M1 Project - Image

Simon Mauras
Victor Hublitz

March 29, 2016

Contents

1 Introduction

This report describe the implementation of a feature-based shape-classifier. We suppose that we solved the segmentation problem and we want to classify binary shapes. This project is part of the evaluation of a master's degree lecture at ENS Lyon.

In a first part we are going to describe the actual implementation of our tool. After a brief part on the machine learning problem, we will speak about normalization of images. Then we will explain how we designed features and we will finally conclude analysing the results we get.

2 Implementation

2.1 Installation

To install our shape classifier, you will need to install several Python 3 packages (scipy, numpy, sklearn). You will also need the C++ library DGtal.

To compile our project you need to have cmake installed.

```
mkdir build && cd build
cmake .. -DDGtal_DIR=/path-to-dgtal-folder/build
```

2.2 Feature extraction

The feature extraction is done in C++ using the library DGtal. The file `features.cpp` provides several functions to extract a vector of features from an image. The program `shape_indexing` takes as input an image and write a vector of values in $\mathbb{R}^d$ on the standard output. The program `shape_distance` computes the similarity between the vectors of features of two images and write it on the standard outut. The python script `classifier_corpus.py` builds a vector of feature for each image of the directory `database` and store it in `corpus.csv`.

2.3 Machine learning

Once we have extracted a vector of values for each image of the database, we train a learning model on the corpus. The script `classifier_learning.py` reads the vectors of features, train a learning model and export it in the file `model.dump`.
We chose to use a k-nearest neighbors classifier (scikit-learn). To have an estimation of the score with the whole database, we use cross-validation.

3 Learning models

As machine learning is not the objective of the project, this part will be brief. The goal of model selection is to find the best machine learning algorithm (and the best parameters) for our dataset. We tested several models (k-neighbors, Gaussian naïve bayes, Logistic regression, Decision tree, Random forest, ...). To evaluate the score we use cross-validation. We chose to use a k-neighbors classifier with $k = 25$.

3.1 k-nearest neighbors classifier

Let's introduce the problem of machine learning. We are given a set E of points, each of them being labeled by a class $c \in C$. We assume that the distribution of the classes follows some pattern. We want an algorithm able to decide in which class a point $e \in E$ belongs. To do that, we are given a corpus (X, y) with $X \in E^n$ a family of points and $y \in C^n$ the classes to which they belong.

We chose the k-nearest neighbors classifier algorithm. For each query $e \in E$ we look for the k-nearest point to e in X (using a distance d). The class with the biggest frequency in this set of k points is the most likely class. To compute efficiently the k-nearest neigbors of a point we typicaly store X in a datastructure like a Kd-tree.

3.2 Cross-validation

Now that we have a learning algorithm, we want to evaluate its efficiency. But we only know the classes of the points of the corpus. And computing the score of our algorithm on the corpus is a very bad approximation. Indeed our algorithm may be very good inside the corpus and very bad outside of it. One solution to this problem is cross-validation (N-fold cross validation).

- Shuffle the corpus, then split it in N parts (of equal size).
- For each part, train the algorithm on the $N - 1$ remaining parts and compute the score with the current one.
- Compute the average (or the minimum) of those scores.

We can expect that the score outside of the training corpus is close to the cross-validation score.

4 Normalization

During the classification, we need to analyse some noisy images. A preprocessing is therefore needed. Our preprocessing is done in several steps.

- Scale the image.
- Smooth with a median filter.
- Compute the contour of the shape.
- Fill-in the shape.

We normalize images using function `normalize` (in `features.cpp`). The scaling is important because even if our features are scale-invariant, the digitization is a huge problem. Indeed after a few experiments, we realize that when the order of magnitude of the digitization is close to the order of magnitude of the shape, our results are not consistent. So scaling then smoothing is better than just using the initial image. The result of normalization on a very noisy shape is satisfactory.

