PLearn User's Guide

How to use the PLearn Machine-Learning library and tools

August 3, 2022

Copyright © 1998-2006 Pascal Vincent, Yoshua Bengio
Copyright © 2002 Julien Keable
Copyright © 2003 Réjean Ducharme
Copyright © 2004 Martin Monperrus
Copyright © 2005 Olivier Delalleau

Permission is granted to copy and distribute this document in any medium, with or without modification, provided that the following conditions are met:

1. Modified versions must give fair credit to all authors.
2. Modified versions may not be written with the aim to discredit, misrepresent, or otherwise taint the reputation of any of the above authors.
3. Modified versions must retain the above copyright notice, and append to it the names of the authors of the modifications, together with the years the modifications were written.
4. Modified versions must retain this list of conditions unaltered, and may not impose any further restrictions.

Contents

Introduction

PLearn is an Open Source C++ library and framework with an associated collection of software tools developped and used for *research* in *statistical machine learning*.

The emphasis here is on *"research"*: it was built by researchers mostly for their own use, i.e. not too much with the general public in mind. It is not for the faint of heart, and you are more likely to find here exotic algorithms at the forefront of research, rather than a comprehensive collection of all the "standard proven and well tested" algorithms of Machine Learning. This being said, if you want to program your new idea of a learning algortihm in efficient modern C++, the PLearn framework offers a solid foundation.

Chapter 1

Tutorial

This chapter is a tutorial that will walk you through the basic concepts from a user-level perspective.

We assume you have a copy of the plearn distribution, and a working plearn executable accessible through yout PATH. All the files in this tutorial are in `examples/Tutorial/` so you should first `cd` to this directory.

1.1 The plearn Commands and Help

Usual PLearn executables such as `plearn` or `plearn_light` are typically called in command-line fashion.

```
valhalla:~/PLearn/examples/Tutorial> plearn
plearn 0.92.0  (Jun 21 2005 12:04:50)
Type 'plearn help' for help


valhalla:~/PLearn/examples/Tutorial> plearn help
plearn 0.92.0  (Jun 21 2005 12:04:50)
To run a .plearn script type:                       plearn scriptfile.plearn
To run a command type:                              plearn command [ command a

To get help on the script file format:              plearn help scripts
To get a short description of available commands:   plearn help commands
To get detailed help on a specific command:         plearn help <command_name>
```

```
To get help on a specific PLearn object:           plearn help <object
To get help on datasets:                           plearn help dataset
```

The `plearn` executable can be invoked either with a *PLearn script* (more on that later) or with a *PLearn command*.
To get the list of available commands:

```
valhalla:~/PLearn/examples/Tutorial> plearn help commands
plearn 0.92.0  (Jun 21 2005 12:04:50)
To run a command, type:  % plearn command_name command_arguments

Available commands are:
FieldConvert    :  Reads a dataset and generates a .vmat file based on

autorun :  watches files for changes and reruns the .plearn script
help    :  plearn command-line help
htmlhelp        :  Output HTML-formatted help for PLearn
jdate   :  Convert a Julian Date into a JJ/MM/YYYY date
ks-stat :  Computes the Kolmogorov-Smirnov statistic between 2 matrix c
learner :  Allows to train, use and test a learner
read_and_write  :  Used to check (debug) the serialization system
run     :  runs a .plearn script
server  :  Launches plearn in computation server mode
test-dependencies       :  Compute dependency statistics between input
test-dependency :  Compute dependency statistics between two selected c
vmat    :  Examination and manipulation of vmat datasets


For more details on a specific command, type:
  % plearn help <command_name>
```

PLearn commands accept a number of arguments that are command specific. Very often the first argument is itself a sub-command…

`help` is actually a *PLearn command*! Thus we can ask help on help!

```
valhalla:~/PLearn/examples/Tutorial> plearn help help
plearn 0.92.0  (Jun 21 2005 12:04:50)
```

```
*** Help for command 'help' ***
plearn command-line help
help <topic>
Run the help command with no argument to get an overview of the system.
```

The `help` command can give detailed help on any available PLearn *command*, as well as on any PLearn *object class*.

There is an on-line html version of the help provided by the help command... See *PLearn help on user-level commands and objects* on the PLearn homepage...

1.2 Data Matrices

Machine-learning algorithms learn from data and are then used for prediction on new data. In this tutorial, we'll concentrate on the simplest and most usual form of data samples: vectors in $\mathbb{R}^d$.

A dataset of l samples is then simply an $l \times d$ matrix of reals. In PLearn such datasets are implemented through the concept of a **VMatrix** (or **VMat** in short).

A **VMat** is essentially:

- A $l \times d$ matrix of reals (l is its *length*, d its *width*),
- optionally with an associated *fieldname* for each column (or *field*),
- optionally with associated *inputsize, targetsize, weightsize, extrasize*
- optionally with strings associated to specific values of a given column

The *inputsize, targetsize, weightsize, extrasize* are important information for learning algorithms, as they specify which part of each row is to be considered the known input (the first *inputsize* elements), which part is the target to predict (the next *targetsize* elements), and whether or not they are followed by a sample weight (*weightsize = 0 or 1*). The *extrasize* fields can be used to store any extra information.

For the traditional *tasks* of statistical machine learning, we have the following conventions regarding datasets and "sizes":

- **regression:**
  inputsize = number of known inputs ("variables", "factors" or "features", i.e.

dimensioality of “x”)
targetsize = number of values to predict (i.e. dimensionality of “y”)

- **classification:**
  inputsize = number of known inputs
  targetsize = 1: the target is the class number (between 0 and nclasses-1)

- **density estimation:**
  inputsize = dimensionality of x
  targetsize = 0

For ex., let's create a simple data set for 1D regression, i.e. to predict a real y from a real x. Open a file `1d_reg.amat` with your favorite editor, and and enter the following text definint a 5×2 matrix:

```
#size: 5 2
#: x y
#sizes: 1 1 0 0

0     3
0.5   4
1     5
2     6
3     7.5
```

This represents a 5×2 matrix whose columns are named *x* and *y*, and whose *inputsize=1, targetsize=1, weightsize=0, extrasize=0*.

