Technical tips: Step 3 (continued)

1. Checking histogram features (continued)

The last technical tips document introduced a few simple ideas to check the features you have been computed. We elaborate on these below, specifically for features represented by *histograms*, such as the shape property distributions (A3,D1,…,D4). Walk through these tips in turn in the order below:

(1) Normalizing histograms

To meaningfully compare two histograms computed on different number of elements, these need to be *normalized.* This 'factors out' the number of elements from the computation. This is especially important for the shape property distributions which are typically computed for different number of points (D1), point-pairs (D2), point-triplets (A3,D3), and sets of four points (D4). Moreover, normalized histograms will be easier to use when computing shape distances (see Module 4).

Normalizing a histogram $H = \{h_i\}$ is simple: Replace each value h_i by $h_i / \Sigma_i h_i$. This way, each bar becomes a *percentage* in [0,1], regardless of the total number of samples $\Sigma_i h_i$.

(2) Sampling the space of possibilities

All shape-property distributions use combinations of 1, 2, 3, or even 4 vertices from the N vertices of a shape. Obviously, when N is large (thousands), this yields too many combinations to compute (N to N^4). To reduce this space of possibilities to something manageable, we can use *smart subsampling*. The key idea behind is to consider only n (from the N^k possible ones) so that we don't favour, in this choice, *specific* parts of the space of possibilities.

A simple algorithm to do this is as follows (example given for the D3 descriptor):

```
Algorithm input: n, where n < N^3              //number of samples to evaluate
int random();                                   //returns a random integer in 0..N-1
int k = pow(n, 1.0/3.0)                         //compute number of samples along each of the three dimensions
for (int i=0;i<k;++i)
{
  int vi = random();
  for (int j=0;j<k;++j)
  {
    int vj = random();
    if (vj==vi) continue;                       //do not allow duplicate points
    for (int l=0;l<k;++l)
    {
      int vl = random();
      if (vl==vj || vl==vi) continue;
      construct triangle with vertices (vi,vj,vl); evaluate its area;
    }
  }
}
```

(3) Interpreting histogram features

As explained in the last tech tips, a *plausible* feature (meaning, a feature which likely performs well for shape retrieval) should take similar values for similar shapes, and different values for different shapes, respectively. Hence, a simple way to check the plausibility of features is to compute histograms for various shapes *whose similarity we know* and then see if the histograms reflect this similarity.

Consider the division of shapes into classes in the database. Arguably, shapes in the same class are more similar than shapes in different classes. Hence, we can visually check if the histograms of shapes in the same class are more similar than those of shapes in different classes.

Two problems can occur here:

- We have shapes in *different* classes with *similar-looking* histograms; this may create **false positives** during retrieval;
- We have shapes in the *same* class with *different-looking* histograms; this may create **false negatives** during retrieval.

This is illustrated by some concrete examples:

