DiSH

Detection and Visualization of Subspace Cluster Hierarchies

Documentation

Gruppe 1
Daan Scheepens
Johannes Stangl
Sophie Pingitzer
Lukas Feierl

1. Introduction 2

2. Overview of the algorithm 3

3. Hierarchical Subspace Clustering Basics 4

4. Subspace-preference-based Distances 4

5. Main Algorithm 5

6. Extracting Clusters 6

7. Results and Discussion 8

1. Introduction

DiSH (Detecting Subspace cluster Hierarchies) is a subspace clustering algorithm proposed in 2007[1]. It tries to detect patterns in the data using a density-based approach similar to OPTICS[2]. One of the special features of DiSH is that it is able to detect clusters of significantly different dimensionalities in the data. Plus, it also provides a way of assigning a hierarchy between the clusters. For example, DiSH is able to detect the four clusters visualized in Fig.1 and can deduce that the 1-dimensional cluster C is a part of both 2-dimensional clusters A and B, too.

One of the drawbacks of the clustering algorithm is that it is a axis-parallel subspace algorithm, which means that it performs only well on data with clusters that are parallel to the feature-dimensions (like in the example).

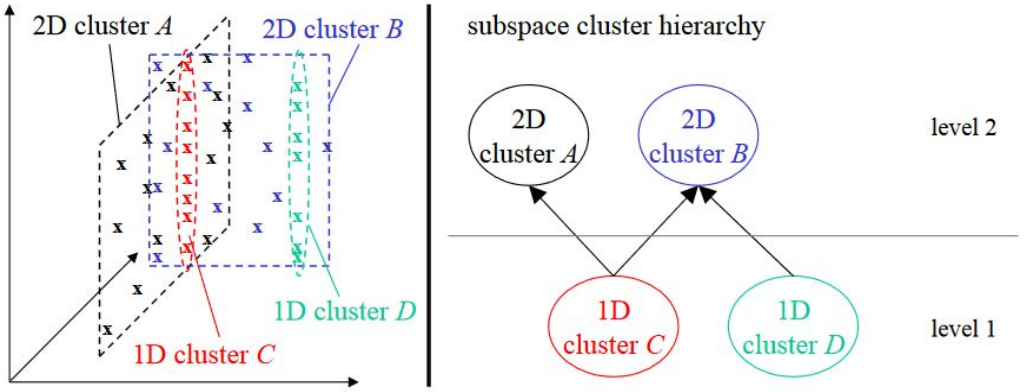

Fig.1: Schematic drawing of two 2D-clusters (A, B) and two 1D clusters (C, D) on the left side. The hierarchy of the clusters is displayed on the right.

[1] Achert, E., Böhm, C., Kriegler, H.P., Müller-Gorman, I., Zimek, A.: Detection and Visualization of Subspace Cluster Hierachies. In: Proc. DASFAA (2007).

[2] Ankerst, M., Breunig, M.M., Kriegel, H.P., Sander, J.: OPTICS: Ordering points to identify the clustering structure. In: Proc. SIGMOD (1999).

2. Overview of the algorithm

Algorithm 1 DiSH

Input: *data*, ϵ, μ
Output: *cluster_list*

preference_vectors ← **get_preference_vectors()**
priority_queue ← **get_pq()**
cluster_list ← **extract_clusters()**

In the first step, the algorithm assigns a **preference vector** to each datapoint. This Boolean vector indicates which features are relevant for spanning a cluster. It looks at the neighborhood of the considered point and checks if enough points are inside the neighbourhood (projected onto each feature-dimension). If so, the datapoint may participate in a cluster projected onto this feature. By **best-first-search**, the most promising of the features are selected for the best subspace and the preference vector for the considered point is set accordingly.

Next, these "local" cluster-preferences are used to measure the similarity between points and combine them to clusters: Starting from one arbitrary point of the data, the **subspace-distance** is computed to all other points, which is a measure of how closely-related one point is to another, in terms of being in the same cluster. From now on, the algorithms "walks" through the dataset, always selecting the nearest point in reference to the subspace-distance (or, so-to-speak, to the cluster-similarity). In each step of the walk, the subspace-distance gets updated such that the point "closest" to all the previously visited points is used for the next step.

The order induced by this walk can then be used for **extracting** the clusters, because the walk was performed in a way that each subsequent point is as close as possible to the previous in terms of subspace-distance. A new cluster can hence be detected if there is a "jump" during the walk where either the preference vector changes from one point to the next one or if there is no existing cluster "near" enough to explain the data.

3. Hierarchical Subspace Clustering Basics

All clustering methods use some measure of distance or closeness. The main idea of subspace clustering is that the distances are considered in projections onto subspaces rather than in the entire feature space.
In this algorithm, the projection is initially done for all one-dimensional axis-parallel subspaces, i.e. for each feature, and is then successively done for subspaces of larger dimensionality.

In the lower dimensional projections, the number of nearest neighbours is considered in order to determine subspaces of low variance. In particular, this algorithm uses the **Euclidean distance of the projected points** (`DIST()`) to determine their ε**-neighbourhoods in these projections** (`get_neighbors()`). Subspaces of a point are of low variance when the point has many neighbours within some radius ε in the subspace, i.e. the cardinality of the projected ε-neighbourhood around a point *o* is at least μ:

$$| N_{\varepsilon}^{\{a_i\}}(o) = \{x \mid DIST^{\{a_i\}}(o,x) \leq \varepsilon\} | \geq \mu$$

The one-dimensional subspaces fulfilling this criterion are candidates for constructing the **best subspace** for a given point. This is done in `get_best_subspace()` using a best-first search where subspaces are combined as long as the ε-neighbourhood in the resulting projection remains larger than μ. The best subspace is thus the largest combination of candidate subspaces for which the ε-neighbourhood is still larger than μ. The term 'best-first search' refers to the strategy of always adding the one-dimensional subspace in which the ε-neighbourhood is largest.

From this, for each point a **preference vector** with a length equal to the overall number of features can be constructed. The value is 1 if the corresponding feature is in the best subspace of the point and 0 otherwise. The 0-entries thus describe the high-variance dimensions and the number of 0-entries is called the subspace dimensionality of the point. In our implementation the preference vectors of all points are calculated using the function `get_preference_vectors()`.

Algorithm 2 get_preference_vectors()

```
Input: data, ϵ, μ
Output: preference_vectors
for each point ∈ data do
    preference_vector ← False // Each entry is False
    best_subspaces ← get_best_subspace()
    preference_vector[best_subspaces] ← True
return preference_vectors
```

Algorithm 3 get_best_subspace()

```
Input: point, data, ϵ, μ
Output: best_subspace
for each feature do
    neighbors ← get_neighbors(point, feature) // list of neighbors
    candidate_features ← ∅
    if neighbors.count ≥ μ then
        candidate_features ← feature

best_subspace ← ∅
for each feature ∈ candidate_features do // best-first search
    best_feature ← neighbors.count.argmax()
    proposed_feature_space ← best_subspace + best_feature
    neighbors ← get_neighbors(point, proposed_feature_space)
    if neighbors.count ≥ μ then
        best_subspace ← proposed_feature_space
return best_subspace
```

4. Subspace-preference-based Distances

The main idea of the DiSH algorithm is the definition of a so called **subspace distance** that assigns small values if two points *(p,q)* are in a common low-dimensional subspace cluster and high values if two points are in a common high-dimensional subspace cluster (or if they are not in a subspace cluster at all). Subspace clusters with small subspace distances are embedded within clusters with higher subspace distances.

In order to compute the subspace distances of any two points *(p,q)* using the previously obtained preference vectors *w(p)*, *w(q)* we first implement a function `DIST_projected()` that returns the Euclidean distances between the points *(p,q)* projected on a lower dimensional subspace according to their preference vectors. This projected distance makes up a part of the first of the two measures (d_1, d_2) that define the subspace distance. The projected distance is also used when building the subspace clustering graph for visualizing the hierarchies of the found subspace clusters.

We implement the function `get_subspace_distance()` in order to determine the subspace distance for a given point and all the other points in our dataset according to the criteria defined for the notion of a subspace distance. The subspace distance between two points *p, q* consists of two parts d_1 and d_2, where $d_1 = \lambda(p,q) + \Delta(p,q)$ and $\lambda(p,q)$ is defined as the subspace dimensionality in the combined subspace of *p* and *q,* having subspace preference vector *w(p,q).* $\Delta(p,q)$ is defined as*:*

$$\Delta(p,q) = \begin{cases} 1 & \text{if } (w(p,q) = w(p) \vee w(p,q) = w(q)) \wedge \text{DIST}^{S(w(p,q))}(p,q) > 2\varepsilon \\ 0 & \text{else,} \end{cases}$$

where $DIST^{S(w(p,q))}(p,q)$ is given by the projected distance `DIST_projected(p,q)` as implemented in our code. The subspace distance consists of the two parts d_1 and d_2 in order to define a sorting relation (details can be found in the paper) which is of importance for the function `get_reachability_distance()`.

Every algorithm based on grouping in a bottom up fashion by combining clusters can suffer from the ‘single linkage effect’ when clusters end up becoming long and thin or separate clusters are undesirably linked. In order to avoid this the **subspace reachability** is defined to measure the similarity of two points *p* and *q*. The subspace reachability is defined as:

$$REACHDIST_{\mu}(p,\ q)\ =\ max\,(SDIST(p,r),\ SDIST(p,q))$$

where *μ* is the minimum number of points in a cluster and *r* is the *μ* - nearest neighbor of *p* (w.r.t the subspace distance).

5. Main Algorithm

The main algorithm that we utilize, named `dish()`, performs the following steps: First, all preference-vectors of all data points are calculated using `get_preference_vectors()` and from that a priority queue (pq) is set up using `get_pq()` which returns a matrix object that holds all data points sorted according to reachability distance. This serves to give us the proper way to walk through our data points to extract clusters. Namely, new clusters are created using the joint subspace preference vector of the current point p and the previous point q in the priority queue, which only gives meaningful clusters if p and q are nearest neighbours in the sense of reachability. The clusters are extracted from the priority queue using `extract_cluster()`.

In our implementation, the priority queue pq is a matrix with the following 3 columns: data point index, reachability-distance r_1 and reachability-distance r_2. In the function `get_pq()` the following is done. We initialize $pq_{current}$ with all reachability distances set to infinity (NaN). We then go through all points, by index, and calculate $r_{1,new}$ and $r_{2,new}$ from this point to all other points. Once we have these distances, we call $pq_{new} \leftarrow$ `update_pq`($pq_{current}, r_{1,new}, r_{2,new}$). This compares all reachability distances in our current $pq_{current}$ matrix with the newly calculated ($r_{1,new}, r_{2,new}$) pair. It updates each r_1 entry of $pq_{current}$ by taking *min*($r_{1,current}, r_{1,new}$) and updates each r_2 entry by taking the r_2 value of whichever r_1 value was chosen. If $r_{1,current} == r_{1,new}$, the smaller r_2 value of the two is chosen.
Before we move on to the next point, pq_{new} is re-ordered in ascending order according to r_1 and then to r_2, for all not-yet-visited points. This is done so that the next index that we choose becomes that of the nearest, not-yet-visited point to our previous point ('nearest' in the sense of reachability). In this way we hop from one point to its nearest, not-yet-visited neighbour.
We then move on to the next nearest point, by index, and do the same. The $r_{1,new}$ and $r_{2,new}$ values of this point, to all other points, now contest with the current entries of the $pq_{current}$ matrix and again the $pq_{current}$ matrix is updated according to the choices explained above. In our implementation the pq matrix is again sorted according index before `update_pq()` is called: this may seem redundant, but it is done because r_1 and r_2 are also sorted according to index and so the comparison here is performed more easily.

Algorithm 4 get_pq()

Input: *data, preference_vectors,* ϵ, μ
Output: *cluster_order*

$pq \leftarrow \infty$ // Each distance is set to ∞
for each *point* **do**
  $d_1, d_2 \leftarrow$ **get_subspace_distance**($point, preference\_vector$)
  $r_1, r_2 \leftarrow$ **get_reachability_distance**(d_1, d_2)
  $pq \leftarrow$ **update_pq**(pq, r_1, r_2) // reorder according to reachability distance
  $pq \leftarrow$ **pq.sort()** // sort according to r_1, then r_2
return $cluster\_order \leftarrow pq$

It is important that we hop from neighbour to neighbour in the priority queue, as explained, because of the way that points are added to clusters. This refers to lines 352-357 in the code. When a new cluster is created, it is created with the subspace preference vector of the pair of the current point p and the previous point q, as its cluster preference vector, i.e.

$$w_{cluster} = w(p, q) = w(p) \wedge w(q)$$

which is only sensible if p and q are close in the sense of reachability. If this were not the case, all sorts of clusters would be created with subspace preference vectors of pairs of points that are nowhere near each other. This is explained in more detail in the following chapter. Points are then successively added to existing clusters if the subspace preference vector between it and the previous point is the same as the cluster preference vector of an existing cluster AND the projected distance (`DIST_projected()`) to the cluster center is less than 2ε (lines 344-350). If neither of these two criteria are met, a new cluster is created with the given subspace preference vector as cluster preference vector. In this manner a list of clusters is created (`cluster_list`) which is the returned as the output of `dish()`.

6. Extracting Clusters

After the last step of the main algorithm one gets a *pq*-matrix sorted by the order in which the data points were visited throughout the walk. Because this order is based on the reachability distance of each point to the previous ones, points of the same cluster are always subsequent to another. The only exception is when a subcluster of lower dimensionality (hence lower d_1) is found in between another higher dimensional cluster.
One can identify the clusters by looking at two criteria - if both criteria are satisfied for a cluster *C* then the considered point is added to this cluster. If no existing cluster is found, a new cluster is formed and the point is added.

1. The joint preference vector *w(p,q)* of one point *p* and its predecessor *q* matches with the preference vector of the cluster *w(C)*.
$$w(p,q) = w(C)$$

2. The position of the point *p* is less than 2ε away from the center of the cluster in terms of the distance with respect to the preference vector *w(p,q)*.
$$DIST^{w(C)}(p,\ C.center) \leq 2\varepsilon$$

Take for example the points *p*, *q* and *z* from **Fig.2**, which are subsequent points according to the pq-matrix. According to the figure, let's assume that *p* and *q* are in the same cluster and *z* is in another one:
If *p* were the first point in the *pq*-matrix, both criteria wouldn't hold because there would be no cluster yet. Hence, *p* would form a new cluster with $w(C) \equiv w(p)$.
The second point *q* is then the most similar point to *p* according to the *pq*-matrix and is considered next. Because they are in the same cluster, their $w(p,q)$ is the same and $w(p,q) = w(C)$. Thus, the first criterion is satisfied. The second criterion is also satisfied, because point *p* and *q* are very close to each other.
Point *z* is next in the priority queue, as it also has the same preference vector as *p* and *q* (with the specific choice of $\mu = 5$ and $\varepsilon = 0.5$). Hence criterion 1 is fulfilled. However, it does not suffice for criterion two, as point *z* is clearly too far away from the blue cluster to belong to it. Thus point *z* belongs to a new (noise) cluster.
Finally, all points inside the green cluster have preference vector $w(C) = (1,\ 0)$ and form a new cluster.

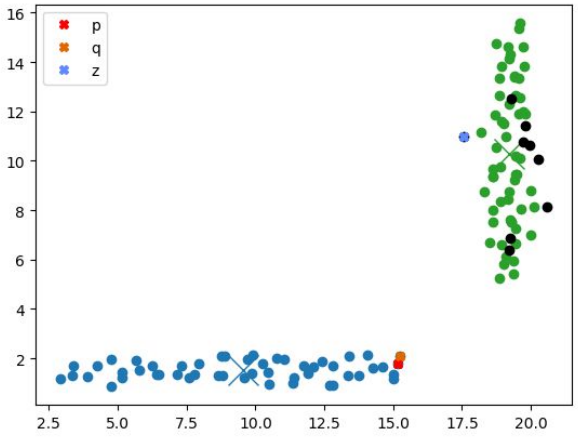

Fig. 2: Dataset containing two clusters. Three example points p, q and z are highlighted where p and q belong to the same cluster while point z has a different preference vector and is too far away.

With this example in mind, there are just two more things to consider:
Firstly, one would think that each cluster is finished in the walk before the next one is entered. However, for example in **Fig.1**, the walk may start in cluster *A*, then find the lower-dimensional cluster *C* and continue with that one when its close enough (as d_1 is smaller based on λ). When cluster *C* is finished, cluster *A* will be continued again. Thus, one has to check all clusters in the **extract_clusters** function, not only the most recently created one. Secondly, after the assignment of each point to a cluster, the ones with less than μ points should be considered as noise points.

Algorithm 5 extract_cluster()

```
Input: cluster_order co
Output: cluster_list
cluster_list ← ∅
for each object ∈ co do
    w_o ← object.preference_vector
    w_p ← object.previous().preference_vector // previous object in co
    w_op ← w_o ∧ w_p
    for each cluster do
        if (w_op == w_c) ∧ (DIST_projected(object, cluster.center) ≤ 2ε) then
            cluster.append(object)
        else
            cluster_list.append(new cluster)
return cluster_list
```

7. Results

For evaluating our implementation of DiSH we tried to run DiSH on the Mouse dataset from ELKI https://elki-project.github.io/datasets/ and compared them to the ELKI-implementation of DiSH.
We found that choosing ε within the domain $\varepsilon \in [0.10,\ 0.14]$ and μ within the domain $\mu \in [25, 40]$ returns very closely the three true clusters of the dataset. The plots in **Fig.3** show on the left the three clusters with the true labels and on the right the three clusters found by DiSH. The three clusters are colored in red, green and blue and noise in black. The cluster centers are plotted as crosses in the right figure as well. As can be seen, there is very satisfying agreement between the two plots.

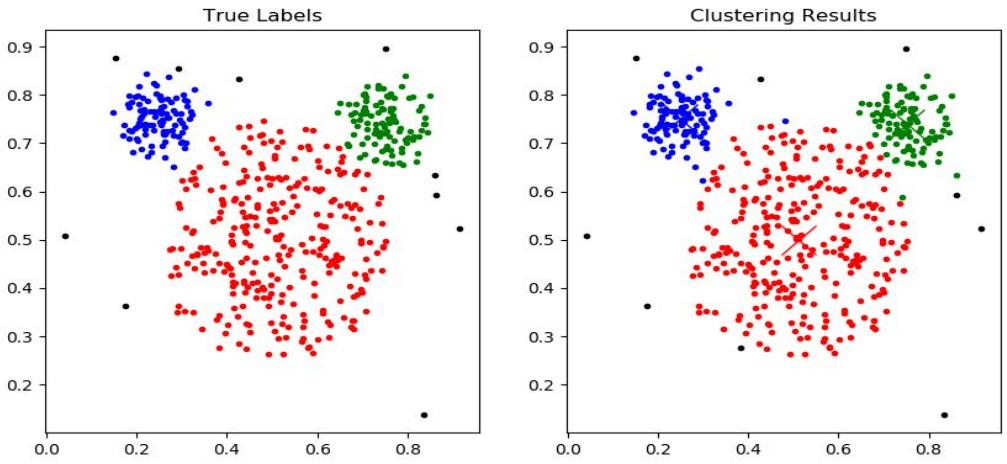

Fig.3 True clusters (left) and clusters obtained by DiSH with $\mu = 25$ and $\varepsilon = 0.13$ (right).

To analyse the performance quantitatively we use the **mutual information score** (MIS) as well as the **adjusted mutual information score** (AMIS) of the clustering result. The MIS is an often used measure of similarity between two clusterings of some data which is based on the amount of information that can be obtained about one random variable from observing the other random variable. The AMIS takes into account the fact that mutual information is usually higher for clusterings with a large number of clusters, regardless of whether the clusters actually share more information or not. For the result presented in **Fig.3**, a MIS of 0.945 and a AMIS of 0.910 are obtained, which are nice results.
Compared to the ELKI implementation, however, there are some differences with the results of our implementation. While the clusters look more or less the same with higher parameter values $\mu = 40,\ \varepsilon = 0.1$ (see **Fig.4** left and middle) the clusters for our optimal parameters $\mu = 25,\ \varepsilon = 0.13$ differ considerably (compare **Fig.3** and **Fig.4**). We assume that this is the case because the algorithm probably starts from a different data point than in our implementation, which will have quite an influence on the final result because of the different walk through the data and thus the order in which clusters are created. That is, the ELKI implementation seems to start at the “ears” of the mouse dataset, assigning a greater portion of the “body” to the blue and yellow “ear”-clusters, as compared to our implementation which starts at the “body”.

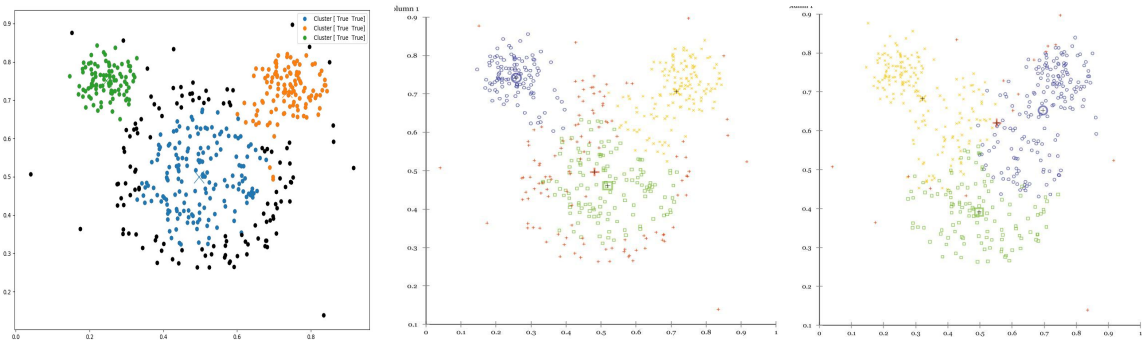

Fig.4 Comparison for DiSH implementation results (left) with ELKI (middle) for $\mu = 40$ and $\varepsilon = 0.1$ .and the ELKI result for $\mu = 25$ and $\varepsilon = 0.13$ (right).

8. Discussion

Although very good results were obtained using the parameters as mentioned above, the choice of μ and ε is not at all trivial and requires either very good a priori knowledge of the dataset or some optimization of the parameters (grid-search) against some performance measure such as the above-mentioned mutual information score.

The different behaviour of our implementation compared to ELKI is bothersome. However, running DiSH on various other datasets always showed at least similar results for ELKI and our implementation, such that it can be assumed that the essential parts of DiSH are performed adequately. Hence, either different starting positions or additional implementation details not included in the paper might be the reason for the differences.

As mentioned in the overview, DiSH is less successful if applied to data where clusters are present in arbitrary orientations, as can be seen when applied to non-axis-parallel data (see **Fig.5**). With high values for μ and ε, only rough portions of the diagonal lines were detected. One part of the right diagonal contained too few objects and was even labeled as “noise”, while a lot of noise objects were assigned to clusters.
One could change μ and ε to very small values to get rid of the noise near the lines, but this naturally creates a lot of small clusters in the process (see **Fig.6**). Here we can see the issue that DiSH often generates a large amount of clusters. This quickly becomes hard to interpret, even with hierarchies. However, even though this result looks messy at first, the lines and noise would be quite perfectly separated, if only there were an algorithm after the cluster extraction which combines clusters that are near to each other (w.r.t. $DIST^{w(c)}$) while having the same preference vectors and similar density. This could be a way to improve the expressiveness of the clustering result (but we did not get to test this in our implementation).

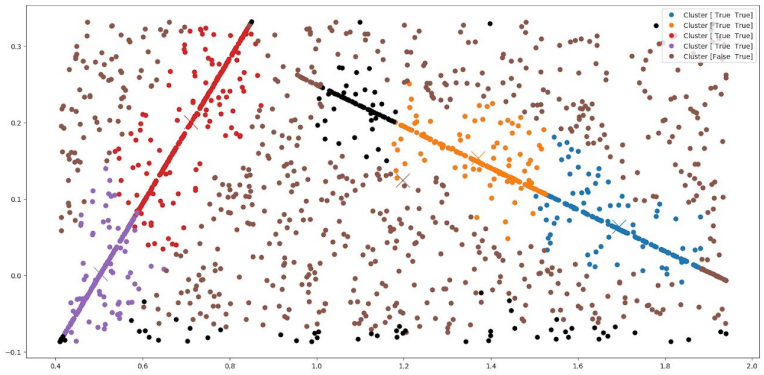

Fig. 5: Clustering results with DiSH using $\mu = 90$ and $\varepsilon = 0.1$ on a dataset containing noise and two arbitrary oriented lines.

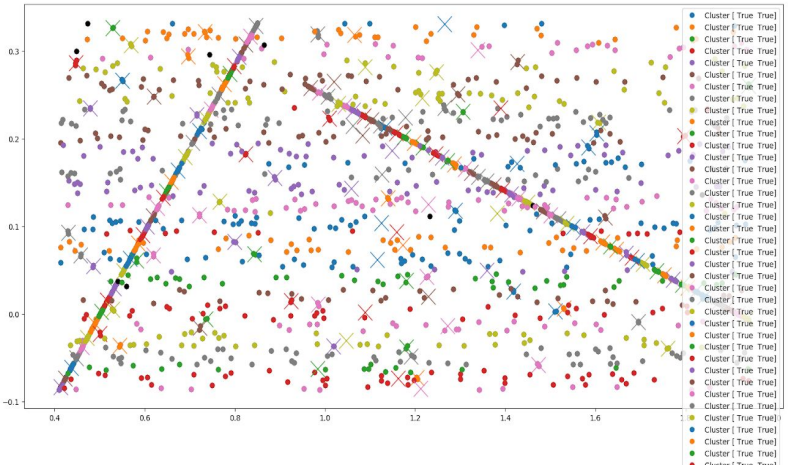

Fig. 6: Clustering results with DiSH using $\mu = 2$ and $\varepsilon = 0.005$ on a dataset containing noise and two arbitrary oriented lines.

Moreover, we found that noise points are sometimes found in places where the preference should be clear based on the preference-vectors of the surrounding points. Unless this is an error in our implementation, it could be considered to be improved in the best subspace generation. This could be done for example by checking the preference vectors in the local neighborhood of points and reassigning outliers (i.e all other preference vectors but one belong are the same).

Finally, we should mention that although we have implemented the calculation of the hierarchies of the clusters in the function `build_hirarchy()`, which is the final part of DiSH, we did not find a good way to display the hierarchies in the kind of tree diagrams that are presented in the paper. Therefore we present no results on this part, although it is acknowledged to be an important last part of the algorithm. The implemented function works, however, and returns a hierarchy in agreement with ELKI.