De-noising for Topological Data Analysis

Uriya Jankurazova
Advisors - Prof. Deanna Donohoue, Prof. Scott Corry
Lab Group - Natalie Horgan, Maxim Muter

August 2019
Lawrence University

Currently, most methods of analyzing data involve a statistical approach. While these methods are powerful, many of these methods share similar assumptions and limitations. This work focuses on using a different approach, Topological Data Analysis (TDA). TDA is a suite of methods which focus on analyzing and retrieving the shape of a dataset and then using that shape to obtain valuable information. One TDA technique is Persistent Homology, where the dataset is evaluated for shape under a series of conditions. Shapes which persist over a large range of conditions are identified as key features. One limitation of this technique is that if the data is noisy, retrieving the underlying shape is difficult. As we evaluate the TDA methods for use in geoscience applications, the issue of noise is significant. Therefore, here we have focused on incorporating a pre-processing or de-noising step before analysis of the persistent homology.

Overview

To de-noise the data as preprocess for following TDA Persistent Homology technique, the package `denoiseTDA` was created, which is based on the Mean Shift Algorithm and Kloke's Algorithm. This paper is mostly focused on how you can use the package in the **Tutorial** section. You are welcome to look at how the algorithm was implemented into code, and suggest changes, improvement, and fixes.

More about issues and future work in the **Future Work** section.

Content

……………………..

Tutorial
- Setting Up
- Two Points
- Circle
- Other Shapes
Insight and Future Work
Appendix A | Function List
Appendix B | Code

……………………..

Tutorial

Tutorial how to use package `denoiseTDA`. Set up and generate different data sets and de-noise it.

Look at the Appendix A for function description. Also you can type in console `?+function`. For example

```
?NoisyGen
```

For a complete insight of functions look at the `Appendix B` section.

Setting Up

Installing Pakage `denoiseTDA`

If you don't have the package `denoiseTDA`[1] installed, then you can either load it in the console by entering the code below into it. Or just put it in the R code.

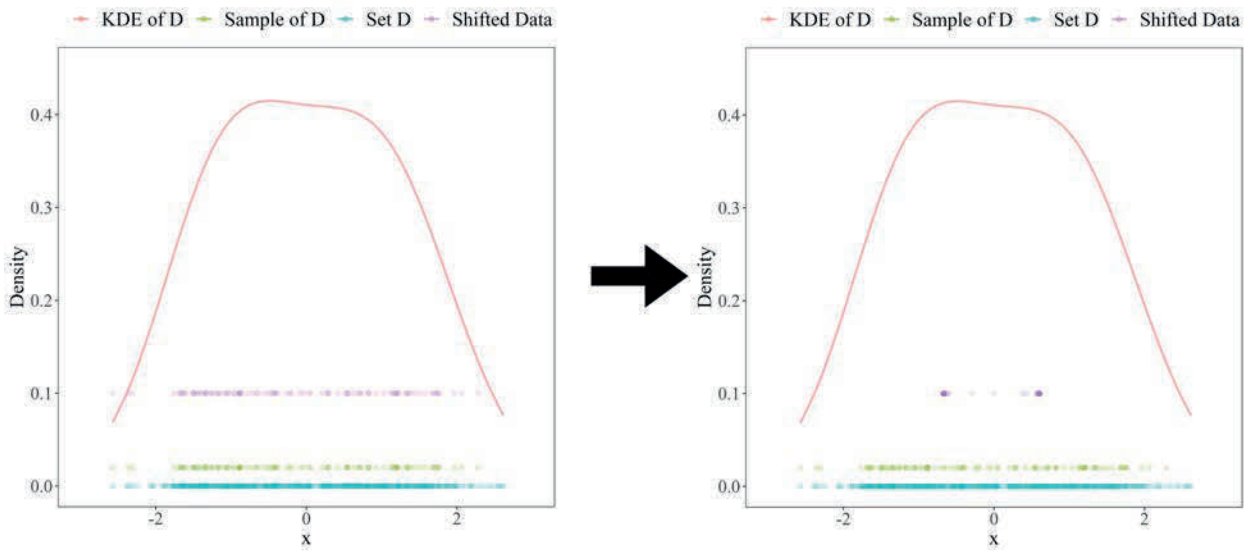
[1] Functions in package `denoiseTDA` are described in the *Appendix A*. At the very end.

```
library(devtools) # A nessesary package to load the the function intall_github()
library(usethis)  # Just a a part of it. Just use it.
install_github("uriyaxd/denoiseTDA") # Finnly we upload package from the github.
```

Loading Packages

First of all lets load nesessary packages.

```
library(denoiseTDA) # the denoising package
library(ggplot2)    # package to plot graphs
```



Two Points

How to use common functions in the R package denoiseTDA, and denoising one-dimensional two-point noisy data set.