1.3 Viewing Data Matrices

Data matrices can be manipulated with the PLearn command *vmat*:

```
valhalla:~/PLearn/examples/Tutorial> plearn help vmat
plearn 0.92.0  (Jun 21 2005 12:04:50)
*** Help for command 'vmat' ***
Examination and manipulation of vmat datasets
Usage: vmat info <dataset>
       Will info about dataset (size, etc..)
   or: vmat fields <dataset> [name_only] [transpose]
```

```
    To list the fields with their names (if 'name_only' is specified, the i
    and if 'transpose' is also added, the fields will be listed on a single
or: vmat fieldinfo <dataset> <fieldname_or_num> [--bin]
    To display statistics for that field
or: vmat bbox <dataset> [<extra_percent>]
    To display the data bounding box (i.e., for each field, its min and max
or: vmat cat <dataset> [<optional_vpl_filtering_code>]
    To display the dataset
or: vmat sascat <dataset.vmat> <dataset.txt>
    To output in <dataset.txt> the dataset in SAS-like tab-separated format
or: vmat view <dataset>
    Interactive display to browse on the data.
or: vmat stats <dataset>
    Will display basic statistics for each field
or: vmat convert <source> <destination> [--cols=col1,col2,col3,...]
    To convert any dataset into a .amat, .pmat, .dmat or .csv format.
    The extension of the destination is used to determine the format you wa
    If the option --cols is specified, it requests to keep only the given c
    (no space between the commas and the columns); columns can be given eit
    number (zero-based) or a column name (string).  You can also specify a
    such as 0-18, or any combination thereof, e.g. 5,3,8-18,Date,74-85
    If .csv (Comma-Separated Value) is specified as the destination file, t
    following additional options are also supported:
      --skip-missings: if a row (after selecting the appropriate columns) c
                       one or more missing values, it is skipped during exp
      --precision=N:   a maximum of N digits is printed after the decimal p
      --delimiter=C:   use character C as the field delimiter (default = ',
or: vmat gendef <source> [binnum1 binnum2 ...]
    Generate stats for dataset (will put them in its associated metadatadir
or: vmat genvmat <source_dataset> <dest_vmat> [binned{num} | onehot{num} |
    Will generate a template .vmat file with all the fields of the source p
    with the processing you specify
or: vmat genkfold <source_dataset> <fileprefix> <kvalue>
    Will generate <kvalue> pairs of .vmat that are splitted so they can be
    The first .vmat-pair will be named <fileprefix>_train_1.vmat (all sourc
    and <fileprefix>_test_1.vmat (the first 1/k of <source_dataset>
or: vmat diff <dataset1> <dataset2> [<tolerance> [<verbose>]]
    Will report all elements that differ by more than tolerance (defauts to
    If verbose==0 then print only total number of differences
or: vmat cdf <dataset> [<dataset> ...]
```

```
       To interactively display cumulative density function for each fi
       along with its basic statistics
   or: vmat diststat <dataset> <inputsize>
       Will compute and output basic statistics on the euclidean distan
       between two consecutive input points

<dataset> is a parameter understandable by getDataSet:
Dataset specification can be one of:
 - the path to a matrix file (or directory) .amat .pmat .vmat .dmat or
 - ...
```

OK, too many subcommands here, but let's concentrate on the few ones you're most likely to use:

```
valhalla:~/PLearn/examples/Tutorial> plearn vmat info 1d_reg.amat
plearn 0.92.0  (Jun 21 2005 12:04:50)
5 x 2
inputsize: 1
targetsize: 1
weightsize: 0
extrasize: 0


valhalla:~/PLearn/examples/Tutorial> plearn vmat fields 1d_reg.amat
plearn 0.92.0  (Jun 21 2005 12:04:50)
FieldNames:
0: x
1: y


valhalla:~/PLearn/examples/Tutorial> plearn vmat fieldinfo 1d_reg.amat
plearn 0.92.0  (Jun 21 2005 12:04:50)
[------------------------------------ Computing statistics (5) -------
[....................................................................
Field #1:  y     type: UnknownType
nmissing: 0
nnonmissing: 5
sum: 25.5
mean: 5.09999999999999964
stddev: 1.74642491965729807
```

```
min: 3
max: 7.5


valhalla:~/PLearn/examples/Tutorial> plearn vmat cat 1d_reg.amat
plearn 0.92.0  (Jun 21 2005 12:04:50)
0 3
0.5 4
1 5
2 6
3 7.5
```

If you want to browse the data matrix interactively, you can use the command `plearn vmat view 1d_reg.amat` (This is most useful for huge data sets.... `plearn` need to be compiled with `curse`.)

You can also see the points graphically by using the `pyplot` script `pyplot plot_2d 1d_reg.amat`

1.4 vmat File Formats

The *V* in **VMatrix** stands for *Virtual*, because **VMatrix** is a C++ virtual base class of which there are several concrete derived classes (do a `plearn help VMatrix` if you want to see how many...).

Accordingly, there are several file formats that represent real data matrices, distinguished by their file extension:

extension	**format description**
.amat	Simple ascii format
.pmat	Simple raw binary format with 1 line ascii header
.dmat	Directory containing compressed binary data (possibly split in several files for huge data)
.vmat	Contains the specification of a C++ VMatrix object (in PLearn's ascii serialisation format)
.pymat	Python preprocessing code that generates the specification of a C++ VMatrix object (a la .vmat)

In addition, several of those tend to have an associated `.metadata` directory, that will contain associated data that is not held within the file itself (for ex: fieldnames, inputsize and targetsize, field statistics, etc...)

You can convert from any format to `.amat, .pmat, .dmat, .csv` with PLearn command `vmat convert`:

```
plearn vmat convert 1d_reg.amat 1d_reg.pmat
plearn vmat view 1d_reg.pmat
```

1.5 PLearn Objects, Their Serialization and Specification

PLearn is first and foremost a C++ class library. PLearn also provides a mechanism to serialize such objects to and from files (i.e. write a representation of an in-memory object to a file, or later reload such a saved object from that file). PLearn serialization supports both an ASCII human-readable format (`plearn_ascii`), and a more efficient binary format (`plearn_binary`).

As a result of this capability, it is also possible to *specify* a PLearn object by simply writing its ASCII serialized form by hand. This is basically what a `.vmat` file contains: *the ASCII serialised form of a C++ subclass of VMatrix*.

For example, create a file `selected_rows.vmat` with the following content:

```
SelectRowsVMatrix(
  source = AutoVMatrix( specification = "1d_reg.amat" ),
  indices = [ 1 1 3 0 3 4],
  inputsize =   1,
  targetsize =  0,
  weightsize =  1
);
```

The serialised form of most PLearn objects, as can be seen here, is:

```
ObjectName(
  optionname = optionval
  optionname = optionval
  ...
)
```

Note that in `plearn_ascii` format, in general, spaces, newlines, commas and semicolons are ignored (any sequence of those is considered a single separator).

There is typically a one to one correspondance between an object's *options* (in its serialised form) and the fields of the corresponding C++ object. A PLearn object often has many options, but they always have a default value, so that there is no need to explicitly set those for which the default value is fine.

