Project3

Anton Holm

2021-03-03

Note: I wrote this project during high fever and a ongoing Covid infection so I apologize in advance that the quality may not be as high as I wanted it to be and if it is hard to follow some parts since it was quite hard to concentrate and think during the project.

LLE

In chosing the right value of K we can have this mindset. When K is very small, each neighbor is quite close to the original point and as such, the reconstruction error (eq 1 in Roweis & Saul) we are trying to minimize is decreasing if we keep adding more neighbors. However, if we start to include too many neighbors, the validity of using euclidean distance is no longer certain and as such, the reconstruction error could increase. Therefor, we don't want to chose K to be too small or too large.

I use the function `calc_k` in the package `lle` in R. Below is how they decide their optimal K according to the reference they leave. However note that in reality, they calculate the residual variance for each value of K, not only those with local/global minima for some reason as the reference writes . Look at the reconstruction error of each K up to some max number of neighbors (I chose 30 since more than 30 should never be a good choice with the number of data we have and we could probably pick an even lower number but I chose this size so I can vizualize it better later). Pick the values of K that resulted in any global or local minima of the reconstruction error and call them $K_1^*, K_2^*, ..., K_{N_S}^*$ where N_S is the number of local and global minimas. Calculate the residual variance $1 - R^2(\hat{D}_M, D_Y)$ (see reference and notes 42 Tenenbaum et al. 2000). Lower residual variance means that high-dimensional data is better represented in the embedding space. So the optimal number of neighbors is the K_i^* for which the residual variance is the lowest. Keep in mind that in this algorithm they use regularisation where they add $Tr(S)/k$ to the grammatrix S in order to prevent any singularities. Therefor, we will end up with a big K but in reality, most of the neighbors will have a very small weight and as such will not influence the results too much, i.e. we will in practice have much fewer neighbors influencing the results than the given K which can be seen in Figure 2. This method takes quite some time since we need to calculate the LLE for each K_i which can take some time since the function solves the eigenvector problem of the $N \times N$ matrix for every K_i but running the algorithm on several cores of the computer increase the speed.

Below I perform the mentioned algorithm to find the optimal K and plot in Figure 1 the number of neighbors against the residual variance. We can see that we have a few contenders but since $K = 13$ or $K = 16$ does not give any major changes to the embedding I decide to proceed with $K = 13$. We can also see in Figure 2 that after $K = 13$ our weights are all 0 which makes sense if we have 13 neighbors. We can see that after 8 the weights tend to drop quite fast and for example the neighbors with the tenth largest weight have quite small weights and does not influence the procedure too much.

```
## Warning: package 'lle' was built under R version 4.0.4
```

```
## Warning: package 'snowfall' was built under R version 4.0.4
## Warning: package 'snow' was built under R version 4.0.4
```

```
## Warning: package 'fpc' was built under R version 4.0.4
```

```
## Warning: package 'dbscan' was built under R version 4.0.4
```

```
## Warning in searchCommandline(parallel, cpus = cpus, type = type, socketHosts =
## socketHosts, : Unknown option on commandline: rmarkdown::render('C:/Users/anton/
## Documents/Unsupervised Learning/Project3.Rmd', encoding
```

```
## R Version:  R version 4.0.3 (2020-10-10)
```

```
## snowfall 1.84-6.1 initialized (using snow 0.4-3): parallel execution on 4 CPUs.
```

```
## Library lle loaded.
```

```
## Library lle loaded in cluster.
```

```
##
## Stopping cluster
```

```
## best k: 16 19 20
```

Figure 1: K versus residual variance

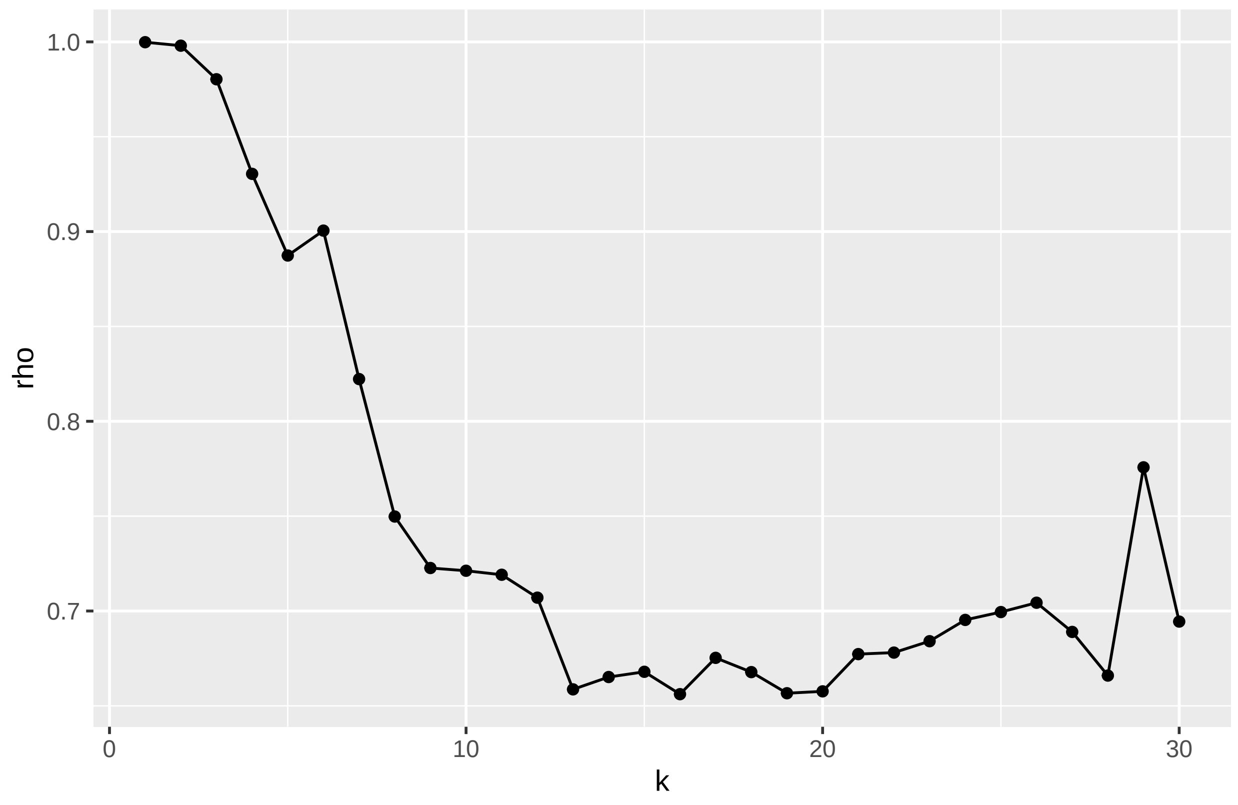
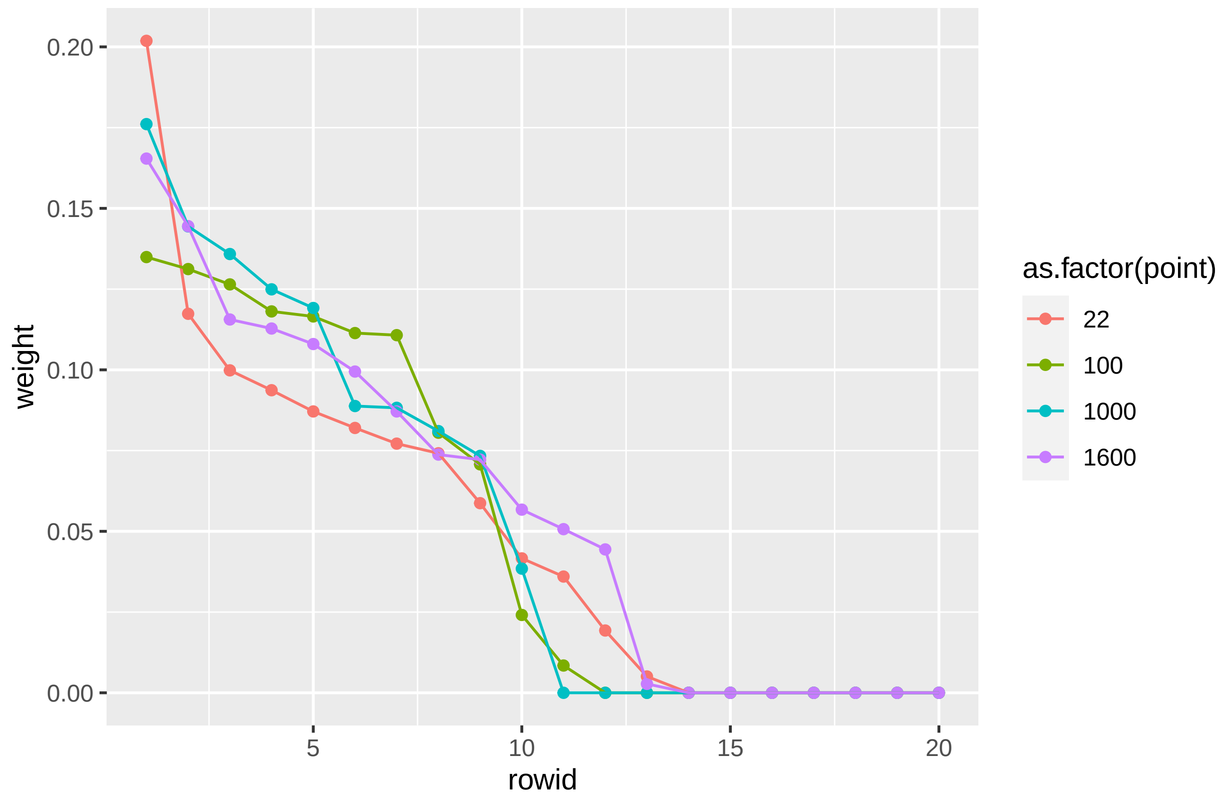
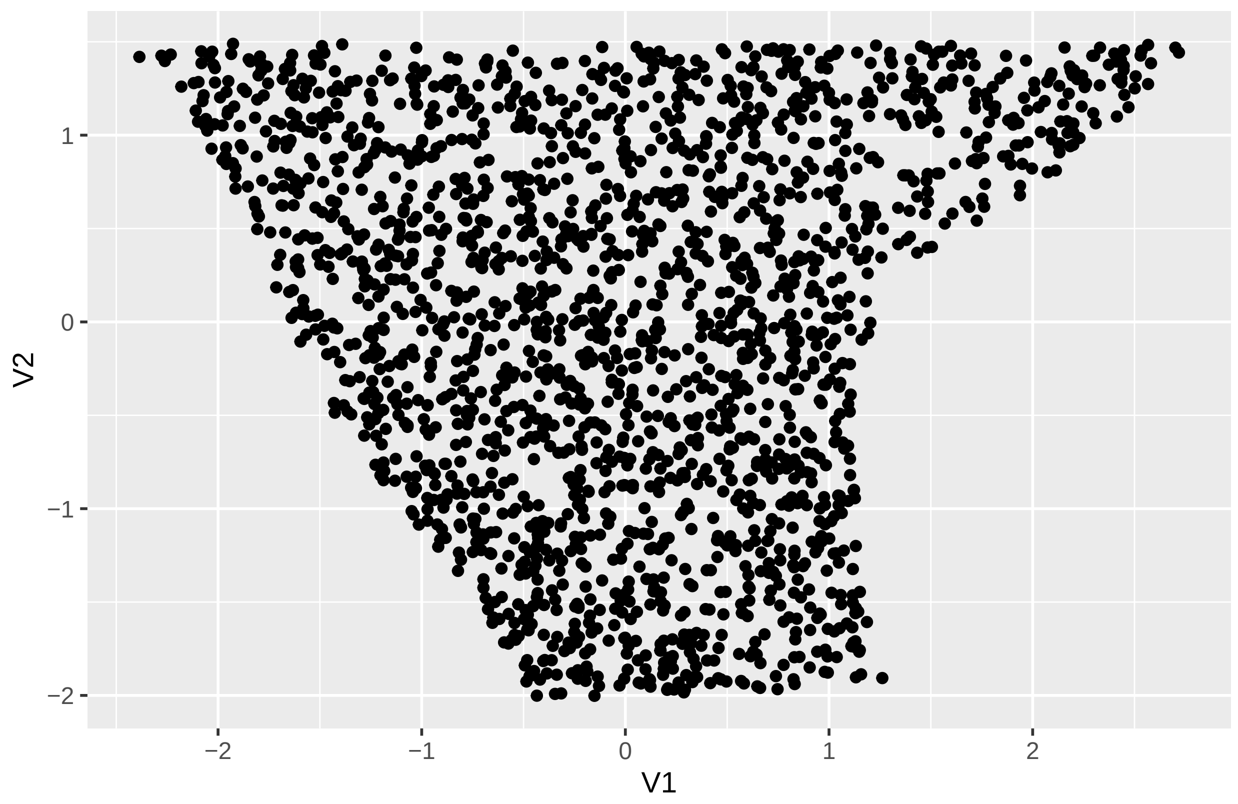


Figure 2: Weights of some datapoints



```
## finding neighbours
## calculating weights
## computing coordinates
```

Figure 3: Swissroll in 2−dim LLE space



In Figure 3 we see the swissroll data visualized in the 2 dimensional LLE space. It would seem like we can easiliy see the four corners of the swiss roll in this new visualization as the four corners of this deformed rectangle. Since there is no patch of a high density of datapoints it would seem like there are no overlaps, i.e. the swissroll is fully unfolded. However, in this visualization euclidean distance is assumed and as such, it is hard to interpret the distance between those points around the fold on the right side of the figure, i.e. where measuring the distance between two points using euclidean distance mean we leave the figure represented by the datapoints. Therefor, at least for the datapoints which ended up on the right side of the projected space, the rank is most likely not well preserved. One of the reasons to why we are not getting a perfect rectangle could be due to the aspect ratio of the swissroll. The length of the swiss roll is much larger than that of the width which makes it difficult to unfold it perfectly.

I think that LLE is more interpretable for people without mathematical background than CTD. For CTD you would have to explain about all the coordinate changes and Laplacian theory. For LLE it is easier to explain that you want to preserve the local structure of the data on smaller patches. In LLE we are able to preserve the local structure better as mentioned but the global structure, i.e. rank or distance between points far away in the original space are not preserved well when using LLE. We can see for example that as mentioned above, the aspect ratio of the swissroll is such that it is much longer than it is wide (if we consider it unfolded). When using CTD embedding we could see this aspect ratio in the projected space. However, when we use LLE, we can see that the unfolded roll is almost as long as it is wide (see figure 3). So the aspect ratio in the LLE space is not kept.

When we pick the neighbors in LLE this is based on euclidean distance. However, here we could use some geodestic distance when chosing the neighbors in order to prevent prossible short circuits. So this is one way to implement geodestic distance into the LLE algorithm.

Part 2

When using the “Fast Search” clustering method by Rodriguez and Laio (2014) we first of all need to decide how we chose neighbors. I will use the epsilon neighborhood recommended in the paper since this method is implemented in the R function. However, we could also use for example a KNN-graph approach. I could not find any theory regarding this but from my own thinking I would propose to chose a number of neighbors such that we do not have problems with short circuits if euclidean distance is used. Then we create a weight matrix with some weight measure, e.g. one over the average euclidean distance of the K neighbors. In this way we get an estimated density of each point. This way we would have real-valued densities instead of having integer valued densities (rho’s) and by so removing the problem of tied densities.

In using the epsilon neighborhood we need to decide on the parameter epsilon. Since I will be using euclidean distance we do not want epsilon to be too big since this could create short circuits. The authors of the paper recommends to choose epsilon such that we have on average $1-2\%$ of the data as neighbors to a point. If this method is the best or not can be discussed. One drawback of doing this could be that it works for some parts of the data but not for others since epsilon is held constant. There is no “fit for all” epsilon to choose. Using epsilon neighborhood could also be problematic if the data is not spherical as in this case. However, since writing an entire function by myself seems like way too much work for just a small project I will still use the method proposed in the paper.

In Figure 4 we see the decision plot from the fast search method. We can see 6 clear points sticking out in the upper right part of the graph and as such, following the analysis of Rodriguez and Laio (2014) we should have 6 clusters.

In Figure 5 we can see the result if we consider points as being corepoints or part of the cluster halos (points considered as noise). Here most of the data has been considered as noise and as such clearly something is not right. In Figure 6 we plot the data considering them all as corepoints and see that the method has problems distinguishing the clusters. There could be several reasons to why this is the case. One of these reasons seem to be that the distance between some of the parts of several shapes is so small that we suffer from some short circuits (see e.g. where the two horse shoes fit together). In Figure 5 we see where the cluster centers are according to the fast search method. We see that they are not centered in the shapes. For example, look at cluster 2 in Figure 5. The data of the upper horseshoe is much closer to the center of the lower horseshoe than most of the data of the lower horseshoe. Using euclidean distance here is therefor not applicable and is probably one of the reasons our results fails. Here we could instead perhaps use some geodestic distance measure instead of euclidean distance and the method would work better. However, we can also see that the fast search method even misses some shapes and as such the method seems faulty in this scenario.

```
## Warning: package 'densityClust' was built under R version 4.0.4
```

Figure 4: Decision

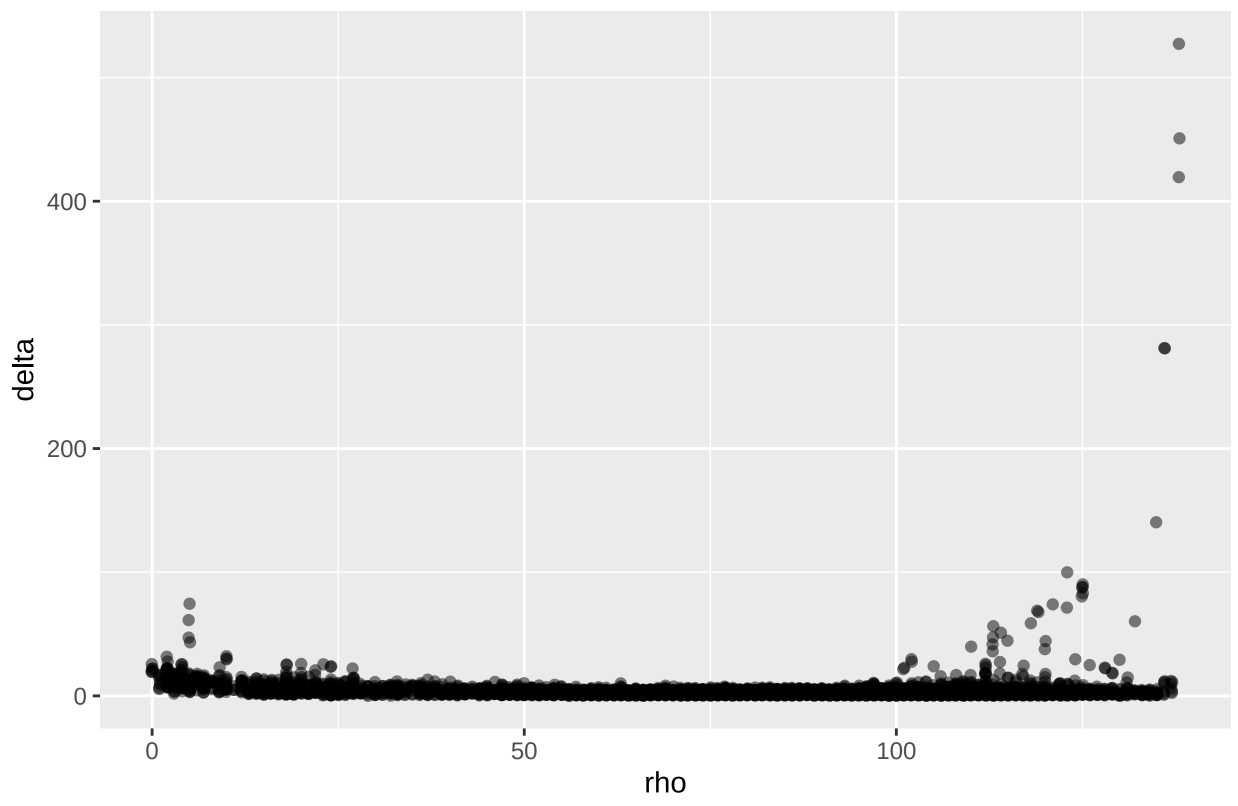


Figure 5: Result of Fast Search considering Halos (cluster 0)

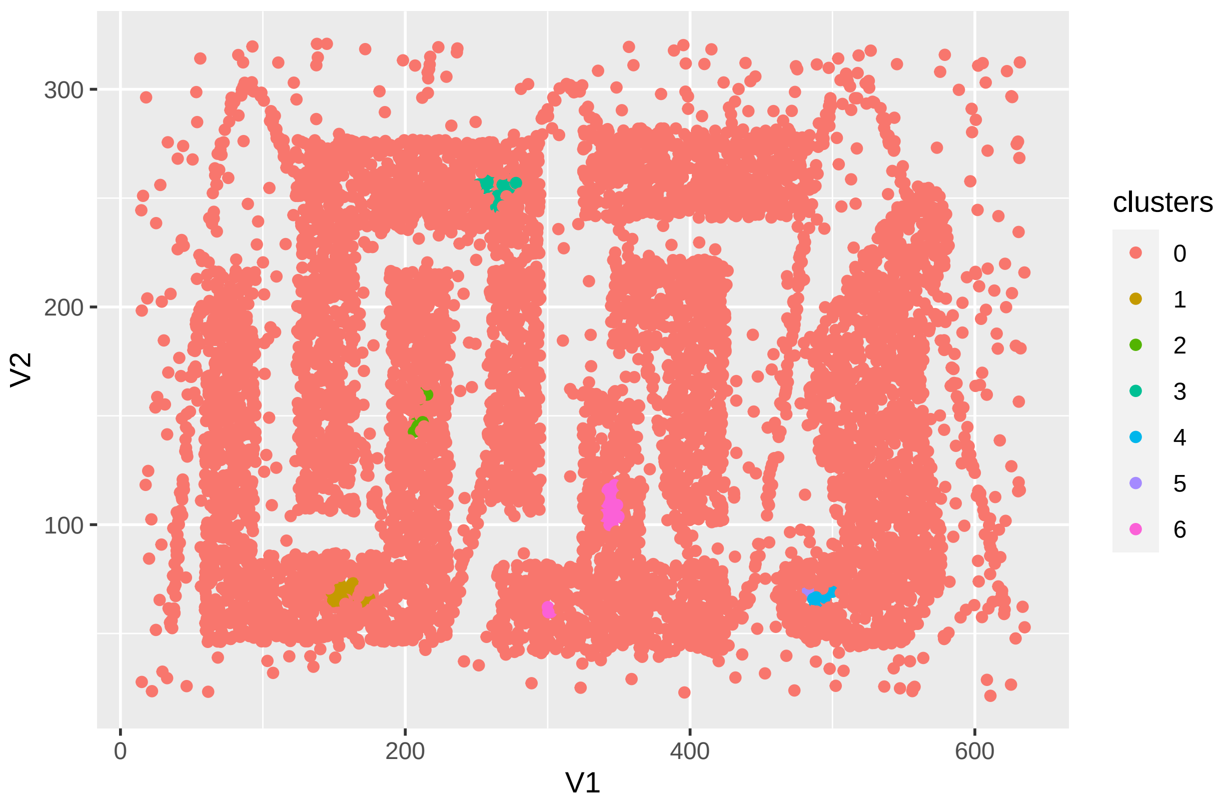
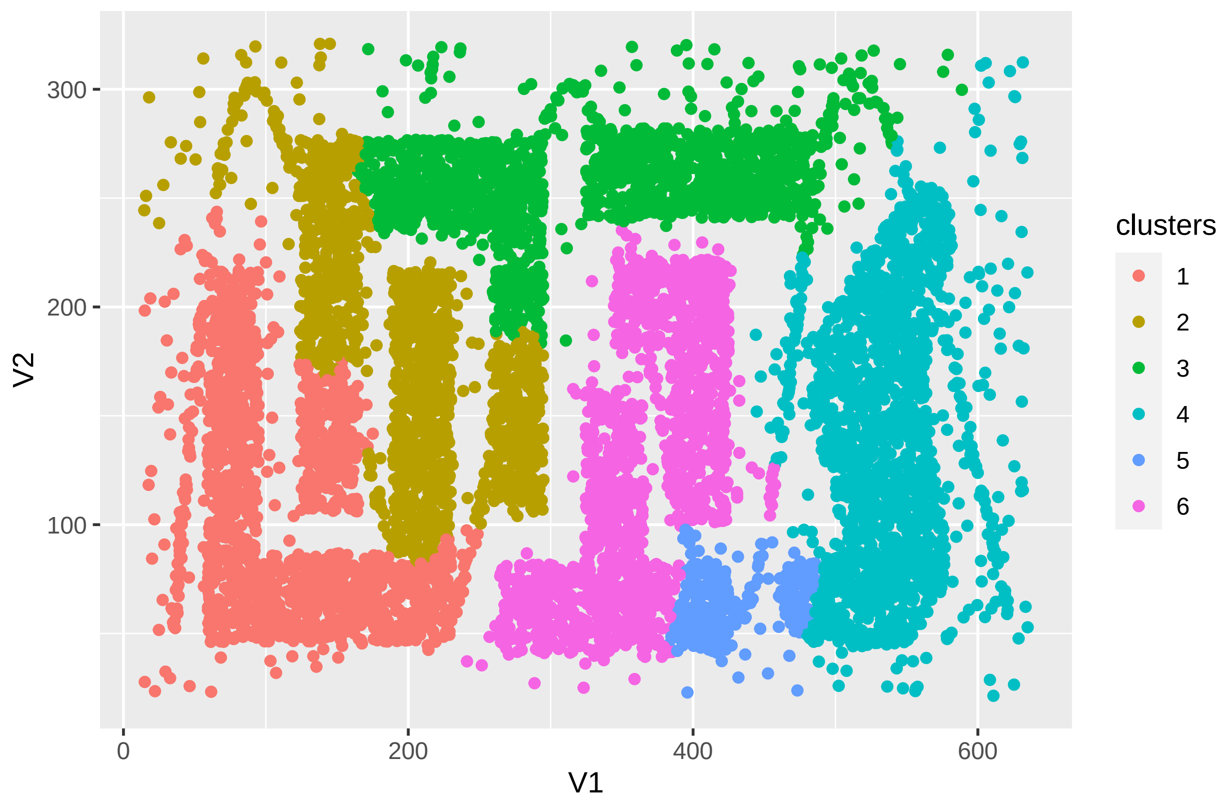


Figure 6: Result of Fast Search not considering Halos



Since the fastsearch method did not perform well I instead turn to DBSCAN and try this method out instead.

DBSCAN

To perform DBSCAN we need to decide the value of two parameters, epsilon for the epsilon neighborhood and MinPts which is used to decide if a point is a corepoint, borderpoint or outlier. According to Ester et al. (1996) the MinPts value should be set to 4. The reasoning is as follows: We want to produce a k-dist graph. This is done by mapping each observation to the value of the distance from the point to its k:th nearest neighbor (its k-distance). Then we sort the values in descending order and plot the index on the x-axis and the distance on the y-axis. We then want to find the "valley" or as we have used in the course "elbow" in this plot.

If we now set MinPts to be equal to this k the following holds: All points with a k-distance smaller than the k-distance of the point we choose based on the "elbow" criteria is a core point, i.e. all points to the right of this point is a core point and assigned a cluster while the points with a larger distance (to the left of the valley point) is considered noise.

Since the k-dist graph does not change much between $k = 4$ and $k > 4$ they propose to keep MinPts at 4 (for 2-dim data) since we require a lot more computational time for larger k. The k-distance of the valleypoint is then chosen as the epsilon value.

In Figure 7 we see the above mentioned 4-dist plot. The elbow can be seen at around the datapoint with the 750:th largest 4-distance with a value of around 5. Therefor I procede with the dbscan function using MinPts = 4 and a epsilon value equal to 5. The result can be seen in Figure 8. Here we see that the 6 geometric shapes are clustered seperately as we wanted. We can also see that this method can not distinguish the sine-curve as its own cluster. This makes sense considring that if we use an epsilon neighborhood for a point on the sine-curve, we will capture both noise and data from other clusters in this neighborhood. The density of the points at each small neighborhood of the sinecurve is also very small and as such many of the points are classified as its own cluster. We can see that by using DBScan, we manage to distinguish the noise from the clusters and sine-curve for the most part. Some of the data present in the 6 clusters actually gets considered as outliers.

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

Figure 7: 4−dist graph of arbitary shape data

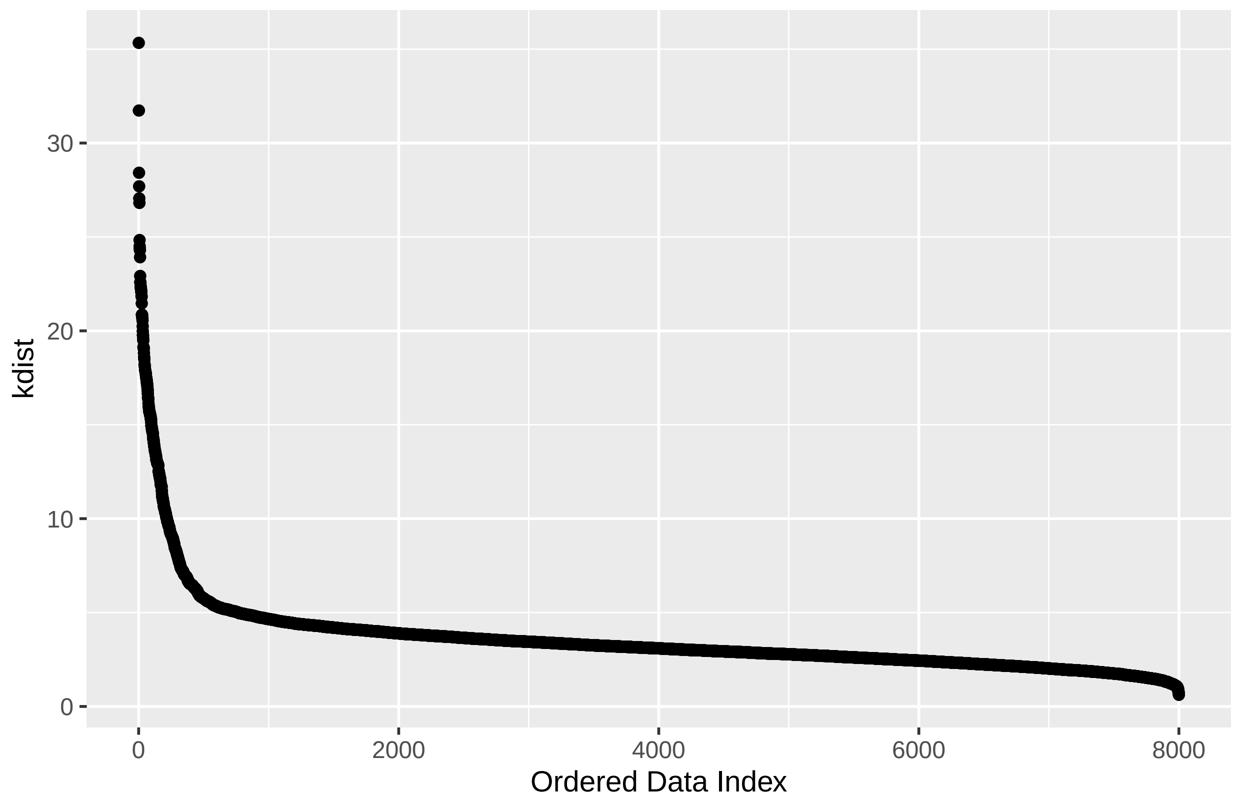
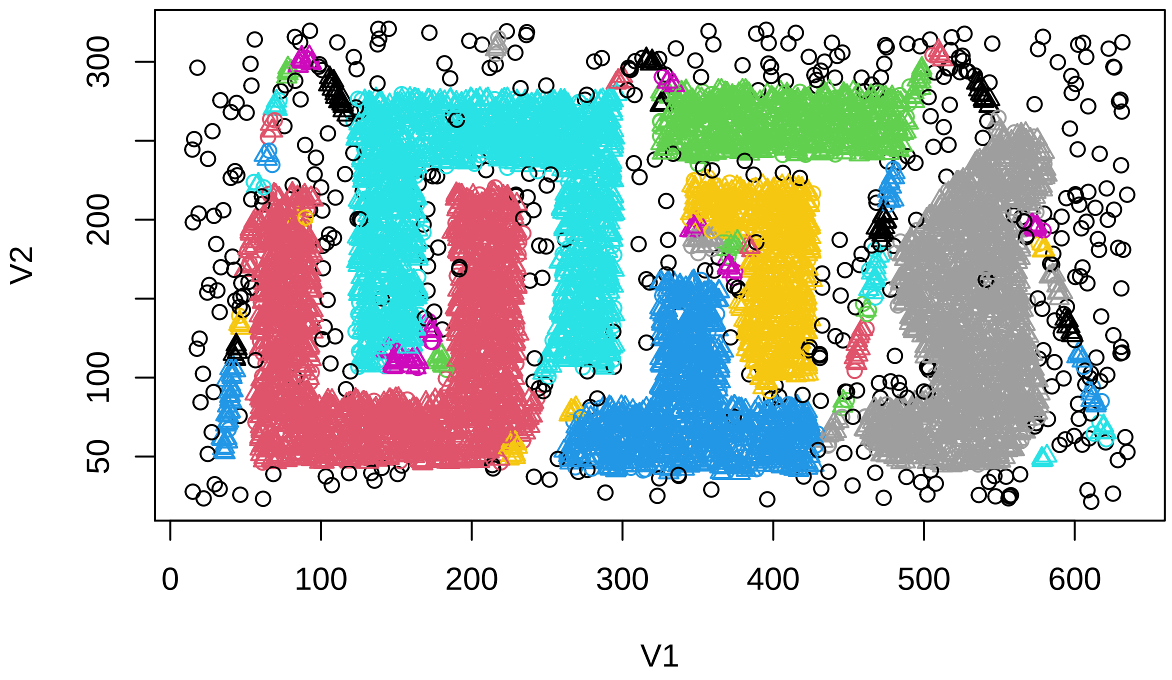


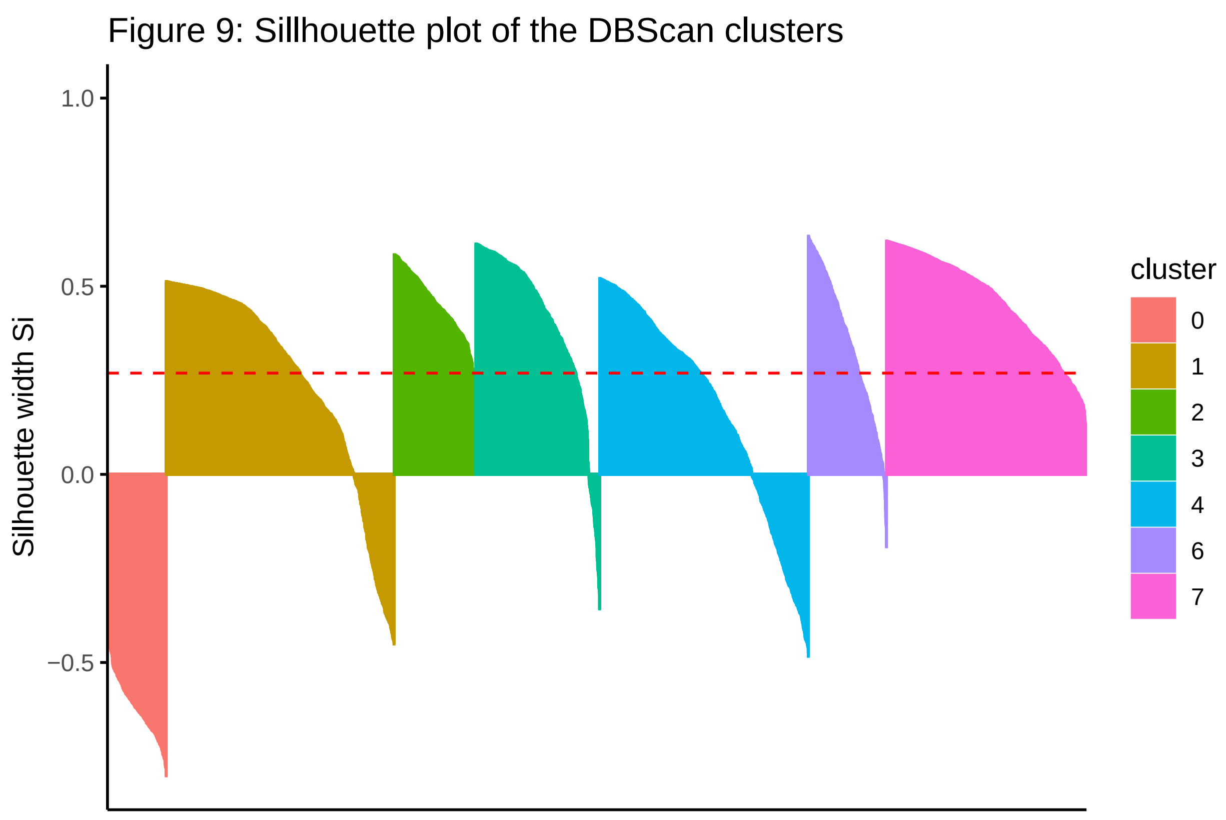
Figure 8: Result of DBScan



Since the sine-curve impacts the number of clusters we get, we end up with around 50 clusters. However, most of them have less than 30 observations in them which is very small compared to our 8000 observations in the dataset and as such I will disregard these clusters in order to make the silhouette plot more interpretable. In the silhouette plot below we see 7 clusters where cluster 0 is the points that got considered as noise.

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## Selecting by count
```



Since it was not possible to change the colors of the sillhouette plot to match that of the cluster colors in the DBScan result these are the clusters represented in the sillhouette plot (see Figure 9). Clusters are Outliers (0), U-shape (1), hyphen shape (2), upsidedown T (3), upsidedown U-shape (4), 7-shape (6), and S-shape (7).

We can see that even though we manage to cluster the distinct shapes well (excluding the sine-curve) in the dataset using DBScan the sillhouette plot does not show us that this is the case. The reason for this is the fact that when calculating the sillhouette coefficient for each observation the euclidean distance is used. This is the reason to why we see that the sillhouette coefficent for many observations in the two U-shapes/horseshoes are negative. These two shapes comes together and therefor the euclidean distance between these two clusters are very small at the "upper-parts" of the U-shapes, i.e. the two vertical bars. Actually the between distance of the right vertical bar of the U-shape cluster and the vertical bars of the upsidedown U-shape is half as large as the distance between the two vertical bars of the U-shape cluster (in the euclidean space). Since the sillhouette coefficent uses two variables, the within distance and the between distance of the closest cluster, obviously the sillhouette coefficent will be missleading in that case. The same reasoning holds for the upsidedown T-shape since it fits well into the 7-shape. Here all points on the right-side and upper part of the upsidedown T-shape lies so entangled together with the 7-shape that the between distance of those point in the T-shape will have a smaller between distance than within (in the euclidean space). Therfor this cluster suffers from many negative sillhouette coefficents. This is also why we see that the S-shape seem to have been clustered well according to the sillhouette plot. The S-shape lies quite alone in the dataspace without any entanglement with other shapes. One way to fix the problem of the sillhouette plot is to use some graphdistance instead, e.g. geodestic distance.

```r
suppressPackageStartupMessages(library(lle))
suppressPackageStartupMessages(library(tidyverse))
suppressPackageStartupMessages(library(factoextra))
suppressPackageStartupMessages(library(fpc))
suppressPackageStartupMessages(library(dbscan))

#load data
swiss_df <- read.table("Swiss_Roll.txt", header = FALSE, sep = " ") %>%
  dplyr::select(V1, V6, V11)
#Calculate residual variance for K = 1,2,...,30
optim_k <- calc_k(as.matrix(swiss_df), m = 2, kmin = 1, kmax = 30, parallel = TRUE, cpus = 4, plotres =
#Plot above
ggplot(optim_k, aes(x = k, y = rho)) +
  geom_point() +
  geom_line() +
  labs(title = "Figure 1: K versus residual variance")
```

```r
#Finds neighbors to all datapoints
neighbors <- find_nn_k(swiss_df, k = 13)
#Calclate all weights of all datapoints
weights <- find_weights(neighbors, swiss_df, m = 2)

#Take a subset of data and chose their 20 highest weights
weights_22 <- tibble(point = 22, weight = weights$wgts[,22]) %>%
  arrange(desc(weight)) %>%
  slice(1:20) %>%
  rowid_to_column()
weights_100 <- tibble(point = 100, weight = weights$wgts[,100]) %>%
  arrange(desc(weight)) %>%
  slice(1:20)%>%
  rowid_to_column()
weights_1000 <- tibble(point = 1000, weight = weights$wgts[,1000]) %>%
  arrange(desc(weight)) %>%
  slice(1:20)%>%
  rowid_to_column()
weights_1600 <- tibble(point = 1600, weight = weights$wgts[,1600]) %>%
  arrange(desc(weight)) %>%
  slice(1:20)%>%
  rowid_to_column()
#Join data
subset_weights <- full_join(weights_22, weights_100, by = c("rowid", "point", "weight")) %>%
  full_join(weights_1000, by = c("rowid", "point", "weight")) %>%
  full_join(weights_1600, by = c("rowid", "point", "weight"))

#Plot top 20 weights
subset_weights %>%
  ggplot(aes(x = rowid, y = weight, col = as.factor(point))) +
  geom_point() +
  geom_line() +
  labs(title = "Figure 2: Weights of some datapoints")
```

```r
#Perform LLE with 13 neighbors and m = 2 since swissroll is a 2-D manifold
lle_obj <- lle(as.matrix(swiss_df), m = 2, k = 13, nnk = TRUE)

as.data.frame(lle_obj$Y) %>%
  ggplot(aes(x = V1, y = V2)) +
  geom_point() +
  labs(title = "Figure 3: Swissroll in 2-dim LLE space")
```

```r
suppressPackageStartupMessages(library(densityClust))

#Load Arbitrary Shape data and calculate distance
arb_df <- read.table("Arbitrary_Shape.txt")
arb_sim <- dist(arb_df, method = "euclidean")

#Perform Fast Search with dc according to Rodriguez and Laio (2014)
densi <- densityClust(arb_sim, dc = 19)

tibble(rho = densi$rho, delta = densi$delta) %>%
  ggplot(aes(x = rho, y = delta)) +
  geom_point(position=position_jitter(h=0.1, w=0.1), alpha = 0.5) +
  labs(title = "Figure 4: Decision ")

#Find Cluster Centers and assign data to clusters
clusters <- findClusters(densi, rho = 125, delta = 100)

#Data using Halo
arb_halo_df <- arb_df %>%
  mutate(Halo = clusters$halo) %>%
  mutate(clusters = ifelse(Halo == TRUE, 0, clusters$clusters)) %>%
  mutate(clusters = as.factor(clusters))
#Data without Halo
arb_clust <- arb_df %>%
  mutate(clusters = as.factor(clusters$clusters))

ggplot(arb_halo_df, aes(x = V1, y = V2, col = clusters)) +
  geom_point() +
  labs(title = "Figure 5: Result of Fast Search considering Halos (cluster 0)")

ggplot(arb_clust, aes(x = V1, y = V2, col = clusters)) +
  geom_point() +
  labs(title = "Figure 6: Result of Fast Search not considering Halos")
```

```r
suppressPackageStartupMessages(library(reshape2))
suppressPackageStartupMessages(library(Rfast))

#Create the k-dist graph
dist_df <- as.matrix(arb_sim) %>% melt(varnames = c("row", "col"))
k_dist <- dist_df %>%
  group_by(row) %>%
  summarise(kdist = Rfast::nth(value, 4, descending = F))

k_dist %>%
  arrange(desc(kdist)) %>%
  ggplot(aes(x = 1:8000, y = kdist)) +
  geom_point() +
  labs(title = "Figure 7: 4-dist graph of arbitary shape data", x = "Ordered Data Index")

#Perform dbscan with parameters according to text
dbobj <- fpc::dbscan(arb_sim, eps = 5, MinPts = 4, method = "dist")
```

```r
plot.dbscan(dbobj, arb_df)
title(main = "Figure 8: Result of DBScan")
```

```r
#Get the biggest clusters
big_clusters <- arb_df %>%
  mutate(cluster = dbobj$cluster) %>%
  group_by(cluster) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  top_n(7) %>%
  select(cluster) %>%
  unlist(use.names = FALSE)

#Filter out all small clusters
arb_cluster_df <- arb_df %>%
  mutate(cluster = dbobj$cluster) %>%
  filter(cluster %in% big_clusters) %>%
  select(V1, V2)
#Distance matrix of the data from large clusters
arb_cluster_sim <- arb_cluster_df %>%
  dist(method = "euclidean")
#Remove small clusters
clusts <- dbobj$cluster %>%
  tibble(cluster = .) %>%
  filter(cluster %in% big_clusters) %>%
  as.matrix()

#Get the sillhouette coefficients and plot the sillhouette plot
sil_coeff <- cluster::silhouette(clusts, dmatrix = as.matrix(arb_cluster_sim))
fviz_silhouette(sil_coeff, ggtheme = theme_classic(), print.summary = FALSE, title = "Figure 9: Sillhou
```