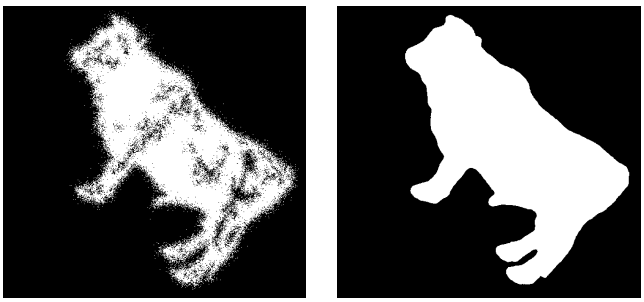


Figure 1: Normalization of a noisy shape

If there are several connex components, the biggest one is choose to be the shape. The filling part of the normalization is very usefull for classes like butterfly, cattle or dog as some images contains a lot of gaps and other don't.

Figure 2: Filling the shape

5 Feature design

5.1 Distance Transformation Percentile

Let's take a look at the distance transformation histogram for several shapes.

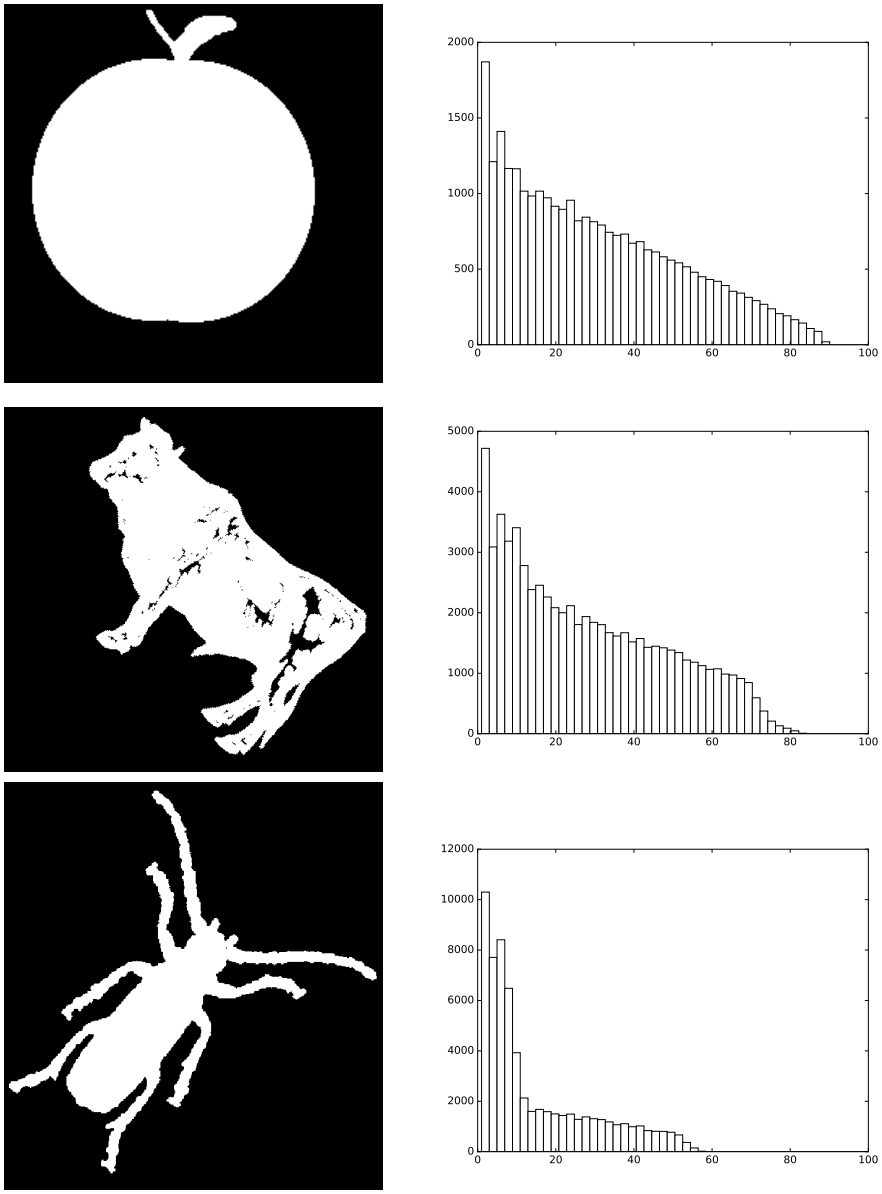


Figure 3: Distance Transformation Histogram

We can observe that a shape can be characterized by the distribution of its distance transformation.

