# Harbinger Machine Learning Toolkit Manual*

B. Barla Cambazoglu and Cevdet Aykanat

Bilkent University, Department of Computer Engineering,
06800, Ankara, Turkey
{berkant,aykanat}@cs.bilkent.edu.tr

**Abstract.** This manual is the primary guide to the Harbinger Machine Learning Toolkit (HMLT), which provides implementations for some well-known and frequently used machine learning classifiers. The main concerns in development of HMLT are correctness, effectiveness, transparency, modularity, and re-usability. At the moment, efficiency is not claimed to be a primary concern in any part of the toolkit. This is basically due to the fact that all supported classifier implementations use common representations and data structures, preventing further utilization and employment of some classifier-specific optimizations. However, we believe that the toolkit is quite robust and supposed to successfully execute under most circumstances.

## 1 Introduction

The Harbinger Machine Learning Toolkit (HMLT) is a general-purpose machine learning toolkit. It currently supports ten different machine learning classifiers including the naive Bayesian and k-nearest neighbor classifiers. The toolkit uses a common format for storing and reading the data sets. In this format, aliases can be defined for attribute values preventing repetition of long strings. Moreover, this format allows using both dense matrix and sparse matrix representations for the data sets. Hence, for problems with high attribute dimensions (like text categorization), sparse matrix format can be used to save storage space and reduce the I/O time. As well as classifiers, the toolkit offers a wrapper program to ease the validation process. By means of this wrapper, the user can easily perform cross-validation, leave-1-out validation, and some other validation techniques over the data set. This eliminates the need for writing an extra piece of code to partition the instance set into train and test sets for each data set at hand.

The outline of the manual is as follows. In Section 2, we provide some background information on the supported classifiers. In Section 3, the details of the HMLT data set format used by the classifiers are presented on an example. In Section 4, the installation procedure is described. In Section 5, we provide the command line options for the classifiers. In Section 6, we briefly mention the wrapper program and its options. In Section 7, we underline the limitations of the current version and point at the work under progress. Finally, in Section 8, we provide contact addresses and pointers to machine learning sites in the Web.

* The original and up-to-date version of this document can be obtained from http://www.cs.bilkent.edu.tr/~berkant/coding/harbinger.

## 2 Supported Classifiers

We present the classifiers supported by HMLT under four main headings: instance-based classifiers, probabilistic classifiers, symbolic learning classifiers, and neural network classifiers. The reader is assumed to be aware of the theoretical and practical details of these classifiers. Hence, here we provide only a very brief and rough description of the implemented classifiers.

### 2.1 Instance-Based Classifiers

k-nearest neighbor (k-NN), k-nearest neighbor with feature projections (k-NN-FP), and k-means classifiers are the supported classifiers of this type. k-NN and its derivations (like weight adjusted k-NN) are among the most frequently used classifiers. This type of classifiers are able to capture the local properties in the data, but fail to capture the global features of a data set. These classifiers perform the entire work in the test phase, where all test instances are compared with the training instances and, for each test instance, the most similar $N$ training instances are determined. Depending on the classes of these $N$ training instances, a prediction about the class of the test instance is made.

For the k-NN classifier, HMLT supports three different distance measures for finding the similarity of two instances: cosine similarity measure, Euclidean distance measure, and Manhattan distance measure. After the most similar $N$ instances are found, the class of an instance can be predicted using majority voting or similarity score summing. Similarly, for k-NN-FP, majority voting or similarity score summing can be used to make the final decision on class selection.

### 2.2 Probabilistic Classifiers

Currently, the only supported classifier under this category is the naive Bayesian classifier. In contrast to instance-based classifiers, the naive Bayesian classifier tries to capture the global properties of a data set. It operates only on data sets with categorical attributes. In the training phase of this classifier, the probability that a class value will be observed when an input attribute value is observed is calculated. In the test phase, for each instance, these probabilities are multiplied depending on the attribute values of the test instance. For each class value, a probability is calculated and the class with the highest probability is selected as the predicted class. Despite its assumption that attributes appear independent of each other, naive Bayesian performs quite well in most data sets.

