVE method files is `mmore`.

Command Summary

`mmore` *mpath ...*

The `mmore` command displays the contents of text files that reside in DELVE method directories. The `mpath` arguments are the method path names of the files to be displayed. Files are displayed on the terminal, one screenful at a time.

To view the files, `mmore` passes its output through a pager. The default pager is `more`, but it can be changed by setting the environment variable `PAGER` to the name of the command you wish to use.

The `mmore` command recognizes only the `-h` "help" option described in the introduction to this section.

Example

To view the source program for the implementation of the `lin-1` method, you could use the command:

```
unix> mmore /lin-1/Source/lin-1.c
/* lin-1.c: Robust linear method for regression.
 *
 * Reads training examples from "train.n", test inputs from "test.n" and
 * targets from "targets.n".  Produces point predictions in "cguess.n" and
 * densities of targets under a predictive distribution in "clptarg.L.n".
 * Here "n" is the instance number, supplied as a command argument. Handles
 * badly conditioned cases where inputs are (close to) linearly dependent.
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <stdio.h>
...
```

mstats — Calculate or compare loss statistics

Once the loss files for a given method have been generated, you can see how well or poorly the method performed, either in absolute terms, or in comparison to another method. This is done with the `mstats` command.

Command Summary

> `mstats` [`-c` *methods*] [`-i` *base*] [`-l` *losses*] [*mpath*]

The `mstats` command prints summary statistics about a method's loss files, or compares the loss files of two methods, and prints summary statistics about the comparison. The *mpath* argument is the DELVE method path name of the method whose losses are to be summarized. If it is omitted, it defaults to '`.`' which must be a DELVE method directory.

Loss files are generated by `mloss` and are normally named `loss.`$l.n$, where l is a single character describing the loss function used to generate the file, and n is an integer describing the training instance the loss file corresponds to. See the description of `mloss` for further details.

Full details of the statistics used to summarize the losses are described in Section 8; however a quick summary is given here.

When summarizing the loss files for a single method, `mstats` returns:

- the estimated expected loss.
- the standard error of the estimate.
- the standard deviation of the losses between training sets.
- the standard deviation of the losses between testing cases (if applicable).
- the standard deviation of the residuals.

Both the raw values and a standardized version of these terms are printed. See section 8.3 for a discussion of the standardization used.

When comparing the performance of two methods, `mstats` returns:

- the estimated expected loss for both methods.
- the estimated expected difference in the losses.
- the standard error of the estimate for the difference.
- the standard deviation of the losses between training sets.

- the standard deviation of the residuals.

Both the raw values and a standardized version of these are printed. The report also includes a probability describing the significance of the differences of the two loss estimates (calculated using either a T-test, or F-test as appropriate).

Both reports include a listing of how many training sets and cases, and test sets and cases were used to calculate the statistics.

The `mstats` command recognizes the `-h` "help" option described in the introduction to this section, as well as:

`-c` *methods* This flag causes `mstats` to compare the current method with the selected methods. Summary statistics about the differences of the loss files in `mpath` and those of the other methods are returned. The `method` arguments may be proper mpaths for a method, e.g. `/lin-1`, or you may omit the initial slash.

`-i` *base* This allows you to change the base name of the input loss files. These files are generated by `mloss` and normally have the base name `loss`.

`-l` *losses* This allows you to change what loss functions `mstats` attempts to summarize. You can specify any combination of `A`, `S`, `L`, `Q`, and `Z`. By default, it attempts to summarize all of them, using whatever files exist. `Mstats` will only print warnings if it attempts to summarize a loss measure for which there are no loss files.

