Digital Images Project

Fabrice Lécuyer and Etienne Desbois

Introduction

Given a database containing several images for several classes, we want to build a software that takes a new image and states probabilistically to which class it belongs. We have split the work into two big steps : extraction of features for a given image, and classification of features vector.

The usage of all our scripts can be found in `README.md`.

1 Features extraction

The first part deals with images and aims to extract a list of numbers called *features* to describe this image. We got a total of 12 features, including handmade features (intuitive description of images) and image moments (a well-known means to qualify images).

This part is achieved by the C++ program `invariants.cpp`

1.1 Preprocessing

The goal here is to reduce noise as much as possible. It can be an issue to estimate perimeter or skeleton for example. We chose here to use what we have seen in class, e.g. erosion/dilation, which can be combined to form closing/opening. A good noise-filter is to combine those once again. With a 3x3 block we retrieve a connex shape (most of the time) but the shape itself is not that great (not as smooth as we would want it to be). Yet it erase noise which can be useful for some features. It preceeds feature extraction in `invariants.cpp`.
The filecode `preprocessing.cpp` can be used on its own to see some outputs and possibilities (currently designed for opening, closing, opening(closing) and closing(opening)).

1.2 Handmade features

1.2.1 Thickness

This feature is $\frac{perimeter}{area}$. Its goal is to roughly estimate the thickness of the object depicted. Its sole purpose is to be able to differentiate an apple (low coefficient) from a bone (high coefficient). It is obviously invariant under translation, rotation, rescaling. Yet it is subject to noise. As said before, pretreatment results in a connex but not smooth image, which increase the perimeter and thus this feature.

1.2.2 Skeleton based features

We thought of different way to characterize shapes. Contours, "branches", etc. Since except from few animal classes, shapes don't contain holes, the Freeman code contains all the information on the shape. However, we could not see any ways to characterize it as a feature (except having 1000 features, one with a distance to each other Freeman code of the database, but we would have an issue with dimensionnality in our knn). Furthemore, our preprocessing is not efficient enough.
So we tought of simpler feature, that carries nearly as much information : the skeleton.

These features are based on the skeleton of the shape. To produce it, the algorithm used is an adaptation of Zhang-Suen. However, it is not as fast as we hoped for, so we strongly recommend you not to compute data.csv, since it requires more than an hour. Furthermore, it was harder to use than we expected. We had far more anchor and hinge points than we expected. It might be due to an error in the algorithm though but as seen in lectures, the skeleton is not always like we want it to be.

The time needed to compute the skeleton varies much : it can be nearly immediate like around 15sec for some classes (classic, some devices like device7, etc.).
We extracted three features from the skeleton :

- Its size, divided by the image size : its goal is not really clear, since it is not obvious that different classes will have different skeleton sizes. It is invariant to rescaling and rotation, probably not noise but differences should be negligeable.
- Its number of anchor points : here the idea is to have the number of branches. It is not invariant to rotation/rescaling, but once again differences should be negligeable
- Its number of hinge points (points linked to at least 3 others) : it is another measure of the same idea. Same invariancy results

1.2.3 Trick : size

This one is a little "trick". It doesn't give any information on the object itself but on the image. It is $\frac{area}{size}$. Bells for example cover most of the image but bones only a small part. It helps distinguishing them.
This trick is obviously invariant to everything. Noise doesn't affect it much.
It should reach its limits with new images (a bell will still be bigger than a bottle though).

1.3 Image moments

This kind of features is called image moments. It is a purely mathematical way to describe the distribution of pixels in an image. The p, q moment of a 2D set of pixels $\mathcal{S}$ is defined by :

$$m_{p,q} = \sum_{(x,y)\in\mathcal{S}} x^p y^q$$

Low moments have intuitive explanations : $m_{0,0}$ is the number of pixels, $\hat{x} = \frac{m_{1,0}}{m_{0,0}}$ is the mean value of abscisse, $\hat{y} = \frac{m_{0,1}}{m_{0,0}}$ is the mean value of ordinate.

To obtain translation invariance, we take a new origin $(\hat{x}, \hat{y})$ and define central moments :

$$\mu_{p,q} = \sum_{(x,y)\in\mathcal{S}} (x-\hat{x})^p (y-\hat{y})^q$$

Now we would like to have features with invariance by rotation. Hu's paper gives seven of them (page 7). Let us prove that the first one is indeed invariant :

First, we suppose the image is centered, which means $\hat{x} = \hat{y} = 0$.

$$\phi_1 = \mu_{2,0} + \mu_{0,2} = \sum_{(x,y)\in\mathcal{S}} (x-\hat{x})^2 (y-\hat{y})^0 + (x-\hat{x})^0 (y-\hat{y})^2 = \sum_{(x,y)\in\mathcal{S}} x^2 + y^2$$

Now define $(x' = x\cos\theta + y\sin\theta, y' = -x\sin\theta + y\cos\theta)$ obtained from (x, y) by rotation of angle θ.

$$\begin{aligned}
\phi_1' &= \sum_{(x,y)\in\mathcal{S}} x'^2 + y'^2 = \sum_{(x,y)\in\mathcal{S}} (x\cos\theta + y\sin\theta)^2 + (-x\sin\theta + y\cos\theta)^2 \\
&= \sum_{(x,y)\in\mathcal{S}} (x^2\cos^2\theta + y^2\sin^2\theta + 2xy\cos\theta\sin\theta) + (x^2\sin^2\theta + y^2\cos^2\theta - 2xy\cos\theta\sin\theta) \\
&= \sum_{(x,y)\in\mathcal{S}} (x^2+y^2)\underbrace{(\sin^2\theta + \cos^2\theta)}_{=1} \\
&= \phi_1
\end{aligned}$$

The last step is scale invariance. We define the standardardized moments by $\eta_{p,q} = \frac{\mu_{p,q}}{\mu_{0,0}^{\left(1+\frac{i+j}{2}\right)}}$.