The above `.vmat` specifies an object of type `SelectRowsVMatrix`, which is a sort of vmat that will select desired rows from another "source" vmat. `selected_rows.vmat` will thus be an *altered view* of `1d_reg.amat`, for which we also change the values of *inputsize, targetsize, weightsize*.

```
valhalla:~/PLearn/examples/Tutorial> plearn vmat info selected_rows.vmat
plearn 0.92.0  (Jun 22 2005 19:42:18)
6 x 2
inputsize: 1
targetsize: 0
weightsize: 1
```

```
valhalla:~/PLearn/examples/Tutorial> plearn vmat cat selected_rows.vmat
plearn 0.92.0  (Jun 22 2005 19:42:18)
0.5 4
0.5 4
2 6
0 3
2 6
3 7.5
```

Help on any plearn object can be obtained, as usual, by invoking `plearn help` *objectclass*. This will output a commented serialised object, with all its build *options* and their default value. This help is also available in online html form. For ex. try:

```
plearn help SelectRowsVMatrix
```

This makes for a good starting point for writing a `.vmat` (or `.plearn`), as you can issue:

```
plearn help SelectRowsVMatrix > mymat.vmat
```

and then edit the file to your liking (removing unnecessary options that are to keep their default value, etc...)

`.vmat` is not the only file extension associated with specifications of PLearn objects in serialised form. Here are the other extensions you may encounter:

extension	format description
.vmat	specification of a subclass of VMatrix in `plearn_ascii` serialization format (with rudimentary macro-processing)
.plearn	specification of any PLearn object in `plearn_ascii` format (with rudimentary macro-processing)
.psave	serialized PLearn object in `plearn_ascii` or `plearn_binary` format (does not undergo macro-explansion)
.pymat	Python preprocessing code that generates the `plearn_ascii` specification of a VMatrix subclass
.pyplearn	Python preprocessing code that generates the `plearn_ascii` specification of any PLearn object

While `.vmat` and `.plearn` support some rudimentary macro-processing, this is deprecated in favor of the power of the Python preprocessing of `.pymat` and `.pyplearn` files. We will get back to this later.

1.6 plearn Learner

The concept of a learning algorithm in PLearn is implemented through the **PLearner** class. Conceptually a **PLearner** is an object that:

- can be *trained* using a training data set (which contains input and target)
- can then be *used* by *computing outputs* corresponding to new inputs
- can be *tested* on a test set (containing input and target) and report statistics on some *costs* (ex: classification error rate).
- can be saved to and loaded from file (like any PLearn object)

The meaning and form of the output vector are learner-dependant, but in PLearn we try to respect the following convention for standard tasks:

- **regression:** output is the *predicted* target (i.e. same dimension as terget)

- **classification:** target is a scalar between 0 and *nclasses-1*; output is a vector of length *nclasses* giving a score for each class (the higher, the more likely).
- **density estimation:** output is typically the log of the estimated density at x (but this can be controlled by an option, if you want for ex. the density instead of the log).

For ex. let us create a file `linreg.plearn` with the following content:

```
LinearRegressor(
  weight_decay = 1e-6
  )
```

LinearRegressor is a subclass of **PLearner** and as such, it can be trained, used, tested with the `plearn learner` command:

```
valhalla:~/PLearn/examples/Tutorial> plearn help learner
plearn 0.92.0  (Jun 22 2005 19:42:18)
*** Help for command 'learner' ***
Allows to train, use and test a learner
learner train <learner_spec.plearn> <trainset.vmat> <trained_learner.psave>
  -> Will train the specified learner on the specified trainset and save the r
     trained_learner.psave

learner test <trained_learner.psave> <testset.vmat> <cost.stats> [<outputs.pma
  -> Tests the specified learner on the testset. Will produce a cost.stats fil
     command) and optionally saves individual outputs and costs

learner compute_outputs <trained_learner.psave> <test_inputs.vmat> <outputs.pm

learner compute_outputs_on_1D_grid <trained_learner.psave> <gridoutputs.pmat>
  -> Computes output of learner on nx equally spaced points in range [xmin, xm
     in gridoutputs.pmat

learner compute_outputs_on_2D_grid <trained_learner.psave> <gridoutputs.pmat>
  -> Computes output of learner on the regular 2d grid specified and writes th

learner compute_outputs_on_auto_grid <trained_learner.psave> <gridoutputs.pmat
  -> Automatically determines a bounding-box from the trainset (enlarged by 5%
     regular 1D grid of <nx> points or a regular 2D grid of <nx>*<ny> points.
```

```
     bbox to determine the bounding-box by yourself, and then invoke le

learner analyze_inputs <data.vmat> <results.pmat> <epsilon> <learner_1>
  -> Analyze the influence of inputs of given learners. The output of e
     when each input is perturbed, so as to estimate the derivative of
     is averaged over all samples and all learners so as to estimate th
     file, are stored the average, variance, min and max of the derivat

The datasets do not need to be .vmat they can be any valid vmatrix (.am
```

To train this linear regressor on our data-set `1d_reg.amat` and save the resulting trained learner as `linreg_trained.psave` we issue the following command:

```
plearn learner train linreg.plearn 1d_reg.amat linreg_trained.psave
```

To get the predicions of the trained learner on new data that was not in the training set, (for ex. $x = 0.25, x = 1.5, x = 2.5$) we can create a file `1d_reg_test.amat` containing

```
#size: 3 1
#: x
#sizes: 1 0 0
0.25
1.5
2.5
```

and issue the commands

```
valhalla:~/PLearn/examples/Tutorial> plearn learner compute_outputs lin
plearn 0.92.0  (Jun 22 2005 19:42:18)
[-------------------------------------- Using learner (3) ----------
[.................................................................

valhalla:~/PLearn/examples/Tutorial> plearn vmat cat 1d_reg_test_output
plearn 0.92.0  (Jun 22 2005 19:42:18)
3.58836232959270118
5.3879309848394854
6.82758590903691243
```

We thus get the predictions output by the learner.

To see the learnt parameters of the trained learner, we can examine the file `linreg_trained.psave` :

```
*1 ->LinearRegressor(
include_bias = 1 ;
cholesky = 1 ;
weight_decay = 9.99999999999999955e-07 ;
output_learned_weights = 0 ;
weights = 2  1  [
3.22844859854334443
1.43965492419742724
]
;
AIC = -2.53047027031051597 ;
BIC = -2.6866951053368755 ;
resid_variance = 1 [ 0.0596271276504959716 ] ;
expdir = "" ;
stage = 0 ;
n_examples = 5 ;
inputsize = 1 ;
targetsize = 1 ;
weightsize = 0 ;
forget_when_training_set_changes = 0 ;
nstages = 1 ;
report_progress = 1 ;
verbosity = 1 ;
nservers = 0  )
```

We can see that there are many more *options* in the saved learner than what we specified. In particular the *weights* option gives us the parameters tuned by the learning (i.e. the regression weights).