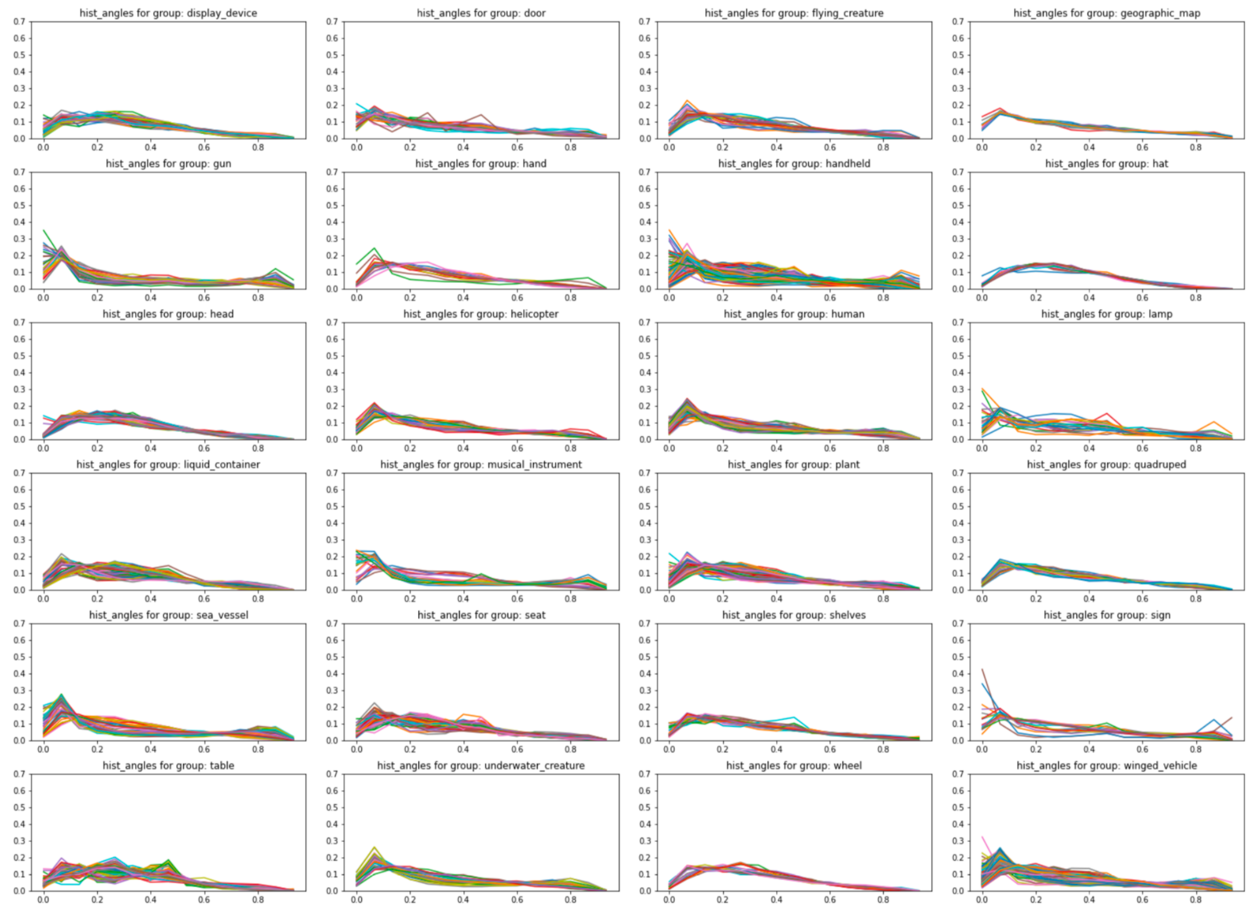

Figure 1: *A3 descriptors for all shapes, grouped per class*

In this image, we'd like to have (1) very similar graphs within each plot (similar histograms of shapes in same class) and different overall graphs in different plots (different histograms of shapes in different classes). We see that the latter property holds reasonably well, though, not for all classes (some plots are quite similar). Within each plot, the curves are quite well grouped, which is good – same-class shapes yield similar histograms.

The plot below shows a different story:

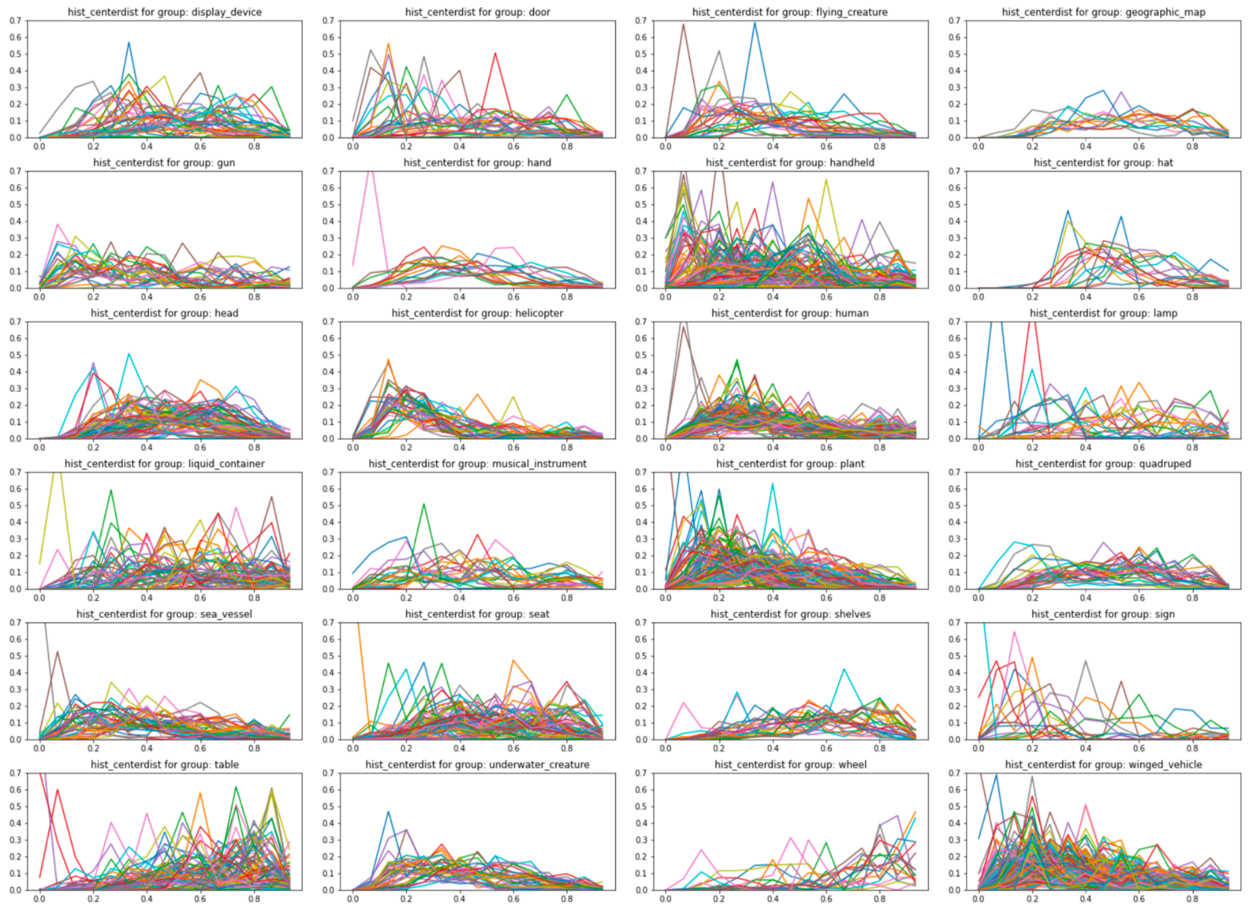

Figure 2: *D1 histograms for shapes, grouped per class.*

Compared to the A3 histograms, we see now a wide variation in distances between shapes, even within the same class. This means that either A3 is very sensitive to the actual shape, or it is suboptimally computed. Either way, this is not good for retrieval, since it will create likely many false negatives. Also, it is hard to say that each class exhibits a particular 'signature' formed by its histograms. This implies the possibility to have a lot of false positives during retrieval. In each case, seeing the above indicates there is likely trouble with the A3 features computed.

(4) Improving histogram features

Say that you observed some of the problems of histogram features outlined above. How to *improve* the computation of such features? There are several simple steps one should follow in any case:

- **Sample count:** Make sure you use sufficient samples for computing the respective feature. Consider e.g. D3: if you use a total of, say, 10000 combinations (of the 3 points), uniformly distributed over the space of possibilities, this means roughly that only pow(10000,1/3)= 21 vertices are considered (!!!) Clearly, this is a too low number. A far more reasonable number is to consider 100 such vertices, thus a total of 100^3 = 1 million combinations. If coded smartly, this is not a too long computation.
- **Debugging the computation:** Take two particular shapes which you find extremely similar. Compute and compare their histograms. You should really get two very identical plots. If not, there is surely a problem in the histogram computation. Check that the shapes are *uniformly* sampled; and that you are normalizing the

histograms properly. These are two often-met mistakes. In particular, the shape property distributions are quite sensitive to a *good spread* of the sample points over a shape's surface.