Example

Suppose you wished to know how well the `/lin-1` method did on the `/demo/age/std.128` task, using the squared error loss measure. You could use the command:

```
unix> mstats -l S /lin-1/demo/age/std.128
/lin-1/demo/age/std.128
Loss: S (Squared error)
                                                Raw value   Standardized

                       Estimated expected loss:    400.73      0.819745
                   Standard error for estimate:   28.6111     0.0585277

     SD from training sets & stochastic training:    40.898     0.0836622
 SD from test cases & stoch. pred. & interactions:   790.029       1.61611

     Based on 8 disjoint training sets, each containing 128 cases and
              8 disjoint test sets, each containing 128 cases.
```

If you then wanted to compare its performance to the `/knn-cv-1` method, you could use:

```
mstats -c knn-cv-1 -l S /lin-1/demo/age/std.128
/lin-1/demo/age/std.128
Loss: S (Squared error)
                                                      Raw value   Standardized

               Estimated expected loss for lin-1:      400.73      0.819745
           Estimated expected loss for /knn-cv-1:     368.003      0.752798
                   Estimated expected difference:      32.727     0.0669473
          Standard error for difference estimate:      14.075     0.0287922

     SD from training sets & stochastic training:     27.6978     0.0566594
 SD from test cases & stoch. pred. & interactions:    323.515      0.661792

     Significance of difference (t-test), p = 0.052988

     Based on 8 disjoint training sets, each containing 128 cases and
              8 disjoint test sets, each containing 128 cases.
```

Files Used

/*dataset*/`Dataset.spec`
: contains specifications describing the dataset with dpath /*dataset*.

/*dataset*/`Dataset.data`
: contains all the data for the dataset with dpath /*dataset* in a DELVE standard format. The contents of the file are not used, but its existence is checked.

/*dataset*/*prototask*/`Prototask.spec`
: contains specifications describing the prototask with dpath /*dataset*/*prototask*.

/*dataset*/*prototask*/`*.prior`
: files contain prior information to be used when generating tasks for the prototask with dpath /*dataset*/*prototask*.

/*method*/*dataset*/*prototask*/*task*/`loss.`*l*`.`*n*
: contains the losses for the `n`'th training instance of the task calculated using the loss function *l* with dpath /*dataset*/*prototask*/*task*/, as made by the method with mpath /*method*.

/*method*/*dataset*/*prototask*/*task*/`Test-set-stats`
: contains statistics of the testing data for the task with dpath /*dataset*/*prototask*/*task*.

D GLOSSARY OF DELVE TERMINOLOGY

absolute-error loss — A loss function for regression tasks in which the loss is the absolute value of the difference between the guess and the target. When there is more than one target, the absolute loss is the sum of such absolute differences for all the targets.

angular attribute — An attribute whose value is an angle or some other circular quantity, such as time-of-day. By default, such attributes are encoded as the sine and the cosine of the equivalent angle, so as to avoid introducing an artificial discontinuity.

artificial dataset/prototask — A dataset generated by a program (usually with a random component) on the basis of some mathematical specification, without any connection with a real-world problem. Prototasks based on such datasets are also referred to as artificial.

assessment dataset — A dataset that is recommended for use in formally assessing learning methods.

attribute — One of the quantities associated with each case in a dataset. The dataset specification classifies attributes as **controlled** or **uncontrolled**, according to how their values were determined. The prior information for a task will characterize attributes as **binary**, **nominal**, **ordinal**, **integer**, **angular**, or **real**.

binary attribute — A categorical attribute that can take on exactly two possible values (not counting missing values) — for example, an attribute with possible values of "male" and "female", or one with values of "0" and "1".

categorical attribute — An attribute that takes on values from some finite set. The targets for a classification prototask must be categorical. The prior information for a task further characterizes categorical attributes as **binary**, **nominal**, or **ordinal**, and may designate one of the values as **passive**.

case — A collection of attribute values that all apply to the same thing. For example, in a dataset of medical tests on patients, a case might consist of all the test results for a particular patient.

censored value — An indication of the value for an attribute in a case that says only that the value is known to be greater than or equal to (or less than or equal to) a specified value. In DELVE dataset files, a censored value is recorded as "*number*:" (if the actual value is greater than or equal to *number*), or as ":*number*" (if the actual value is less than or equal to *number*).

classification prototask/task — A prototask (or task) in which all the target attributes are **categorical**.

controlled attribute An attribute whose values were fixed by the investigators who gathered the data. For example, the amount of fertilizer applied to an agricultural test plot would likely be a controlled attribute.

common testing scheme An experimental set-up in which a single common test set is used to assess the performance of a method with all the training sets; distinguished from a **hierarchical** testing scheme.

