LVQ_PAK

The Learning Vector Quantization Program Package

Version 3.1 (April 7, 1995)

Prepared by the
LVQ Programming Team of the
Helsinki University of Technology
Laboratory of Computer and Information Science
Rakentajanaukio 2 C, SF-02150 Espoo
FINLAND

Teuvo Kohonen
Jussi Hynninen
Jari Kangas
Jorma Laaksonen
Kari Torkkola

Contents

1 Introduction **4**

1.1 Contents of this package 4

1.2 Precautions 4

1.3 The LVQ-algorithms 5

1.3.1 The LVQ1 5

1.3.2 The LVQ2.1 5

1.3.3 The LVQ3 6

1.3.4 Differences between the basic LVQ1, LVQ2.1 and LVQ3 . 7

1.3.5 The optimized-learning-rate LVQ1 (OLVQ1) 7

2 General considerations **8**

2.1 Initialization of the codebook vectors 8

2.2 Learning 9

2.3 Stopping rule 10

3 Installation of the program package **10**

3.1 Getting the program code 10

3.2 Installation in UNIX 11

3.3 Installation in DOS 12

3.4 Hardware requirements 12

4 File formats **13**

4.1 Data file formats 13

4.2 Codebook file formats 14

5 Application of this package **14**

5.1 The interface program *lvq_run* 14

5.2 Using the programs directly 14

5.3 Program parameters 15

5.4 Using the command lines (an example) 17

5.4.1 First stage: Codebook initialization 17

5.4.2 Second stage: Codebook training 18

5.4.3 Third stage: Evaluation of the recognition accuracy . . . 18

5.4.4 Fourth stage: Codebook visualization 19

6 Description of the programs of this package **19**

6.1 Initialization programs 19

6.2 Training programs 20
6.3 Recognition accuracy program 21
6.4 Classification program 21
6.5 Monitoring programs 21
6.6 Auxiliary subprograms 22

7 Advanced features 23

8 Comments and experiences of the use of this package 26
8.1 Changes in the package 26

References 30

1 Introduction

1.1 Contents of this package

The package at hand, Version 3.1 of LVQ_PAK, contains amendments to earlier versions, developed to attack large problems. The most important of them is the possibility to read data files in pieces (see Section 7).

This package contains all programs necessary for the correct application of certain LVQ (Learning Vector Quantization) algorithms in an arbitrary statistical classification or pattern recognition task, as well as a program for the monitoring of the codebook vectors at any time during the learning process [Kohonen et al. 1992] [Kohonen 1995].

NEW BOOK:

Complete description, with over 1500 literature references, of the SOM (Self-Organizing Map) and LVQ algorithms can be found in the recently published book Kohonen: Self-Organizing Maps (Springer Series in Information Sciences, Vol 30, 1995). 362 pp.

To this package four options of the algorithms, the LVQ1 (as described in [Kohonen 1990b]), the LVQ2.1 (as specified, e.g., in [Kohonen 1990a] and [Kohonen 1990c]), the LVQ3 (as described in [Kohonen 1990b]) and the OLVQ1 ([Kohonen 1992]), have been selected.

1.2 Precautions

Few traditional 'neural network' algorithms have been meant to directly operate on raw data, such as pixels of an image, or samples of speech waveforms picked up from the time domain. Most pattern recognition tasks are preceded by a preprocessing transformation that extracts invariant features from the raw data, such as spectral components of acoustical signals, or elements of co-occurrence matrices of pixels. Selection of a proper preprocessing transformation for a particular task usually requires careful consideration, and no general rules can be given here. It is cautioned that if this LVQ-package is used for benchmarking against other methods, a proper preprocessing should always be used.

In performing statistical experiments, a separate data set for training, and another separate data set for testing must be used. If the number of required learning steps is bigger than the number of training samples available, the samples must be used re-iteratively in training, either in a cyclical or in a randomly-sampled order.

1.3 The LVQ-algorithms

1.3.1 The LVQ1

Assume that a number of 'codebook vectors' m_i (free parameter vectors) are placed into the input space to approximate various domains of the input vector x by their quantized values. Usually several codebook vectors are assigned to each class of x values, and x is then decided to belong to the same class to which the nearest m_i belongs. Let

$$c = \arg\min_i \{||x - m_i||\} \tag{1}$$

define the nearest m_i to x, denoted by m_c.

Values for the m_i that approximately minimize the misclassification errors in the above nearest-neighbor classification can be found as asymptotic values in the following learning process. Let $x(t)$ be a sample of input and let the $m_i(t)$ represent sequences of the m_i in the discrete-time domain. Starting with properly defined initial values (cf. Sec. 2.1), the following equations define the basic LVQ1 process:

$$\begin{aligned}
m_c(t+1) &= m_c(t) + \alpha(t)[x(t) - m_c(t)] \\
&\quad \text{if } x \text{ and } m_c \text{ belong to the same class,} \\
m_c(t+1) &= m_c(t) - \alpha(t)[x(t) - m_c(t)] \\
&\quad \text{if } x \text{ and } m_c \text{ belong to different classes,} \\
m_i(t+1) &= m_i(t) \;\; \text{for } i \neq c.
\end{aligned} \tag{2}$$

Here $0 < \alpha(t) < 1$, and $\alpha(t)$ may be constant or decrease monotonically with time. In the above basic LVQ1 it is recommended that α should initially be smaller than 0.1; linear decrease in time is used in this package. In Version 3.1 it is possible to use also an inverse-time type function (see Section 5.3). (In the optimized LVQ1, cf. Sec. 1.3.5, different values of α are used.)

1.3.2 The LVQ2.1

The classification decision in this algorithm is identical with that of the LVQ1. In learning, however, two codebook vectors, m_i and m_j that are the nearest neighbors to x, are now updated simultaneously. One of them must belong to the correct class and the other to a wrong class, respectively. Moreover, x must fall into a zone of values called 'window', which is defined around the midplane of m_i and m_j. Assume that d_i and d_j are the Euclidean distances of x from m_i and m_j, respectively; then x is defined to fall in a 'window' of relative width w if

$$\min(\frac{d_i}{d_j}, \frac{d_j}{d_i}) \quad > \quad s, \text{ where } s = \frac{1-w}{1+w}. \tag{3}$$

A relative 'window' width w of 0.2 to 0.3 is recommendable.

