Principal Component Analysis
Task #2

Student: Sergi Blanco Cuaresma

January 12, 2011

Abstract

Solutions to the classification tasks (OCR) from the subject Statistics, Monte Carlo Methods and Data Processing - Master in Astrophysics, Particle Physics and Cosmology (Universitat de Barcelona).

Contents

1 Introduction

1.1 Description

In order to provide a solution for the different tasks and facilitate future results validations, several automatic processes has been implemented using the R language and Ruby. This document presents the results for each task and its respective analysis.

Weka[1] has been used for attribute analysis and classification algorithms testing, while neural networks has been designed with JavaNNS[2].

Apart from this report, the following directories has been included:

1. "Original ARFF Statistical Analysis": Original input data and statistical results produced by the R script.
2. "Weka Experimenter": Original input data + two reduced versions (PCA 95% and 75%) and experimenter results.
3. "Convert + rescale ARFF to PAT Train + Test": Original input data + two reduced versions (PCA 95% and 75%) and the 12 output files produced by the Ruby script:
   (a) Train (70%) and test subdatasets (30%) for each input file (original, PCA95 and PCA70), using:
      i. Binary representation of letters
      ii. Numeric representation of letters
4. "JavaNNS results":
   (a) Input: Data converted in point 3 and initial neural networks.
   (b) Output: Trained networks and data results.
5. "JavaNNS comparison": Analysis of JavaNNS data results done by the Ruby script.

2 Letter Image Recognition Data (OCR)

2.1 Description of the sample

Disclaimer: Most of the information presented in this section has been obtained directly from the paper "Letter Recognition Using Holland-Style Adaptive Classifiers" (see reference section).

2.1.1 Context

The sample used in this study has been obtained from the paper "Letter Recognition Using Holland-Style Adaptive Classifiers", where the authors generated a set of 20,000 unique letter images by randomly distorting pixel images of the 26 uppercase letters from 20 different commercial fonts. Concretely, this study will use a subset of 9.940 letters from the original one.

[1] http://www.cs.waikato.ac.nz/ml/weka/
[2] http://www.ra.cs.uni-tuebingen.de/software/JavaNNS/

2.1.2 Acquisition process

For the letters generation, the 20 different used fonts (Roman alphabet) represent five different stroke styles (simplex, duplex, triplex, complex, and Gothic) and six different letter styles (block, script, italic, English, Italian, and German). Each character were generated with random uniformly distributed parameter values for font type, letter of the alphabet, linear magnification (from 1.0 to 1.6), aspect ratio (or horizontal magnification from 1.0 to 1.5), and horizontal and vertical "warp" (stretching/shrinking a region). The resulting image for each character consisted in a picture of 45 pixels high by 45 pixels wide, which was fairly recognizable by humans.

2.1.3 Objects

The features of each of the 9.940 characters (composed by 26 uppercase letters) are summarized in terms of a class and 16 primitive numerical attributes:

1. Uppercase letter that represents.
2. The horizontal position, counting pixels from the left edge of the image, of the center of the smallest rectangular box that can be drawn with all "on" pixels inside the box.
3. The vertical position, counting pixels from the bottom, of the above box.
4. The width, in pixels, of the box.
5. The height, in pixels, of the box.
6. The total number of "on" pixels in the character image.
7. The mean horizontal position of all "on" pixels relative to the center of the box and divided by the width of the box. This feature has a negative value if the image is "left-heavy" as would be the case for the letter L.
8. The mean vertical position of all "on" pixels relative to the center of the box and divided by the height of the box.
9. The mean squared value of the horizontal pixel distances as measured in 6 above. This attribute will have a higher value for images whose pixels are more widely separated in the horizontal direction as would be the case for the letters W or M.
10. The mean squared value of the vertical pixel distances as measured in 7 above.
11. The mean product of the horizontal and vertical distances for each "on" pixel as measured in 6 and 7 above. This attribute has a positive value for diagonal lines that run from bottom left to top right and a negative value for diagonal lines from top left to bottom right.
12. The mean value of the squared horizontal distance tunes the vertical distance for each "on" pixel. This measures the correlation of the horizontal variance with the vertical position.

13. The mean value of the squared vertical distance times the horizontal distance for each "on" pixel. This measures the correlation of the vertical variance with the horizontal position.

14. The mean number of edges (an "on" pixel immediately to the right of either an "off pixel or the image boundary) encountered when making systematic scans from left to right at all vertical positions within the box. This measure distinguishes between letters like "W" or "M" and letters like "I" or "L."

15. The sum of the vertical positions of edges encountered as measured in 13 above. This feature will give a higher value if there are more edges at the top of the box, as in the letter "Y."

16. The mean number of edges (an "on" pixel immediately above either an "off pixel or the image boundary) encountered when making systematic scans of the image from bottom to top over all horizontal positions within the box.

17. The sum of horizontal positions of edges encountered as measured in 15 above.

2.1.4 General characteristics

The subset sample is composed by 9.940 characters which represent 26 uppercase letters. They are in a ARFF file format (Weka) with attributes stored in the same order they have been presented in the above section.

2.2 Statistical analysis

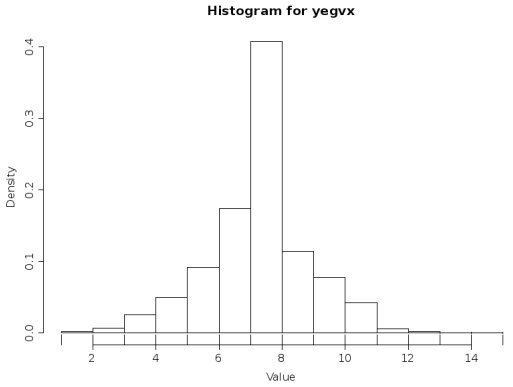
R is the perfect environment for data treatment and general statistics generation. A small R script has been created for this purpose (code in section 7 on page 29) and the results are presented in this section.

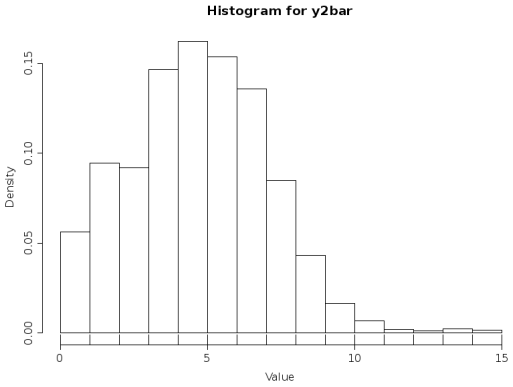
2.2.1 Key indicators

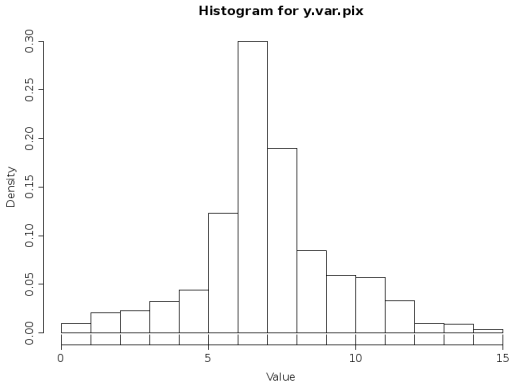
A brief statistical analysis has been perform on the attributes of the 9.940 characters:

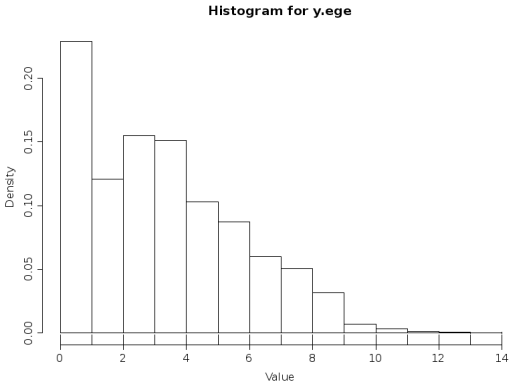
Attribute	Mean	Variance	Std Deviation	Min	Max	Median
yegvx	7.78	2.56	1.60	1	15	1
y2bar	5.18	5.70	2.39	0	15	0
y.var.pix	7.52	5.46	2.34	0	15	0
y.ege	3.69	6.62	2.57	0	14	0
y.box	6.99	11.02	3.32	0	15	0
xybar	8.30	6.22	2.49	0	15	0
xy2br	7.93	4.32	2.08	0	15	0
xegvy	8.34	2.39	1.55	0	15	0
x2ybr	6.49	6.91	2.63	0	15	0
x2bar	4.63	7.45	2.73	0	15	0
x.var.pix	6.88	4.10	2.02	0	15	0
x.ege	3.03	5.45	2.33	0	15	0
x.box	4.01	3.64	1.91	0	15	0
width.box	5.11	4.05	2.01	0	15	0
npix	3.50	4.85	2.20	0	15	0
height.box	5.35	5.14	2.27	0	15	0