I. Creating Toy Data : Two Point Data

What is a Toy Data? Toy Data is the data with the undelying shape that you will be generating, which also has some noise in it. To generate noise will be used function `NoisyGen(Data, var, mean=0)` from `denoiseTDA` package.

1. First let's generate 2 point data in 1D.

```
shape<-rep(c(-1,1), 1000)
```

2. Then to be more random, we will sample that vector, by using function `sample()`.

```
dataRandom<-sample(shape, 500)
```

3. Since the data stored as a vector, therefore we need to transform it to the matrix, by using `matrix()`. So we could apply function `NoisyGen()` from `denoiseTDA`. Which uses matrix input.

```
dataOriginal<-matrix(dataRandom, ncol=1)
```

4. And finally we will apply some noise into it, by using function from `denoiseTDA` package, a `NoisyGen(matrix, var)`[2]. Which generates noise over the matrix. Where `var` is an indicator of how noisy you want the data be (usual values from 0.1 to 1).

[2] **Play** with variable **var**, by setting it to the higher value, and then try to denoise it

```
sigma<-0.5
dataNoisy<- NoisyGen(dataOriginal, var = sigma)
```

Plotting data

Another useful function in the `denoiseTDA` is `MatToData(matrix)`. Which transforms matrix with columns up to 3 into the data frame. Where the 1st column is evaluated as `x` , 2nd as `y` and 3rd as `z`.

```
ggplot()+
  geom_point(MatToData(dataOriginal),
             mapping=aes(x=x, y=0, col="Original Data"),
             alpha=0.1, size=2)+
  geom_jitter(MatToData(dataNoisy),
              mapping=aes(x=x, y=0.025, col="Noisy Data"),
              alpha=0.1, size=2,
              width = 0, height = 0.001)+
  ylim(c(-0.015, 0.05))+
  scale_color_discrete(name="")+
  theme_minimal()+
  theme(axis.title.y = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_blank(),
        legend.position = "top",
        text = element_text(family="serif", size = 16))
```

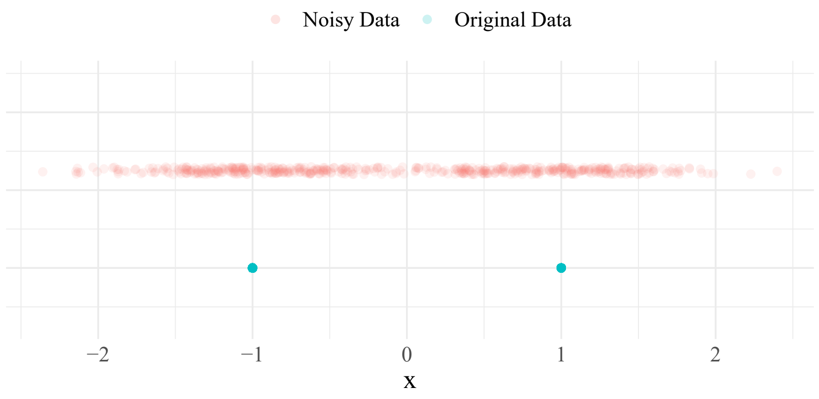


Figure 1: Data Points - the red points are the noisier version of the original points, which are blue.

II. Plotting Kernel Density Estimator

One of important part of the de-noising algorithms is **Kernel Density Estimator (KDE)**. It computes how dense is the data. Which is implemented in the function `KDE(set, sigma)`.

1. Applying the **KDE** over our dataset with the same sigma as the var of data.

```
dataKDE<-KDE(set=dataNoisy, sigma = sigma)
```

2. Plotting **KDE** with the data

```
ggplot()+
  geom_point(MatToData(dataOriginal),
             mapping=aes(x=x, y=0, col="Original Data"),
             alpha=0.1, size=2)+
  geom_jitter(MatToData(dataNoisy),
              mapping=aes(x=x, y=0.05, col="Noisy Data"),
              alpha=0.1, size =2 ,
              width = 0, height = 0.005)+
  geom_line(MatToData(dataKDE),
            mapping= aes(x=x, y=y, col="KDE of Data"))+
  theme_minimal()+
  scale_color_discrete(name="")+
  ylab("Density")+
  theme(legend.position = "top",
        text = element_text(family="serif", size = 16))
```

