Technical guide step 5 - Performance

As in the previous guides, exercises are included in this guide to help you think about the discussed topics. You don't need to hand in any solutions to these exercises; they are meant only for personal study.

KNN

In order to reduce query times, there's a number of things we can do. One of them is to use a precomputed data structure which partitions our set of feature vectors such that we no longer need to compare each feature vector in our database with the query vector. Instead, we can do a fast query on the precomputed data structure. Such a query then returns the k Nearest Neighbors (KNN) which are closest to our query vector in the search space.

Multiple types of data structures can be used for this purpose. In the provided sample script '`knn.py`', a KNN search using a KDTree data structure is implemented. A KDTree (meaning k-dimensional tree) partitions the feature space into a hierarchy of k-dimensional 'boxes' (this 'k' should not be confused with the 'k' in KNN), which allows for quick look-ups of samples inside those boxes. In our case the *samples* are the feature vectors in the database.

We provide the entire list of feature vectors to the KDTree constructor as a single feature matrix. After construction, we can call 'query' on the KDTree and specify the query shape's feature vector as well as the number of neighbors we'd like to retrieve (k). This function returns the distances from the found neighbor vectors to the query vectors, as well as the row indices into the feature matrix we provided to the KDTree constructor. These indices allow us to retrieve the names and labels of the neighbors.

If we run the script using the mini database provided in the previous guide to obtain some sample feature vectors, we see that the KNN search indeed returns shapes that are in the same class as the query shape (at least when we use a shape from the 'Humanoid' class).

```
Building KDTree... Finished.
3 nearest neighbors for shape Humanoid/m140.obj (label='Humanoid'):
    Humanoid/m118.obj (label='Humanoid', distance=0.21525849524002533)
    Humanoid/m159.obj (label='Humanoid', distance=0.22696160056273837)
    Humanoid/m151.obj (label='Humanoid', distance=0.2682109968103471)
```

Figure 1: Console output of the script 'knn.py'

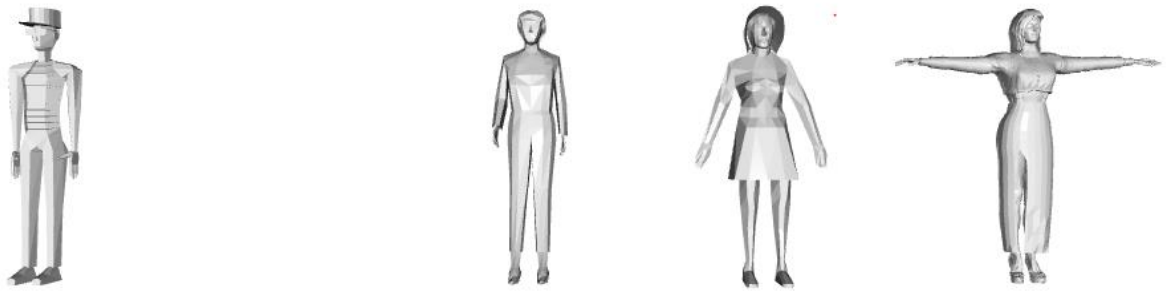

Figure 2: UI output of the script 'knn.py' showing the query shape (left) and k=3 neighbors in feature space (right)

Exercise 1: Use the mini database to compute the k (say, 5) most similar shapes to the humanoid shape, using KNN. Display them in order of similarity. Then use your own distance function (implemented in the previous assignment step) to compute, for the same query shape, the k most similar shapes. Display them. Are the results identical to the KNN ones? If not, which of the two sets of retrieved shapes do you find better? Where from do you think the differences come?

T-SNE

In order to visualize how the feature vectors are distributed in the feature space, we can reduce the feature space's dimensionality to 2, and then plot the points corresponding to the feature vectors. The dimensionality reduction can be done with t-SNE.

The script '`tsne.py`' applies t-SNE to the mini database using the library scikit-learn. The code generates an embedded version of the 'features_total' matrix (which contains the complete feature vector of each shape in the mini database), in which the dimensionality of the feature space is reduced from 68 to 2 dimensions. The embedded features are then plotted using matplotlib.

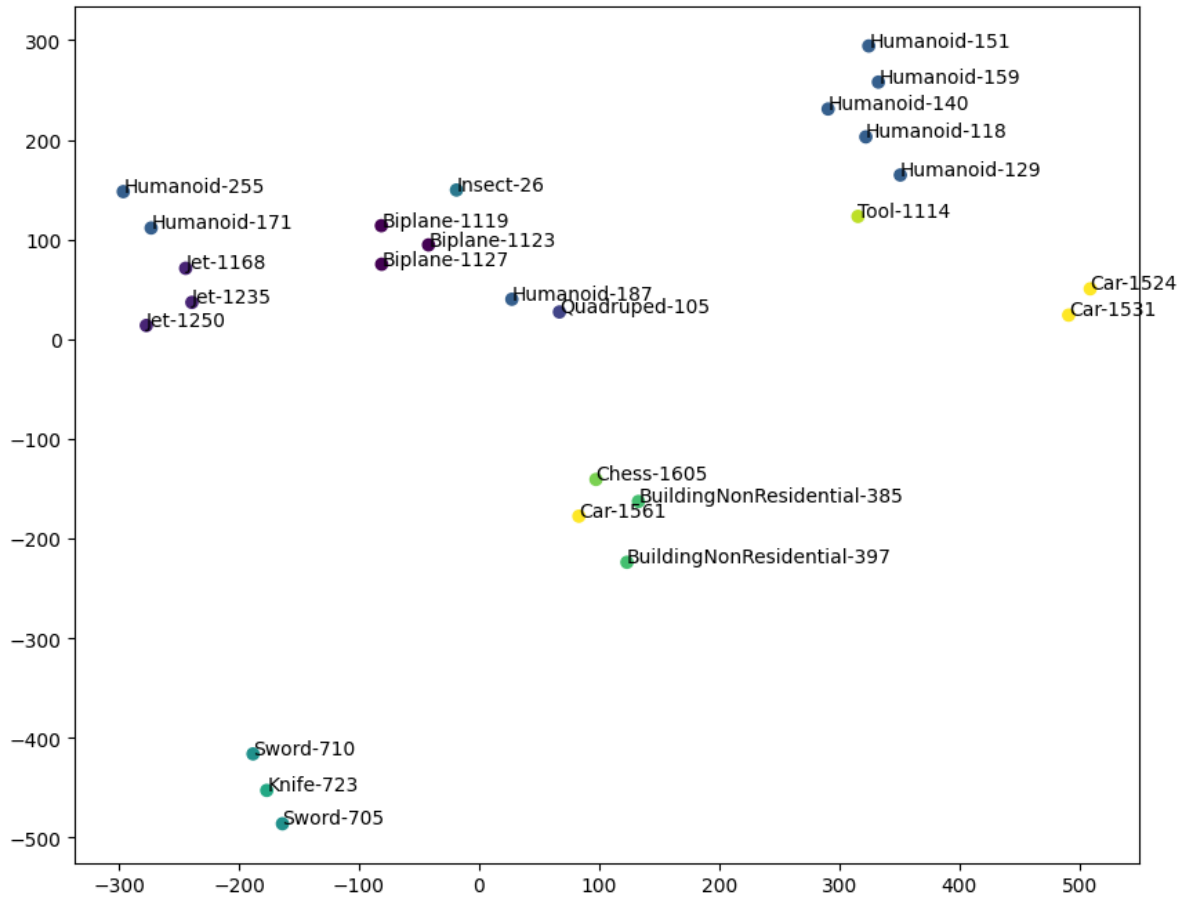

Figure 3: t-SNE plot generated by the 'tsne.py' script, showing clusters of similar shapes (in terms of shape features, not necessarily in terms of classes, although this is often the case as well).

An important parameter of t-SNE is the perplexity. This parameter is (loosely) "*a guess about the number of close neighbors each point has*" (How to Use t-SNE Effectively (distill.pub)).

The value of this parameter can have a major influence on the distribution of embedded feature vectors.

Exercise 2: Try changing the perplexity setting of the TSNE constructor. What effects do you see? And what do you think is the best setting?

Exercise 3: Try visualizing some of the shapes that are close neighbors in the t-SNE plot but belong to different classes (for example using the provided script '`visualize_class_shapes.py`'). Can you think of an explanation for why they are so close together in (embedded) feature space despite their semantic difference?