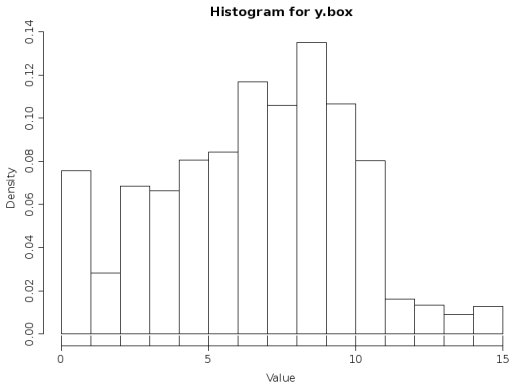
2.2.2 Histograms

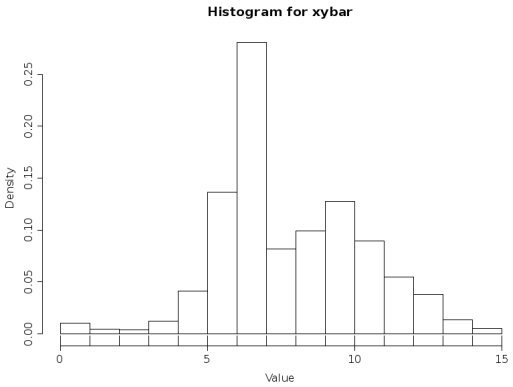

Histogram for yegvx
Density
Value

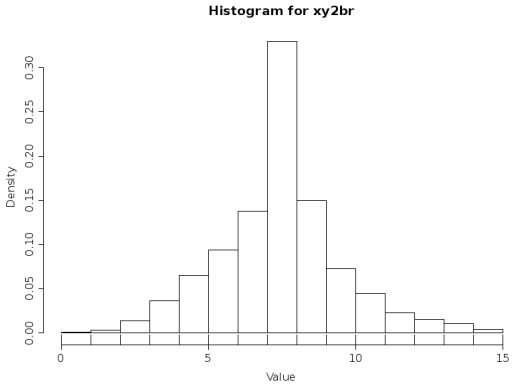

Histogram for y2bar
Density
Value

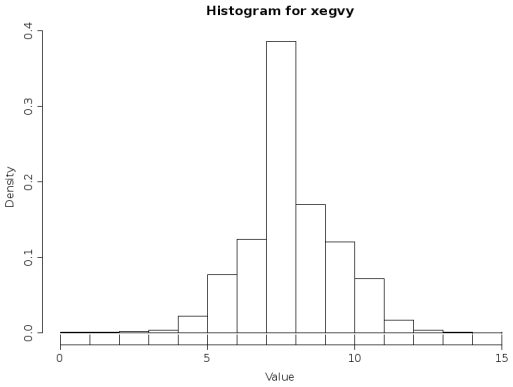

Histogram for y.var.pix
Density
Value

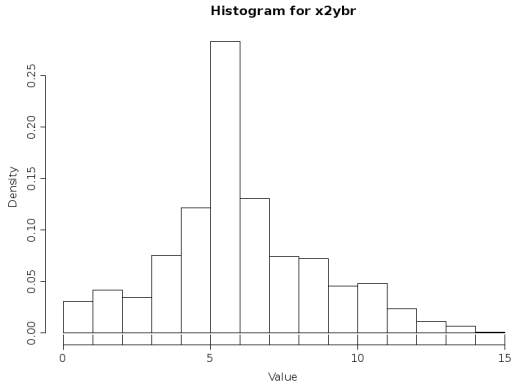

Histogram for y.ege
Density
Value

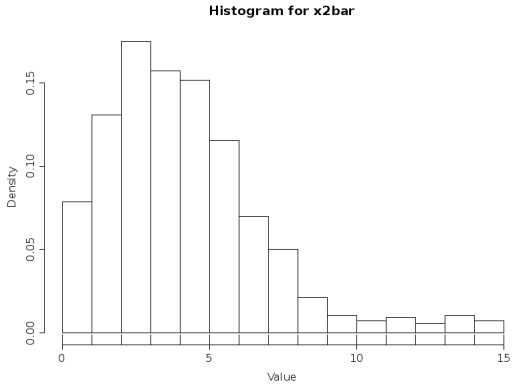

Histogram for y.box
Density
Value

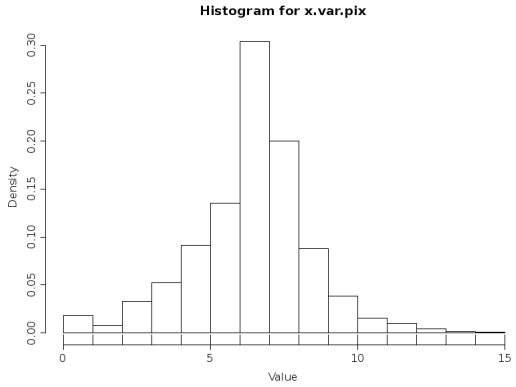

Histogram for xybar
Density
Value

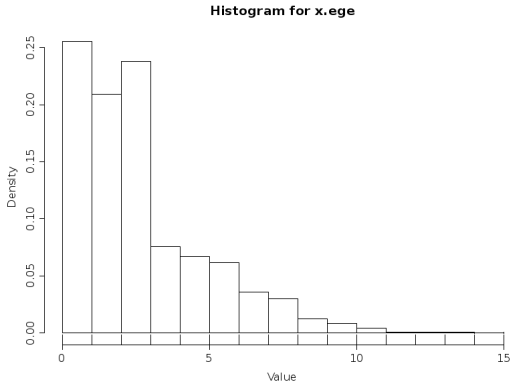

Histogram for xy2br
Density
Value

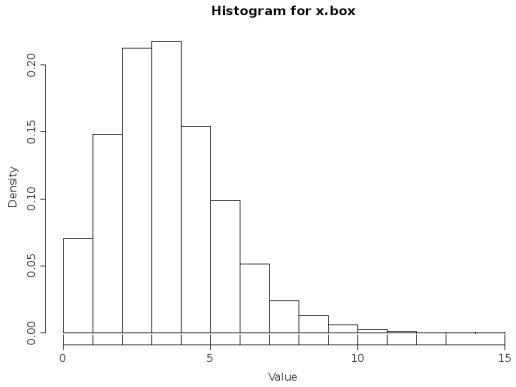

Histogram for xegvy
Density
Value

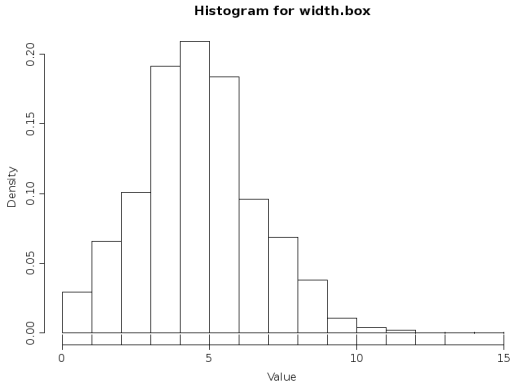

Histogram for x2ybr
Density
Value

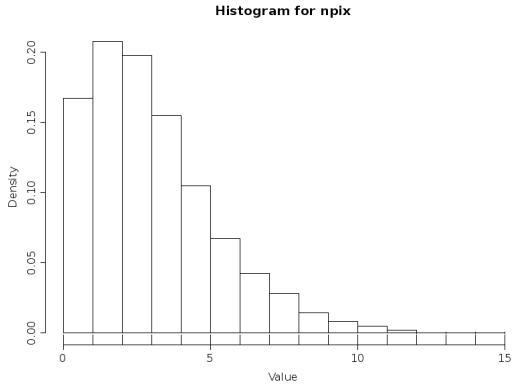

Histogram for x2bar
Density
Value

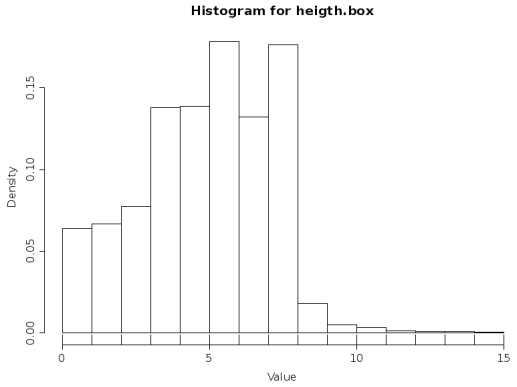

Histogram for x.var.pix
Density
Value


Histogram for x.ege
Density
Value


Histogram for x.box
Density
Value


Histogram for width.box
Density
Value


Histogram for npix
Density
Value


Histogram for heigth.box
Density
Value

2.3 Pretreatment

2.3.1 Principal Components Analysis: 95% of variance coverage

Using Weka it is possible to perform a Principal Components Analysis that will allow us to reduce the dimensionality of the problem by identification of linear correlation between attributes. With 95% of variance coverage, it is possible to make a reduction from 16 attributes to 12:

```
=== Run information ===

Evaluator:     weka.attributeSelection.PrincipalComponents -R 0.95 -A 5
Search:        weka.attributeSelection.Ranker -T -1.7976931348623157E308 -N -1
Relation:      letter-recognition
Instances:     9940
Attributes:    17
               class
               x-box
               y-box
               width-box
               heigth-box
               npix
               x-var-pix
               y-var-pix
               x2bar
               y2bar
               xybar
               x2ybr
               xy2br
               x-ege
               xegvy
               y-ege
               yegvx
Evaluation mode:    evaluate on all training data


=== Attribute Selection on all input data ===

Search Method:
        Attribute ranking.

Attribute Evaluator (unsupervised):
        Principal Components Attribute Transformer

Correlation matrix
   1       0.76    0.85    0.68    0.62   -0.03    0.04    0.02    0.06    0.15    0.03   -0.05    0.48    0.08    0.28   -0.1
   0.76    1       0.67    0.82    0.56    0.04   -0.04   -0.02    0.09    0.16   -0.05   -0.01    0.27   -0.01    0.24   -0.05
   0.85    0.67    1       0.66    0.77    0.07    0.02   -0.1     0.06    0.12    0.01   -0.05    0.55    0.03    0.27   -0.11
   0.68    0.82    0.66    1       0.65    0.03   -0.01    0.08    0.06    0.01   -0.01    0.02    0.27    0.02    0.3    -0.02
   0.62    0.56    0.77    0.65    1       0.14   -0.03   -0.01   -0.06   -0.07   -0.08   -0.04    0.63    0       0.5    -0.06
  -0.03    0.04    0.07    0.03    0.14    1      -0.35   -0.05   -0.13    0.08   -0.33   -0.03    0.16   -0.25    0.13    0.25
   0.04   -0.04    0.02   -0.01   -0.03   -0.35    1      -0.13   -0.04    0.19    0.6    -0.28   -0.05    0.55   -0.08   -0.21
   0.02   -0.02   -0.1     0.08   -0.01   -0.05   -0.13    1      -0.2    -0.32    0.04    0.08    0.15   -0.09    0       0.19
   0.06    0.09    0.06    0.06   -0.06   -0.13   -0.04   -0.2     1       0.14   -0.06    0.12   -0.38   -0.04    0.28   -0.06
   0.15    0.16    0.12    0.01   -0.07    0.08    0.19   -0.32    0.14    1       0.06   -0.1    -0.18    0.03   -0.09   -0.12
   0.03   -0.05    0.01   -0.01   -0.08   -0.33    0.6     0.04   -0.06    0.06    1       0.06    0.04    0.52   -0.23   -0.24
  -0.05   -0.01   -0.05    0.02   -0.04   -0.03   -0.28    0.08    0.12   -0.1     0.06    1      -0.01   -0.19    0.05    0.25
   0.48    0.27    0.55    0.27    0.63    0.16   -0.05    0.15   -0.38   -0.18    0.04   -0.01    1      -0.02    0.12   -0.04
   0.08   -0.01    0.03    0.02    0      -0.25    0.55   -0.09   -0.04    0.03    0.52   -0.19   -0.02    1      -0.06   -0.19
   0.28    0.24    0.27    0.3     0.5     0.13   -0.08    0       0.28   -0.09   -0.23    0.05    0.12   -0.06    1       0.14
  -0.1    -0.05   -0.11   -0.02   -0.06    0.25   -0.21    0.19   -0.06   -0.12   -0.24    0.25   -0.04   -0.19    0.14    1


eigenvalue      proportion      cumulative
   4.31144         0.26946         0.26946        -0.436width-box-0.426x-box-0.417npix-0.4heigth-box-0.399y-box...
   2.62272         0.16392         0.43338        -0.509y-var-pix-0.467x2ybr-0.453xegvy+0.324x-var-pix+0.314yegvx...
   1.7285          0.10803         0.54142        -0.556y2bar+0.452x2bar-0.433xybar+0.43 x-ege+0.184x2ybr...
   1.35958         0.08497         0.62639        -0.497xy2br+0.421x-var-pix-0.383y2bar+0.322xybar-0.313x2bar...
   1.05434         0.0659          0.69229        0.613y-ege-0.445xy2br-0.371xybar-0.239y-box+0.232npix...
   0.97429         0.06089         0.75318        -0.636yegvx-0.435x-var-pix-0.281xegvy-0.278xybar-0.257y-var-pix...
   0.89406         0.05588         0.80906        -0.518xy2br+0.494x2bar-0.36x-ege+0.296yegvx+0.273y-box...
   0.62559         0.0391          0.84816        0.547x2bar+0.503xybar-0.304xegvy+0.293y-ege-0.239heigth-box...
   0.59126         0.03695         0.88511        -0.648x-var-pix+0.42 yegvx-0.294xegvy-0.253x2ybr+0.221y-var-pix...
   0.49325         0.03083         0.91594        -0.536y2bar+0.357y-ege+0.314heigth-box-0.303width-box-0.291yegvx...
   0.4304          0.0269          0.94284        0.633xegvy-0.414y-var-pix-0.338x2ybr+0.291xybar-0.205y2bar...
   0.26207         0.01638         0.95922        0.566npix-0.512x-box-0.332y-ege+0.27 heigth-box-0.236y-box...

Eigenvectors
 V1      V2      V3      V4      V5      V6      V7      V8      V9      V10     V11     V12
-0.426  -0.0881 -0.0373 -0.0235 -0.1447  0.042   0.0723  0.0563  0.1042 -0.2668  0.1816 -0.5116 x-box
-0.3987 -0.0243 -0.1375 -0.0605 -0.2388  0.0507  0.2728 -0.2255 -0.0924  0.1449 -0.039  -0.236  y-box
-0.4361 -0.0588 -0.0328  0.064  -0.073   0.0142 -0.1302  0.0138  0.0943 -0.3033 -0.0409  0.0826 width-box
-0.4001 -0.0171 -0.0455 -0.155  -0.1254  0.0333  0.2726 -0.2388 -0.1872  0.3144 -0.1778  0.2701 heigth-box
-0.4171  0.032   0.0999  0.0349  0.232  -0.0433 -0.1958  0.0428  0.0405  0.1449 -0.0535  0.5661 npix
-0.0572  0.3236 -0.0163  0.4211  0.0502 -0.435   0.0046  0.0614 -0.6479 -0.0668 -0.1938 -0.1222 x-var-pix
 0.0197 -0.5094  0.0349 -0.0177  0.1565 -0.2573  0.105   0.113   0.2206  0.1424 -0.4137 -0.0305 y-var-pix
-0.0021  0.123   0.4517 -0.3125 -0.0738  0.1359  0.4942  0.5473 -0.2038 -0.0761  0.1235  0.1316 x2bar
-0.0101  0.003  -0.5557 -0.3833  0.1559  0.1109 -0.0651  0.1656 -0.2011 -0.5362 -0.2051  0.1897 y2bar
-0.0263 -0.1553 -0.4327  0.3216 -0.3713 -0.278   0.1125  0.5031  0.1638  0.1328  0.2906  0.2188 xybar
 0.0305 -0.4672  0.1843 -0.206  -0.1763 -0.2248 -0.1646  0.1699 -0.2528 -0.0126 -0.3382 -0.1604 x2ybr
 0.0178  0.1897  0.0103 -0.4966 -0.4445 -0.2315 -0.5175  0.0003 -0.0847  0.2275  0.1625  0.0067 xy2br
-0.2883  0.0305  0.4295  0.1984  0.0181 -0.0114 -0.3597  0.1944  0.1414 -0.2485  0.0026 -0.0037 x-ege
-0.0006 -0.4535  0.0968 -0.0795  0.2156 -0.2811  0.0588 -0.3043 -0.2937 -0.1846  0.6331  0.1501 xegvy
-0.2131  0.1542 -0.1672 -0.2532  0.6126 -0.2017 -0.0576  0.2926  0.0627  0.3573  0.1619 -0.3325 y-ege
 0.0339  0.3142  0.096  -0.2048 -0.0423 -0.6364  0.2958 -0.213   0.42   -0.2914 -0.105   0.0798 yegvx

Ranked attributes:
 0.7305    1 -0.436width-box-0.426x-box-0.417npix-0.4heigth-box-0.399y-box...
 0.5666    2 -0.509y-var-pix-0.467x2ybr-0.453xegvy+0.324x-var-pix+0.314yegvx...
 0.4586    3 -0.556y2bar+0.452x2bar-0.433xybar+0.43 x-ege+0.184x2ybr...
 0.3736    4 -0.497xy2br+0.421x-var-pix-0.383y2bar+0.322xybar-0.313x2bar...
 0.3077    5 0.613y-ege-0.445xy2br-0.371xybar-0.239y-box+0.232npix...
 0.2468    6 -0.636yegvx-0.435x-var-pix-0.281xegvy-0.278xybar-0.257y-var-pix...
```

```
 0.1909    7  -0.518xy2br+0.494x2bar-0.36x-ege+0.296yegvx+0.273y-box...
 0.1518    8  0.547x2bar+0.503xybar-0.304xegvy+0.293y-ege-0.239heigth-box...
 0.1149    9  -0.648x-var-pix+0.42 yegvx-0.294xegvy-0.253x2ybr+0.221y-var-pix...
 0.0841   10  -0.536y2bar+0.357y-ege+0.314heigth-box-0.303width-box-0.291yegvx...
 0.0572   11  0.633xegvy-0.414y-var-pix-0.338x2ybr+0.291xybar-0.205y2bar...
 0.0408   12  0.566npix-0.512x-box-0.332y-ege+0.27 heigth-box-0.236y-box...

Selected attributes: 1,2,3,4,5,6,7,8,9,10,11,12 : 12
```

A file has been generated named "OCR.PCA95.arff" with the transformed data. Consequently, data is represented by 12 different variable, which correspond to a combination of the 16 original attributes that has been presented in previous sections.