commonality index An integer that may be associated with a case, indicating that the case has something in common with the other cases with the same commonality index. For example, in a dataset where a case records features of a handwritten digit, all the digits written by one person might have the same commonality index.

cultivated dataset/prototask A dataset that comes from a real-world source, but has no real-world context, having been collected or selected for the purpose of creating a DELVE dataset rather than from any genuine interest. Natural datasets that have been modified in some way, such as by adding extra noise, are also in this class. Prototasks based on cultivated datasets are also classified as cultivated, as are prototasks that are based on natural datasets but which have little resemblance to the original purpose for which the data was gathered.

dataset A collection of data, consisting of a number of **cases**, each associated with the values of several **attributes**. Datasets are classified as **natural**, **cultivated**, **simulated**, or **artificial** according to the data's relationship to the real world. DELVE also distinguishes among **development datasets**, **assessment datasets**, and **historical datasets**, on the basis of recommended usage.

default encoding The encoding of an attribute that DELVE will use by default if a particular learning method does not specify otherwise. The default encoding is based on the **prior information** for the task.

dependency (between cases) A situation where knowledge of the values of the targets in one case would be informative regarding the values of the targets in other cases with the same **commonality index**, or that are nearby in a **sequential prototask**. Here, it is assumed that the inputs in all cases are already known, and that the true nature of the general relationship between inputs and targets is also fully understood — ie, the dependency is between the "noise" or "residuals" in the related cases (the part of the variation not explainable by the relationship between inputs and targets).

development dataset A dataset that is recommended for use in developing learning methods. To avoid bias, such datasets should not also be used in formal assessments of performance.

encoding (of an attribute) The way that DELVE represents the value of an attribute (usually as one or more numbers) when generating data files for task instances. The encoding to use is part of the specification of a learning method, but DELVE provides a **default encoding** that will often be appropriate.

estimated expected loss An estimate for the expected loss of a learning method on some task, based on the results of a learning experiment. At present, DELVE's estimates are simply the average loss over training sets and test cases tried. Each estimate has an associated **standard error**, that is indicative of its likely accuracy.

expected loss The expected performance of a **learning method** on some **task** as judged by a specified **loss function**, the expectation being with respect to random selection of a training set and a test case. Put another way, the performance the method would achieve on average if it were applied a great many times to training sets and test cases obtained from the same source as the actual dataset. Note that the true expected loss cannot be determined exactly, but an **estimated expected loss** can be computed from the results of a learning experiment.

guess (for a test case) A prediction for the targets in a test case consisting of a single value for each target, these values being chosen by the learning method with the aim of minimizing the expected **absolute-error**, **squared-error**, or **0-1 loss**. If a **no-guess penalty** has been specified, a learning method also has the option of making no guess for a particular target in a particular test case.

hierarchical testing scheme An experimental set-up in which separate, non-overlapping test sets are used to assess the performance of a method as trained on different training sets; distinguished from a **common** testing scheme.

historical dataset A dataset that is included in the DELVE archive because it has been used to assess learning methods in the past, but which is not recommended for future use, except when there is a need to make comparisons with past results in the literature.

input attribute For a particular **prototask**, an attribute that is available for use in predicting the values of the **target attributes** in the same case, but whose values do not themselves need to be predicted.

informative ordering An ordering of cases in a dataset (as originally obtained) that conveys information that may be significant — for instance, an ordering of data on patients by date of admission to hospital.

integer attribute An attribute whose values are integers, and for which the prior information does not specify an interpretation as a **categorical attribute**. Note that the range of an integer attribute may be restricted (eg, to the positive integers).

learning experiment An experiment in which the performance of one or more **learning methods** on one or more **tasks** is assessed by applying the learning methods to several **task instances**. DELVE defines a standard scheme for conducting such experiments.

learning method A well-defined procedure for discovering relationships among attributes on the basis of prior information and empirical data, and for making predictions for new cases using the relationships learned. Learning can be **supervised** or **unsupervised**.

log-probability loss A loss function used with methods whose predictions are predictive distributions over target values. The log-probability loss is minus the log (base e) of the probability or probability density of the target values. This loss function can be used with any task, but for tasks with real-valued targets (such as regression tasks), the loss must be computed by the learning method itself, rather than by DELVE.

loss function A measure of how far off a prediction is, given the actual values of the targets. The standard loss functions DELVE supports are **squared-error loss**, **absolute-error loss**, **0-1 loss**, **squared-probability loss**, and **log-probability loss**. Specialized loss functions can also be constructed that incorporate a **no-guess penalty**, or that are based on a **loss matrix**.

loss matrix For a prototask with one categorical target, a matrix that specifies the loss that is suffered for every possible combination of a guessed value for the target and an actual value for the target. For each actual value of the target, the loss suffered when no guess is made may also be specified.

missing value An indicator that the actual value of an attribute for a particular case is not known. In DELVE dataset files, a missing value starts with a question mark; this may be followed by other characters to distinguish values that are missing for different reasons.

natural dataset/prototask A dataset that comes from a real-world source, and for which there is or was a real interest in learning relationships among the attributes (for either scientific or engineering purposes). A prototask is classified as natural if it is based on a natural dataset, and involves learning relationships that were of interest to the original investigators.

no-guess penalty The loss suffered when a learning method whose predictions take the form of guesses decides to make no guess for a particular target in a particular case.

nominal attribute A categorical attribute with at least three possible values (not counting missing values) for which the prior information does not specify any natural ordering of the values. An example might be an attribute with values of "beef", "pork", and "lamb".

non-standard task instance A task instance in which the training and test sets are not selected according the standard DELVE scheme.

noise level (for a target) The proportion of the variation in a target attribute that is not explained by the variation in the input attributes, even given full knowledge of the true relationship between inputs and targets.

order (of a dataset) An indicator of whether the order of cases in the dataset (as originally obtained) is **informative** or **uninformative**.

ordinal attribute A categorical attribute with at least three possible values (not counting missing values) for which the prior information specifies a natural ordering of the values. An example might be an attribute with values of "no-education", "primary-education", "secondary-education", and "post-secondary-education".

p-value (for a comparison) When comparing the estimated expected loss of two learning methods on some task, the probability that a difference in estimated expected loss of equal or greater magnitude than that observed might arise by chance even if the true expected loss for the two methods is the same. A low p-value may give one confidence that the apparently better method actually is better.

passive value A value for a categorical attribute that is expected on the basis of prior information to play a role different from that of the other value or values of the attribute, with the passive value being associated with a lack of positive influence. If a binary attribute has values of "hockey-player" and "not-a-hockey-player", for example, "not-a-hockey-player" might be regarded as passive.

performance (of method) In the DELVE context, usually the predictive performance of the method on some task, formalized in terms of **expected loss**. One might also be interested in the computational performance of a method (its time and memory requirements).

prediction (for a test case) The output of a learning method for a test case, embodying the method's prediction regarding the likely values of the targets in this case. Predictions may be either single-valued **guesses** for the target values, or **predictive distributions** that say how likely each of the possible target values is.

predictive distribution A probability distribution produced by a learning method as its prediction for the values of the targets in a test case. For classification tasks, the predictive distribution consists of a finite number of probabilities, which may be output in explicit form. For tasks with real targets, the predictive distribution consists of a probability density function, which DELVE does not attempt to represent explicitly; instead, the learning method itself calculates the **log-probability loss** based on its internal representation of the predictive distribution.

prior information Information regarding the the possible or likely nature of the relationship being learned that is obtained from the prior knowledge of the investigator (or a surrogate for the investigator), rather than from the data itself.

prototask A supervised learning problem associated with a **dataset**, consisting of a set of **target attributes** that are to be predicted, a set of **input attributes** that may be used in making predictions, and a pool of **cases** that are seen by the learning method. A prototask can have many associated **tasks**, in which the available prior information and the size of the training set are also specified. Prototasks are classified as **natural**, **cultivated**, **simulated**, or **artificial** according to their relationship to the real world. **Regression** and **classification** prototasks are distinguished by the nature of their target attributes.

range (of attribute) The set of **values** that an attribute could conceivably take on, including the set of **missing values** that are allowed for the attribute.

real attribute An attribute whose values are real numbers, and for which the prior information does not specify an interpretation as an **angular**, **integer**, or **categorical attribute**. Note that the range of a real attribute may be restricted (eg, to some interval).

relevance (of an input) The degree to which variation in an input attribute (within its observed range) affects the values of the target attributes. Put another way, the degree to which knowledge of the input attribute's value helps in predicting the values of the targets, given that the true nature of the relationship between inputs and targets is known.

regression prototask/task A prototask (or task) in which all the targets attributes are **real**.

sequential prototask A prototask based on a dataset with an **informative ordering** in which this ordering has been preserved, and in which there may therefore be **dependencies** between nearby cases.

simulated dataset/prototask A dataset generated by a program (usually with a random component) that simulates some actual phenomenon in a realistic fashion. Prototasks based on such datasets are also referred to as simulated.

squared-error loss A loss function for regression tasks in which the loss is the square of the difference between the guess and the target. When there is more than one target, the squared-error loss is the sum of such squared differences for all the targets.

squared-probability loss A loss function for classification tasks, used with methods whose predictions are predictive distributions over target values. The squared-probability loss is the square of one minus the probability assigned to the correct target value, plus the sum of the squares of the probabilities assigned to all the other target values. Squared-probability loss cannot be used when there is more than one target attribute.

standard error (of estimate) The standard deviation of an estimate (eg, of expected loss) that would be observed if the experiment on which the estimate is based were to be repeated many times with new data randomly obtained from the same source as the actual data. (In practice, the standard errors quoted are themselves estimates, since the true standard deviation usually depends on unknown quantities.)

standard task instance One of the task instances that are used in DELVE's standard scheme for learning experiments.

stratified training set A training set for a classification task in which training cases have been selected in such a way that each of the different possible target values appears the same number of times.

supervised learning Learning whose goal is to discover the relationship of certain **target attributes** to other **input attributes**, and on this basis predict the values of the target attributes for a new case for which only the input attributes are known.

target attribute For a particular **prototask**, an attribute whose values are to be predicted, based on the values of other **input attributes** in the same case.

task A specific learning context for a **prototask**, consisting of the **prior information** regarded as being available for use in learning, and the size and nature of the **training set** that will be provided. A task is sufficiently well specified that each learning method has a well-defined **expected loss** for a given task and loss function. A task may be associated with many **task instances**, in which particular training sets and test cases are specified.

task instance A particular **training set** for a **task**, to which a learning method can be applied as part of a learning experiment, together with a **test set** that is used to evaluate the accuracy of the learning method's predictions. In DELVE's scheme for learning experiments, a set of **standard task instances** are defined; it is possible to define **non-standard task instances** as well.

test case A **case** that is used to evaluate the performance of a learning method applied to a particular **task instance**.

test set The set of all **test cases** for a particular **task instance**. Note that although a task instance will normally include many test cases, the predictions for the targets in each test case are to be made without using information from any other test case.

training case A **case** that is part of the **training set** made available to a learning method.

training set The set of **training cases** that are made available to a learning method in a particular **task instance**.

uncontrolled attribute An attribute whose values were not fixed by the investigators who gathered the data, but by some random process. For example, the amount of rainfall on various agricultural test plots would be an uncontrolled attribute (even though the investigators influence the amount of rainfall by where they decide to put the plots).

uninformative ordering An ordering of cases in a dataset (as originally obtained) that does not convey any useful information — for instance, a random ordering, or an ordering that is sorted by the value of one of the attributes.

unsupervised learning Learning whose goal is to discover the relationships amongst all attributes, without distinguishing some attributes as "inputs" and others as "targets". DELVE does not currently handle methods for unsupervised learning, but may do so in future.

value (of an attribute) The actual numerical or non-numerical quantity taken on by an **attribute** in a particular **case**. Some cases may have attributes with **missing values**, for which the actual value is not known, or with **censored values**, for which the actual value is known only to be beyond some given value.

0-1 loss A loss function for classification tasks in which the loss is 0 when a guess matches the actual target value and 1 when the guess does not match the actual target value. When there is more than one target, the total loss is the number of mis-matches between guesses and actual values.