### 2.3 Symbolic Learning Classifiers

The supported classifiers of this type are the covering rules (PRIM) and 1-rule classifiers. The covering rules classifier aims to produce human-understandable rules for classifying the test instances. During the training phase, each class value is visited. For each class value, using the training instances, a set of rules that will cover all training instances with that class value is generated. Later, these rules are used for classification of test instances. This classifier works on categorical attributes. An instance can be classified by more than one rule as belonging to many classes. In this case, a majority voting scheme can be used, or the most frequently appearing class can be assigned as the predicted class.

1-rule classifier is a similar but simpler version of covering rules classifier. It produces its rules depending on the values of just a single attribute. Although being a rather naive classifier, for small data sets with a few important attributes, this classifier produces surprisingly good results.

### 2.4 Neural Network Classifiers

The supported neural network classifiers include perceptron, back-propagation, Kohonen, and Hopfield networks. All classifiers in this category convert their input attribute values to -1 and 1, by taking the sign of actual input attribute values. Compared to other classifiers, the training phase is quite slow in this type of classifiers. It may take a large number of iterations to find a local optimum. Hence, it is wise to limit the epoch counts in most cases.

Perceptron is the simplest of neural networks. It acts as a black box, which maps a given input instance to an output class value. Back-propagation neural network is a more enhanced classifier. It is known to be outperforming perceptron neural network in many applications. However, due to the massive amount of computations performed, it is relatively slower. Our implementation of back-propagation neural network constructs a three layer (input, hidden, and output layers) network. This classifier can be used both for classification and regression problems.

Kohonen and Hopfield networks are examples of unsupervised classifiers. In fact, these two are clustering algorithms rather than classification algorithms. However, created clusters can be used for classification purposes. In Hopfield neural networks, the class values of the training instances is not utilized at all. In Kohonen network, they are used just for calibration purposes.

## 3 Harbinger Data Set Format

Throughout this discussion on the data set format, we assume that we work on a data set containing a total of $m$ instances (examples), $n$ input attributes (features), and a single output attribute (class). All classifiers expect the information about the data set and its content to be initially distributed and stored under four separate files in the disk. These four files are pure text files, each starting with a common name, `<data set>`, where `<data set>` is a name representing the data set. The file extensions for the files are fixed and are `.info`, `.insts`, `.attrs`, and `.DMR`. For example, a data set about cancer can be stored under the files `cancer.info`, `cancer.insts`, `cancer.attrs`, and `cancer.DMR`. In all files, the lines starting with a # character are treated as comment lines and are ignored together with white spaces. All files are case-sensitive.

We describe the details of these files on an example. Assume that we have a data set about humans. Each instance in our data set represents a human being. Let each human being has the input attributes skin color, age, weight, and the output attribute gender. In other words, by using the skin color, age, and weight of a human, we are trying to predict its gender. Since we have four attributes, in this particular example, $n = 4$.

**.insts file:**

`<data set>.insts` file contains information about the labels of the instances. Each line in this file corresponds to a label identifying an instance. In our case, it contains human names:

```
# human.insts
# containing human names
# m=5

Berkant
Barla
John
Marry
Sandra
```

The use of this file is not obligatory. In the absence of the `<data set>.insts` file, each instance is given a unique name starting from `Inst`*1* through `Inst`$m$.