2.3.2 Principal Components Analysis: 70% of variance coverage

As exposed in the previous section, Weka can perform a Principal Components Analysis that will allow us to reduce the dimensionality of the problem by identification of linear correlation between attributes. In this case a variance coverage of 70% has been applied, reducing the number of attributes from 16 to 6:

```
=== Run information ===

Evaluator:     weka.attributeSelection.PrincipalComponents -R 0.7 -A 5
Search:        weka.attributeSelection.Ranker -T -1.7976931348623157E308 -N -1
Relation:      letter-recognition-weka.filters.unsupervised.attribute.Reorder-R2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,1
Instances:     9940
Attributes:    17
               x-box
               y-box
               width-box
               heigth-box
               npix
               x-var-pix
               y-var-pix
               x2bar
               y2bar
               xybar
               x2ybr
               xy2br
               x-ege
               xegvy
               y-ege
               yegvx
               class
Evaluation mode:    evaluate on all training data


=== Attribute Selection on all input data ===

Search Method:
        Attribute ranking.

Attribute Evaluator (unsupervised):
        Principal Components Attribute Transformer

Correlation matrix
   1       0.76    0.85    0.68    0.62   -0.03    0.04    0.02    0.06    0.15    0.03   -0.05    0.48    0.08    0.28   -0.1
   0.76    1       0.67    0.82    0.56    0.04   -0.04   -0.02    0.09    0.16   -0.05   -0.01    0.27   -0.01    0.24   -0.05
   0.85    0.67    1       0.66    0.77    0.07    0.02   -0.1     0.06    0.12    0.01   -0.05    0.55    0.03    0.27   -0.11
   0.68    0.82    0.66    1       0.65    0.03   -0.01    0.08    0.06    0.01   -0.01    0.02    0.27    0.02    0.3    -0.02
   0.62    0.56    0.77    0.65    1       0.14   -0.03   -0.01   -0.06   -0.07   -0.08   -0.04    0.63    0       0.5    -0.06
  -0.03    0.04    0.07    0.03    0.14    1      -0.35   -0.05   -0.13    0.08   -0.33   -0.03    0.16   -0.25    0.13    0.25
   0.04   -0.04    0.02   -0.01   -0.03   -0.35    1      -0.13   -0.04    0.19    0.6    -0.28   -0.05    0.55   -0.08   -0.21
   0.02   -0.02   -0.1     0.08   -0.01   -0.05   -0.13    1      -0.2    -0.32    0.04    0.08    0.15   -0.09    0       0.19
   0.06    0.09    0.06    0.06   -0.06   -0.13   -0.04   -0.2     1       0.14   -0.06    0.12   -0.38   -0.04    0.28   -0.06
   0.15    0.16    0.12    0.01   -0.07    0.08    0.19   -0.32    0.14    1       0.06   -0.1    -0.18    0.03   -0.09   -0.12
   0.03   -0.05    0.01   -0.01   -0.08   -0.33    0.6     0.04   -0.06    0.06    1       0.06    0.04    0.52   -0.23   -0.24
  -0.05   -0.01   -0.05    0.02   -0.04   -0.03   -0.28    0.08    0.12   -0.1     0.06    1      -0.01   -0.19    0.05    0.25
   0.48    0.27    0.55    0.27    0.63    0.16   -0.05    0.15   -0.38   -0.18    0.04   -0.01    1      -0.02    0.12   -0.04
   0.08   -0.01    0.03    0.02    0      -0.25    0.55   -0.09   -0.04    0.03    0.52   -0.19   -0.02    1      -0.06   -0.19
   0.28    0.24    0.27    0.3     0.5     0.13   -0.08    0       0.28   -0.09   -0.23    0.05    0.12   -0.06    1       0.14
  -0.1    -0.05   -0.11   -0.02   -0.06    0.25   -0.21    0.19   -0.06   -0.12   -0.24    0.25   -0.04   -0.19    0.14    1


eigenvalue      proportion      cumulative
   4.31144         0.26946         0.26946        -0.436width-box-0.426x-box-0.417npix-0.4heigth-box-0.399y-box...
   2.62272         0.16392         0.43338        -0.509y-var-pix-0.467x2ybr-0.453xegvy+0.324x-var-pix+0.314yegvx...
   1.7285          0.10803         0.54142        -0.556y2bar+0.452x2bar-0.433xybar+0.43 x-ege+0.184x2ybr...
   1.35958         0.08497         0.62639        -0.497xy2br+0.421x-var-pix-0.383y2bar+0.322xybar-0.313x2bar...
   1.05434         0.0659          0.69229        0.613y-ege-0.445xy2br-0.371xybar-0.239y-box+0.232npix...
   0.97429         0.06089         0.75318        -0.636yegvx-0.435x-var-pix-0.281xegvy-0.278xybar-0.257y-var-pix...

Eigenvectors
 V1       V2       V3       V4       V5       V6
-0.426   -0.0881  -0.0373  -0.0235  -0.1447   0.042   x-box
-0.3987  -0.0243  -0.1375  -0.0605  -0.2388   0.0507  y-box
-0.4361  -0.0588  -0.0328   0.064   -0.073    0.0142  width-box
-0.4001  -0.0171  -0.0455  -0.155   -0.1254   0.0333  heigth-box
-0.4171   0.032    0.0999   0.0349   0.232   -0.0433  npix
-0.0572   0.3236  -0.0163   0.4211   0.0502  -0.435   x-var-pix
 0.0197  -0.5094   0.0349  -0.0177   0.1565  -0.2573  y-var-pix
-0.0021   0.123    0.4517  -0.3125  -0.0738   0.1359  x2bar
-0.0101   0.003   -0.5557  -0.3833   0.1559   0.1109  y2bar
-0.0263  -0.1553  -0.4327   0.3216  -0.3713  -0.278   xybar
 0.0305  -0.4672   0.1843  -0.206   -0.1763  -0.2248  x2ybr
 0.0178   0.1897   0.0103  -0.4966  -0.4445  -0.2315  xy2br
-0.2883   0.0305   0.4295   0.1984   0.0181  -0.0114  x-ege
-0.0006  -0.4535   0.0968  -0.0795   0.2156  -0.2811  xegvy
```

```
 -0.2131   0.1542  -0.1672  -0.2532   0.6126  -0.2017 y-ege
  0.0339   0.3142   0.096   -0.2048  -0.0423  -0.6364 yegvx

Ranked attributes:
 0.731    1 -0.436width-box-0.426x-box-0.417npix-0.4heigth-box-0.399y-box...
 0.567    2 -0.509y-var-pix-0.467x2ybr-0.453xegvy+0.324x-var-pix+0.314yegvx...
 0.459    3 -0.556y2bar+0.452x2bar-0.433xybar+0.43 x-ege+0.184x2ybr...
 0.374    4 -0.497xy2br+0.421x-var-pix-0.383y2bar+0.322xybar-0.313x2bar...
 0.308    5  0.613y-ege-0.445xy2br-0.371xybar-0.239y-box+0.232npix...
 0.247    6 -0.636yegvx-0.435x-var-pix-0.281xegvy-0.278xybar-0.257y-var-pix...

Selected attributes: 1,2,3,4,5,6 : 6
```

A file has been generated named "OCR.PCA70.arff" with the transformed data. Consequently, data is represented by 6 different variable, which correspond to a combination of the 16 original attributes that has been presented in previous sections.

3 Weka classification algorithms

Weka supports a great variety of supervised learning algorithms (classification) and they are classified in the following categories (for each one, at least one algorithm has been selected for testing purposes):

- Bayesian classifiers
  - NaiveBayesSimple: Uses the normal distribution to model numeric attributes and the Bayes theorem for probabilistic classification.
- Trees
  - J48: Implements C4.5 algorithm for decision tree generation.
- Rules
  - OneR: Finds the one attribute to use to classify considering which makes fewest prediction errors.
- Functions: classifiers that can be written down as mathematical equations in a reasonably natural way (although NaiveBayes is an exception).
  - SimpleLogistic: Linear regression for non-continuous classes (which is our case with 26 letters).
  - MultilayerPerceptron: Neural network with hidden layers.
- Lazy classifiers
  - IBk: Implementation of the K-Nearest Neighbours method (KNN).
- Miscellaneous category
  - Hyperpipes: Identifies classes by using the attribute ranges of the training data.

3.1 Bayesian classifier: NaiveBayesSimple

The NaiveBayes uses the Bayes theorem establishing a priori probability that an object $\overrightarrow{x_0}$ belongs to a group G_i:

$$P\left(G_i|\overrightarrow{x_0}\right) = \frac{P\left(\overrightarrow{x_0}|G_i\right) \cdot P\left(G_i\right)}{\sum_{i=1}^{N} P\left(\overrightarrow{x_0}|G_i\right) \cdot P\left(G_i\right)} \tag{1}$$

It is worth to mention that numeric attributes are modeled by a normal distribution.

3.2 Tree: J48 (C4.5)

J48 implements C4.5 revision 8, an algorithm that generates decision trees that can be used for classification. It uses the concept of information entropy (measure of disorder/unpredictability).

At each node of the tree, C4.5 chooses one attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The algorithm is as follows:

```
Check for base cases such as:
        All the samples in the list belong to the same class.
                Create a leaf node for the decision tree saying to choose that class.

For each attribute a
        Find the normalized information gain (difference in entropy) from splitting on a
Let a_best be the attribute with the highest normalized information gain
Create a decision node that splits on a_best
Recur on the sublists obtained by splitting on a_best, and add those nodes as children of node

# At the end
Pruning tree after creation:
        Go back through the tree and attempts to remove branches that do not help
        by replacing them with leaf nodes
```

For the case of study, the confidence threshold for pruning is one of the relevant parameters which will be set to 0.25 (general recommended value).

3.3 Rules: OneR

The one-attribute-rule, or OneR, is an algorithm for finding association rules. The algorithm is as follows:

```
For each attribute A:
        For each value V of that attribute, create a rule:
                1. count how often each class appears
                2. find the most frequent class, c
                3. make a rule "if A=V then C=c"
        Calculate the error rate of this rule
Pick the attribute whose rules produce the lowest error rate
```

3.4 Functions

3.4.1 SimpleLogistic

SimpleLogistic builds logistic regression models for non-continuous classes (e.g. 26 letters). Mathematically, the resulting model is:

$$Pr\left(C_k|a_1, a_2, ..., a_n\right) = \frac{1}{1 + e^{-w_0 - w_1 a_1 - ... - w_n a_n}} \tag{2}$$

where C_k is one possible class, $(a_1, ..., a_n)$ are the attributes of a new instance and $(w_0, ..., w_n)$ are the weights found from the training data.

SimpleLogistic fits the regression models by using the LogitBoost algorithm[3] with simple regression functions as base learners and determining how many iterations to perform using cross-validation[4] (technique for estimating the performance of a predictive model).

3.4.2 MultilayerPerceptron

MultilayerPerceptron is a neural network that trains using backpropagation (propagation of error for teaching of artificial neurons). For the case of study, the "autoBuild" option will be set so hidden layers are added and connected up automatically.

[3] http://en.wikipedia.org/wiki/LogitBoost
[4] http://en.wikipedia.org/wiki/Cross-validation_(statistics)

3.5 Lazy: IBk

IBk implements the K-Nearest Neighbours method (KNN). Given a sample of objects classified on N groups, every new instance is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its k nearest neighbors.

Weka IBk implementation uses Euclidean distance and the number of nearest neighbors (default $k = 1$) can be specified. For the case of study $k = 5$ will be used, which has been determined as a good choice after various manual tests.

3.6 Miscellaneous: Hyperpipes

For each class, a Hyperpipe is constructed that contains all points of that class (essentially records the attribute limits observed for each category). Then, the new instances are classified according to the class that "most contains the instance".

This is an extremely simple algorithm that has the advantage of being extremely fast and works quite well when there are a huge amount of attributes.

3.7 Metalearning algorithms

3.7.1 Combining classifiers: Vote

Vote provides a method for combining classifiers by averaging their probability estimates (classification). The algorithm is as follows:

```
# Model generation
Let n be the number of instances in the training data.
For each t classifier:
        Apply learning algorithm.
        Store the resulting model.

# Classification
For each of the t models:
        Predict class of instance using model.
Return class that has been predicted most often.
```

For the case of study, the average probability given by the following algorithms will be considered for object classification:

- NaiveBayesSimple: Probabilistic type
- J48: Decision tree type
- IBk: Geometric type

The classifiers has been chosen considering they are different enough as to complement each other efficiently.

3.7.2 Boosting: AdaBoost.M1

Generally, it is considered a good option to combine multiple models generated by a given classifier when these models are significantly different from one another and each one treats a reasonable percentage of the data correctly. In that sense, the boosting method can perform this combination by explicitly seeking models that complement one another.

AdaBoost.M1 is a boosting algorithm designed specifically for classification and the algorithm is as follows:

```
# Model generation
Assign equal weight to each training instance.
For each of t iterations:
        Apply learning algorithm to weighted dataset and store resulting model.
        Compute error e of model on weighted dataset and store error.
        If e equal to zero, or e greater or equal to 0.5:
                Terminate model generation.
```

```
        For each instance in dataset:
                If instance classified correctly by model:
                        Multiply weight of instance by e / (1 - e).
        Normalize weight of all instances.

# Classification
Assign weight of zero to all classes.
For each of the t (or less) models:
        Add -log(e / (1 - e)) to weight of class predicted by model.
Return class with highest weight.
```

It is worth to mention that in case any of the selected classifiers cannot handle weighted instances (e.g. C4.5 algorithm can accommodate weighted instances without modification but others don't), AdaBoost.M1 resamples the data with the weights before applying the classifier again.