Algorithm:

$$\begin{aligned} m_i(t+1) &= m_i(t) - \alpha(t)[x(t) - m_i(t)], \\ m_j(t+1) &= m_j(t) + \alpha(t)[x(t) - m_j(t)], \end{aligned} \tag{4}$$

where m_i and m_j are the two closest codebook vectors to x, whereby x and m_j belong to the same class, while x and m_i belong to different classes, respectively. Furthermore x must fall into the 'window'.

1.3.3 The LVQ3

The LVQ2 algorithm was based on the idea of *differentially* shifting the decision borders towards the Bayes limits, while no attention was paid to what might happen to the location of the m_i in the long run if this process were continued. Therefore it seems necessary to include corrections that ensure that the m_i continue approximating the class distributions, at least roughly. Combining these ideas, we now obtain an improved algorithm that may be called LVQ3:

$$\begin{aligned} m_i(t+1) &= m_i(t) - \alpha(t)[x(t) - m_i(t)], \\ m_j(t+1) &= m_j(t) + \alpha(t)[x(t) - m_j(t)], \end{aligned}$$

where m_i and m_j are the two closest codebook vectors to x, whereby x and m_j belong to the same class, while x and m_i belong to different classes, respectively; furthermore x must fall into the 'window';

$$m_k(t+1) = m_k(t) + \epsilon\alpha(t)[x(t) - m_k(t)], \tag{5}$$

for $k \in \{i, j\}$, if x, m_i, and m_j belong to the same class.

In a series of experiments, applicable values of ϵ between 0.1 and 0.5 were found. The optimal value of ϵ seems to depend on the size of the window, being smaller for narrower windows. This algorithm seems to be self-stabilizing, i.e., the optimal placement of the m_i does not change in continual learning.

1.3.4 Differences between the basic LVQ1, LVQ2.1 and LVQ3

The three options for the LVQ-algorithms, namely, the LVQ1, the LVQ2.1 and the LVQ3 chosen to this package, yield almost similar accuracies, although a different philosophy underlies each. The LVQ1 and the LVQ3 define a more robust process, whereby the codebook vectors assume stationary values even after extended learning periods. For the LVQ1 the learning rate can approximately be optimized for quick convergence (as shown in Sec. 1.3.5). In the LVQ2.1, the relative distances of the codebook vectors from the class borders are optimized whereas there is no guarantee for the codebook vectors being placed optimally to describe the forms of the class borders. Therefore the LVQ2.1 should only be used in a differential fashion, using a small value of learning rate and a relatively low number of training steps.

1.3.5 The optimized-learning-rate LVQ1 (OLVQ1)

The basic LVQ1 algorithm is now modified in such a way that an individual learning rate $\alpha_i(t)$ is assigned to each m_i. We then get the following discrete-time learning process. Let c be defined by Eq. (1). Then

$$\begin{aligned}
m_c(t+1) &= m_c(t) + \alpha_c(t)[x(t) - m_c(t)] \\
&\quad \text{if } x \text{ is classified correctly,} \\
m_c(t+1) &= m_c(t) - \alpha_c(t)[x(t) - m_c(t)] \\
&\quad \text{if the classification of } x \text{ is incorrect,} \\
m_i(t+1) &= m_i(t) \;\; \text{for } i \neq c.
\end{aligned} \tag{6}$$

Next we address the problem of whether the $\alpha_i(t)$ can be determined optimally for fastest possible convergence of (6). If we express (6) in the form

$$m_c(t+1) = [1 - s(t)\alpha_c(t)]m_c(t) + s(t)\alpha_c(t)x(t) \tag{7}$$

where $s(t) = +1$ if the classification is correct and $s(t) = -1$ if the classification is wrong, we first directly see that $m_c(t)$ is statistically independent of $x(t)$. It may also be obvious that the statistical accuracy of the learned codebook vector values is optimal if the effects of the corrections made at different times, when referring to the end of the learning period, are of equal weight. Notice that $m_c(t+1)$ contains a "trace" from $x(t)$ through the last term in (7), and "traces" from the earlier $x(t'), t' = 1, 2, \ldots, t-1$ through $m_c(t)$. The (absolute) magnitude of the last "trace" from $x(t)$ is scaled down by the factor $\alpha_c(t)$, and, for instance, the "trace" from $x(t-1)$ is scaled down by $[1-s(t)\alpha_c(t)] \cdot \alpha_c(t-1)$. Now we first stipulate that these two scalings must be identical:

$$\alpha_c(t) = [1 - s(t)\alpha_c(t)]\alpha_c(t-1) \;\; . \tag{8}$$

If this condition is then made to hold for all t, by induction it can be shown that the "traces" collected up to time t from all the earlier x will be scaled down by an equal amount at the end, and thus the "optimal" values of $\alpha_i(t)$ are determined by the recursion

$$\alpha_c(t) = \frac{\alpha_c(t-1)}{1 + s(t)\alpha_c(t-1)} \quad . \tag{9}$$

Any user of the LVQ_PAK can easily become convinced about that (9) really provides for fast convergence. A precaution must be made, however: since $\alpha_c(t)$ can also increase, it is especially important that it does not rise above the value 1; the learning program *olvq1* in this package is even more restrictive, it never allows any α_i to rise above its initial value. With this provision, the initial values of the α_i can be selected rather high, say, 0.3, whereby learning is significantly speeded up, especially in the beginning, and the m_i quickly find their approximate asymptotic values.

It must be warned, too, that (9) is not applicable to the LVQ2, since thereby the α_i, on the average, would not decrease, and the process would not converge.

2 General considerations

In the LVQ algorithms, vector quantization is not used to approximate to density functions of the class samples (described, e.g., in [Makhoul et al. 1985]), but to directly define the class borders according to the nearest-neighbor rule. The accuracy achievable in any classification task to which the LVQ algorithms are applied and the time needed for learning depend on the following factors:

- an approximately optimal number of codebook vectors assigned to each class and their initial values,
- the detailed algorithm, a proper learning rate applied during the steps, and a proper criterion for the stopping of learning.

2.1 Initialization of the codebook vectors

In many practical applications such as speech recognition, even when the a priori probabilities for the samples falling in different classes are very different, a very good strategy already is to start with the same number of codebook vectors in each class. An upper limit to the total number of codebook vectors is set by the restricted recognition time and computing power available.

