Data Mining
Programming Assignment 1: Correlation Clustering

Sonja Biedermann Thilo Hannes Fryen
Chan Dat Dennis Huyen David Turner

November 18, 2018

Contents

1 ORCLUS

We decided to implement the ORCLUS algorithm. ORCLUS is a spin on k-means which utilizes eigensystems to adapt the distance measure to non axis parallel correlation clusters.

We picked this algorithm because the paper is easy to read and contains good pseudo code which is of utmost importance when trying to actually implement the described algorithm.

Furthermore, since k-means was the first clustering algorithm we studied, it will forever have a special Gaussian-shaped spot in our heart—which makes ORCLUS, as a spiritual successor, a natural choice.

1.1 Description

The algorithm proceeds in rounds in which the dimensionality of the clusters and the number of clusters is gradually reduced. The rate at which this happens should not be too quickly, to which end the authors propose two parameters α and β and a relation between these two, by which one can be calculated from the other. They suggest an α of 0.5, which we've also adopted.

In each round, the algorithm undertakes an assignment step which is the same as in k-means. Every point is assigned to the closest centroid. However, as a next step, the eigenvectors associated with the smallest spread are computed on a per cluster basis—the rationale behind this is that those vectors define a subspace in which the points cluster well, i.e. have a low contribution to the overall cluster energy. The cardinality of this vector set dictates the dimensionality of the subspace to which the points are projected. The points are then projected into the subspace defined by these vectors, which modifies their distances to each other.

To reduce the amounts of clusters—the algorithm starts with more seeds than requested by the user, we've chosen to start with 5 times as many as the authors offer no suggestions—a merging step is performed next. Although quite lengthy, this operation is pretty simple. The objective is to find pairs of clusters which can be merged such that the overall cluster energy stays low. For this, all pairs of clusters are examined, their centroid and energy computed and then the cluster-pairs with the lowest energy are picked to be merged, being careful to update all relevant data structures after each merge.

These three steps are repeated until the desired dimensionality and number of clusters are reached. After a final assignment step the clustering is returned.

1.2 Pseudocode

The core algorithm is defined in Algorithm 1. Algorithms 2, 3 and 4 cover the assignment, subspace determination, and merging of clusters.

1.3 Implementation

We chose to implement the algorithm in Python 3. The implementation is rather straight forward and only depends on NumPy and some functionality from Python's standard library.

Notably we use `numpy.linalg.eigh` to decompose the cluster matrices into eigenvalues and eigenvectors. We know PCA could also be used but online literature[1] suggests that the LAPACK routines utilized in NumPy's implementation of `eigh` perform a tiny sliver better than SVD, which we expect to be used in `sklearn.decomposition.PCA`. However, this probably would make no practical difference whatsoever.

The initial seeds are distributed using the kmeans++ initialization strategy, although we have also implemented a completely random initialization strategy. If the initial seed count is chosen to be high enough, this strategy actually seems to be beneficial, as it returns

[1] `https://stackoverflow.com/questions/50358310/how-does-numpy-linalg-eigh-vs-numpy-linalg-svd`

Algorithm 1 ORCLUS

```
function ORCLUS(Database: DB, # of Clusters: k, # of Dimensions: l)
    ▷ $C_i$ ⇒ current cluster $i$
    ▷ $\varepsilon_i$ ⇒ set of vectors defining subspace for cluster $C_i$
    ▷ $k_c$ ⇒ current # of seeds
    ▷ $l_c$ ⇒ current dimensionality
    ▷ $S = \{s_1, s_2, ...s_{k_c}\}$ ⇒ current set of seeds
    $k_c = k_0$; $l_c = d$;    ▷ $k_0$ ⇒ initial # of seeds    ▷ $d$ ⇒ dimension of the data
    Pick $k_0 > k$ points from $DB$ an denote by $S$;    ▷ e.g. randomly or K-means++
    for all $i \in \{1...k_c\}$ do $\varepsilon_i = D$;    ▷ initially, $\varepsilon_i$ is the original axis-system
    $\alpha = 0.5$; $\beta = e^{-\log(l_c/l)\log(1/\alpha)/\log(k_c/k)}$;
    while $k_c > k$ do
        ▷ partition the database by assigning each object to its closest seed
        $s_1...s_{k_c}$, $C_1...C_{k_c}$ = ASSIGN($DB$, $S$, $\varepsilon$);
        ▷ determine current subspace associated with each cluster
        for all $i \in \{1...k_c\}$ do $\varepsilon_i$ = FINDVECTORS($C_i$, $l_c$);
        ▷ reduce number of seeds and dimensionality associated with each seed
        $k_{new}$ = MAX($k$, $k_c * \alpha$); $l_{new}$ = MAX($l$, $l_c * \beta$);
        $s_1...s_{k_{new}}$, $C_1...C_{k_{new}}$, $\varepsilon_1...\varepsilon_{k_{new}}$ = MERGE($C_1...C_{k_c}$, $k_{new}$, $l_{new}$);
        $k_c = k_{new}$; $l_c = l_{new}$;
    $s_1...s_{k_c}$, $C_1...C_{k_c}$ = ASSIGN($DB$, $S$, $\varepsilon$);
    return ($C_1...C_{k_c}$);
```

Algorithm 2 Assignment step

```
function ASSIGN($DB$, $s_1...s_{k_c}$, $\varepsilon_1...\varepsilon_{k_c}$)
    for all $i \in \{1...k_c\}$ do $C_i = \emptyset$;
    for all $p \in DB$ do
        ▷ distance of point $p$ to $s_i$ in subspace $\varepsilon_i$
        for all $i \in \{1...k_c\}$ do PDIST($p$, $s_i$, $\varepsilon_i$);
        Determine the seed $s_i$ with the least value of PDIST($p$, $s_i$, $\varepsilon_i$) and add $p$ to $C_i$;
    for all $i \in \{1...k_c\}$ do $s_i = \bar{X}(C_i)$;    ▷ set $s_i$ to new cluster center
    return ($s_1...s_{k_c}$, $C_1...C_{k_c}$)
```

Algorithm 3 Procedure for determining cluster subspaces

```
function FINDVECTORS(Cluster of Points: C, Dimensionality of Projection: q)
    Determine the $d * d$ covariance matrix $M$ for $C$;
    Determine the eigenvectors of matrix $M$;
    $\varepsilon$ = Set of eigenvectors corresponding to smallest $q$ eigenvalues;
    return ($\varepsilon$)
```

Algorithm 4 Procedure for merging clusters

```
function MERGE($C_1 ... C_{k_c}$, $k_{new}$, $l_{new}$)
    for all $i, j \in \{1 ... k_c\}, i < j$ do
        ▷ define $\varepsilon'_{ij}$ by eigenvectors for $l_{new}$ smallest eigenvalues
        $\varepsilon'_{ij}$ = FINDVECTORS($C_i \cup C_j$, $l_{new}$);
        $s'_{ij} = \bar{X}(C_i \cup C_j)$;                                   ▷ Centroid of $C_i \cup C_j$
        ▷ compute projected energy of $C_i \cup C_j$ in subspace $\varepsilon'_{ij}$
        $r_{ij}$ = CLUSTERENERGY($C_i \cup C_j$, $\varepsilon'_{ij}$);
    while $k_c > k_{new}$ do
        for all $i, j \in \{1 ... k_c\}, i < j$ do Find smallest value of $r_{i'j'}$;
        ▷ merge the corresponding clusters $C_{i'}$ and $C_{j'}$
        $s_{i'} = s'_{i'j'}$; $C_{i'} = C_{i'} \cup C_{j'}$; $\varepsilon_{i'} = \varepsilon'_{i'j'}$;
        Discard $s_{j'}$ and $C_{j'}$;
        Renumber the seeds/clusters indexed larger than $j'$ by subtracting 1;
        for all $i, j \geq j'$ do Renumber the values of $s'_{ij}$, $\varepsilon_{ij}$, $r_{ij}$ correspondingly;
        ▷ recompute $r_{i'j}$ pairwise for new cluster $i'$
        for all $j \neq i' \in \{1 ... k_c - 1\}$ do
            $\varepsilon'_{i'j}$ = FINDVECTORS($C_{i'} \cup C_j$, $l_{new}$);
            $s'_{i'j} = \bar{X}(C_{i'} \cup C_j)$;
            $r_{i'j}$ = CLUSTERENERGY($C_{i'} \cup C_j$, $\varepsilon'_{i'j}$);
        $k_c = k_c - 1$;
    return ($s_1 ... s_{k_{new}}$, $C_1 ... C_{k_{new}}$, $\varepsilon_1 ... \varepsilon_{k_{new}}$)
```

good partitions and is also faster. Thus we also included a random seed picking startegy, although we didn't use it during the evaluation.

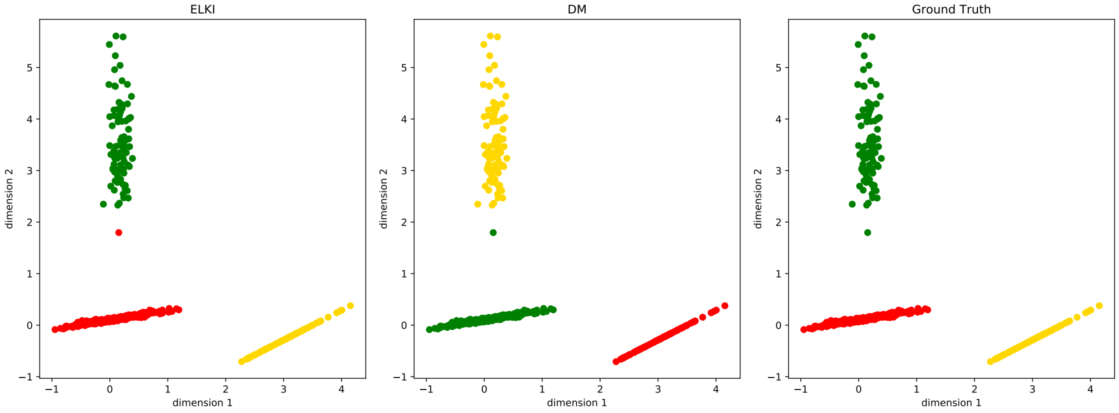


Figure 1: Results on the `test` dataset

2 Evaluation

We will be comparing ourselves to the ELKI implementation on 4 synthetic datasets using the NMI as scoring method. The datasets were generated using ELKI's data generator. We used flat Gaussian distributions as correlation clusters and added noise points to disturb the algorithm.

2.1 Methodology

The evaluation should be fully reproducible. File `eval.sh` contains the calls made to our test scripts and ELKI. `summary.rb` is a Ruby script which processes the results, outputs an overview table and also generates a `gnuplot`-ready plot file for plotting a boxplot of the results.

The scripts `show_data.py` can be used to view the datasets contained in `data/`, `elki_results_to_csv.py` consolidates the output of ELKI into one single `.csv` file which can be plotted and compared to our results using the `plot_results.py` script.

We execute the test runs 5 times and note their resulting NMIs as well as the clusterings with the best NMI score. We average using the arithmetic mean.

2.2 Simple dataset

As a simple first dataset we generate a 2-dimensional data set containing 3 clusters that are very clearly separated, one of which is a bit fatter than the stereotypical correlation cluster, and which is axis-parallel. Figure 1 shows the results obtained by our implementation and the result obtained by the implementation contained in ELKI. They are identical—notably misclassifying one single point which would belong to the fat axis-parallel cluster.

2.3 Simple dataset with added noise

We want to torment ORCLUS by adding noise to the previous trivial dataset. It should perform much worse, since ORCLUS has no notion of noise by default (and, although

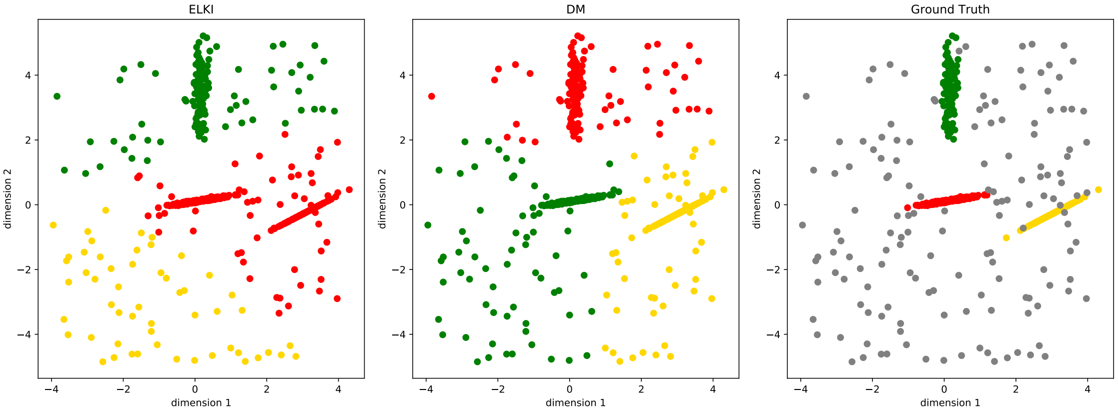


Figure 2: Results for dataset `test_noisy`

the authors recommend a simple scheme for implementing outlier detection, this does not seem to be implemented in ELKI), but it is interesting to see how the two implementation compare on a problematic dataset.

Figure 2 show the results of our implementation and ELKI's implementation as well as the ground truth.

2.4 Clusters that are noise in some subspaces

It seems like a common situation in real-world data that clusters are present in some subspaces, but are very dispersed in other subspaces, where other clusters may reside—thus clusters in some subspaces are noise in others. We would thus expect the algorithm to struggle. Figure 3a depicts the ground truth for this data set.

Figure 3b shows the results for both implementations. Interestingly, our implementation manages to obtain clusterings that look tidier. However, the NMIs are pretty close, as the tidiness of the clustering does not match the actual structure in the data.

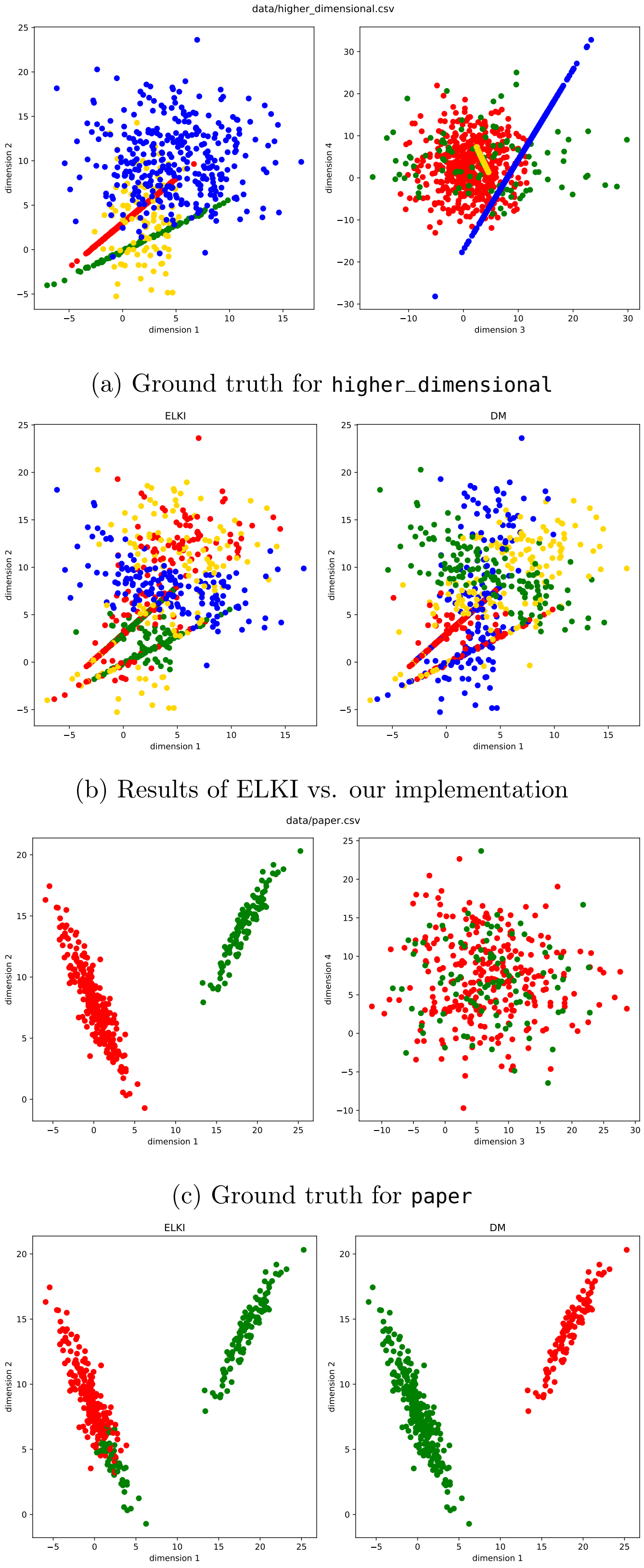
Note that we used more intial seeds for this dataset, as this seems to help a lot here.

2.5 Clusters with different rotations in some subspaces

The final dataset is somewhat based on the example of projected clusters in the paper. This dataset has two rotated clusters in the first two dimensions, and Gaussian noise in the last two dimensions. We expect the algorithm to correctly focus on the leading two dimensions and discard the other two. Figure 3c shows this data set with the correct labels.

Note that this data set has an axis-parallel representation when picking the first and third dimension (i.e. a noise dimension). This subspace seems to be preferred by ELKI's implementation, as misclassified points are often at the border of the correct cluster in this subspace. It could be that these outliers (that are somehow assigned to the wrong cluster, e.g. by badly placed seeds) destroy the PCA result, causing the algorithm to fall into a local minimum it cannot escape from.

Figure 3d shows the results. As this is a very simple data set, we are a bit worried that ELKI does not manage to find this clustering in every try—out of 5 runs, only one run



(a) Ground truth for `higher_dimensional`

(b) Results of ELKI vs. our implementation

(c) Ground truth for `paper`

(d) Results of ELKI vs. our implementation

Figure 3: Results on the `higher_dimensional` and `paper` datasets compared to their ground truth

	DM			ELKI		
dataset	min	max	avg	min	max	avg
test	0.9869	0.9869	**0.9869**	0.5372	0.9869	0.8106
test noisy	0.539	0.6596	**0.6352**	0.5372	0.6585	0.6342
higher dimensional	0.1892	0.3368	**0.2631**	0.1881	0.3561	0.2578
paper	0.6466	1.0	**0.8863**	0.0046	1.0	0.5594

Table 1: Overview over NMI scores

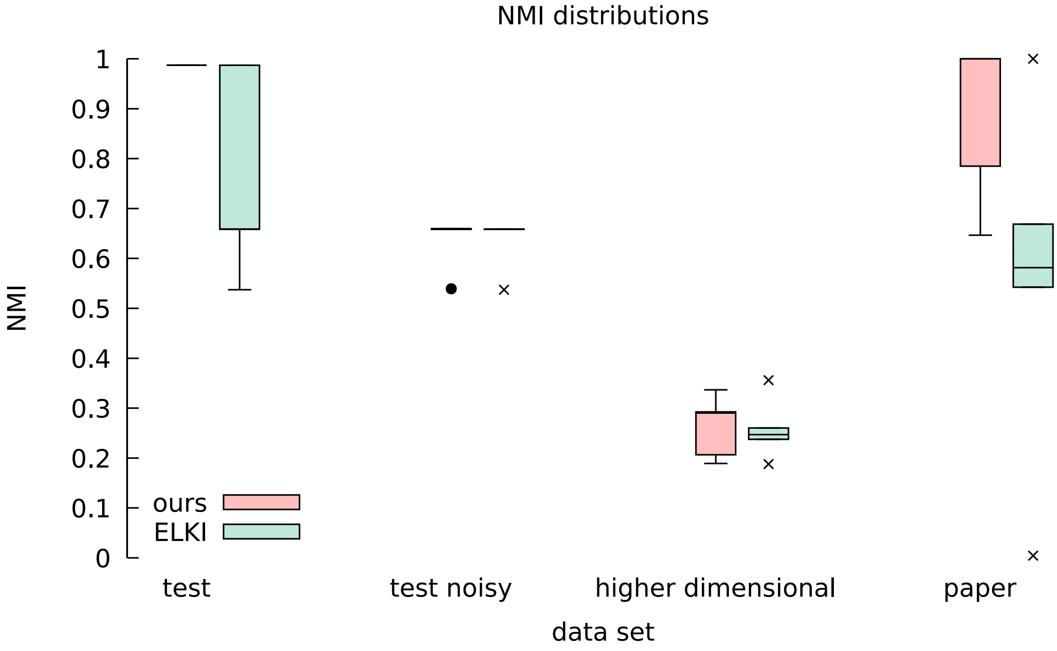


Figure 4: Boxplot illustrating stability

managed to reach a NMI of 1. One run was even only able to reach a NMI of 0.005, which is catastrophically small.

2.6 Summary

Table 1 shows the results of our evaluation. The minimum, maximum and average over the 5 runs are shown split between our implementation (DM) and the implementation available in ELKI. Averages that are better than ELKI's average NMI are written in bold, although note that some of the differences are rather slight.

Interestingly, our implementation appears to have a slight advantage over ELKI's. Notice especially the discrepancy with dataset `paper`, where ELKI's implementation appears extremely unstable, while ours uncovers the cluster structure perfectly 3 out of 5 times, with the two other results being of fairly similar quality. The biggest difference in our implementation is that we use a different initialization strategy, namely kmeans++—this might result in our algorithm having an easier time converging to a good minimum since the cluster centers are already well-separated. However, our implementation performed better even when using a random seed picking strategy—we suspect that perhaps the eigenvalue decomposition is implemented differently, or we are just lucky. We also want to note that we use fewer initial seeds than ELKI, which defaults to $30 \cdot k$.

Figure 4 shows a boxplot of the obtained NMI scores. A more stable implementation would have a shorter box, i.e. less variance in the results. Our implementation appears a bit more stable on the given datasets.