For the case of study, the following classifiers will be used with AdaBoost.M1:

- NaiveBayesSimple: Probabilistic type
- J48: Decision tree type
- IBk: Geometric type

4 Weka experimenter

4.1 Input data

The following input files has been used with Weka experimenter, each of them has 9.940 objects:

- OCR.arff: Objects with the 16 original attributes presented in section 2 on page 3
- OCR.PCA95.arff: Objects with 12 attributes determined after applying a Principal Components Analysis with 95% of covered variance, presented in section 2.3.1 on page 8.
- OCR.PCA70.arff: Objects with 6 attributes determined after applying a Principal Components Analysis with 70% of covered variance, presented in section 2.3.2 on page 9.

Weka divides each dataset into K folds/subsamples for cross-validation. Of the K subsamples, a single subsample is retained as the validation data for testing the model, and the remaining K − 1 subsamples are used as training data. For the case of study, $K = 10$ has been established.

4.2 Classification algorithms

The selected algorithms for object classification are those presented in section 3 on page 10. Concrete list of algorithm with parameters:

```
bayes.NaiveBayesSimple ''

trees.J48 '-C 0.25 -M 2'

rules.OneR '-B 6'

functions.SimpleLogistic '-I 0 -M 500 -H 50 -W 0.0'

functions.MultilayerPerceptron '-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a'

lazy.IBk '-K 5 -W 0
    -A "weka.core.neighboursearch.LinearNNSearch
    -A "weka.core.EuclideanDistance -R first-last""'

misc.HyperPipes ''

meta.Vote '-S 1
    -B "bayes.NaiveBayesSimple "
    -B "trees.J48 -C 0.25 -M 2"
    -B "lazy.IBk -K 5 -W 0
```

```
21          -A "weka.core.neighboursearch.LinearNNSearch
22          -A "weka.core.EuclideanDistance -R first-last""
23      -R AVG'
24
25  meta.AdaBoostM1 '-Q -P 100 -S 1 -I 10 -W bayes.NaiveBayesSimple'
26
27  meta.AdaBoostM1 '-P 100 -S 1 -I 10 -W trees.J48 -- -C 0.25 -M 2'
28
29  meta.AdaBoostM1 '-P 100 -S 1 -I 10 -W lazy.IBk -- -K 5 -W 0
30          -A "weka.core.neighboursearch.LinearNNSearch
31          -A "weka.core.EuclideanDistance -R first-last""'
```

In order to get statistically meaningful results, the number of repetitions of each algorithm has been established to 10. For the case of study, 100 calls will be done of one classifier with training data and tested against test data (10 folds/subsamples x 10 repetitions).

4.3 Results analysis

Weka includes an experiment analyzer (T-Test) that can be used to study the results of experiments. For the case of study, the statistical significance has been established to 0.05 which refers to the result of a pair-wise comparison of schemes using the corrected resampled T-Test. It is worth to mention that as the significance level is decreased, the confidence in the conclusion increases.

Results from analysis of the percentage of correct classification per algorithm and dataset (standard deviation in parenthesis):

Dataset	(1) 'OCR original'	(2)'OCR PCA 95'	(3)'OCR PCA 70'
meta.AdaBoostM1 (J48)	(100) 93.04(0.86)	87.87(0.95) *	76.80(1.24) *
lazy.IBk	(100) 92.59(0.93)	91.94(0.79) *	78.95(1.16) *
meta.AdaBoostM1 (IBk)	(100) 92.42(0.96)	91.85(0.82)	78.95(1.16) *
meta.Vote	(100) 89.94(0.93)	85.67(0.93) *	74.34(1.28) *
trees.J48	(100) 84.17(1.26)	73.97(1.44) *	67.66(1.42) *
functions.MultilayerPerce	(100) 81.46(1.22)	77.11(1.51) *	66.42(1.63) *
functions.SimpleLogistic	(100) 77.24(1.02)	72.05(1.24) *	50.17(1.39) *
bayes.NaiveBayesSimple	(100) 63.69(1.44)	64.65(1.34) v	48.48(1.51) *
meta.AdaBoostM1 (NBayes)	(100) 63.63(1.42)	64.37(1.41)	48.33(1.56) *
misc.HyperPipes	(100) 24.60(1.28)	24.09(1.11)	17.98(0.87) *
rules.OneR	(100) 16.98(0.90)	16.92(1.06)	16.92(1.06)
	(v/ /*)	(1/4/6)	(0/1/10)

At the bottom of each column after the first column is a count (v/ /*) of the number of times that a scheme was better than (v), the same as ( ), or worse than (*), the baseline dataset ('OCR original').

It is possible to conclude that applying PCA to the original dataset and reducing the number of attributes that define an object, affects the efficiency (in terms of correct classification) of any of the selected classifiers and the impact is not proportional for all of them.

On the other hand, despite that the execution of AdaBoost.M1 using J48 as classifier gets the better result for the original data, IBk is getting the best results for all the datasets as we can see in the overall algorithm ranking:

'>' Number of times the scheme was significantly more than the other schemes

'<' Number of times the scheme was significantly less than the other schemes

'>-<' Difference between the above numbers

>-<	>	<	Resultset
26	26	0	meta.AdaBoostM1 (IBk)
26	26	0	lazy.IBk
20	24	4	meta.AdaBoostM1 (J48)
12	21	9	meta.Vote (NBayes+J48+IBk)
3	16	13	functions.MultilayerPerceptron
3	16	13	trees.J48
-6	12	18	functions.SimpleLogistic
-15	6	21	meta.AdaBoostM1 (NBayes)
-15	6	21	bayes.NaiveBayesSimple
-24	3	27	misc.HyperPipes
-30	0	30	rules.OneR

Finally, the computational efficiency of each algorithm can be evaluated by the overall ranking of CPU time used for training and testing (lower position is better performance):

```
# Training time
>-<   >   < Resultset
 28   29    1 functions.SimpleLogistic
 25   27    2 functions.MultilayerPerceptron
 19   24    5 meta.AdaBoostM1 (IBk)
 12   21    9 meta.AdaBoostM1 (J48)
  2   16   14 meta.AdaBoostM1 (NaiveBayes)
  1   15   14 trees.J48
 -3   13   16 meta.Vote (NBayes+J48+IBk)
-12    9   21 rules.OneR
-18    6   24 bayes.NaiveBayesSimple
-27    0   27 misc.HyperPipes
-27    0   27 lazy.IBk

# Testing time
>-<   >   < Resultset
 30   30    0 meta.AdaBoostM1 (IBk)
 23   26    3 meta.Vote (NBayes+J48+IBk)
 19   24    5 lazy.IBk
  7   18   11 functions.SimpleLogistic
  7   18   11 meta.AdaBoostM1 (NaiveBayes)
  2   15   13 meta.AdaBoostM1 (J48)
 -5   11   16 bayes.NaiveBayesSimple
-11    9   20 functions.MultilayerPerceptron
-24    0   24 misc.HyperPipes
-24    0   24 rules.OneR
-24    0   24 trees.J48
```

In order to have a better understanding of how much better/worst is each algorithm, here is the time expressed in minutes of a single execution using one core of a CPU Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz. For obtaining the total time consumed for this study, it should be multiplied by 100 (10 folds x 10 iterations):

```
# Testing time
Dataset                        (1) 'OCR original'    | (2) 'OCR PCA 95'   (3) 'OCR PCA 70'
-------------------------------------------------------------------------------------------
meta.AdaBoostM1 (J48)          (100)     5.62( 0.05) |    14.83( 0.10) v     8.63( 0.05) v
lazy.IBk                       (100)     0.00( 0.00) |     0.00( 0.00)       0.00( 0.00)
meta.AdaBoostM1 (IBk)          (100)   134.69(23.08) |   106.44(22.37) *   53.60( 8.60) *
meta.Vote                      (100)     0.49( 0.01) |     1.44( 0.02) v     0.82( 0.01) v
trees.J48                      (100)     0.52( 0.02) |     1.45( 0.03) v     0.85( 0.02) v
functions.MultilayerPerce(100)   147.14( 5.89) |   122.71( 0.84) *   97.04( 0.75) *
functions.SimpleLogistic       (100)   184.13(51.42) |   191.67(58.11)      76.25(19.71) *
bayes.NaiveBayesSimple         (100)     0.01( 0.01) |     0.01( 0.00)       0.00( 0.01) *
meta.AdaBoostM1 (NBayes)       (100)     2.78( 0.67) |     2.28( 0.47)       0.14( 0.01) *
misc.HyperPipes                (100)     0.00( 0.00) |     0.00( 0.00)       0.00( 0.00)
rules.OneR                     (100)     0.02( 0.01) |     0.02( 0.01)       0.01( 0.00) *
-------------------------------------------------------------------------------------------
                                     (v/ /*) |           (3/6/2)           (3/2/6)
```

```
# Testing time
Dataset                        (1)  OCR original |(2) OCR PCA 95 (3) OCR PCA 70
-----------------------------------------------------------------------------
meta.AdaBoostM1 (IBk)          (100)   0.04(0.01) |   0.04(0.01)      0.04(0.01)
lazy.IBk                       (100)   1.22(0.02) |   0.88(0.02) *   0.64(0.01) *
meta.AdaBoostM1 (NBayes)       (100)   7.00(1.34) |   5.61(1.32) *   2.73(0.50) *
meta.Vote                      (100)   1.26(0.02) |   0.91(0.02) *   0.65(0.02) *
trees.J48                      (100)   0.00(0.00) |   0.00(0.01)      0.00(0.01)
functions.MultilayerPerce(100)   0.01(0.00) |   0.01(0.00)      0.01(0.00)
functions.SimpleLogistic       (100)   0.05(0.01) |   0.06(0.01) v   0.03(0.01) *
bayes.NaiveBayesSimple         (100)   0.04(0.01) |   0.03(0.00) *   0.01(0.01) *
meta.AdaBoostM1 (J48)          (100)   0.13(0.04) |   0.11(0.03)      0.02(0.00) *
misc.HyperPipes                (100)   0.00(0.00) |   0.00(0.00)      0.00(0.00)
rules.OneR                     (100)   0.00(0.00) |   0.00(0.00)      0.00(0.00)
-----------------------------------------------------------------------------
                                     (v/ /*) |         (1/6/4)         (0/5/6)
```

IBk is the fastest algorithm in the training process but comparative slower for testing (classifying new objects). If it is important to choose an algorithm with a fast testing process and good classification capacities, the boosted version of J48 should be considered.

With respect to the metalearning algorithms, Vote has been used to combine the following classifiers:

- NaiveBayesSimple: Probabilistic type

- J48: Decision tree type
- IBk: Geometric type

Percentage of correction and CPU time consumption for this Vote combination is worst that the offered by the best individual classifier (IBk), therefore its not a good choice for this case of study.

On the other hand, despite that AdaBoost.M1 with NaiveBayesSimple and IBk does not present any improvement at all, when applied to J48 the results are improved significantly (from 84.17% to 93.04% with the original dataset) increasing the CPU time consumption.

5 Neural networks

5.1 Input data

A ruby script has been implemented (code in 8 on page 29) in order to transform the following input files (each of them with 9.940 objects):

- OCR.arff: Objects with the 16 original attributes presented in section 2 on page 3
- OCR.PCA95.arff: Objects with 12 attributes determined after applying a Principal Components Analysis with 95% of covered variance, presented in section 2.3.1 on page 8.
- OCR.PCA70.arff: Objects with 6 attributes determined after applying a Principal Components Analysis with 70% of covered variance, presented in section 2.3.2 on page 9.

The transformations that the script performs are:

1. Finds the maximum and minimum value for each attribute
2. Rescales/normalizes the values between 0..1 for each attribute
   $new\,value = \frac{value - minimum}{maximum - minimum}$
3. Builds a binary representation of the letter (class). For example:

   ```
   V K W L A X M B Y N C Z O D P E Q F R G S H T I U J
   0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ```

4. Defines a numeric representation between 0 and 1 with increments of 0.04 (1/25)

   ```
             V K W L A X M B Y N C Z O D P E Q F R G S H T I U J
   Num = 0.0 ..................................................... 1.0
   ```

5. Writes new files with the required JavaNNS/SNNS format
   (a) Directories:
      i. binary_classes: output neuron (class/letter) represented in binary form
      ii. numeric_classes: output neuron (class/letter) represented in numeric form
   (b) Files on each directory: OCR.pat, OCR.PCA95.pat and OCR.PCA70.pat
6. For each of the above directory/files, generates subdatasets for training (70% of the original data) and validating (30%)
   (a) Files on each directory:
      i. OCR.Train.pat, OCR.Test.pat
      ii. OCR.PCA95.Train.pat, OCR.PCA95.Test.pat
      iii. OCR.PCA70.Train.pat, OCR.PCA70.Test.pat

7. Saves the conversion table of letters representation:

* Conversion table: Class, numeric and binary values *

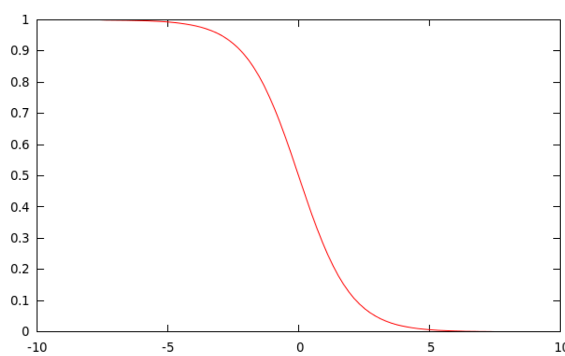
Class		Num.		V	K	W	L	A	X	M	B	Y	N	C	Z	O	D	P	E	Q	F	R	G	S	H	T	I	U	J
V	=	0.0	=	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K	=	0.04	=	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	=	0.08	=	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	=	0.12	=	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	=	0.16	=	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X	=	0.2	=	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	=	0.24	=	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	=	0.28	=	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Y	=	0.32	=	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	=	0.36	=	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	=	0.4	=	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Z	=	0.44	=	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O	=	0.48	=	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
D	=	0.52	=	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
P	=	0.56	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
E	=	0.6	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Q	=	0.64	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
F	=	0.68	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
R	=	0.72	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
G	=	0.76	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
S	=	0.8	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
H	=	0.84	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
T	=	0.88	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
I	=	0.92	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
U	=	0.96	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
J	=	1.0	=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

5.2 General design decisions

5.2.1 Neuron

The chosen activation function is Logistic, which applies the following equation to the input value (x) generating a smoothed step-function bound between 0 and 1 :

$$y = \frac{1}{1 + e^{\sum x_{inputs}}} \tag{3}$$



On the other hand, Identity has been established as the output function ($Identity\,(y) = y_{output}$). Therefore, each neuron will output the value calculated by the equation above.

5.2.2 Patterns (data)

The data is divided into a training set (for estimating the weights) and a validation set (for estimating the optimal number of hidden nodes). In section 5.1 on the previous page it has been already explained how this data has been created.

The shuffle option will be activated, so JavaNNS will randomly shuffle the patterns before each cycle of backpropagation training. Thanks to that it is possible to reduce the probabilities for the network to oscillate constantly between certain states by the use of equally ordered patterns for each training epoch.

5.2.3 Layers

Each layer will be composed by the following number of neurons:

- Input layer (N_{input}): 26, 12 and 6 attributes depending if the input data is original, PCA 95% or PCA 70%.
- Output layer (N_{output}): 26 or 1 depending if the letter is going to be represented in binary or numeric.
- Hidden layer (N_{hidden}): the number of neurons is recommended to be placed between the following range:

$$\frac{N_{input} + N_{output}}{2} < N_{hidden} < N_{input} + N_{output}$$

  - The initial strategy defined consist on choosing the largest number among input and output neurons $N_{hidden} = max\left(N_{input}, N_{output}\right)$

5.2.4 Learning

Backpropagation is the chosen learning algorithms that iteratively will alter the value of the weights until some error function is minimized, it will use a learning rate $\eta = 0.2$ and a maximum difference between the desired output and the real output of $d_{max} = 0.1$. Each iteration is called a cycle, and a minimum of 1.000 cycles will be done for each test (limited by computational power and time). As update method "Topological_Order" has been chosen with one update step (sufficient to propagate information from input to output).

In this process, while the training error reduces monotonically, d_{max} will have an important role in order to avoid overfitting because it establishes a limit for which error should not be propagated. Additionally, the validation pattern helps to control this kind of problems.

On the other hand, the learning algorithm can get caught in a local minimum. To validate in simple way that this is not apparently happening, several different weights initializations will be performed (random weights between -1 and 1).

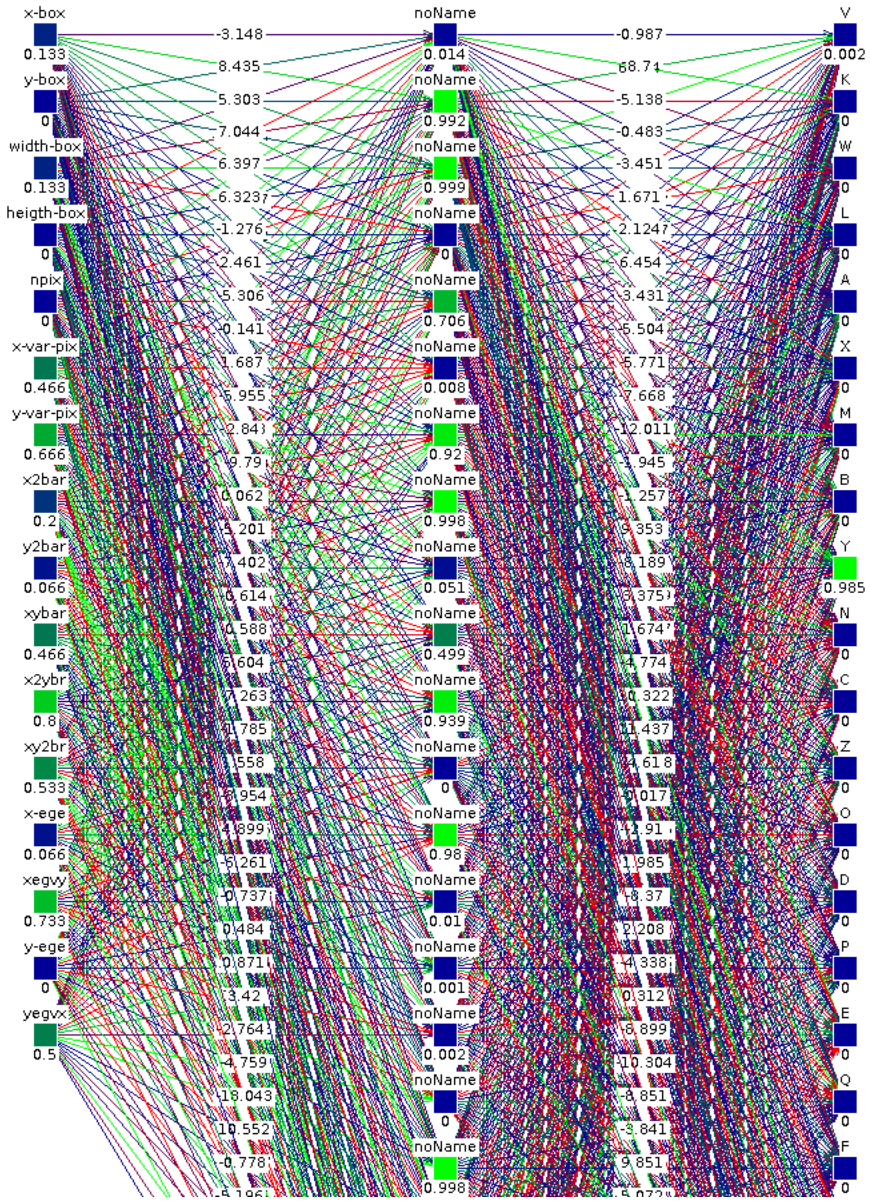
5.2.5 Pruning

Networks can be pruned (neuron and connection elimination) in order to improve efficiency. For the cases where this option is applied, it will be performed a Magnitude Based Pruning. This is the simplest weight pruning algorithm: After each training, the link with the smallest weight is removed. It's worth to mention that though this method is very simple, it is known to rarely yield worse results than the more sophisticated algorithms.

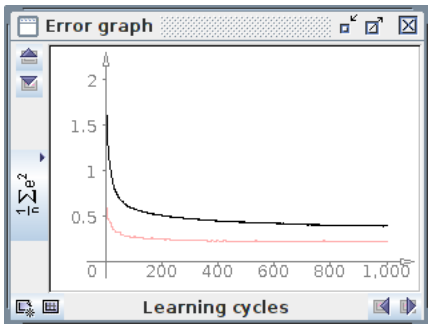
5.3 Networks with binary representation of letters

5.3.1 Original

The following network has been designed and trained (it does not fit entirely in the image, it has 26 hidden neurons and 26 output neurons):



During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):

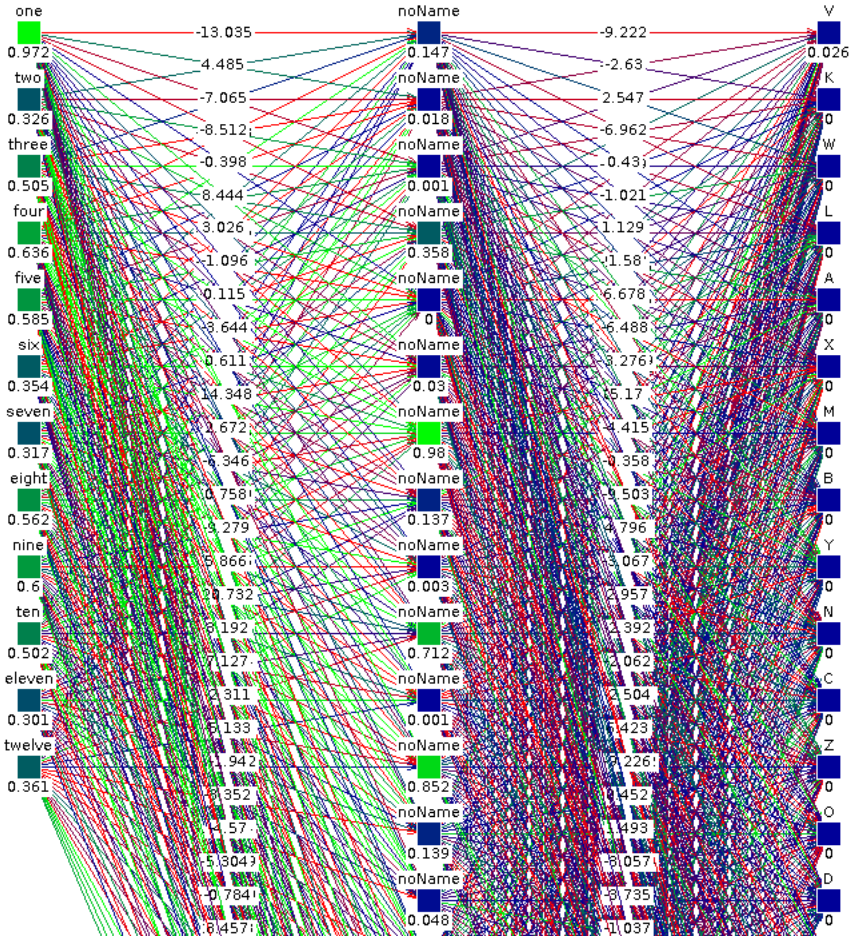



```
Step  100 MSE:     0.5734590240007755          validation:     0.2613144688443159
Step  200 MSE:     0.4994512566938087          validation:     0.2362549728710806 6
Step  300 MSE:     0.46331340910043595         validation:     0.2229920047629527 2
Step  400 MSE:     0.44003850577422 87         validation:     0.2222846560155202
```

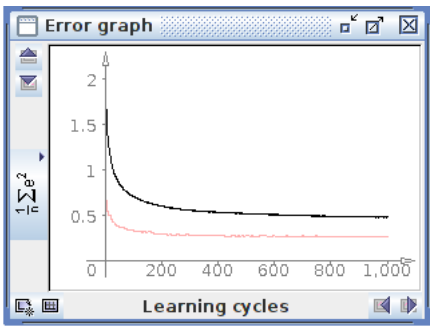
```
Step  500 MSE:    0.4256005188029217      validation:     0.218283189534341 48
Step  600 MSE:    0.4155173963704 0996    validation:     0.218843169695254 86
Step  700 MSE:    0.4021933521703 1355    validation:     0.213629003022038 5
Step  800 MSE:    0.394033103241846 9     validation:     0.213942795772028
Step  900 MSE:    0.3885376058433937      validation:     0.214445179377043 52
Step  1000 MSE:   0.3843637181159876      validation:     0.209005313620001 27
```

5.3.2 PCA 95 %

The following network has been designed and trained (it does not fit entirely in the image, it has 26 hidden neurons and 26 output neurons):



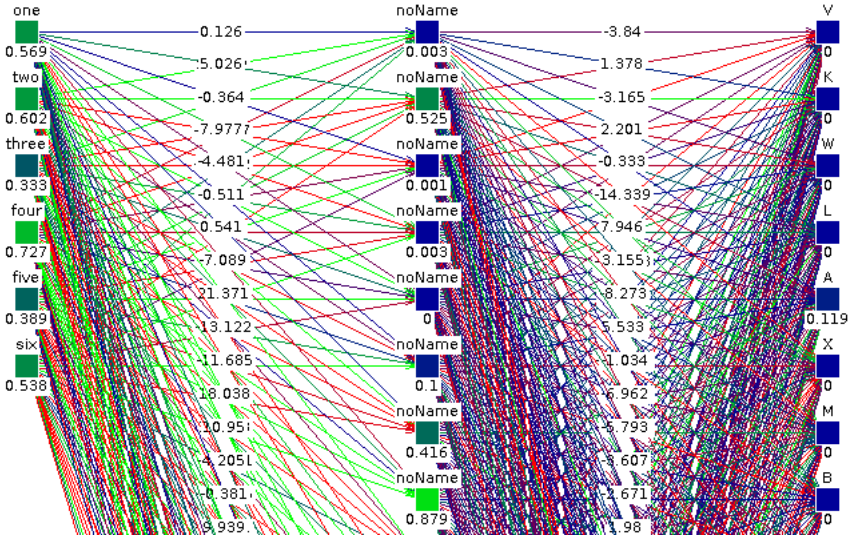
During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):



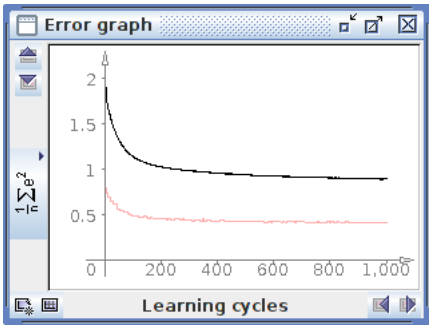
```
Step  100 MSE:    0.7010856643768864      validation:     0.33005264251524774
Step  200 MSE:    0.5932119818520498      validation:     0.280224662591434
Step  300 MSE:    0.5524064389592765      validation:     0.2702997771786972
Step  400 MSE:    0.5250195427120011      validation:     0.2706575553581908
Step  500 MSE:    0.5107304389487 97      validation:     0.2771468491941231
Step  600 MSE:    0.4988691507770582      validation:     0.2713542812866784
Step  700 MSE:    0.4891950171078554 6    validation:     0.2662246059363996
Step  800 MSE:    0.4800335214411468 6    validation:     0.2618340045153735
Step  900 MSE:    0.4738439180001886 6    validation:     0.2641874194385060 4
Step  1000 MSE:   0.4720729160116951 5    validation:     0.2579010439590669
```

5.3.3 PCA 70 %

The following network has been designed and trained (it does not fit entirely in the image, it has 26 hidden neurons and 26 output neurons):



During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):

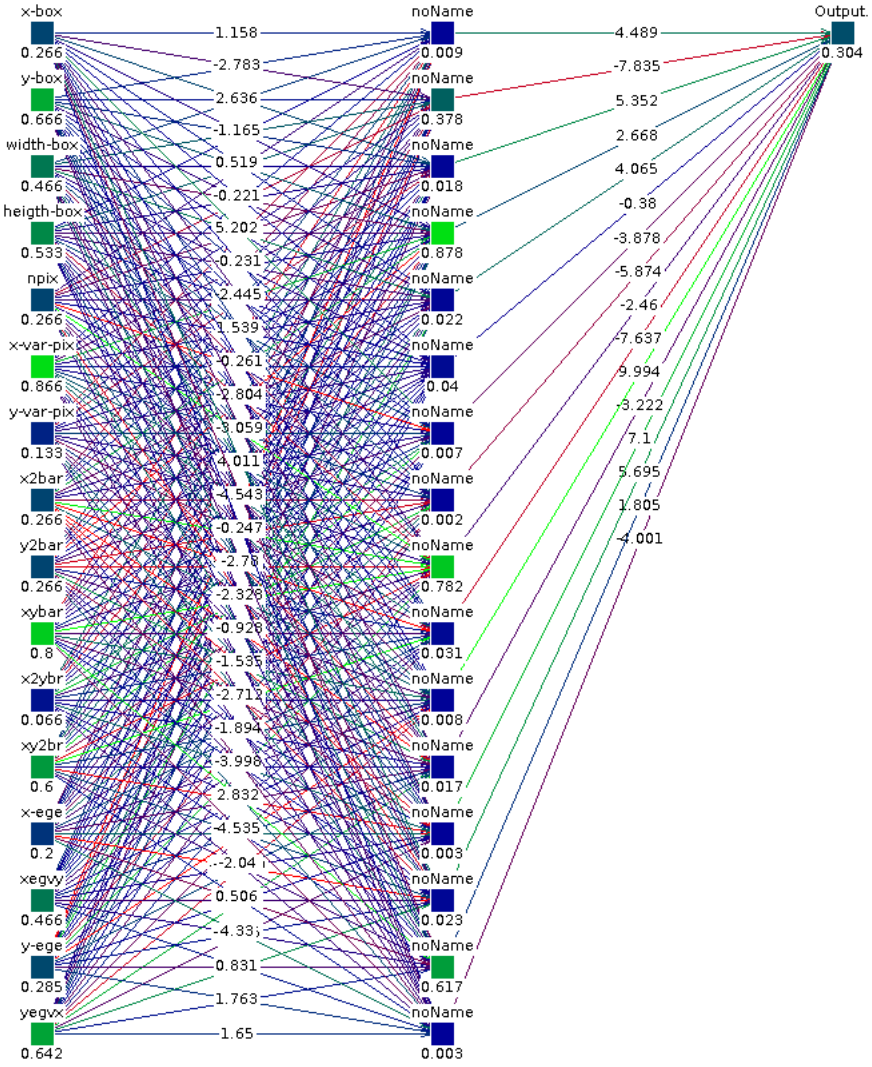