**.attrs file:**

This file keeps the labels used for the attributes and optionally the labels for the attribute values. The `<data set>.attrs` file is also optional. If the file is not present, default attribute names `Attr`*1* through `Attr`$n$ are assigned as the labels for the attributes. In our human data set, `human.attrs` file contains something like the following:

```
# human.attrs
# containing human attributes
# n=4

eyeColor
age
weight
gender
```

In the HMLT data set format, attributes may have three types of values: categorical, ordinal, or numeric. In our example, eye color and gender are categorical attributes, age is an ordinal attribute, and weight is a numeric attribute. We can further include this information in the `human.attrs` file as follows:

```
# human.attrs
# containing human attributes
# n=4

eyeColor C
age O
weight N
gender C
```

The letters C, O, and N indicate the attribute being categorical, ordinal, and numeric, respectively. In the absence of this information, the convention is to assume all attributes as categorical. However, even if a single numeric value is detected for an attribute, while reading the data set content, that attribute is assumed to be of type numeric. For example, by reading the weight 54.5 for the instance Marry, the code can decide that the weight attribute is numeric.

It is possible to define aliases for categorical attribute values. This can be done by inserting `<value>:<alias>` pairs in the `<data set>.attrs` file. This way, we can avoid repeating the same string in the `<data set>.DMR` file and save some storage space. For example, we can use the value 0 to represent male, and 1 to represent female genders, and then define them as aliases in the `human.attrs` file. Hence, we do not repeat the strings male and female in the original data file `<data set>.DMR`. The sample `human.attrs` file can be created like this:

```
# human.attrs
# containing human attributes
# n=4

eyeColor C 0:black 1:brown 2:green 3:blue
age O
weight N
gender C 0:male 1:female
```

**.DMR and .SMR files:**

The attribute values are stored in the `<data set>.DMR` file. This file contains an $m \times n$ matrix, where the rows represent the instances and the columns are the attributes. Our example `human.DMR` file is as follows:

```
# human.DMR
# containing attribute values
# mXn=5X4

1 34 67.3 0
3 48 53.2 1
0 34 78.0 0
0 78 51.2 1
2 49 55.2 1
```

For some applications, DMR (dense matrix representation) format is not appropriate. In case the number of attributes is large and attribute values are mostly zero, using SMR (sparse matrix representation) format may be better. Hence, as an option, it is possible to store attribute values in SMR format in the `<data set>.SMR` file. The above example can be stored in the `human.SMR` file as follows:

```
# human.SMR
# containing non-zero attribute values

1 1 2 34 3 67.3
1 3 2 48 3 53.2 4 1
2 34 3 78.0
2 78 3 51.2 4 1
1 2 2 49 3 55.2 4 1
```

Note that, in this representation, for the first instance, the value of the output attribute is not stored. While the data is read, it is implicitly assumed to be zero. For small data sets, DMR format is usually the better choice and vice versa.

**.info file:**

In the `<data set>.info` file, some general information about the data set is supplied. This information includes the type of the storage format used (DMR or SMR) and the total number of instances and attributes in the data set. The `<data set>.info` file also contains information about the partitioning of training and test instances, and selection of the attributes that will be used as input and output attributes. The sample `human.info` file is as follows:

```
# human.info

representationType DMR

totalInstanceCount 5
trainInstances 1-3 5
testInstances 4

totalAttributeCount 4
inputAttributes 1-3
outputAttribute 4
```

The tags used in this file and their meanings are as follows:

– *representationType*: The storage format used for keeping the attribute values. It can be DMR or SMR. Depending on this information, the appropriate `<data set>.DMR` or `<data set>.SMR` file is fetched from the disk.

– *totalInstanceCount*: Shows how many instances are expected. Instance labels beyond this count are ignored.

– *trainInstances*: Shows which instances will be used for training. “-” sign can be used to denote intervals, as in 1-3.

– *testInstances*: Shows the instances to be predicted.

– *totalAttributeCount*: Shows how many attributes are expected. Attribute labels beyond this count are ignored.

– *inputAttributes*: Shows the input attributes that will be used for prediction.

– *outputAttribute*: Shows the output attribute we are trying to predict.

Any tag other than these is accepted to be an erroneous tag. All indices in the `<data set>.info` file start from 1. For instances and attributes, the indices beyond *totalInstanceCount* and *totalAttributeCount* are treated as errors, respectively. The output attribute need not be the last one. We can simply modify the `<data set>.info` file to predict the eye color of a human by using its age and gender attributes as follows:

```
# human.info

representationType DMR

totalInstanceCount 5
trainInstances 1-5
testInstances 1-5

totalAttributeCount 3
inputAttributes 2 3
outputAttribute 1
```

In this example, *totalAttributeCount* is 3 since we no longer use the weight attribute. We use all instances both for training and testing purposes. No other modification is necessary in any of the remaining three files.

## 4 Installation

HMLT has been successfully installed, compiled, and executed on Linux, Unix, and Windows platforms. For Windows installation, Cygwin was used. Installation of HMLT is rather straightforward:

– Download and move `harbinger.tar.gz` file into the directory where you want to install HMLT.

- Type the following to unzip the file:

```
gunzip harbinger.tar.gz
```

- Now, extract the files by:

```
tar xvf harbinger.tar.gz
```

- This will create a directory named `Harbinger` in the current directory and extract the source files under it.
- Now, move into the `Harbinger` directory and run the installation script:

```
cd Harbinger
./install
```

- This will compile the source codes. For each classifier, there is an associated directory. Both the source codes and executables of a classifier are kept in corresponding directories. Also, a library, which is common to all classifiers is compiled under the `lib` directory. Symbolic links are created under the `bin` directory. The wrapper program is installed in the `tester` directory.
- You may need to modify some parts of the installation script depending on your system configuration (the lines at the top of the script).

## 5 Toolkit Options

In this section, we provide a list of the command line parameters that classifiers accept. Some options are common to all classifiers, whereas some are classifier-specific.

### 5.1 Options Common to All Classifiers

**-h** : Prints the command line options for a classifier.

**-iv** `<verbosity level>`: Sets instance verbosity (1:Low, 2:Medium, 3:High). If verbosity is low only the name of the instance is displayed. When medium, the class value is also displayed. If verbosity is high, the input attribute values are also displayed. But, this may be annoying if there are too many attributes in the data set.

**-cv** `<verbosity level>`: Sets classifier verbosity (0:None, 1:Low, 2:Medium, 3:High). The meaning of classifier verbosity depends on the classifier used. But, in general, if verbosity is none, only the accuracy and timing information is displayed. If it is low, test instances together with predictions made for them is printed.

**-M** `<classification model>`: Sets classification model (1, 2, 3, 4). In the first model, all attribute values are used both for training and testing. In the second model, only the non-zero values of the test instances are used. In the third model, only the non-zero values of the training instances are used. In the fourth model, only the non-zero values of the instances are used.

**-f** `<path>`: Sets the location of the dataset to be read. For example, to read the human data set from the current directory, this option should take the parameter `./human`.

## 5.2 Classifier-Specific Options

**Options for k-NN classifier:**

**-N** `<number of neighbors>`: Sets the number of neighbors to be found.

**-dm** `<distance metric>`: Sets the distance metric used (c:cosine similarity, e:Euclidean distance, m:Manhattan distance).

**-vm** `<voting metric>`: Sets the voting metric used (m:majority voting, s:similarity voting).

**Options for k-NN-FP classifier:**

**-N** `<number of neighbors>`: Sets the number of neighbors to be found.

**-vm** `<voting metric>`: Sets the voting metric used (m:majority voting, s:similarity voting).

**Options for k-means classifier:**

No options.

**Options for naive Bayesian classifier:**

No options.

**Options for covering rules classifier:**

No options.

**Options for 1-rule classifier:**

No options.

**Options for perceptron neural network classifier:**

**-tr**: Trains the network.

**-ts**: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

**-lc** `<learning constant>`: Sets the learning constant.

**-er** `<minimum error>`: Sets the minimum error before convergence. If this error is obtained over the training set, the training algorithm stops.

**-ep** `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

**Options for back-propagation neural network classifier:**

**-tr**: Trains the network. Saves the weight matrix to the disk.

**-ts**: Tests the network. If the -tr option is not used, the testing is performed using the weight matrix read from the disk.