For 1D regression problems such as this, we can easily display the predicted output along the real line:

```
pyplot 1d_regression 1d_reg.amat linreg.plearn
```

This will train the given learner on the given training set, compute the output prediction along the real line, and plot the result.

1.7 A density estimation example

Let's make a new data matrix `spiral.vmat` containing:

```
VMatrixFromDistribution(
  distr = SpiralDistribution(),
  # nsamples=10600,
  nsamples=200,
  inputsize=2,
  targetsize=0,
  weightsize=0);
```

```
valhalla:~/PLearn/examples/Tutorial> plearn vmat view spiral.vmat
```

```
valhalla:~/PLearn/examples/Tutorial> pyplot plot_2d spiral.vmat
```

Now let's make `parzen.plearn`

```
ParzenWindow(
sigma_square = 0.06;
outputs_def = "d"  ;
);
```

and check how well it estimates the density:

```
valhalla:~/PLearn/examples/Tutorial> pyplot 2d_density spiral.vmat par
```

1.8 A classification example

See the older tutorial.2

Note that we can make a classification data set by issuing

```
pypoints 2d_classif.amat
```

1.9 Running a Full Experiment: PTester

The class `PTester` is used to wrap the action of running a complete experiment in a single runnable `PLearn` object. The goals of this class are as follows:

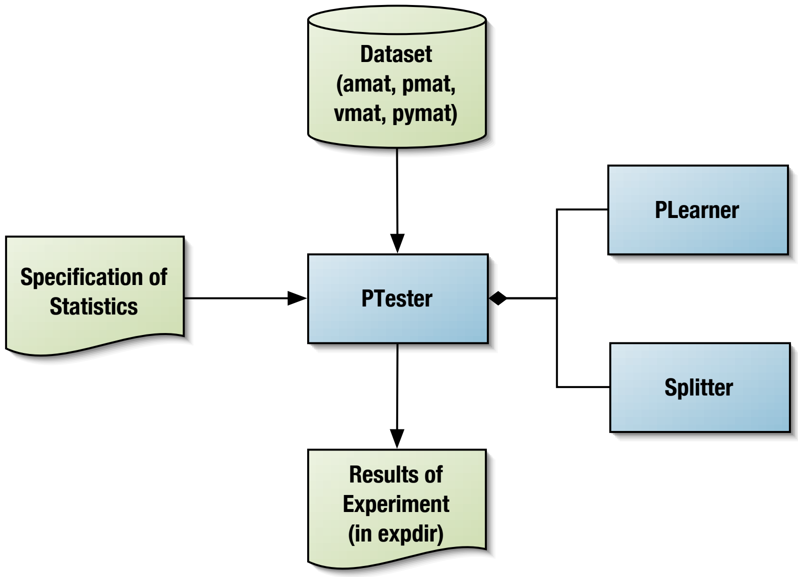

Figure 1.1: Relationship among the classes taking part in the experiment run by PTester. The PLearner must actually be an instance of a class derived from PLearner; likewise, the Splitter must be an instance of a class derived from Splitter. The desired statistics are specified as options of the PTester object, and the experiment results are stored in the experiment directory.

- Take a dataset (either a .amat, .vmat, .pmat or .pymat) and *split it* into one or more training and test sets. We shall denote the k-th such split as *Split-k*.
- For each split, the `PTester` trains an associated learner (which must be of a class derived from `PLearner`) on the training set of the split.
- For each split, the `PTester` then tests the trained learner on the testset data. Afterwards, it can compute performance statistics and report.

The relationship among the various parts is illustrated in Figure 1.1.

1.9.1 Process Underlying PTester

The process underlying PTester is illustrated in Figure 1.2.

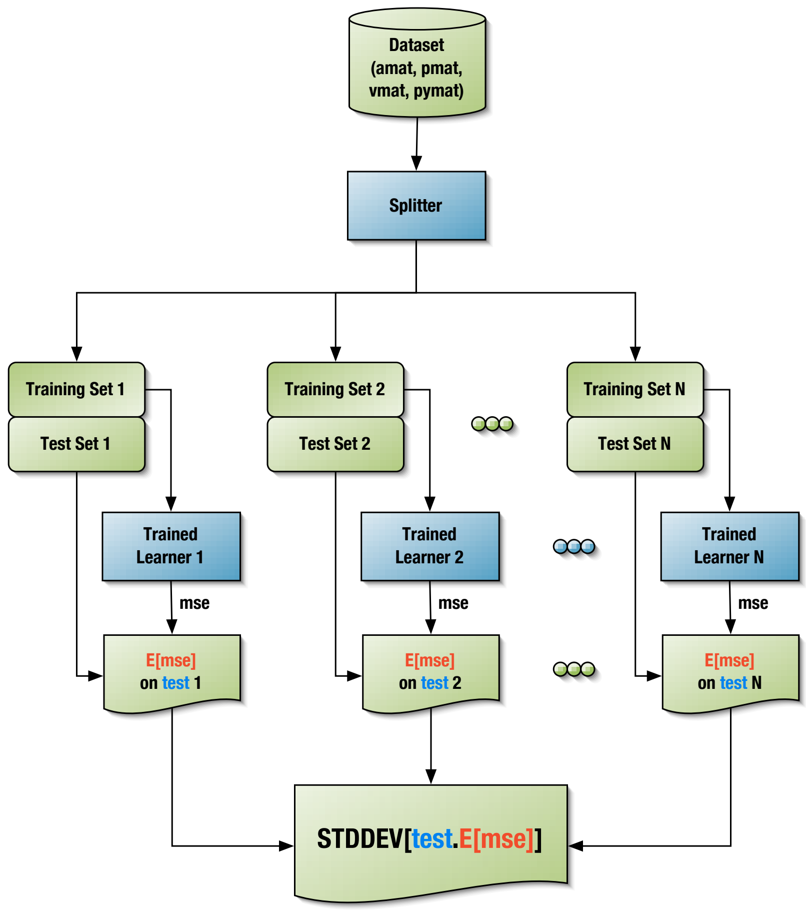

Figure 1.2: Process Underlying PTester

1.9.2 Experiment Directory

`PTester` executes its experiment in a designated *experiment directory* (often abbreviated `expdir`, the name of the option used to specify it within the `PTester` object.) This directory should be empty at the beginning of the experiment (if it does not exist, it is created automatically); if it contains the results of a previous experiment, `PTester` complains loudly and exits immediately.

Note that if you run your experiments from `.pyplearn` scripts, a synthetic experiment directory of the form `expdir_YYYY_MM_DD_HH:MM:SS` is created for you automatically, which pretty much guarantees uniqueness of the name.