```
Step  100 MSE:    1.1341376710785547      validation:     0.4836847009793134
Step  200 MSE:    1.0188979744351525      validation:     0.4454418978061075
Step  300 MSE:    0.9731453499643695      validation:     0.4345495580907318 7
Step  400 MSE:    0.9484801058957915      validation:     0.4244462134132922 5
Step  500 MSE:    0.9275684215972921      validation:     0.4142286580813003
Step  600 MSE:    0.9145770619812625      validation:     0.4139139852133635 3
Step  700 MSE:    0.9078236508897133      validation:     0.4181919315371395
Step  800 MSE:    0.899911431479502       validation:     0.4048590042861335
Step  900 MSE:    0.8893915249308351      validation:     0.4051175952197560 4
Step  1000 MSE:   0.8832434712441315      validation:     0.4077373583951626
```

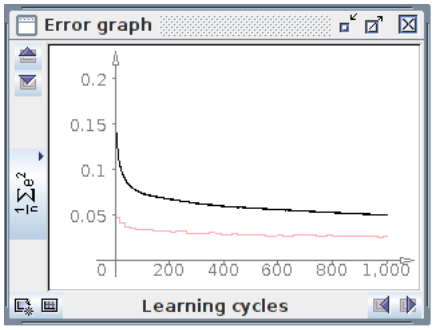
5.4 Networks with numeric representation of letters

5.4.1 Original

The following network has been designed and trained:



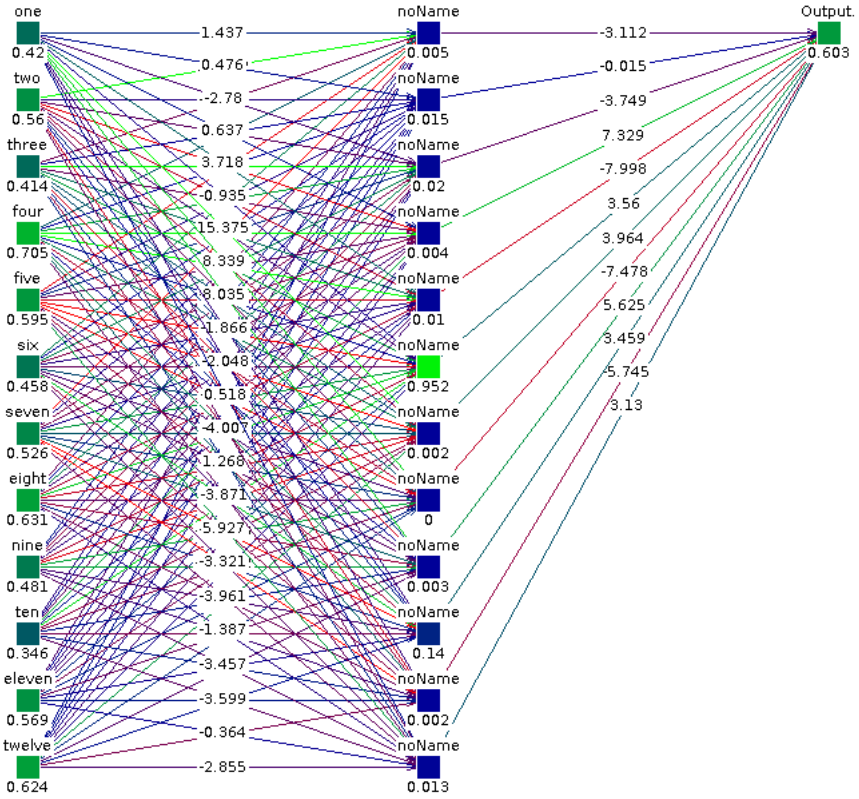
During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):



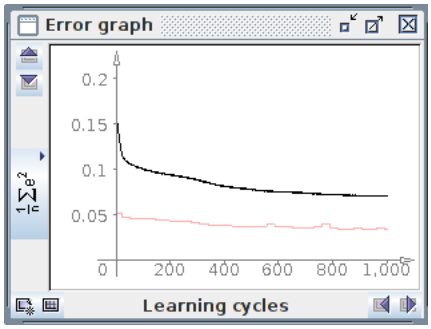

```
Step  100 MSE:    0.07316514647782528      validation:     0.03274918881780265
Step  200 MSE:    0.06685264889303268      validation:     0.03111019940443441 8
Step  300 MSE:    0.06176537783013183 5    validation:     0.02884077690017055 5
Step  400 MSE:    0.05876463615678445 6    validation:     0.02723863986736172 3
Step  500 MSE:    0.05699404352547577      validation:     0.02738178695791284
Step  600 MSE:    0.05502045066623701      validation:     0.02680796540718923
Step  700 MSE:    0.05338100584300071      validation:     0.02661614318888592
Step  800 MSE:    0.05158073993276062 6    validation:     0.02654731233674835 5
Step  900 MSE:    0.05016698395942058      validation:     0.02550822052677392 5
Step 1000 MSE:    0.04939202475595762      validation:     0.02537576533419265 4
```

5.4.2 PCA 95 %

The following network has been designed and trained:



During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):




```
Step 100 MSE:   0.09940173880195234     validation:     0.045026742076809 82
Step 200 MSE:   0.09305192319280264     validation:     0.042365224468556 13
Step 300 MSE:   0.08753950910868859     validation:     0.040313047822252 53
Step 400 MSE:   0.08033461286108579     validation:     0.036873754121408135
Step 500 MSE:   0.07653567584707463     validation:     0.035841219701837325
Step 600 MSE:   0.07475118042237802     validation:     0.035624853961824 33
Step 700 MSE:   0.07318759460244541     validation:     0.034924697108431844
Step 800 MSE:   0.07165655267870243     validation:     0.034890671850929 8
Step 900 MSE:   0.07014810727155904     validation:     0.033909527108942 72
Step 1000 MSE:  0.06980477428372317     validation:     0.033752717722349 72
```

5.4.3 PCA 70 %

In the design of every presented network, the initial strategy for establishing the number of neurons in the hidden layer was determined by the maximum number:

$$N_{hidden} = max\left(N_{input}, N_{output}\right)$$

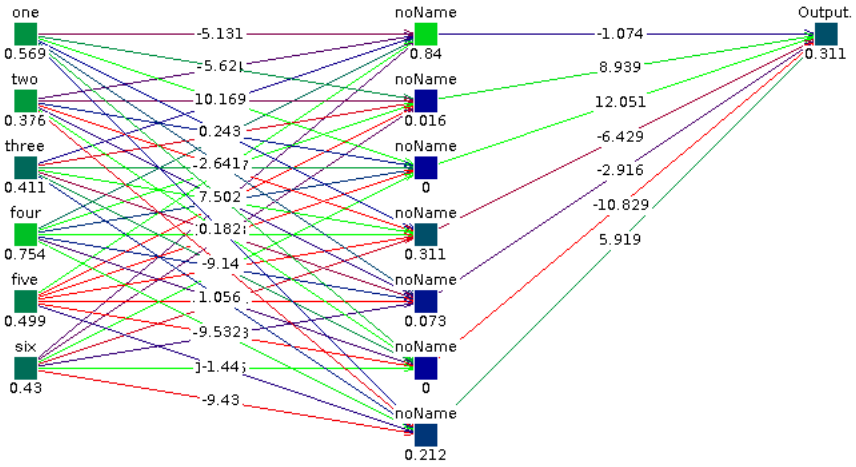
For this case, which is an easy to compute network in terms of CPU time consumption, three scenarios has been created in order to evaluate the correctness of the strategy. Three networks has been defined using the recommended bounds for the number of neurons in the hidden layer:

$$\frac{N_{input} + N_{output}}{2} < N_{hidden} < N_{input} + N_{output}$$

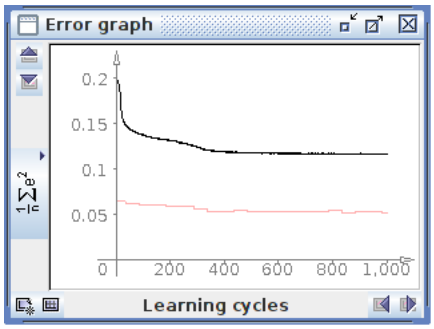
Resulting in three networks with 7, 6 and 4 hidden neurons respectively.

Additionally, pruning (removes neurons and connections) has been applied to the 6 hidden neurons version in order to evaluate this possibility.

7 neurons in the hidden layer The following network has been designed and trained:

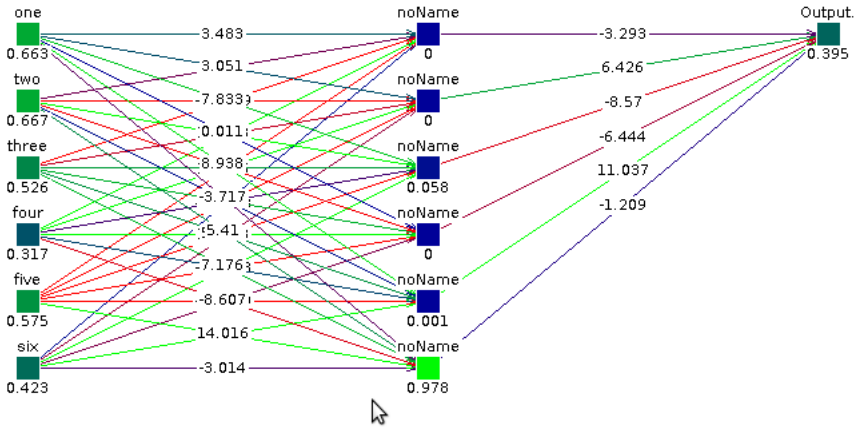


During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):

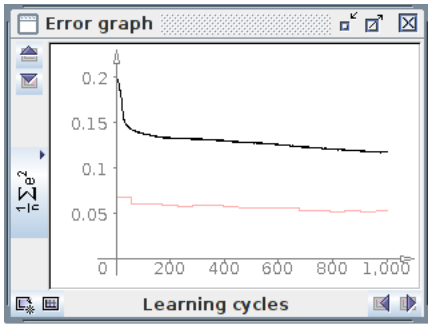