Now we need to build an estimator satisfying the following requirements:

- Must be invariant under translation, scaling and rotation.
- We need some kind of continuity. If there is a small modification of the shape then the new estimation has to be close to the initial one.

We already have invariance by translation and rotation. In order to have the invariance under scaling, we can normalize the distance and the frequencies.

However we can notice that it is not enough to have a good estimator. Indeed the histograms $(1, 0, \ldots, 0)$ and $(0, 1, 0, \ldots, 0)$ are very close but the distance between those two vectors is huge. To fix this problem we are going to compute percentiles. The function `compute_dt` (in `features.cpp`) compute those features.

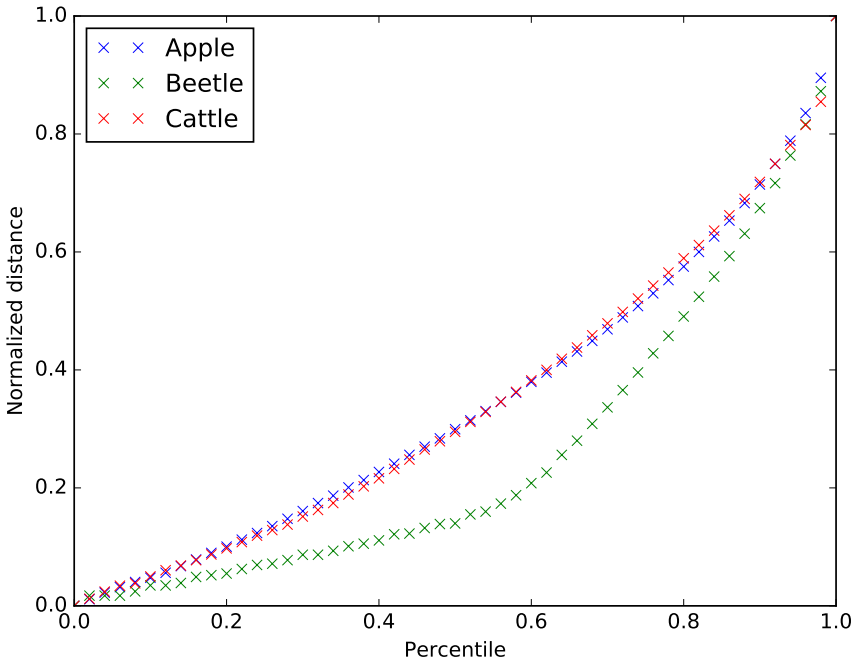


Figure 4: Distance Transformation Percentile

We just build an estimator that can help us to distinguish an apple from a beetle. But we can notice that this estimator is not very good to decide whether a shape is a cattle of an apple.

5.2 Perimeter and Area

Among all the estimators we could implement, we chose to compute the perimeter and the area of the shape. However, a good feature must be invariant under scaling. When a shape is scaled by a factor k, all length are multiplied by k and all areas are multiplied by k^2. So we add the perimeter over the square root of the area to the list of features.

It is implemented in the functions `compute_area` and `compute_perimeter` (in `features.cpp`). We compute the perimeter by decomposing the contour into maximal DSS (DGtal implementation). We only compute the number of white pixels to evaluate the area of the shape.

5.3 Rotation Distance

One caracteristic which is still not used is point reflexion. There is a lot of images with symetry and rotation invariance. Our idea is to compare the initial image with its rotation of θ degrees around its centroid. We can prove that if we can find a invariance under rotation for our image, then the centroid of the shape will be the center of the rotation.

For $\theta = 2\pi/k$ for $k = 2 \dots 10$ we chose to compute our features as follow:

- Compute the centroid c of the shape
- For each point p of the shape
  - Compute its rotation $q = c + e^{i\theta}(p - c)$
  - If there is a point in the shape at distance to q less than r, then add 1 to the score.
- Normalize the score (divide by the number of pixels in the shape)

The function `similarity_rotation` (in `features.cpp`) implement the algorithm we just describe.

The figure 5 contains the value of those features for five different classes. When k goes from 2 to 10, the angle goes from π to $\pi/10$. We can recognize the $\pi/6$ rotation invariance of device 1 and the circular shape of an apple. The results are very relevant for regular shapes (circles, squares, stars, ...). With distance transformation we had trouble to distiguish an apple from a cattle, now we can separate those two classes.

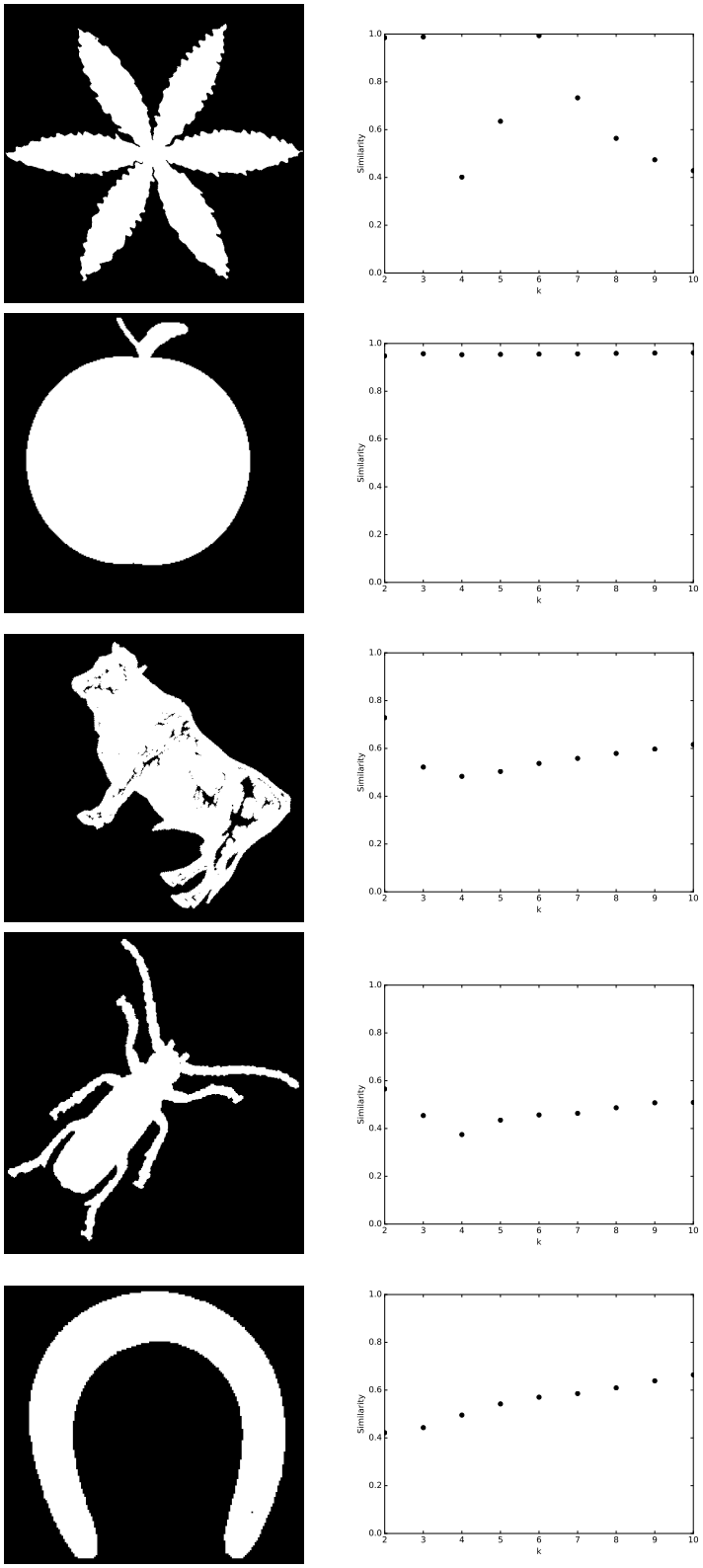


Figure 5: Rotation Distance

5.4 Convex hull

The convex hull of a set of point X is the smallest convex set that contains X. There exists several algorithms to compute the convex hull of a 2D finite set (Graham's scan, ...) We implemented the function `convex_hull` (in `features.cpp`) which computes the convex hull of a 2D shape in an image.