Since the class borders are represented piecewise linearly by segments of midplanes between codebook vectors of neighboring classes (borders of the so-called Voronoi tessellations), it may seem to be an even better strategy for optimal approximation of the borders that the average distances between the adjacent codebook vectors (which depend on their numbers per class) should be the same on both sides of the borders. Then, at least if the class distributions were symmetric, this would mean that the average shortest distances of the codebook vectors (or alternatively, the medians of the shortest distances) should be the same in every class. Because the final placement of the codebook vectors is not known until at the end of the learning process, their distances and thus their optimal numbers cannot be determined before that. This kind of assignment of the codebook vectors to the various classes can therefore only be made iteratively, for which there is a provision (program named *balance*) in this package.

Once the tentative numbers of the codebook vectors for each class have been fixed, for their initial values one can use first samples of the real training data picked up from the respective classes. Since the codebook vectors should always remain inside the respective class domains, for the above initial values too one can only accept samples that are not misclassified. In other words, a sample is first tentatively classified against all the other samples in the training set, for instance by the k-nearest-neighbor (KNN) method, and accepted for a possible initial value only if this tentative classification is the same as the class identifier of the sample. (In the learning algorithm itself, however, no samples must be excluded; they are thereby applied independent of whether they fall on the correct side of the class border or not.)

In the program *balance*, the medians of the shortest distances between the initial codebook vectors of each class are first computed. If the distances turn out to be very different for the different classes, new codebook vectors may be added to or old ones deleted from the deviating classes, and a tentative training cycle based on the optimized-learning-rate LVQ1 algorithm (cf. Sec. 1.3.5) is run once. This procedure can be iterated a few times. (The exact numbers of codebook vectors are not critical; the shortest distances may differ by a factor of, say, 2 but not significantly more.)

For good piecewise linear approximation of the borders, the medians of the shortest distances between the codebook vectors should also be somewhat smaller than the standard deviations (= square roots of variances) of the input samples in all the respective classes. These figures are displayed by the program *mindist* for checking.

2.2 Learning

It is recommended that learning be always started with the optimized LVQ1 algorithm (cf. Sec. 1.3.5), which has very fast convergence; its asymptotic recognition accuracy will be achieved after a number of learning steps that is about 30 to 50 times the total number of codebook vectors. If the initial learning period, as described in Sec. 2.1, is included in the initialization of the codebook vectors, the optimized LVQ1 algorithm can be continued from those codebook vector values that have been obtained in the initialization phase.

Often the optimized LVQ1 learning phase alone may be sufficient for practical applications, especially if the learning time is critical. However, in an attempt to improve recognition accuracy, one may continue with either the basic LVQ1, the LVQ2.1 or the LVQ3, using a low initial value of learning rate, which is then the same for all the classes.

2.3 Stopping rule

It often happens that the neural-network algorithms 'overlearn'; i.e., when learning and test phases are alternated, the recognition accuracy is first improved until an optimum is reached; after that, when learning is continued, the accuracy starts to decrease slowly. A possible explanation in the present case is that when the codebook vectors become very specifically tuned to the training data, the ability of the algorithm to generalize for new data suffers from that. It is therefore necessary to stop the learning process after some 'optimal' number of steps, say, 50 to 200 times the total number of the codebook vectors (depending on particular algorithm and learning rate). Such a stopping rule can only be found by experience, and it also depends on the input data.

Let us recall that the optimized-learning-rate LVQ1 may generally be stopped after a number of steps that is 30 to 50 times the number of codebook vectors (c.f. Sec. 2.2).

3 Installation of the program package

In the implementation of the LVQ programs we have tried to use as simple a code as possible. Therefore the programs are supposed to compile in various machines without any specific modifications made on the code. All programs have been written in ANSI C.

No graphics are included in this package so that the programs may be run equally well in all computers ranging from PC:s to Cray supercomputers. The monitoring program *sammon* generates a list of coordinates of points (and an encapsulated postscript code for visual inspection).

3.1 Getting the program code

The latest version – currently Version 3.1 – of the *lvq_pak*-program package is available for anonymous ftp user at the Internet ftp-site *cochlea.hut.fi (130.233.168.48)*. All programs and this documentation are stored in the directory */pub/lvq_pak*. The files are in multiple formats to ease their downloading and compiling.

The directory */pub/lvq_pak* contains the following files:

README	–	short description of the *lvq_pak* package
lvq_doc.ps	–	this document in ©PostScript format
lvq_doc.ps.Z	–	same as above but compressed
lvq_doc.txt	–	this document in ASCII format
lvq_p3r1.exe	–	self-extracting MS-DOS archive file
lvq_pak-3.1.tar	–	UNIX tape archive file
lvq_pak-3.1.tar.Z	–	same as above but compressed

An example of FTP access is given below

unix> ftp cochlea.hut.fi (or 130.233.168.48)
Name: anonymous
Password: <your email address>
ftp> cd /pub/lvq_pak
ftp> binary
ftp> get lvq_pak-3.1.tar.Z
ftp> quit
unix>

3.2 Installation in UNIX

The archive file *lvq_pak-3.1.tar.Z* is intended to be used when installing *lvq_pak* in UNIX systems. It needs to be uncompressed to get the file *lvq_pak-3.1.tar*. If your system doesn't support the BSD *compress* utility, you may download the uncompressed file directly.

The *tar* archive contains the source code files, makefiles, and example data sets of the package, all in one subdirectory called *lvq_pak-3.1*. In order to create the subdirectory and extract all the files you should use the command *tar xovf lvq_pak-3.1*. (The switches of *tar* unfortunately vary, so you may need omit the '*o*'.)

The package contains a makefile called makefile.unix for compilation in UNIX systems. Before executing the *make* command, it has to be copied to the name *makefile*. The file, *makefile.unix*, should work as such in most systems.

We have written the source code for an ANSI standard C compiler and environment. If the *cc* compiler of your system doesn't fulfill these requirements, we recommend you to port the public domain GNU *gcc* compiler in your computer. When using *gcc*, the makefile macro definition CC=cc has to be changed accordingly to CC=gcc. The *makefile* also contains some other platform specific definitions, like optimizer and linker switches, that may need to be revised.