```
Step  100 MSE:    0.13660209343626864      validation:    0.0591191497127295
Step  200 MSE:    0.13051012168387932      validation:    0.057464875925474 72
Step  300 MSE:    0.12306074197463426      validation:    0.05503580667763089
Step  400 MSE:    0.11800580501236586      validation:    0.052716787991949 09
Step  500 MSE:    0.11717367396108422      validation:    0.05187454581820349
Step  600 MSE:    0.11655469519591508      validation:    0.05173947329812206
Step  700 MSE:    0.11599225889189302      validation:    0.052479913456978854
Step  800 MSE:    0.11596967260922304      validation:    0.05422891761375545
Step  900 MSE:    0.11561868975899668      validation:    0.05160117197324572
Step  1000 MSE:   0.11547352146734574      validation:    0.05138052845065183
```

6 neurons in the hidden layer The following network has been designed and trained:

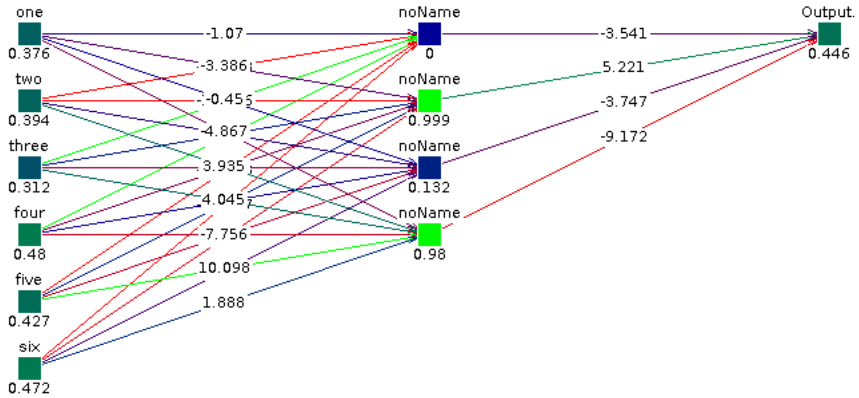


During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):

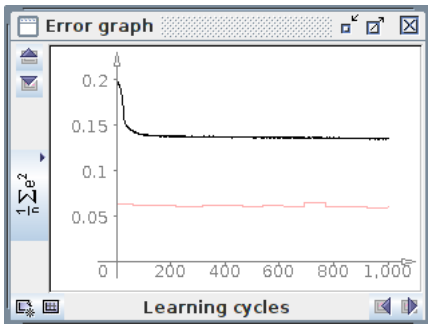



```
Step 100 MSE:    0.13697006460966318     validation:    0.059857630074064 17
Step 200 MSE:    0.1320928603030946      validation:    0.057992578272368 59
Step 300 MSE:    0.1309209832563087      validation:    0.058379077975339 01
Step 400 MSE:    0.12859853620420 44     validation:    0.057271214158322 33
Step 500 MSE:    0.12701329027223235     validation:    0.055518858389595 39
Step 600 MSE:    0.12496408914096559     validation:    0.054744579742452 11
Step 700 MSE:    0.12101870459410106     validation:    0.05195517383270 98
Step 800 MSE:    0.12009227811530643     validation:    0.050503199249206175
Step 900 MSE:    0.11790827566309314     validation:    0.052013686325308624
Step 1000 MSE:   0.11635975930452506     validation:    0.051971885839778256
```

4 neurons in the hidden layer The following network has been designed and trained:

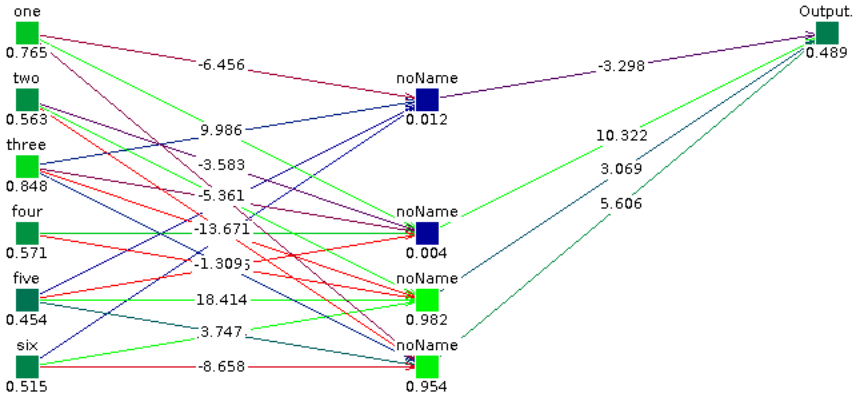


During the training, the evolution of the mean square error was the following (red line for the test/validation pattern):

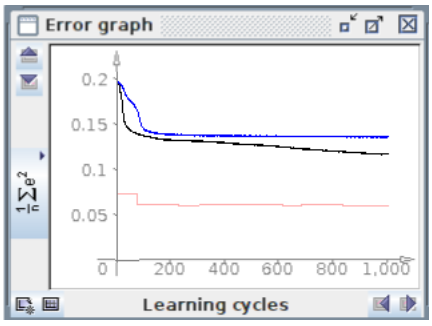



```
Step 100 MSE:    0.13853663362007984      validation:    0.060795211536143 95
Step 200 MSE:    0.13670098389818075      validation:    0.060862908980576 16
Step 300 MSE:    0.13641086222740087      validation:    0.059859303321556 94
Step 400 MSE:    0.13615051105788376      validation:    0.060952246069188566
Step 500 MSE:    0.13564532335615254      validation:    0.059557908977301 31
Step 600 MSE:    0.13592388068646252      validation:    0.060719357649805 08
Step 700 MSE:    0.1349311588115935       validation:    0.063592743186004 04
Step 800 MSE:    0.1345791423344756       validation:    0.059172591293201 92
Step 900 MSE:    0.13402450428482351      validation:    0.059513161120040556
Step 1000 MSE:   0.13392397639417233      validation:    0.059203441474039 55
```

Pruning Additionally, in this case the network version with 6 hidden neurons has been pruned searching for efficiency improvements:



The evolution of the mean square error for the pruned network (blue is the pruned version, black is the 6 hidden neuron version and red is the verification pattern):




```
Step 100 MSE:    0.14501143621167106      validation:    0.061091760114724 81
Step 200 MSE:    0.13774829355363954      validation:    0.060987880612123 26
Step 300 MSE:    0.13703827375057637      validation:    0.060071595957581 43
Step 400 MSE:    0.13710864225063926      validation:    0.060927080836254505
```

```
Step 500 MSE:    0.13628441179302378      validation:      0.060561340659516995
Step 600 MSE:    0.13617224792439533      validation:      0.060528198078444624
Step 700 MSE:    0.13630848199548856      validation:      0.060994931789631016
Step 800 MSE:    0.13587077682566756      validation:      0.06170215261294328
Step 900 MSE:    0.1356409637021347       validation:      0.059108311981262575
Step 1000 MSE:   0.13570556026589223      validation:      0.05914588073689533
```

5.5 CPU time consumption

JavaNNS does not allow to perform a exhaustive calculation of the CPU time consumption, but none of the learning process (1.000 cycles) has last more than 5 minutes using one core of a CPU Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz. Simpler networks have performed significantly faster, the learning of the simplest one (PCA 70% and network with 4 neurons in the hidden layer) lasted less than 1 minute.

5.6 Results analysis

5.6.1 Data improvements and network design

Comparing the 3 analyzed datasets with the networks that use the binary representation, it has been demonstrated that reducing the number of original attributes by applying Principal Components Analysis has lead to worst classification results by the neural networks:

```
# Step 1000 in binary representation networks
 Original - MSE: 0.3843637181159876       validation:      0.20900531362000127
PCA 95%   - MSE: 0.47207291601169515       validation:      0.2579010439590669
PCA 70%   - MSE: 0.8832434712441315       validation:      0.4077373583951626
```

In that sense, the same happens for networks with numeric representation of letters:

```
# Step 1000 in numeric representation networks
 Original - MSE: 0.04939202475595762      validation:      0.025375765334192654
PCA 95%   - MSE: 0.06980477428372317      validation:      0.03375271772234972
PCA 70%   - MSE: 0.11635975930452506      validation:      0.051971885839778256
```

In the case of the network with numeric representation and dataset PCA 70 %, extra analysis has been performed as described in section 5.4.3 on page 24. They show that between the 7, 6 and 4 hidden neurons versions the second one is a perfectly good choice. Although the 7 version reaches a lower mean square error in Step 1.000, it is not a huge difference compared to the computational time that supposes adding a neuron.

```
# Step 1000 in numeric representation networks with PCA 70%
7 Hidden neurons - MSE: 0.11547352146734574      validation:      0.05138052845065183
6 Hidden neurons - MSE: 0.11635975930452506      validation:      0.051971885839778256
4 Hidden neurons - MSE: 0.13392397639417233      validation:      0.05920344147403955
```

Therefore, for this case of study, it seems a good design strategy to choose the maximum $N_{hidden} = max\,(N_{input}, N_{output})$ as the number of neurons for the hidden layer.

On the other hand, from the 6 hidden neuron version, the pruning process has generated a 4 hidden neurons scenario with some less connections. The error evolution is quite similar to the 4 hidden neurons version with all the connections:

```
# Step 1000 in numeric representation networks with PCA 70%
4 Hidden neurons - MSE: 0.13392397639417233      validation:      0.05920344147403955
Pruned version   - MSE: 0.13570556026589223      validation:      0.05914588073689533
```

It is a computational improvement that could be interesting if the increase in the error margin is not a big problem.

5.6.2 Binary vs numeric representation

A ruby script (check section 9 on page 31) has been develop to analyze the best classification results for the JavaNNS networks with binary and numeric representations, which correspond to the original dataset (16 attributes).

For the network with binary representation of letters, the script performs the following steps:

1. For each binary value (26) of each object, consider correct classification by default and treat the following cases:

   (a) Desired output is one

i. Consider incorrect if the real output equal or less than "1 - margin_error"

(b) Desired output is zero

i. Consider incorrect if the real output more than "0 + margin_error"

2. Calculate total (in)correct classification and percentages

3. Calculate mean error $e = \frac{\sum_{i=0}^{26 \cdot total\ objects} |ouput_{real} - output_{desired}|}{26 \cdot total\ objects}$

Margin error has been established to 0.5, therefore any number greater or equal to 0.5 is considered a 1, otherwise it is interpreted as a 0. These are the analysis results:

```
Correct classifications:        5880 (84.51%)
Incorrect classifications:      1078 (15.49%)
Mean error:                     0.01
```

On the other hand, for the network with numeric representation of letters, the script performs the following steps:

1. For each object:

   (a) Consider correct classification if:

   $$ouput_{desired} - margin\,error \leq output_{real} \leq ouput_{desired} + margin\,error$$

   (b) Otherwise, consider incorrectly classified

2. Calculate total (in)correct classification and percentages

3. Calculate mean error $e = \frac{\sum_{i=0}^{total\ objects} |ouput_{real} - output_{desired}|}{total\ objects}$

The separation between consecutive letters is 0.04 in the numeric representation, therefore a margin error of 0.02 has been established. These are the analysis results:

```
Correct classifications:        915 (13.15%)
Incorrect classifications:      6043 (86.85%)
Mean error:                     0.11
```

It is clear that the design presented in this study for numeric representation of letters does not help to produce good classification results when working with neural networks, although they are much more efficient in term of CPU time than binary ones.

6 Conclusions

In general the K-Nearest Neighbours method (Weka IBk), as a geometric classifier type, seems to be the better choice for letter classification. It has obtained the best percentage of correct classification for almost every used dataset (92.59% for original, 91.94% for PCA95 and 78.95% for PCA70).

On the other hand, comparing neural networks, it has been possible to design a network that correctly classifies letters in the 84.51% of the cases with JavaNNS, while the Weka MultilayerPerceptron (with autoBuild activated) reached 81.46%. Therefore results are very aligned for this type of algorithms with independence of the implementation.

Finally, it is worth to mention how the percentage of correct classification has decreased (especially in the 70% of covered variance case) for those datasets where Principal Components Analysis has been performed. Although this method helps in simplifying the problem, it is important to check the reduction in the classification capacities of the algorithms. For example, in the case of K-Nearest Neighbours method and PCA 95%, the reduction was small (from 92.59% to 91.94%) but for PCA 70% the percentage fell significantly (from 92.59% to 78.95%).

7 Annex I. R script for statistical analysis

```r
library(rkward)
library(GDD)

# Read input file
ocr.data <<- read.table (file="input/OCR.arff", header=FALSE, sep=',', quote='"', dec='.', fill=FALSE, comment.char="#", col.names = c("class", "x-box", "y-box", "width-box", "heigth-box", "npix", "x-var-pix", "y-var-pix", "x2bar", "y2bar", "xybar", "x2ybr", "xy2br", "x-ege", "xegvy", "y-ege", "yegvx"), na.strings = "NA", nrows = -1, skip = 59, check.names = TRUE, strip.white = FALSE, blank.lines.skip = TRUE)

# Select variable to be analized
vars <- list (substitute (ocr.data[["yegvx"]]), substitute (ocr.data[["y2bar"]]), substitute (ocr.data[["y.var.pix"]]), substitute (ocr.data[["y.ege"]]), substitute (ocr.data[["y.box"]]), substitute (ocr.data[["xybar"]]), substitute (ocr.data[["xy2br"]]), substitute (ocr.data[["xegvy"]]), substitute (ocr.data[["x2ybr"]]), substitute (ocr.data[["x2bar"]]), substitute (ocr.data[["x.var.pix"]]), substitute (ocr.data[["x.ege"]]), substitute (ocr.data[["x.box"]]), substitute (ocr.data[["width.box"]]), substitute (ocr.data[["npix"]]), substitute (ocr.data[["heigth.box"]]))

# Prepare result vector
results <- data.frame ('Variable Name'=rep (NA, length (vars)), check.names=FALSE)

for (i in 1:length (vars))  {
        var <- eval (vars[[i]], envir=globalenv());

        results[i, 'Variable Name'] <- rk.get.description(vars[[i]], is.substitute=TRUE)