**-lc** `<learning constant>`: Sets the learning constant.

**-mc** `<momentum constant>`: Sets the momentum constant.

**-N** `<hidden layer neuron count>`: Sets the number of hidden layer neurons.

**-er** `<minimum error>`: Sets the minimum error before convergence. If this error is obtained over the training set, the training algorithm stops.

**-ep** `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

**-tt** `<test type>`: Sets the test type (r:regression, c:classification). If the test type is classification, output attribute must have categorical values. Otherwise, it must have ordinal or numeric values.

**Options for Kohonen neural network classifier:**

**-tr**: Trains the network.

**-ts**: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

**-lc** `<learning constant>`: Sets the learning constant.

**-ep** `<maximum epoch count>`: Sets the maximum epoch count before convergence. If this epoch count is reached, the training algorithm stops.

**Options for Hopfield neural network classifier:**

**-tr**: Trains the network.

**-ts**: Tests the network. If the -tr option is not used, the testing is performed using the initial, randomly generated weight matrix.

## 6 The Wrapper

It is possible to run each classifier as a stand-alone application. However, HMLT also supplies a wrapper program to release the burden of modifying the `.info` file for each experiment. if the wrapper is not used, in order to perform cross-validation over a data set, the `.info` file must be edited before each run. The wrapper offers several validation techniques and hides the details of partitioning the instance set. The options for the wrapper are as follows:

**-h** : Prints the command line options for the wrapper.

**-v** `<verbosity level>`: Currently this option is not used.

**-vt** `<validation type>`: Sets the validation type (can be one of exact, cv, scv, l1o, all). If it is set to exact, the current `.info` file is used without any modification. If it is set to cv or scv, the data set is $N$-fold cross validated, by partitioning the data set into $N$ pieces and running the classifier $N$ times. In each run a different piece of data set is used for testing, and the average of the results is calculated as the final result. scv is different than cv in that the instance set is shuffled before partitioning. l1o stands for leave-1-out validation. This is equivalent to $m$-fold cross validation. If validation type is set to all, the entire instance set is used for both training and testing.

**-N** `<fold count>`: Sets the fold count in cross validation.

**-e** `<command>`: Sets the command (i.e., the classifier) to be executed. This option must always be given as the last option to the wrapper.

The example below executes the k-NN classifier over the human data set and performs shuffled, 10-fold cross-validation.

```
tester -v 3 -vt scv -N 10 -e ../knn/knn -iv 2 -cv 2 -M 1 -N 5
       -dm e -vm m -f ../data/human
```

## 7 Limitations and Future Work

The following are the limitations of the current version:

- No handling of missing values. This will hopefully be implemented in the future versions.
- No support for feature extraction (or weighting).
- The parser code for the input files should be enhanced and made more flexible.

The following are among our future plans:

- Addition of new classifiers, including C4.5 decision tree algorithm and classification rules algorithm.
- Optimization of the code.
- A graphical front-end to the classifiers.

## 8 Contact Points and References

The Harbinger Machine Learning Toolkit is under continuous modification. Please, feel free to report any improvements, suggestions, and bugs you came across. Any contributions are welcome. The following are the contact points:

- Harbinger Machine Learning Toolkit homepage:
  `http://www.cs.bilkent.edu.tr/~berkant/coding/harbinger`
- Email address:
  `berkant@cs.bilkent.edu.tr`
- Phone:
  +90 312 2902092

The following are some pointers to the machine learning sites in the Web:

- `http://www.kdnuggets.com/`
- `http://www.ics.uci.edu/~mlearn/`
- `http://www-dsi.ing.unifi.it/neural/w3-sites.html`
- `http://directory.google.com/Top/Computers/Artificial_`
  `Intelligence/Machine_Learning/Datasets/`
- `http://dir.yahoo.com/Science/computer_science/artificial_`
  `intelligence/machine_learning/`