In order to summarize, the installation procedure is as follows:

> *uncompress lvq_pak-3.1.tar.Z*

> *tar xovf lvq_pak-3.1.tar*

> *cd lvq_pak-3.1*

> *cp makefile.unix makefile*

> *make*

After a successful make of the executables, you may test them by executing

> *make example*

which performs the commands as listed in section "5.4 Using the command lines (an example)".

3.3 Installation in DOS

The archive file *lvq_p3r1.exe* is intended to be used when installing *lvq_pak* in MS-DOS computers. It is a self-extracting packed archive compatible with the public domain *lha* utility. If your system supports UNIX *tar* archiving and *compress* file compressing utilities, you may as well use *lvq_pak-3.1.tar* and *lvq_pak-3.1.tar.Z* archives.

The *lvq_p3r1.exe* archive contains the source code files, makefiles, and example data sets of the package, all in one subdirectory called *lvq_pak.3r1*. In order to create the subdirectory and extract all the files simply use the command *lvq_p3r1*.

The package contains a makefile called *makefile.dos* for building up the object files. Before using the *make* command, *makefile.dos* has to be copied to the name *makefile*. It is intended to be used with the Borland Make Version 3.6 and the Borland C++ compiler Version 3.1, and may need to be revised if used with other compilation tools. Even with Borland C you may want to set some compiler switches, e.g., floating point options, according to your hardware.

In order to summarize, the installation procedure is as follows:

> *lvq_p3r1*

> *cd lvq_pak.3r1*

> *copy makefile.dos makefile*

> *make*

After a successful make of the executables, you may test them by executing

> *make example*

which performs the commands as listed in section "5.4 Using the command lines (an example)".

3.4 Hardware requirements

The archive files are about 270 kbytes in size, whereas the extracted files take about 750 kbytes. When compiled and linked in MS-DOS, the executables are about 65 kbytes each. It is recommended to have at least 640 kbytes RAM, when using *lvq_pak* in MS-DOS.

4 File formats

All data files (input vectors and codebooks) are stored as ASCII files for their easy editing and checking. The files that contain training data and test data are formally similar, and can be used interchangeably.

4.1 Data file formats

The input data is stored in ASCII-form as a list of entries, one line being reserved for each vectorial sample. Each line consists of n floating-point numbers followed by the class label (that can be any string). The first line of the file is reserved for status knowledge of the entries; in the present version it is used to define the dimensionality of the data vector. The data files can contain comment lines that begin with '#', and are ignored.

The program *classify* can read unlabeled data vectors. So in these data files the class label can be ignored (the program *classify* can read also previously labeled data vectors). In all other cases the input data vectors must have labels.

If some components of some data vectors are missing (due to data collection failures or any other reason) those components should be marked with 'x' (replacing the numerical value). For example, a part of a 5-dimensional data file might look like:

1.1	2.0	0.5	4.0	5.5	aa
1.3	6.0	x	2.9	x	aa
1.9	1.5	0.1	0.3	x	aa

When vector distances are calculated for winner detection and when codebook vectors are modified, the components marked with x are ignored (see Section 7).

An example: Consider a hypothetical data file *exam.dat* that represents shades of colors in a three-component form. This file contains four samples, each one comprising a three-dimensional data vector. (The dimensionality of the vectors is given on the first line.) The labels can be any strings; here 'yellow' and 'red' are the names of the classes. The second line and the fifth line are comment lines that are ignored while reading the file.

exam.dat:

```
3
# First the yellow entries
181.0  196.0   17.0  yellow
251.0  217.0   49.0  yellow
# Then the red entries
248.0  119.0  110.0  red
213.0   64.0   87.0  red
```

4.2 Codebook file formats

The codebook vectors are stored in ASCII-form. The format of the entries is similar to that used in the input data files.

An example: The codebook file *code.dat* contains two codebook vectors that approximate to the samples of the *exam.dat* file, one codebook vector being assigned to each class.

code.dat:

```
3
224.2  209.0  36.8  yellow
232.2   94.2  99.6  red
```

5 Application of this package

5.1 The interface program *lvq_run*

The easiest way to use the *lvq_pak*-programs is to run them through the *lvq_run* interface program, whereby no separate command lines are needed. The *lvq_run* interactively asks the user about the needed parameters and takes care of running the recommended subprograms in the correct order. Therefore it is advised that the user should first use the *lvq_run* to learn the procedures, even if he or she intends to apply the subprograms directly later on.

5.2 Using the programs directly

It is also possible to run each of the subprograms contained in this package separately and directly from the console using command lines defined in Sec 6. The user should, however, take care of that the programs are then run in the correct order: first proper initialization, then training, and then tests; and that correct parameters are given (correspondence of the input and output files of subsequent programs is particularly important). Direct use facilitates, e.g., combined application of learning algorithms.

Each program needs some parameters: file names, learning parameters, sizes of codebooks, etc. All these must be given to the program in the beginning; the programs are not interactive in the sense that they do not ask for any parameters during their running.

5.3 Program parameters

Various programs need various parameters. All the parameters that are required by any program in this package have been listed below. The meaning of the parameters is obvious in most cases. The parameters can be defined in any order in the commands.

-noc	Number of codebook vectors in the codebook.
-din	Name of the input data file.
-dout	Name of the output data file.
-cin	Name of the file from which the codebook vectors are read.
-cout	Name of the file to which the codebook vectors are stored.
-rlen	Running length (number of steps) in training.
-alpha	Initial learning rate parameter.
-epsilon	Relative learning rate parameter (needed in the *lvq3* program).
-win	Window width parameter (needed in the *lvq2* and *lvq3* programs).
-knn	Number of neighbors used in knn-classification.
-alpha_type	The learning rate function type (in training routines). Possible choices are linear function (*linear*, the default) and inverse function (*inverse_t*). The linear function is defined as $\alpha(t) = \alpha(0)(1.0 - t/rlen)$ and the inverse function as $\alpha(t) = \mathrm{C}\alpha(0)/(\mathrm{C} + t)$ to compute $\alpha(t)$ for an iteration step t. In the package the constant C is defined to be $C = rlen/100.0$.
-version	Gives the version number of LVQ_PAK.

It is always possible to give the *-v* parameter (verbose parameter), which defines how much diagnostic output the program will generate. The values can range from 0 upwards, whereby greater values will generate more output; the default value is 1.

-v	Verbose parameter defining the output level.

In most programs it is possible to give the *-help 1* parameter, which lists the required and optional parameters for the program.

-help	Gives a list where the required and optional parameters are described.

In the program *lvq_run* the user is prompted by bell sound when input is requested. The bell can be silenced with the parameter *-silent* by giving some greater value than 0.

-silent	Silent parameter defining if the program *lvq_run* will give bell sound in prompt.

In the initialization and training programs the random-number generator is used to select the order of the training samples, etc. The parameter *-rand* defines whether a new seed for the random number generator is given; when any other number than zero is given, that number is used as seed, otherwise the seed is read from the system clock. Default value is zero (system clock is used).

-rand	Parameter that defines whether a new seed for the random number generator is defined.

Some auxiliary programs, not usually needed by the user, require the following parameters.

-label	Class label (string).
-cfout	Name of the classification information file.

Some examples of the use of parameters:

> *eveninit -noc 200 -din exam1.dat -cout code1.cod -knn 3*

An initialization program was called above to create a total of 200 entries into the codebook. The input entries out of which the codebook vectors were formed were read from the file *exam1.dat* and the codebook was stored to the file *code1.cod*. The entries selected for initial values of the codebook vectors were supposed to fall inside the class borders, which was tested automatically by knn-classification using the value $k = 3$.

> *lvq1 -din exam1.dat -cin code1.cod -cout code1.cod -rlen 10000 -alpha 0.05*

A training program (*lvq1*) was called. The training entries were read from the file *exam1.dat*; the codebook to be trained was read from the file *code1.cod* and the trained codebook vectors were resaved (in this case) to the same file *code1.cod*. Training was defined to take 10000 steps, but if there are fewer entries in the input file, the file is iterated randomly a sufficient number of times. The initial learning rate was set to 0.05.

> *accuracy -din exam2.dat -cin code1.cod*

The recognition accuracy achieved with the codebook vectors stored in the file *code1.cod* was tested using the test data file *exam2.dat*.

5.4 Using the command lines (an example)

The example given in this section demonstrates the direct use of command lines (it is thus not run under *lvq_run*). It is meant for an introduction to the application of this package, and it may be helpful to study it in detail. (The example may be run directly by the command *make example*.)

The data items used in the example are contained in this package. They consist of two data sets, *ex1.dat* and *ex2.dat*, one for training the codebook and the other for testing, respectively. Each data set contains 1962 cepstral-coefficient vectors picked up from continuous Finnish speech, from the same speaker. Each vector has a dimensionality of 20 and has been labeled to represent one phoneme.

Below, the data sets are processed, the codebooks are formed, and the recognition accuracy is evaluated.

5.4.1 First stage: Codebook initialization

For the initialization of the codebook vectors one has to select first a set of vectorial initial values that are picked up from the training data, one at a time. All the entries used for initialization must fall within the borders of the corresponding classes, which is automatically checked by knn-classification. The initialization program takes care of that. The number of codebook vectors also has to be decided at that point. For this speech recognition example a total number of about 200 codebook vectors seems to be a good choice.

The program *eveninit* selects the initial codebook entries from a given file with the same number of entries allocated to each class. (The program *propinit* would select the initial values so that their numbers in the respective classes are proportional to their a priori probabilities.)

> *eveninit -din ex1.dat -cout ex1e.cod -noc 200*

Now the codebook entries have been selected. This time we did not get the same number of entries to all classes because in certain classes there were not enough sample entries (e.g. in class *D* there were only four samples).

We can now check the number of entries selected for each class and the medians of the shortest distances using the program *mindist*.

> *mindist -cin ex1e.cod*

The recognition accuracy depends on the number of codebook entries allocated to each class. There does not exist any simple rule to find out the best distribution of the codebook vectors. In this example we use the method of iteratively balancing the medians of the shortest distances in all classes.

The program *balance* first computes the medians of the shortest distances for each class and corrects the distribution so that into those classes in which the distance is greater than the average, entries are added, and from those classes in which the distance is smaller than the average, some entries are deleted. Thereafter one learning cycle of the optimized-learning-rate LVQ, or the *olvq1* procedure, is automatically run within the program. After this the medians of the shortest distances are computed again and displayed. (This program may be iterated, if necessary.)

> *balance -din ex1.dat -cin ex1e.cod -cout ex1b.cod*

A global-parameter file for the learning-rate parameters, relating to the codebook *-cout*, is also created by this program, and the recursively updated values of these parameters are left in this file, from which they are automatically read when next time calling the *olvq1* program (c.f. below).

Now the codebook has been initialized and learning can begin.

5.4.2 Second stage: Codebook training

The codebook will now be trained by the fastest and most robust of all the Learning Vector Quantization algorithms, namely, the optimized-learning-rate LVQ1, *olvq1.*

> *olvq1 -din ex1.dat -cin ex1b.cod -cout ex1o.cod -rlen 5000*

The length of the training run has to be decided in the beginning. One should notice that if the program *balance* was used in initialization, one cycle of *olvq1* was already included in it; therefore this training phase can be shorter by that amount. The initial values for the learning-rate parameters are automatically read from the global-parameter file, which was created by the program *balance.* For the run length, 5000 steps have here been chosen. In this example we do not use any fine-tuning by additional training programs.

It may have become obvious from the general description of the LVQ that an unknown vector is always classified by determining its nearest neighbor in the trained codebook.

5.4.3 Third stage: Evaluation of the recognition accuracy

Now the codebook entries have been trained to their final values and the resulting recognition accuracy relative to the codebook can be tested. In the package there exists another speech entry file, *ex2.dat*, that is statistically independent of the file *ex1.dat* used in training. This file may be used for testing the trained codebook. The program *accuracy* can be used to test the recognition accuracy relating to any codebook vector file and test data file.

> *accuracy -din ex2.dat -cin ex1o.cod*

This program computes the recognition accuracy for each class separately and also the average over all the classes. The recognition accuracy resulting in this example is expected to be 90.1 %.

5.4.4 Fourth stage: Codebook visualization

NOTE: This stage is helpful but not necessary.

The trained codebook is now ready to be used for classification. In this package there are some visualization programs by which the distribution and clustering of any data entries (training samples or codebook vectors) can be checked.

The program *sammon* generates a mapping [Sammon Jr. 1969] from an n-dimensional data space to the two-dimensional plane. The two-dimensional mapping approximates to Euclidean distances of the data space, and thus visualizes the clustering of the data. The list of the mapped points can be visualized two-dimensionally. If option *-eps* is given an encapsulated postscript image of the result is produced.

> *sammon -cin ex1o.cod -cout ex1o.sam -rlen 100*

The *sammon* program will store the two-dimensional image points in a similar fashion as the input data entries are stored.

6 Description of the programs of this package

6.1 Initialization programs

The initialization programs initialize the codebook vectors. The total number of codebook vectors is given as one parameter (*-noc*).

- *eveninit* - This program selects an equal number of codebook vectors to each class and sets their initial values. The codebook vectors are picked up from the file defined by the parameter *-din* and the selected vectors are left in the file defined by the parameter *-cout*. Misclassified entries are rejected by the knn-classification check automatically (whereby k may be defined by the parameter *-knn*; its default value is 5).

  > *eveninit -noc 200 -din file.dat -cout file.cod [-knn 7]*

  The codebook vectors must be complete in all cases. Therefore, if the data input vectors are incomplete, *eveninit* first runs a replacement routine, where missing components are replaced by mean values of the corresponding components taking the class membership into account.

- *propinit* - This program defines the number of codebook vectors for each class in proportion to the a priori probabilities of the classes.

  > *propinit -noc 200 -din file.dat -cout file.cod [-knn 7]*

  The codebook vectors must be complete in all cases. Therefore, if the data input vectors are incomplete, *propinit* first runs a replacement routine, where missing components are replaced by mean values of the corresponding components taking the class membership into account.

- *balance* - This program adjusts the numbers of codebook vectors stored in the file defined by the parameter *-cin* and using training data stored in the file defined by the parameter *-din* so that the medians of the shortest distances between the codebook vectors in all classes are equalized. One learning cycle (all the data vectors in the training file are used once) based on the optimized-learning-rate LVQ1 algorithm is included in this program, whereby the 'learned' codebook vectors are left in the file indicated by the *-cout* parameter. Every time when this program is called, the initial learning rate parameters of the optimized-learning-rate LVQ1 algorithm, for reasons explained in Sec. 1.3.5, are set to 0.3, and when the program has been run, the values determined by Eq. (1) are left in a global parameter file associated with the respective codebook vector file. Notice that even if the program *balance* is iterated several times, only the last iteration is taken into account in *olvq1* learning.

  > *balance -din file.dat -cin code1.cod -cout code2.cod [-knn 7]*

6.2 Training programs

- *olvq1* - This is the optimized-learning-rate LVQ1 algorithm, recommended for the main learning algorithm. It must always be preceded by an initialization program *eveninit* or *propinit* and possibly by the program *balance*, too. No explicit learning rate parameters are defined in the command. If the initialization stage was terminated with the *balance* program, optimized default values for the learning rate parameters were left by that program in the respective parameter file, from which they are automatically read by the *olvq1*. If for initialization only the *eveninit* or the *propinit* program was used, the default values of the initial learning rates were set equal to 0.3 in those programs.

  The training data is taken from the file defined by the parameter *-din*, and the codebook vectors from the file defined by the parameter *-cin*, respectively. The trained codebook vectors are left in the file defined by the parameter *-cout* (which can be the same as *-cin*). The number of training steps is defined by the parameter *-rlen*.

  > *olvq1 -din file.dat -cin file1.cod -cout file2.cod -rlen 10000 [-alpha_type inverse_t] [-snapinterval 1000] [-snapfile file.snap]*

- *lvq1* - The original LVQ1 algorithm. It can be used (with low *-alpha* value) for an additional fine-tuning stage in learning. The training data is taken from the file defined by *-din* and the codebook vectors to be fine tuned from the file defined by *-cin*; the tuned codebook vectors are left in the file defined by *-cout*.

  > *lvq1 -din file.dat -cin file1.cod -cout file2.cod -alpha 0.05 -rlen 40000 [-alpha_type inverse_t] [-snapinterval 1000] [-snapfile file.snap]*

- *lvq2* - The LVQ2.1 version of the LVQ algorithms. It can be used (with low *-alpha* value) for another additional fine-tuning stage in learning. The relative width of the 'window' into which the training data must fall is defined by the parameter *-win*.

  > *lvq2 -din file.dat -cin file1.cod -cout file2.cod -alpha 0.05 -rlen 40000 -win 0.3 [-alpha_type inverse_t] [-snapinterval 1000] [-snapfile file.snap]*

- *lvq3* - The LVQ3 version of the LVQ algorithms. It can be used (with low *-alpha* value) for additional fine-tuning stage in learning. The relative learning rate parameter *-epsilon* is used (multiplied by the parameter *-alpha*) when both of the nearest codebook vectors belong to the same class. The relative width of the 'window' into which the training data must fall is defined by the parameter *-win*.

  > *lvq3 -din file.dat -cin file1.cod -cout file2.cod -alpha 0.05 -epsilon 0.1 -rlen 40000 -win 0.3 [-alpha_type inverse_t] [-snapinterval 1000] [-snapfile file.snap]*

6.3 Recognition accuracy program

- *accuracy* - The recognition accuracy is evaluated. The codebook vectors are taken from the file defined by the parameter *-cin*, and the test entries from the file defined by the parameter *-din*, respectively. Optionally this program creates a classification information file needed in testing the statistical significance of the difference between two classifiers by using program *mcnemar*.

  > *accuracy -din file.dat -cin file.cod [-cfout file.cfo]*

6.4 Classification program

- *classify* - The classifications of unknown data vectors are found. The codebook vectors are taken from the file defined by the parameter *-cin*, and the entries to be classified from the file defined by the parameter *-din*, respectively. The classification results are saved to the file defined by the parameter *-dout*. Optionally this program creates a classification file that contains only the labels of classified vectors.

  > *classify -din file.dat -cin file.cod -dout file.cla [-cfout file.cfo]*

6.5 Monitoring programs

- *showlabs* - Displays the class labels and the numbers of entries in each class of a given file.

  > *showlabs -cin file.cod*

- *mindist* - Displays the medians of the shortest distances between codebook vectors in each class and the standard deviations of entries in each class in the corresponding input data file (if given).