    # Stats
        results[i, 'Number of obs'] <- length(var)
        results[i, 'Number of missing values'] <- sum(is.na(var))
        results[i, 'Mean'] <- mean(var,na.rm=TRUE)
        results[i, 'Variance'] <- var(var,na.rm=TRUE)
        results[i, 'Sd'] <- sd(var,na.rm=TRUE)
        results[i, 'Minimum'] <- min(var,na.rm=TRUE)
        results[i, 'Maximum'] <- max(var,na.rm=TRUE)
        if (length (var) >= 1) {
                results[i, 'Minimum values'] <- paste (sort(var, decreasing=FALSE, na.last=TRUE)[1:1], collapse=" ")
        }
        if (length (var) >= 1) {
                results[i, 'Maximum values'] <- paste (sort(var, decreasing=TRUE, na.last=TRUE)[1:1], collapse=" ")
        }
        results[i, 'Median'] <- median(var,na.rm=TRUE)
        results[i, 'Inter Quartile Range'] <- IQR(var,na.rm=TRUE)
        temp <- quantile (var,na.rm=TRUE)
        results[i, 'Quartiles'] <- paste (names (temp), temp, sep=": ", collapse=" ")

        # Histograms
        filename <- paste(i, results[i, 'Variable Name'], sep=".")
        plotname <- results[i, 'Variable Name']
    GDD(file=paste("output/Histogram.", filename, sep=""), type="png", w=800, h=600)
    hist(var, probability=TRUE, breaks=15, main=paste("Histogram for", plotname), xlab="Value")
    rug(var)
    dev.off()

}

# Save stats
write.table(results, file = "output/OCR-stats.csv", sep = ",", row.names = FALSE)
```

8 Annex II. Ruby script for ARFF transformation into PAT

```ruby
#!/usr/bin/ruby

files = ["OCR","OCR.PCA95","OCR.PCA70"]
alphabet = {"A",0,"B",0,"C",0,"D",0,"E",0,"F",0,"G",0,"H",0,"I",0,"J",0,"K",0,
"L",0,"M",0,"N",0,"O",0,"P",0,"Q",0,"R",0,"S",0,"T",0,"U",0,"V",0,"W",0,"X",0,"Y",0,"Z",0}

files.each { |f|
    infile = File.new("input/" + f + ".arff", "r")
    outfilebin = File.new("output/binary_class/" + f  + ".pat", "w")
    outfilebin_train = File.new("output/binary_class/" + f  + ".Train.pat", "w")
    outfilebin_test = File.new("output/binary_class/" + f  + ".Test.pat", "w")
    outfile = File.new("output/numeric_class/" + f + ".pat", "w")
    outfile_train = File.new("output/numeric_class/" + f  + ".Train.pat", "w")
    outfile_test = File.new("output/numeric_class/" + f  + ".Test.pat", "w")

    # Find min and max for each input parameter
    process = false
    total = 0
    min = []
    max = []
    infile.each { |i|
            if (process) then
                v = i.split(",")

            for j in 0..(v.length - 1 - 1) # Do not process class (last column)
                if max[j].nil? or max[j] < v[j].to_f then
                    max[j] = v[j].to_f
                end
                if min[j].nil? or min[j] > v[j].to_f then
                    min[j] = v[j].to_f
                end
            end
            total = total + 1
            end
            if (i[0,5] == "@data") then
                    process = true
            end
    }

    #puts "max: " + max.join(" ")
    #puts "min: " + min.join(" ")

```

```
43      infile.close
44      infile = File.new("input/" + f + ".arff", "r")
45
46      # Headers
47      outfile.write("SNNS pattern definition file V3.2\n")
48      outfile.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
49      outfile.write("No. of patterns : " + total.to_s + "\n")
50      outfile.write("No. of input units : " + max.length.to_s + "\n")
51      outfile.write("No. of output units : " + "1" + "\n\n")
52      outfile_train.write("SNNS pattern definition file V3.2\n")
53      outfile_train.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
54      outfile_train.write("No. of patterns : " + (total * 0.70).round.to_s + "\n")
55      outfile_train.write("No. of input units : " + max.length.to_s + "\n")
56      outfile_train.write("No. of output units : " + "1" + "\n\n")
57      outfile_test.write("SNNS pattern definition file V3.2\n")
58      outfile_test.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
59      outfile_test.write("No. of patterns : " + (total - (total * 0.70).round).to_s + "\n")
60      outfile_test.write("No. of input units : " + max.length.to_s + "\n")
61      outfile_test.write("No. of output units : " + "1" + "\n\n")
62      outfilebin.write("SNNS pattern definition file V3.2\n")
63      outfilebin.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
64      outfilebin.write("No. of patterns : " + total.to_s + "\n")
65      outfilebin.write("No. of input units : " + max.length.to_s + "\n")
66      outfilebin.write("No. of output units : " + "26" + "\n\n")
67      outfilebin_train.write("SNNS pattern definition file V3.2\n")
68      outfilebin_train.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
69      outfilebin_train.write("No. of patterns : " + (total * 0.70).round.to_s + "\n")
70      outfilebin_train.write("No. of input units : " + max.length.to_s + "\n")
71      outfilebin_train.write("No. of output units : " + "26" + "\n\n")
72      outfilebin_test.write("SNNS pattern definition file V3.2\n")
73      outfilebin_test.write("generated at Mon Jan 10 15:58:23 2010\n\n\n")
74      outfilebin_test.write("No. of patterns : " + (total - (total * 0.70).round).to_s + "\n")
75      outfilebin_test.write("No. of input units : " + max.length.to_s + "\n")
76      outfilebin_test.write("No. of output units : " + "26" + "\n\n")
77
78
79      process = false
80      count = 0
81      infile.each { |i|
82          if (process) then
83              v = i.split(",")
84              # Rescale/Normalize between 0..1
85              for j in 0..(v.length - 1 - 1) # Do not process class (last column)
86                  if (max[j].to_f - min[j].to_f) != 0 then
87                      v[j] = (v[j].to_f - min[j].to_f) / (max[j].to_f - min[j].to_f)
88                  end
89              end
90
91              letter = v.pop # Extract and remove from vector the class (last column)
92                      # Binary array with order: V K W L A X M B Y N C Z O D P E Q F R G S H T I U J
93                      alphabet[letter[0,1]] = 1
94                      # Numerical value:
95                      value = 0
96              for j in 0..26
97                  if alphabet.keys[j] == letter[0,1] then
98                      # Order: V K W L A X M B Y N C Z O D P E Q F R G S H T I U J
99                      #         0.....................................................1
100                     #          with increments of 0.04
101                     value = j.to_f / 25 # From 0..25
102                     break
103                 end
104             end
105
106             outfilebin.write v.join(" ") + " " + alphabet.values.join(" ") + "\n"
107             outfile.write v.join(" ") + " " + value.to_s + "\n"
108             if count < (total * 0.70).round then
109                 outfilebin_train.write v.join(" ") + " " + alphabet.values.join(" ") + "\n"
110                 outfile_train.write v.join(" ") + " " + value.to_s + "\n"
111             else
112                 outfilebin_test.write v.join(" ") + " " + alphabet.values.join(" ") + "\n"
113                 outfile_test.write v.join(" ") + " " + value.to_s + "\n"
114             end
115
116             alphabet[letter[0,1]] = 0
117             count = count + 1
118         end
119         if (i[0,5] == "@data") then
120                 process = true
121             end
122     }
123
124     infile.close
125     outfilebin.close
126     outfilebin_train.close
127     outfilebin_test.close
128     outfile.close
129     outfile_train.close
130     outfile_test.close
131
132 }
133
134 # Print conversion table
135 #         V K W L A X M B Y N C Z O D P E Q F R G S H T I U J
136 #          0.....................................................1
137 outfiletable = File.new("output/conversion_table.tx", "w")
138 outfiletable.write "-------------------------------------------------------------------------------------------\n"
139 outfiletable.write "* Conversion table: Class, numeric and binary values                                    *\n"
140 outfiletable.write "-------------------------------------------------------------------------------------------\n"
141 outfiletable.write "Class\t\tNum.\t\t" + alphabet.keys.join(" ") + "\n"
142 outfiletable.write "-------------------------------------------------------------------------------------------\n"
143 count = 0.0
144 i = 0
145 while i < 26
146     alphabet[alphabet.keys[i]] = 1
147     outfiletable.write alphabet.keys[i] + "\t=\t" + count.to_s + "\t=\t" + alphabet.values.join(" ") + "\n"
148     alphabet[alphabet.keys[i]] = 0
149     count = count + 0.04
150     i = i + 1
```

```
151  end
152  outfiletable.write "-------------------------------------------------------------------------------------------\n"
```

9 Annex III. Ruby script for JavaNNS results analysis

```
1   #!/usr/bin/ruby
2
3   infile = File.new("input/Binary - OCR Original.res", "r")
4
5   margin_error = 0.5
6
7   total_corrects = 0
8   total_incorrects = 0
9
10  data_start = false
11  data_line = 0
12  error = 0
13  desired_output = ""
14  real_output = ""
15  infile.each { |i|
16      # Begin data entry
17      if i[0,1] == "#" then
18          data_start = true
19          data_line = 0
20          desired_output = ""
21          real_output = ""
22      end
23      # Process data entry
24      if data_start then
25          # Build string with desired output
26          if data_line >= 3 and data_line <= 5 then
27              desired_output = desired_output + i[0,i.length-1] + " " # Skip \n
28          end
29
30          # Build string with real output
31          if data_line >= 6 and data_line <= 8 then
32              real_output = real_output + i[0,i.length-1] + " " # Skip \n
33          end
34
35          # End of data entry => Check if it is correctly classfied
36          if data_line == 8 then
37              desired_output = desired_output.split " "
38              real_output = real_output.split " "
39              correct = true # Suppose correct by default
40              for i in 0..desired_output.length-1
41                  if desired_output[i] == "1" then
42                      if real_output[i].to_f <= 1 - margin_error then
43                          correct = false
44                      end
45                  else
46                      if real_output[i].to_f > 0 + margin_error then
47                          correct = false
48                      end
49                  end
50                  error = error + (real_output[i].to_f - desired_output[i].to_f).abs
51              end
52
53              if correct then
54                  total_corrects = total_corrects + 1
55              else
56                  total_incorrects = total_incorrects + 1
57              end
58              data_start = false
59          end
60
61          data_line = data_line + 1
62      end
63  }
64  puts "-------------------------------------------------------------"
65  puts "* Network with binary representation of letters        *"
66  puts "-------------------------------------------------------------"
67  correct_percent = 100 * total_corrects.to_f / (total_corrects + total_incorrects)
68  correct_percent = (100 * correct_percent).round.to_f / 100
69  puts "Correct classifications:\t" + total_corrects.to_s + " (" + correct_percent.to_s + "%)"
70  incorrect_percent = 100 * total_incorrects.to_f / (total_corrects + total_incorrects)
71  incorrect_percent = (100 * incorrect_percent).round.to_f / 100
72  puts "Incorrect classifications:\t" + total_incorrects.to_s + " (" + incorrect_percent.to_s + "%)"
73  mean_error = error / (26 * (total_corrects + total_incorrects))
74  mean_error = (100 * mean_error).round.to_f / 100
75  puts "Mean error:\t\t\t" + mean_error.to_s
76  puts "-------------------------------------------------------------"
77
78
79  infile = File.new("input/Numeric - OCR Original.res", "r")
80
81  margin_error = 0.02
82
83  total_corrects = 0
84  total_incorrects = 0
85
86  data_start = false
87  data_line = 0
88  desired_output = -1
89  real_output = -1
90  error = 0
91  infile.each { |i|
92      # Begin data entry
93      if i[0,1] == "#" then
94          data_start = true
95          data_line = 0
96          desired_output = -1
97          real_output = -1
98      end
99      # Process data entry
```

```
100         if data_start then
101             # Build string with desired output
102             if data_line == 3 then
103                 desired_output = i.to_f
104             end
105
106             # Build string with real output
107             if data_line == 4 then
108                 real_output = i.to_f
109             end
110
111             # End of data entry => Check if it is correctly classfied
112             if data_line == 4 then
113
114                 if real_output <= desired_output + margin_error and real_output >= desired_output - margin_error then
115                     total_corrects = total_corrects + 1
116                 else
117                     total_incorrects = total_incorrects + 1
118                 end
119                 error = error + (real_output - desired_output).abs
120
121                 data_start = false
122             end
123
124             data_line = data_line + 1
125         end
126     }
127     puts "-------------------------------------------------------------"
128     puts "* Network with numeric representation of letters       *"
129     puts "-------------------------------------------------------------"
130     correct_percent = 100 * total_corrects.to_f / (total_corrects + total_incorrects)
131     correct_percent = (100 * correct_percent).round.to_f / 100
132     puts "Correct classifications:\t" + total_corrects.to_s + " (" + correct_percent.to_s + "%)"
133     incorrect_percent = 100 * total_incorrects.to_f / (total_corrects + total_incorrects)
134     incorrect_percent = (100 * incorrect_percent).round.to_f / 100
135     puts "Incorrect classifications:\t" + total_incorrects.to_s + " (" + incorrect_percent.to_s + "%)"
136     mean_error = error / (total_corrects + total_incorrects)
137     mean_error = (100 * mean_error).round.to_f / 100
138     puts "Mean error:\t\t\t" + mean_error.to_s
139     puts "-------------------------------------------------------------"
```

References

[1] Luri, X. (2010). *Multivariate analysis. PCA.* Universitat de Barcelona.

[2] Witten, I.H., Eibe, F (2005). *Data Mining, Practical Machine Learning Tools and Techniques.* 2nd Edition. Elsevier.

[3] Wikipedia. *OneR.* Retrieved January 9, 2010, from the World Wide Web: http://en.wikipedia.org/wiki/Association_rule_learning#One-attribute-rule

[4] Wikipedia. *C4.5 algorithm.* Retrieved January 9, 2010, from the World Wide Web: http://en.wikipedia.org/wiki/C4.5_algorithm

[5] Wikipedia. *K-Nearest neigbor algorithm.* Retrieved January 9, 2010, from the World Wide Web: http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm