# Project 3 - Machine Learning - Hand Gestures

- Enunciado (ENUNCIADO.pdf)

## Instructions

The project instructions are located at ENUNCIADO.pdf (ENUNCIADO.pdf).

## Installation

Instructions for installing OpenCV on Ubuntu 14.04 or 13.10: here (http://www.sysads.co.uk/2014/05/install-opencv-2-4-9-ubuntu-14-04-13-10/).

Instructions for installing OpenCV on Ubuntu 13.04 or below: here (https://help.ubuntu.com/community/OpenCV).

Download the example sets (ftp://mi.eng.cam.ac.uk/pub/CamGesData/) and extract them to the examples folder.

## Generating ARFF files

Run arff_generator.py, providing the number of frames extracted per example (more than 2 -- more frames provides better but slower classification) and the example sets that will be used.

The following call extracts 6 frames per example, uses the Sets 2, 3 and 5, and saves it all on a file:

```
$ python extractor/arff_generator.py 6  2 3 5 > 6-235.arff
```

## Features

For each example, we extract a number of frames from it.

For example, for 2 frames, we get the first and last frames.
For 3 frames, we get the first, the last and the middle frames.

Then, we extract features for each of these individual frames.

The frame image is segmented based on a HSV range roughly corresponding to skin color range: [0, 30, 60] to [50, 150, 255];

Then, from all the contours in the image (found with findContours), we grab the one with the biggest area.

We use this contour to fit a minimum-area rectangle, an ellipse, and a convex hull. We also extract convexity defects between the contour and the convex hull, and the contour's Hu moments.

From these primitives, we can extract all of the features:

```
area
perimeter
convex_hull_area
solidity
rect_center_x
rect_center_y
rect_width
rect_height
rect_angle
rect_aspect_ratio
ellipse_center_x
ellipse_center_y
ellipse_major_axis
ellipse_minor_axis
ellipse_angle
farthest_convex_defect
hu_moment_1
hu_moment_2
```

```
hu_moment_3
hu_moment_4
hu_moment_5
hu_moment_6
hu_moment_7
```

These 23 features are extracted for each frame. So, if an example is composed of 10 frames, it'll actually have 230 features.

In the ARFF file, we append a number to each attribute, corresponding to the frame number.

# Functions

Based on the file generated by the following run, containing 2 frames per example and all 5 sets:

```
$ python extractor/arff_generator.py 2 1 2 3 4 5 > All.arff
```

We tested some classification functions and varied their parameters.

Using all 5 sets is the most challenging, because of the illumination setup by each set.

Extracting 2 frames per example improves the runtime of the classifier. Had we used more frames, we'd have a better result overall. Because right now our objective is to find the best function, we can use only 2.

## Decision Tree (trees/J48)

### Parameters

**binarySplits** had no influence on the results.

**confidenceFactor** did not impact much on classification performance, but higher levels slowed down the classification quite a bit. We maintained it at 0.25.

**minNumObj** was optimal at 10.

**reducedErrorPruning** deemed worse results, so we turned it off.

### Results

```
Time taken to build model: 0.33 seconds
Correctly Classified Instances         634               70.4444 %
Incorrectly Classified Instances       266               29.5556 %
Kappa statistic                          0.6675
Mean absolute error                      0.0876
Root mean squared error                  0.2293
Relative absolute error                 44.3687 %
Root relative squared error             72.971  %
Total Number of Instances              900
```

## K-Nearest Neighbor (lazy/IBk)

### Parameters

**KNN** was optimal at 1.

**distanceWeighting** had no influence on the results.

**meanSquares** had no influence on the results.

**nearestNeighbourSearchAlgorithm** we tried different ones, but all yielded the same result.

**windowSize** was only good at 0.

### Results

```
Time taken to build model: 0 seconds
```

| | | |
|---|---|---|
| Correctly Classified Instances | 712 | 79.1111 % |
| Incorrectly Classified Instances | 188 | 20.8889 % |
| Kappa statistic | 0.765 | |
| Mean absolute error | 0.0481 | |
| Root mean squared error | 0.2143 | |
| Relative absolute error | 24.3407 % | |
| Root relative squared error | 68.1877 % | |
| Total Number of Instances | 900 | |

### SVM (functions/libsvm)

#### Parameters

**SVMType** nu-CSV was better than C-CSV.

**coef0** 1.0 (small influence).

**cost** 1.0 (no influence).

**degrees** was optimal at 3

**nu** was optimal at 0.1

#### Results

| | | |
|---|---|---|
| Time taken to build model: 22.3 seconds | | |
| Correctly Classified Instances | 771 | 85.6667 % |
| Incorrectly Classified Instances | 129 | 14.3333 % |
| Kappa statistic | 0.8388 | |
| Mean absolute error | 0.0363 | |
| Root mean squared error | 0.1603 | |
| Relative absolute error | 18.3595 % | |
| Root relative squared error | 51.0225 % | |
| Total Number of Instances | 900 | |

### Neural Network (functions/MultilayerPerceptron)

#### Parameters

**hiddenLayers** t (attribs+classes)

**learningRate** 0.3

**momentum** 0.5

**trainingTime** 500 (diminishing returns).

#### Results

| | | |
|---|---|---|
| Time taken to build model: 1.75 seconds | | |
| Correctly Classified Instances | 780 | 86.6667 % |
| Incorrectly Classified Instances | 120 | 13.3333 % |
| Kappa statistic | 0.85 | |
| Mean absolute error | 0.0296 | |
| Root mean squared error | 0.1721 | |
| Relative absolute error | 15 % | |
| Root relative squared error | 54.7723 % | |
| Total Number of Instances | 900 | |

## Maximizing

Now that we know that, for this case, using a NN works best, we're going to try to achieve the highest possible correct classification rate.

For this, we'll use only the 1st set, extracting 10 frames from each example.

### Results

```
Time taken to build model: 89.95 seconds
Correctly Classified Instances         173               96.1111 %
Incorrectly Classified Instances         7                3.8889 %
Kappa statistic                          0.9563
Mean absolute error                      0.014
Root mean squared error                  0.0845
Relative absolute error                  7.074  %
Root relative squared error             26.8859 %
Total Number of Instances              180
```

## Implementing

We implemented a Neural Network algorithm with backpropagation in Ruby.

It can be used like this:

```
require './nn'

xor = [
  [[0,0], [0]],
  [[0,1], [1]],
  [[1,0], [1]],
  [[1,1], [0]]
]

# input, hidden, output
n = NeuralNetwork.new(2, 2, 1)

puts "Training..."
n.train(xor)

puts "Evaluating..."
correct = n.test(xor)

puts "Got #{correct}/#{xor.length} -> #{100.0*correct/xor.length}%"
```

There's also a script that can take an ARFF file and pass it through the Neural Network.

```
$ ruby arff_nn.rb ../examples/Set1-2frames.arff
Training...
Evaluating...
Got 162/180 -> 90%
```

(This takes a long time! Our implementation is not that sophisticated)