  > *mindist -cin file.cod [-din data.dat]*

- *stddev* - Displays the medians of the shortest distances between data vectors in each class and the standard deviations of entries in each class.

  > *stddev -din data.dat*

- *sammon* - Generates the Sammon mapping [Sammon Jr. 1969] from n-dimensional input vectors to 2-dimensional points on a plane whereby the distances between the image vectors tend to approximate to Euclidean distances of the input vectors. If option *-eps* is given an encapsulated postscript image of the result is produced. Name of the eps-file is generated by using the output file basename (up to the last dot in the name) and adding the ending *_sa.eps* to the output filename. If option *-ps* is given a postscript image of the result is produced. Name of the ps-file is generated by using the output file basename (up to the last dot in the name) and adding the ending *_sa.ps* to the output filename.

  In the following example, if the option *-eps 1* is given, an eps file named *file_sa.eps* is generated.

  > *sammon -cin file.cod -cout file.sam -rlen 100 [-rand 1] [-eps 1] [-ps 1]*

- *mcnemar* - Computes the statistical significance of the difference between classification results of two classifiers that have been tested with the same data. As input, two classification information files created by *accuracy* are required.

  > *mcnemar file1.cfo file2.cfo*

6.6 Auxiliary subprograms

These programs are normally not applied by the user. They are mainly used by the other programs as subroutines.

- *elimin* - Eliminates those entries in a given file that are classified to the wrong class when using the knn-classifier. The purpose is to ignore those entries that lie on the wrong side of the class borders when initializing the codebook vectors. (Here 7 nearest neighbors are used in the classification.)

  > *elimin -din file.dat -cout file.elim -knn 7*

- *extract* - Selects and saves only those entries that belong to a given class (here class 'K' is given).

  > *extract -din file.dat -cout file.ext -label K*

- *pick* - Picks a given number of entries from a file.

  > *pick -din file.dat -cout file.pic -noc 10*

- *setlabel* - Sets the labels of the codebook vectors in a codebook by finding a given number of nearest entries in the entry file and selecting the label by majority voting over them.

  > *setlabel -din file.dat -cin file1.cod -cout file2.cod -knn 5*

- *knntest* - The recognition accuracy is computed using the k-nearest-neighbors classifier. Each entry is classified using majority voting with respect to a given number of nearest neighbors in the codebook. This algorithm is primarily meant for a subprogram in the initialization of the codebook vectors, but can be used as an independent classifier, too.

  > *knntest -din file.dat -cin file.cod -knn 5*

7 Advanced features

Some more advanced features has been added into the LVQ_PAK program package in Version 3.0. These features are intended to ease the usage of the package by offering ways to use e.g. compressed data files directly and to save snapshots of the map during the training run.

The advanced features include:

- Missing components in input data entries are allowed
- Buffered loading (the whole data file need not be loaded into memory at once)
- Reading and writing of:
  - compressed files
  - stdin/stdout
  - piped command
- Snapshots of the codebook during teaching
- Environment variables

Missing components in input data entries

In many applications, sensor failures, recording errors and resource limitations can prevent data collection to complete each input vector. Such incomplete training examples still contain useful information, however, and can be used in pattern recognition. For example, partial data can still be used to determine the distribution statistics of the available vector components [Samad et al. 1992][Kaski 1995].

For incomplete input data vectors the LVQ_PAK has the possibility to mark the missing values by a predefined string ('*x*' by default). The LVQ_PAK routines will compute the distance calculations and reference vector modification steps using the available data components.

NOTE: If there are missing components in the data files, some functions may produce misleading results. For example, if an input vector is compared against several data vectors where some of the vectors have missing components, the distances are not comparable, because there are then different number of components in different cases. On the other hand, if an incomplete data vector is compared against a set of complete codebook vectors, the distances are comparable, because in all cases there are an identical number of components.

NOTE: If some specific component is missing in *all* input data vectors, the results conserning that component are meaningless. The component should be removed from the data files.

Buffered loading

This means that the whole data set doesn't have to be loaded in memory all the time. LVQ_PAK can be set, for example, to hold max 10000 lines of data in memory at a time. When the 10000 data vectors have been used, the next 10000 data vectors are loaded over the old ones. The buffered reading is transparent to the user and it works also with compressed files.

Note that when the whole file has been read once and we want to reread it, the file has to be rewound (for regular files) or the uncompressing command has to be rerun. This is done automatically and the user need not to worry about it, but some restrictions are enforced on the input file: If the source is a pipe, it can't be rewound. Regular files, compressed files and standard input (if it is a file) work. Pipes work fine if you don't have to rewind them, ie. there is no end in the data, or the number of iterations is smaller than the number of data vectors.

-buffer Defines the number of lines of input data file that are read at a time.

Most programs support the buffered reading of data files. It is activated with the command line option *-buffer* followed with the maximum number of data vectors to be kept in memory. For example, to read the input data file 10000 lines at a time one uses:

> *lvq1 -buffer 10000 ...*

Reading and writing compressed files

To read or write compressed files just put the suffix *.gz* at the end of the filename. The file is automatically uncompressed or compressed as the file is being read or written. LVQ_PAK uses 'gzip' for compressing and uncompressing. It can also read files compressed with regular UNIX compress-command. The commands used for compressing and decompressing can be changed with command line options or at compile time.

Example: with *lvq1*, to use a compressed data file for teaching:

> *lvq1 -din data.dat.gz ...*

Reading and writing stdin/stdout

To use standard input or output, use the minus sign ('-') as a filename. Data is then read from *stdin* and written to *stdout*. For example, to read training data from *stdin* with vsom:

> *lvq1 -din - ...*

Reading and writing piped commands

If you use a filename that starts with the UNIX pipe character ('|'), the filename is executed as a command. If the file is opened for writing the output of the LVQ command is piped to the command as standard input. Likewise, when the file is opened for reading the output of the command is read by the LVQ programs.

For example:

> *lvq1 -cin "|eveninit ... " ...*

would start the program *eveninit* when it wants to read the initial codebook. However, the same thing could be done with:

> *eveninit ... | lvq1 -cin - ...*

Snapshots

Saves snapshots of the codebook during training.

-snapinterval Interval between snapshots.

-snapfile Name of the snapfile. If the name given contains string '%d', the number of iterations taken so far is included to the filename.

The interval between snapshots is specified with the option *-snapinterval*. The snapshot filename can be specified with the option *-snapfile*. If no filename is given, the name of the output code file is used. The filename is actually passed to 'sprintf(3)' as the format string and the number of iterations so far is passed as the next argument. For example:

> *lvq1 -snapinterval 10000 -snapfile "ex.%d.cod" ...*

gives you snapshots files every 10000 iterations with names starting with: *ex.10000.cod, ex.20000.cod, ex.30000.cod*, etc.

Environmental variables

Some defaults can be set with environment variables:

LVQSOM_COMPRESS_COMMAND Defines the command used to compress files.
Default: "gzip -9 -c >%s"

LVQSOM_UNCOMPRESS_COMMAND Defines the command used to decompress files. Default: "gzip -d -c %s"

LVQSOM_MASK_STR Defines the string which is used to replace missing input vector components. Default: "x"

Other new options

-mask_str Defines the string which is used to replace missing input vector components.

-compress_cmd Defines the compress command.

-uncompress_cmd Defines the uncompress command.

By default the components of the data vectors that are marked with '*x*' are ignored. This string can be changed with the *-mask_str* option. For example,
> *lvq1 -mask_str "MIS"* ...
would ignore components thats are marked with string '*MIS*' instead of '*x*'. The string is case insensitive.
The command used to compress files can be changed by giving the option *-compress_cmd*. Similarly the uncompress command can be changed by giving the option *-uncompress_cmd*.

8 Comments and experiences of the use of this package

Comments and experiences of the installation and use of these programs are welcome, and may be sent to the e-mail address *lvq@cochlea.hut.fi*.

8.1 Changes in the package

No changes to the central recognition algorithms have been made; the latter have been used successfully as such over many years. Therefore, if you already have used previous Versions of *LVQ_PAK*, you should not notice any significant differences in accuracies yielded by the Version 3.1, neither. However, the total computing time, on account of the improved *balance* program as well as some improvements in the best matching unit search, will now be shorter. The following are the details that have been changed from the Version 1.0:

0. The only change made to Version 1.1 was a bug fix in the allocation of memory.

1. In Version 2.0 the following changes have been made: For the recursion of $\alpha(t)$ in Eq. (9), another justification (cf. text) has now been found. Thereby the indexing in Eq. (9) is slightly changed from Versions 1.0 and 1.1. This change has a negligible effect on the numerical results, but in publications you should refer to the new form.

2. The program *balance* has been made faster in several ways, by avoiding unnecessary operations. In Versions 1.0 and 1.1 it eventually also changed the number of codebook vectors defined in the command line. In Version 2.0 this number is kept constant. If no samples after the *knn* test are left in some class, one codebook vector, picked from the samples, is anyway taken to it.

3. Since the random-number generators in different computers are not identical, we have programmed our own formula into the procedures; now the examples computed by the different machines are supposed to yield identical results (provided that they use a similar arithmetic).

4. The LVQ3 algorithm has been added.

5. Several amendments, which are invisible to the user but make the system more logical, have been made to control programs.

6. In Version 2.1 the following changes have been made: It is now possible to put comment lines into the data files. The comment lines begin with '#' and they are ignored while reading the data.

7. We have included a program *classify*, that produces the classifications of unknown data vectors.

8. It is possible to use unlabeled data vectors with the program *classify*.

9. If there are equal vectors in the input data set of the *sammon* program, it now discards all of them except one. Previously such vectors corrupted the computation of the Sammon mapping.

10. We have corrected one error in the documentation. The Eq. 5 is now in the correct form.

11. The routines for the search of the best matching unit are improved. For each codebook vector the computation of the distance between the sample vector and the codebook vector is terminated if the subdistance is already greater than the distance for the current best matching unit. This improvement will decrease the computing time considerably for longer vector lengths.

12. In Version 3.0 it is possible to have missing components in input data vectors.

13. In Version 3.0 it is possible to use an inverse function as a learning rate function $\alpha(t)$.

14. In Version 3.0 it is possible to read the input data files in pieces, i.e. to have only a portion of the whole data in main memory at a time. This will enable using the SOM_PAK programs in PC-machines with large data files.

15. In version 3.0 there are several new 'advanced' features to allow reading and writing of compressed files, stdin and stdout, and piped commands.

16. In version 3.0 it is now possible to save 'snapshots' of the state of codebook during training.

17. The only change made to Version 3.1 was a bug fix in the random ordering of data.

References

[Kaski 1995] Sami Kaski, Teuvo Kohonen. *Structures of Welfare and Poverty in the World Discovered by the Self-Organizing Map.* Report A24, Helsinki University of Technology, Laboratory of Computer and Information Technology, 1995.

[Kohonen 1989] Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin-Heidelberg-New York-Tokio, 3 edition, 1989.

[Kohonen 1990a] Teuvo Kohonen. Improved versions of learning vector quantization. In *Proceedings of the International Joint Conference on Neural Networks*, pages I 545–550, San Diego, June 1990.

[Kohonen 1990b] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[Kohonen 1990c] Teuvo Kohonen. Statistical pattern recognition revisited. In *Advanced Neural Computers*, pages 137–144, 1990.

[Kohonen et al. 1992] Teuvo Kohonen, Jari Kangas, Jorma Laaksonen, Kari Torkkola. LVQ_PAK: A program package for the correct application of Learning Vector Quantization algorithms. In *Proceedings of the International Joint Conference on Neural Networks*, pages I 725–730, Baltimore, June 1992. IEEE.

[Kohonen 1992] Teuvo Kohonen. New Developments of Learning Vector Quantization and the Self-Organizing Map. In *Symposium on Neural Networks; Alliances and Perspectives in Senri 1992 (SYNAPSE'92)*, Osaka, Japan.

[Kohonen 1995] Teuvo Kohonen. *Self-Organizing Maps.* Springer-Verlag, Heidelberg, 1995.

[Makhoul et al. 1985] John Makhoul, S. Roucos, H. Gish. Vector quantization in speech coding. *Proceedings of the IEEE*, 73:1551–1588, November 1985.

[Samad et al. 1992] T. Samad, S. A. Harp. Self-organization with partial data. *Network: Computation in Neural Systems*, 3(2):205-212, 1992.

[Sammon Jr. 1969] John W. Sammon Jr. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969.