Suppose we have $(x' = \alpha x, y' = \alpha y)$ obtained frome (x, y) by $\alpha > 0$ scaling.

$$\Phi_1' = \eta_{2,0}' + \eta_{0,2}' = \frac{\mu_{2,0}' + \mu_{0,2}'}{\mu_{0,0}'^2} = \frac{\alpha^2(\mu_{2,0} + \mu_{0,2})}{(\alpha\mu_{0,0})^2} = \frac{\mu_{2,0} + \mu_{0,2}}{\mu_{0,0}^2} = \Phi_1$$

Similarly, we get 7 features $\Phi_1, \ldots, \Phi_7$ as given in Hu, that are invariant under translation, rotation and scale.

After many tests we discovered that the standardized moments η gave bad results compared to centered moments μ. Even though we could not understand this behaviour, we decided to keep using μ, but apply normalization more carefully.

2 Classification

This part aims to recognize the class of a new image, based on the features of images of a given database. It should be able to deal with modifications of a known image or with a new similar image.

This part has been done in Python, using `pandas` library.

2.1 Method

We consider an image that has undergone modifications (scaling, rotation, addition of noise), and the aim is to classify it : we want to know a distribution of probability that it belongs to a given class. To do so, we use a simple **k-nearest neighbors** algorithm.

First of all, for every example of the database a signature is computed and the label is given with respect to the class of objects. Then we compute the signature of the modified image, and compute its distance (see below) with all known signatures.

Each close neighbor of our modified image increases the probability that they are in the same class. The final probability is the sum of three aspects :

— a basic probability for each class, because we know such an algorithm may fail sometimes
— a probability shared among the k closest neighbors
— a probability decreasing with distance do decide between further classes

2.2 Distance

The definition of the distance have been a tough question. Since we were not allowed to use complicated machine learning devices such as metric learning, we had to design something simple.

2.2.1 Euclidean distance

The first idea was to use the Euclidean distance on vector signatures. However, there were several orders of magnitude between the values of different features. It would have given some features a huge importance.

To counter this problem, we used normalization functions. Several have been tested (0-1 normalization, division by median, applying logarithm to spred the values more uniformly...), but we finally decided to just divide each value by the mean of the feature.

Yet the plot of values clearly showed that values where concentrated in some areas, lowering the efficiency of this simple distance. This method is still used in `classificationOld.py`.

2.2.2 Ranking

Instead of keeping the values of each feature, which leads to concentrated points, we replace them by the rank they have in the feature. Smallest value gets rank 0, biggest gets rank N, whatever the range is.

With this solution, we obtain more uniformly distributed points, but it breaks the agglomeration of some classes.

3 Results and further ideas

3.1 Results

Both classifiers seem quite robust to noise.

3.1.1 Which k for our k-nearest-neighbours ?

The lower the k, the better our result. For $k = 1$, we reach a $90 - 95\%$ scoring, which means that our features are indeed quite robust to changes made (rescale, rotation and noise) ! However, for new images not in the database yet, it can be dangerous to keep $k = 1$. We didn't perform any cross-validation to estimate it, so we chose arbitrarily. For $k = 4$, we are around $75 - 80\%$, for $k = 9$, $70 - 75\%$.

We decided to keep $k = 4$ and hope it will be robust with new images. Please note that you can choose k by adding it at the end of the line : `python3 classification.py img.pgm k`.

3.1.2 With the old classifier

This one is with the euclidean distance. Our program ouputs the good answers around 60-65% of the time. When it's proven wrong, it is still nearly always in the top3 or 4. We could not find features relevant enough to break this glass ceiling.
Even if it is not the best for perfect score, it had to be said that when it's mistaking, it is not by far ! The final version seems to have a bug with some features, resulting in a rank 70 (lowest possible in getRank script) sometimes. So we work only with the first ones.
This classifier was not optimized, we prefered to focus on the current classifier to it.

3.1.3 With the current one

On the contrary, this one with the ranking system can be mistaking by far. However, the perfect score rating outperforms the other one. Its performance are around 70%. (to check scores we had to do it on a small sample (50 to 100), so it has a high variance (from 10% to 14%).

3.2 Furtherwork

Other features that we thought about was to work with the convex hull. For example the proportion of empty cell in it could provide information on its hollowness, thickness, and branches (this proportion would be maximazed for a n-branch star, minimized for a circle). We didn't do it for a lack of time, and because our algorithm is already slow enough. We think the skeleton provides more valuable information than the convex hull. One possible breakthrough would be to ponderate

smartly our features. We did it "by hand" yet it could be optimised. A neural network working on our feature-vector (one layer, linear just to identify ponderation) would probably break our current score.
Yet we decided to go without it. We thought of learning a Genetic Algorithm instead, but either we worked on the same processed images batch (and we had a risk of overfitting), either we generated them randomly, and then it would have required too much time due to the long time needed to compute the invariants. Eventually we did nothing of it.