We chose the monotone chain algorithm (aka Andrew's algorithm) as our points are aready sorted, it is indeed a 2D matrix of 0/1. We first compute the upper hull and the lower hull. Then we simultaneously go through the domain, the upper hull and the lower hull to decide whether each point of the image is in the convex hull. The overall complexity is in $\mathcal{O}(n)$ where $n = a \times b$ is the size of the image.

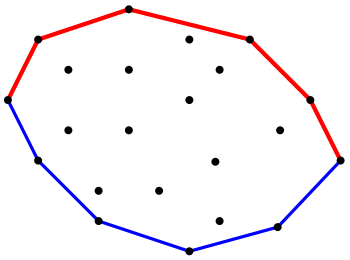
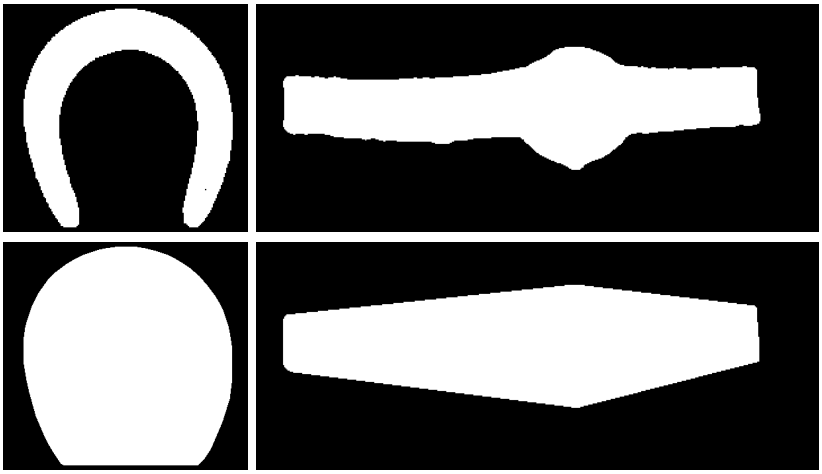


Figure 6: Upper and lower hull

One feature that give very good results is the ratio between the area of the initial image and the area of the convex hull of the shape. It is straighforward to show that this estimator is invariant under translation and rotation. We can also notice that the two area are multiplied by the same factor when scaling an image.



Ratio $= 0,46$ Ratio $= 0.79$

Figure 7: Convex Hull Area

We also duplicated all the features we designed previously and used them on the convex hull of the shape.

6 Conclusion

We have presented in this report the learning model we chose and the features we designed. Now is the time to analyse the results and the limits of our choices.

6.1 Feature normalization

The description of an image is a vector of values in $\mathbb{R}$. One first problem is that some of those values are bigger than others. More annoying, the variances are very different from one coordinate to one other. It is a problem as our learning model uses a "uniform" distance. We fix this problem manually by testing several weights for each feature.

6.2 Robustness results

The robustness of the feature extraction is an indicator showing how resistant to noise/scaling/rotation our features are. From one image we can construct an other one by rotating, scaling and adding noise. If we compute the feature vectors of those two images, their distance should be as small as possible. Let's define the similarity of two vectors $(u_i)_i$ and $(v_i)_i$.

$$\text{Similarity} = 1 - \max_i \frac{|u_i - v_i|}{|u_i| + |v_i|}$$

This definition is interesting because a good robustness certifies that for each coordinate the distance is small. (We use ∞ norm).

Noise	Robustness score	Variance
0.0	0.993	2.0×10^{-4}
0.3	0.989	5.0×10^{-4}
0.7	0.982	7.0×10^{-4}

Figure 8: Robustness results

6.3 Classification results

We don't have the whole database of image so we can't know what our final score will be. However we have some hints. The cross validation score is supposed to be an approximation of the score of the classifier outsite of the training set. The cross validation score we get is 0.980. As the robustness is close to 1, we can expect a final score around 97%.