1.9.3 Example

(See the `.pyplearn` tutorial.)

1.10 Python Preprocessing

See the **pyplearn tutorial**

Chapter 2

Older Tutorial

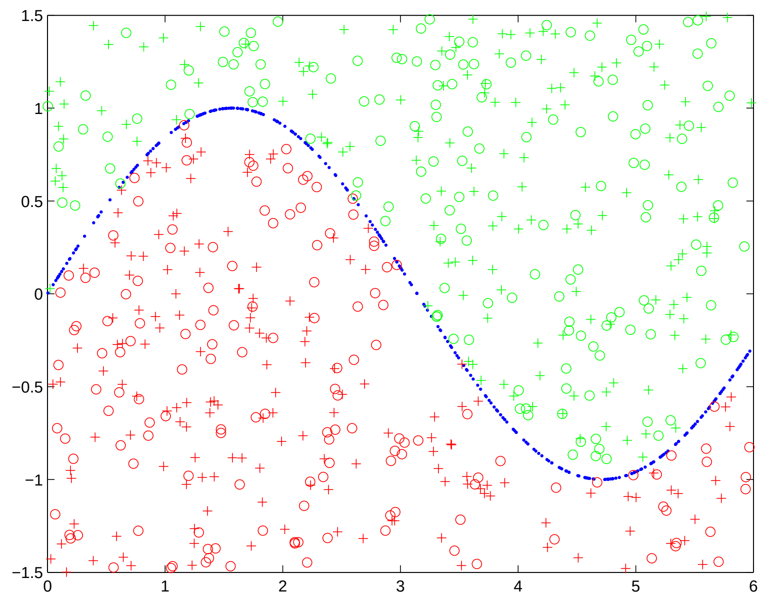

Figure 2.1: In red, the first class. In green, the second one. In blue, the analytic decision boundary. The train examples are 0, the test ones +

2.1 Introduction

PLearn is an open source software for machine learning, with numerous features. It can be used as an runnable software or as a library. This tutorial will help you to discover what is PLearn and how to use it. I assume that:

- you have basic concepts in machine learning.
- you have basic concept in object programming.
- you have a PLearn binary that runs.

2.2 A basic classification problem

2.2.1 First steps

We consider the following classification problem. In a 2-D space, we have two classes. The problem is represented on figure 2.1.

The data are generated with a Matlab script, called task1.m[1].The script generates boundary.amat, train.amat, test.amat, and space.amat.

Now, let's train a neural network on this task with PLearn.

We create the PLearn script, a kind of configuration file:

```
#!plearn

PTester(
  # string: Path of this experiment's directory in which to save all experimen
  expdir = "expdir-nnet";

  # VMat: The dataset to use for training/testing (will be split according to
  dataset = AutoVMatrix(specification="UCI_pima-indians-diabetes all" inputsiz

  # TVec< string >: A list of global statistics we are interested in.
  # These are strings of the form S1[S2[dataset.cost_name]] where:
  #   - dataset is train or test1 or test2 ... (train being
  #     the first dataset in a split, test1 the second, ...)
  #   - cost_name is one of the training or test cost names (depending on data
  #     by the underlying learner (see its getTrainCostNames and getTestCostNa
  #   - S1 and S2 are a statistic, i.e. one of: E (expectation), V(variance),
  #     S2 is computed over the samples of a given dataset split. S1 is over t
  statnames = [ ]
  # TVec< TVec< string > >: A list of lists of masks. If provided, each of the
  # If not provided the statnames are those in the 'statnames' list. See the c
  statmask = [ [ "test#1-2#" ] [ "*.E[stable_cross_entropy]" "*.E[binary_class
  # PP< Splitter >: The splitter to use to generate one or several train/test
  splitter = TrainTestSplitter(
    test_fraction = .10
    append_train = 1
  ) ;

  # PP< PLearner >: The learner to train/test
  learner =
    NNet(
      # int:     number of hidden units in first hidden layer (0 means no hidd
      nhidden = 10  ;
      # int:     number of output units. This gives this learner its outputsiz
```

[1]provided in `Plearn/examples`, as all the other sources for this tutorial

```
#     It is typically of the same dimensionality as the target fo
#     But for classification problems where target is just the cl
#     usually of dimensionality number of classes (as we want to
#     vector, one per class)
noutputs = 1  ;
# double:     global weight decay for all layers
weight_decay = 0.0  ;
# string:     what transfer function to use for ouput layer?
#     one of: tanh, sigmoid, exp, softplus, softmax, log_softmax
#     or interval(<minval>,<maxval>), which stands for
#     <minval>+(<maxval>-<minval>)*sigmoid(.).
#     An empty string or "none" means no output transfer functior
output_transfer_func = "sigmoid"  ;
# Array< string >:     a list of cost functions to use
#     in the form "[ cf1; cf2; cf3; ... ]" where each function is
#       mse (for regression)
#       mse_onehot (for classification)
#       NLL (negative log likelihood -log(p[c]) for classificatio
#       class_error (classification error)
#       binary_class_error (classification error for a 0-1 binary
#       multiclass_error
#       cross_entropy (for binary classification)
#       stable_cross_entropy (more accurate backprop and possible
#       margin_perceptron_cost (a hard version of the cross_entro
#       lift_output (not a real cost function, just the output fo
#     The first function of the list will be used as
#     the objective function to optimize
#     (possibly with an added weight decay penalty)
cost_funcs = [ "stable_cross_entropy" "binary_class_error" ]  ;
# PP< Optimizer >:     specify the optimizer to use
optimizer =
  GradientOptimizer(
    # double:     the initial learning rate
    start_learning_rate = 0.05
    # double:     the learning rate decrease constant
    decrease_constant = 0.001  ;
  );
# int:     how many samples to use to estimate the avergage gradi
#     0 is equivalent to specifying training_set->length()
batch_size = 0  ;
```

```
      # int: The stage until which train() should train this learner and retur
      # The meaning of 'stage' is learner-dependent, but for learners whose
      # training is incremental (such as involving incremental optimization),
      # it is typically synonym with the number of 'epochs', i.e. the number
      # of passages of the optimization process through the whole training set
      # since the last fresh initialisation.
      nstages = 100  ;
      # int: Level of verbosity. If 0 should not write anything on cerr.
      # If >0 may write some info on the steps performed along the way.
      # The level of details written should depend on this value.
      verbosity = 10  ;
      seed = 12345
    );

)
```

We call this file `NNnet.plearn`.

Now we have to specify the dataset. In a classification problem, the dataset is a set of examples and their associated classes. We have already created such a file in Matlab, called train.amat. Now we have to specify to PLearn that the two first columns of "train.amat" contain the data, and the last column the class label, which is in $-1, 1$, corresponding to the output of a tanh.

All this tasks are made with the following file, called `train.vmat`:

```
AutoVMatrix(
specification = "train.amat"
inputsize = 2
targetsize = 1
weightsize = 0
)
```

Now, let's train the network with the folllowing command:

`plearn learner train NNet.plearn train.vmat final.psave`

Then test on the original space:

`plearn learner compute_outputs final.psave space.amat out.pmat`

We need two additionnals commands to view the result of the network with matlab:

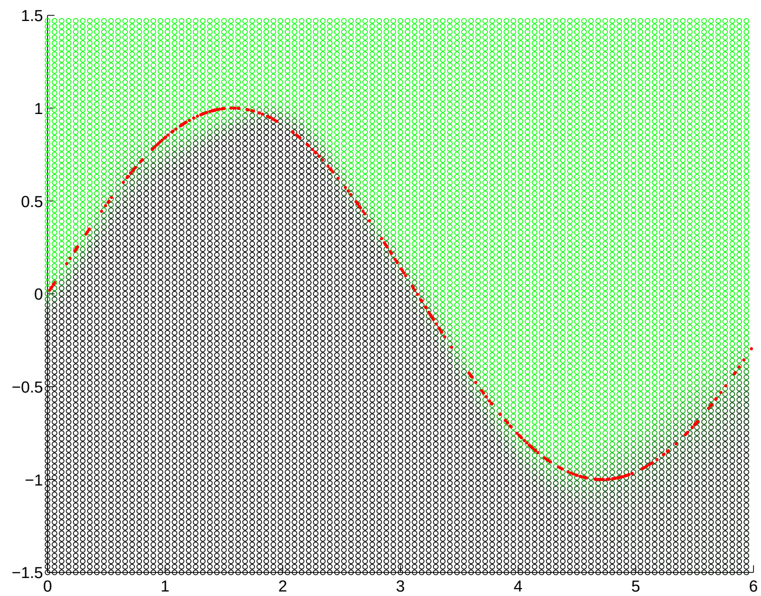

Figure 2.2: Analytic and learned boundary

`plearn vmat convert out.pmat out.amat` converts from a binary to an ascci file.

`tail +3 out.amat > result.mat` removes meta-information at the beginning of the file.

Then we execute the result_task1.m script in Matlab to plot the result, as in figure 2.2.

2.2.2 What have we done?

NNet.plearn contains informations about the neural net. train.vmat contains informations about the the dataset.

PLearn is built with a direct help system. For example:

`plearn help NNet`

`plearn help AutoVMatrix`

`plearn help GradientOptimizer`

This commands gives you an accurate information.

Here are the more important thing about PLearn.

PLearn is an object oriented software. The scripting language is object oriented. "plearn help NameOfTheClass" gives you the corresponding information

Note that a lot of others parameters exist for each classes but they have default values.

2.3 A second example

We now consider a regression problem. We want to train a neural network to predict a sinus.

First, we generate the data with the task2.m matlab file.

Then we perform the task within only one script, the following `regression.plearn`.

```
PTester(

learner = NNet
    (
    nhidden =  10 ;
    noutputs = 1 ;
    output_transfer_func = "";
    hidden_transfer_func = "tanh"  ;
    cost_funcs = 1 [ mse ]  ;
    optimizer = GradientOptimizer(
                    start_learning_rate = .01;
                    decrease_constant = 0;
                    )
    batch_size = 1  ;
    initialization_method = "normal_sqrt"  ;
    nstages = 500 ;
    verbosity = 3;
    );

expdir = "tutorial_task2"  ;

splitter = ExplicitSplitter(splitsets = 1 2 [
    AutoVMatrix(
```

```
        specification = "reg_train.amat"
        inputsize = 1
        targetsize = 1
        weightsize = 0
    )
    AutoVMatrix(
        specification = "reg_test.amat"
        inputsize = 1
        targetsize = 1
        weightsize = 0
    )
    ]
) ;

statnames = ["E[E[train.mse]]" "E[E[test.mse]]" ];

);
```

Let's run Plearn on this script: `plearn regression.plearn`

Then test on the original space: `plearn learner compute_outputs tutorial_task2/Split space2.amat out.pmat`

We need two additionnals commands to view the result of the network with matlab:

`plearn vmat convert out.pmat out.amat`

`tail +3 out.amat > result.mat`

Let's view the result with a matlab script, result_task2.m:

2.3.1 What have we done?

We encapsulated the experiment in a powerful scriptable class called "PTester".

The command `plearn help PTester` gives you the corresponding information. Note that a lot of others parameters exist for PTester but they have default values.

Furthermore, explore all the files that a PTester creates:

```
cd tutorial_task2/
```

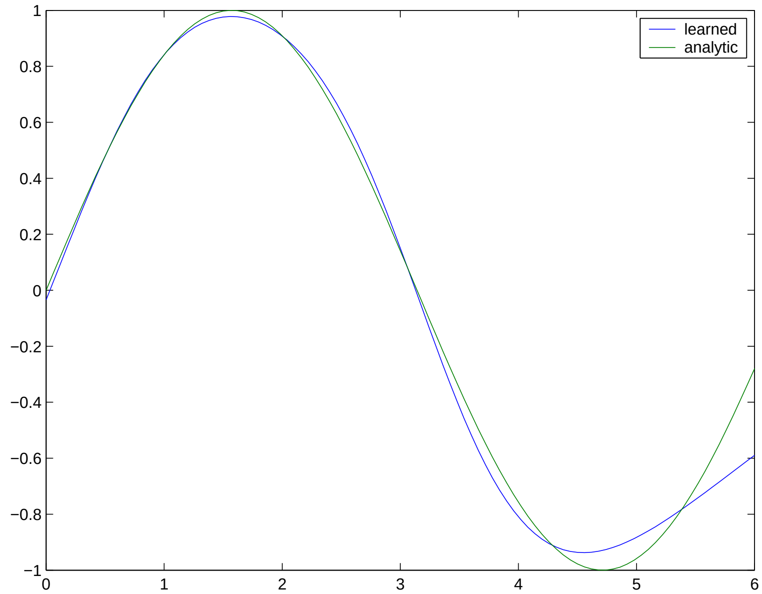

Figure 2.3: Analytic and learned function

```
ls
plearn vmat view split_stats.pmat
plearn vmat view global_stats.pmat
cd Split0
ls
less test1_stats.psave
less final_learner.psave
```

2.4 Conclusion

This short tutorial shows a small part of PLearn. Continue by yourself and have a nice Plearn time!

Chapter 3

Basics

3.1 The plearn Program

The `plearn` program is to be found in PLearn/commands and is used to

- either run a .plearn script
- or run a plearn command

Plearn scripts are essentially text files ending in .plearn that describe a learning experiment to be performed.

Plearn commands are typically little tools that allow you to manipulate or examine datasets or result files, but they can also launch more evolved interactive programs.

The `plearn` program has a simple yet very useful command-line help system. Type `plearn help` to have an overview.

3.2 Essential Commmands

The basic plearn command is `plearn script.plearn`.

The wisest command is `plearn help ClassFoo`.

But there are others:

`plearn vmat view bidule.vmat` to view any .vmat, .pmat or .amat file.

`plearn vmat convert truc.pmat truc.amat` to convert a specific data format in an other.

`plearn learner train`, `plearn learner test`, `plearn learner computes_output` provide useful shortcuts to avoid creating long .plearn script (cf. Tutorial).

If you are interested in more information,

```
plearn help commands
plearn help vmat
plearn help learner
```

3.3 Essential Classes

Here is a list of essential classes.

```
plearn help AutoVMatrix
plearn help PTester
plearn help Optimizer
--- plearn help GradientOptimizer
plearn help PLearner
--- plearn help NNet
```

3.4 The .plearn Object File Format

PLearn uses the same simple file format, both to describe experiments to be performed (in .plearn scripts), and to save and restore objects such as a trained neural-network (in .psave or .spec files).

Essentially these files contain the specifications of PLearn objects.

This is a typical .plearn script:

```
PTester(

learner = NNet
    (
    nhidden =  10 ;
```

```
    noutputs = 1 ;
    output_transfer_func = "";
    hidden_transfer_func = "tanh"  ;
    cost_funcs = 1 [ mse ]  ;
    optimizer = GradientOptimizer(
                    start_learning_rate = .01;
                    decrease_constant = 0;
                    )
    batch_size = 1  ;
    initialization_method = "normal_sqrt"  ;
    nstages = 500 ;
    verbosity = 3;
    );

expdir = "tutorial_task2"  ;

splitter = ExplicitSplitter(splitsets = 1 2 [
    AutoVMatrix(
        specification = "reg_train.amat"
        inputsize = 1
        targetsize = 1
        weightsize = 0
    )
    AutoVMatrix(
        specification = "reg_test.amat"
        inputsize = 1
        targetsize = 1
        weightsize = 0
    )
    ]
) ;

statnames = ["E[E[train.mse]]" "E[E[test.mse]]" ];

);
```

Objects are specified by the name of their type, followed by a list of `option = value` pairs.

Any sequence of spaces, newlines, tabs, comma, or semicolon is considered a separator. So colons and semicolons are just there to ease the reading, spaces would work just as well.

Comments start with a `#` and continue until the end of the line.

The following table sums up the formats that can be used for the values of an option of a given type

Table 3.1: Ascii format for given data-types

Data type	**Format example**
Any subclass of Object	`ObjectType( option1 = value1, option2 = value2, ...`
integer	`-365`
floating number	`-3.2e-4`
string	`"any string"`
character	`'x'`

1D sequences	`[ 10, 20, 30, 40 ]`
	`[ 10 20 30 40 ]`
	`4 [ 10 20 30 40 ]`
	`4 [ "aa", "bb", "cc", "dd" ]`

2D matrices	`3 2 [ 1 2  10 20  30 40 ]`
pairs	`(1, "one")`
tuples	`(1, "one", 3.5)`
maps	`{ 1:"one", 2 :"two", 3: "three" }`
pointers to new object	`*1 -> ObjType( ... )`
reference to pointer	`*1;`

Note for strings: unquoted strings, while not recommended are also supported. They are read until a separator (blank, comma, ...) or opening or closing symbol (parenthesis, bracket, ...) is met.

3.5 The .amat File Format

Ascii data file.

The new format is as follows:

- The size of the matrix is indicated by a line starting with `#size:` and followed by length (number of rows) and width (number of columns).

- An optional line starting with `#sizes:` gives the `inputsize, targetsize, weightsize, extrasize`.
- An optional line starting with `#:` gives the names of the fields (the columns)
- Regular comment lines start with a single `#`.

ex:

```
# Characteristics of a population of 534
#size: 534 3
#sizes: 2 1 0 0
#:  age height weight
     33  1.72   71
    25  1.80   80
```

3.6 The .pmat File Format

PLearn native binary format.

3.7 The .vmat File Format

File containing a description of a virtual dataset.

A .vmat contains the specification of a subclass of VMatrix, in plearn serialization format.

```
AutoVMatrix(
specification = "train.amat"
inputsize = 2
targetsize = 1
weightsize = 0
)
```

Chapter 4

Howto

4.1 How to Build a Neural Network?

You should have learned with the tutorial basic PLearn neural network. The class used is NNet.

Here is a basic NNet script object:

```
NNet(
nhidden =  10 ;
noutputs = 1 ;
output_transfer_func = "";
hidden_transfer_func = "tanh"  ;
cost_funcs = 1 [ mse ]  ;
optimizer = GradientOptimizer(
                start_learning_rate = .01;
                decrease_constant = 0;
                )

batch_size = 1  ;
initialization_method = "normal_sqrt"  ;
nstages = 500 ;
);
```

Chapter 5

Advanced

5.1 The .dmat/ Format

Directory containing compressed data.

Contains:

- 0.data, 1.data, 2.data
- indexfile
- fieldnames

5.2 The VPL language

VPL (vmat processing language) is a home brewed mini-language in postfix notation. As of today, it is used is the {PRE,POST}FILTERING and PROCESSING sections of a .vmat file. It can handle reals as well as dates (format is: CYYMMDD, where C is 0 (1900-1999) or 1 (2000-2099). The language will not be extensively described here. For more info, you can look at plearn/vmat/VMatLanguage.*.

A VPL code snippet is always applied to the row of a VMatrix, and can only refer to data of that row. The result of the execution will be a vector, which is the execution stack at code termination.

When you use VPL in a PROCESSING section, each field you declare must have its associated fieldname declaration. The compiler will ensure that the size of the result vector and the number of declared fieldnames match. This doesn't apply in the filtering sections since the result is always a single value.

To declare a fieldname, use a colon with the name immediately after. To batch-declare fieldnames, use eg.:myfield:1:10. This will declare fields myfield1 up to myfield10.

There are two notations to refer to a field value: the @ symbol followed by the fieldname, or % followed by the field number.

To batch-copy fields, use the following syntax: [field1:fieldn] (fields can be in @ or % notation).

Here's a real-life example:

```
@lease_indicator 88 == 1 0 ifelse :lease_indicator
@rate_class 1 - 7 onehot :rate_class:0:6
@collision_deductible { 2->1; 4->2; 5->3; 6->4; 7->5;
[8 8]->6; MISSING->0; OTHER->0 }
7 onehot :collision_deductible:0:6
@roadstar_indicator 89 == 1 0 ifelse :roadstar_indicator
```

5.3 The Metadata Directory

A metadata directory is associated with each dataset. For the datasets corresponding to a file (`.amat, .pmat, .vmat`) or directory (`.dmat/`) the associated metadata directory is obtained by appending `.metadata/` to the file or directory name.

A metadata directory will typically contain the following cache directories to avoid recomputing costly things

- `STATSCACHE/` contains cached statistics
- `MODELCACHE/<classname>/` contains any pertinent cached data computed on this dataset by objects of class `<classname>`

In addition, the `.metadata` directory associated with a `.vmat` may contain

- `precomputed.dmat/` or `precomputed.pmat` if the `.vmat` description specified `<PRECOMPUTE>`

- `source.index` containing row indexes in the source (resulting from `<PREFILTER>`, `<POSTFILTER>`, `<SHUFFLE>`)

Chapter 6

Appendix A: File Formats

6.1 The .plearn and .psave Formats

6.1.1 Generalities on mixing ascii and binary

The following characters are in many cases skipped before reading any element: space, tab, newline, carriage-return, comma and semicolon. They are essentially ignored. Binary serialized things should always start with a non-printable ascii character.

6.1.2 TVec and TMat

TVec and TMat will be serialized differently depending on the *implicit_storage* flag of the PStream they are being written to.

If *implicit_storage* is set, then serialization won't write the actual whole structure of the TVec or TMat, but will only save the size information and elements as a 1D or 2D *sequence* (see 6.1.4 and 6.1.5), ex:

```
4 [ 1.2 3.5 2.8 5.2 ]

3 2 [
0.1    0.2
0.3    0.4
0.5    0.6
]
```

If *implicit_storage* is false, then the complete structure of the TVec or TMat with the pointer to its storage (possibly shared with others) will be written explicitly. This corresponds to true, deep serialization.

Ex:

```
TVec( 4 0
*1->Storage(4 [ 1.2 3.5 2.8 5.2 ]) )

TMat( 3 2 2 0
*2->Storage(6 [ 0.1 0.2 0.3 0.4 0.5 0.6 ] ) )
```

For TVec, we have *length offset* followed by the storage pointer. For TMat, we have *length width mod offset* followed by the storage pointer.

This allows to keep structure. For example, if we had a submatrix viewing the second column of the previous TMat, we would have:

```
TMat( 3 1 2 1
*2 )
```

6.1.3 Binary PLearn format for base types

To allow mixing of ascii and binary in a file, a non-printable ascii character is used as a one-byte header to identify any binary portion. In Table 6.1 we give the header codes for all basic types

Note that char is considered to be the same as signed char, and long is considered to be the same as int, i.e.: 4-bytes long, which is the case on current architectures.

- booleans are represented the same way in binary mode as in ascii mode: with the character 0 (for false) or 1 (for true). There is no header byte.
- A date (PDate) is written with the header-byte 0xFE followed by a binary serialized double (with appropriate double header) representing the date in YYYYMMDD format.

6.1.4 Ascii PLearn format for a sequence

We consider both one-dimensional sequences (array, vector, . . .) which only have a length, and two-dimensional sequences which have a length and a width.

Table 6.1: Binary-header codes for base types

Base type	Byte order	Header byte	Number of bytes to follow
char	-	0x01	1
signed char	-	0x01	1
unsigned char	-	0x02	1
short	little-endian	0x03	2
short	big-endian	0x04	2
unsigned short	little-endian	0x05	2
unsigned short	big-endian	0x06	2
int	little-endian	0x07	4
int	big-endian	0x08	4
unsigned int	little-endian	0x0B	4
unsigned int	big-endian	0x0C	4
long	little-endian	0x07	4
long	big-endian	0x08	4
unsigned long	little-endian	0x0B	4
unsigned long	big-endian	0x0C	4
float	little-endian	0x0E	4
float	big-endian	0x0F	4
double	little-endian	0x10	8
double	big-endian	0x11	8
PRInt64	little-endian	0x16	4
PRInt64	big-endian	0x17	4
PRUint64	little-endian	0x18	4
PRUint64	big-endian	0x19	4

Ascii-serialized one-dimensional sequences will have the following format:

length `[ ... ... ... ]`

with the elements of the sequence separated by a single space.

However, on reading, several variations of this format are recognized:

- The elements may be separated by any number of blanks (space, tab, newline) and/or commas or semicolons.
- The *length* may be omitted

Ascii-serialized two-dimensional sequences will have the following format:

length width `[`

```
... ... ...
... ... ...
]
```

with the elements of each row separated by a tab, and the rows separated by a newline.

However on reading, blanks, commas and semi-colons between elements are completely ignored (skipped), so you may format the data as you wish.

2D Sequences are used exclusively for TMats. Notice that it's also possible to make a 1D sequence of 1D sequences, but that's different from a 2D sequence.

6.1.5 Binary PLearn format for a sequence

We consider both one-dimensional sequences (array, vector, ...) which only have a length, and two-dimensional sequences which have a length and a width.

The following table gives the corresponding header-byte:

Type of sequence	byte-order	Header byte
one-dimensional	little-endian	0x12
one-dimensional	big-endian	0x13
two-dimensional	little-endian	0x14
two-dimensional	big-endian	0x15

All that follows is supposed to be in the byte-order implied by the header-byte.

The first header-byte is followed by an *element-type* byte giving the nature of the elements in the sequence. It can be either the byte identifying a base-type given in Table 6.1 (the endianness must match), or `'0' = 0x30` to indicate a sequence of booleans (1 byte per boolean) or `0xFF` to indicate a *generic* sequence.

The header bytes are followed by one (for 1D sequences) or two (for 2D) 4-byte int to indicate the length (and possibly width) of the sequence. So the total header size for sequences is 6 bytes for 1D sequences and 10 bytes for 2D sequences.

This header is followed by a dump of the elements of the sequence (in row-major mode for 2D). Notice that a sequence of a base type, may be saved as a *generic* sequence (with the *element-type* byte `0xFF`)

Type of sequence	Header byte	Followed by
Generic on little-endian	0x12	size as 4-byte little-endian int, then binary serialization of the elements
Generic on big-endian	0x13	size as 4-byte big-endian int, then binary serialization of the elements
Sequence of a base-type on little-endian	0x14	size as 4-byte little-endian int, base-type given by header byte in previous table, followed by binary dump of elements
Sequence of a base-type on big-endian	0x15	size as 4-byte big-endian int, base-type given by header byte in previous table, followed by binary dump of elements

License

This document is covered by the license appearing after the title page.

The PLearn software library and tools described in this document are distributed under the following BSD-type license:

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The name of the authors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN
 NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```