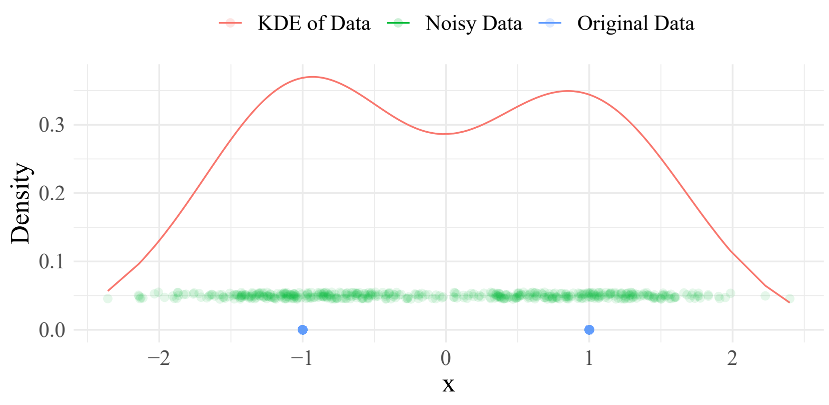


Figure 2: Kernel Density of Noisy Data - the red line indicates how dense is the ‘dataNoisy‘, which is now shown as green points.

III. De-noising Toy Data : Two Point Data

Mean Shift Algorithm

Mean shift algorithm[3] is implemented in the function `MeanShiftAlg( set, sigma, iter, step)`.

[3] Fukunaga, K., & Hostetler, L.D. (1975). The estimation of the gradient of a density function, with applications in pattern recognition. IEEE Trans. Information Theory, 21, 32-40.

1. So lets try to use it on the `dataNoisy`

```
dataShifted<-MeanShiftAlg(set=dataNoisy, sigma = sigma,
                          iter=100, step = 0.035)
```

2. Plotting resulted shifted points

```
ggplot()+
  geom_jitter(MatToData(dataNoisy),
             mapping=aes(x=x, y=0, col="Noisy Data"),
             alpha=0.1, size=2,
             width = 0, height = 0.005)+
  geom_jitter(MatToData(dataShifted),
              mapping=aes(x=x, y=0.05, col="Shifted Data"),
              alpha=0.1, size=2,
              width = 0, height = 0.005)+
  geom_line(MatToData(dataKDE),
            mapping= aes(x=x, y=y, col="KDE of Data"))+
  theme_minimal()+
  scale_color_discrete(name="")+
  ylab("Density")+
  theme(legend.position = "top",
        text = element_text(family="serif", size = 16))
```

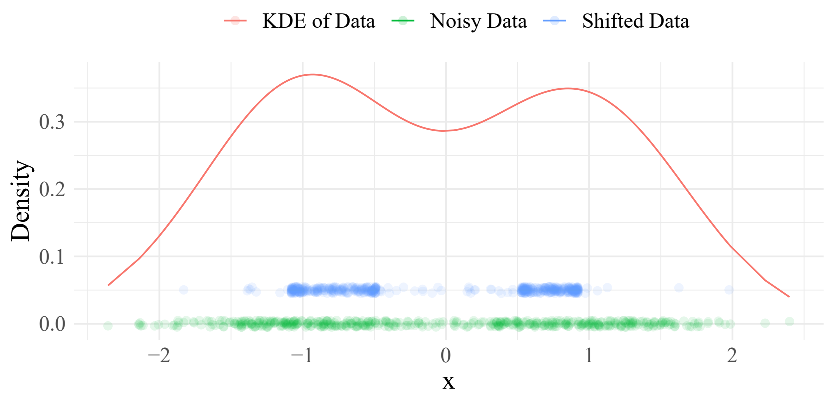


Figure 3: Mean Shift Algorithm over ‘dataNoisy‘- the shifted points are in the blue color, which results in less noisy data.

Kloke's Algorithm

Kloke's Algorithm[4] is implemented in the function `KlokeAlg(sample, set, sigma, omega, iter, step)`[5]

[4] Kloke, Jennifer Novak. (2010) Methods and Applications of Topological Data Analysis. Stanford University.

[5] Play with parameters `step`, `sigma`, `omega`, see how it affects the shifted data.

1. Sampling our matrix

```
dataSample<-MatSample(dataNoisy, N=100)
```

2. Now let's denoise the data by using **Kloke's Algorithm**.

```
dataShifted<-KlokeAlg(sample=dataSample, set=dataNoisy,
                      sigma= sigma, omega = 0.1, iter=100, step=0.7)
```

3. Plotting points

```
ggplot()+
  geom_jitter(MatToData(dataNoisy),
             mapping=aes(x=x, y=0, col="Noisy Data"),
             alpha=0.1, size=2,
             width = 0, height = 0.005)+
  geom_jitter(MatToData(dataShifted),
              mapping=aes(x=x, y=0.05, col="Shifted Data"),
              alpha=0.1, size=2,
              width = 0, height = 0.005)+
  geom_line(MatToData(dataKDE),
            mapping= aes(x=x, y=y, col="KDE of Data"))+
  theme_minimal()+
  scale_color_discrete(name="")+
  ylab("Density")+
  theme(legend.position = "top",
        text = element_text(family="serif", size = 16))
```

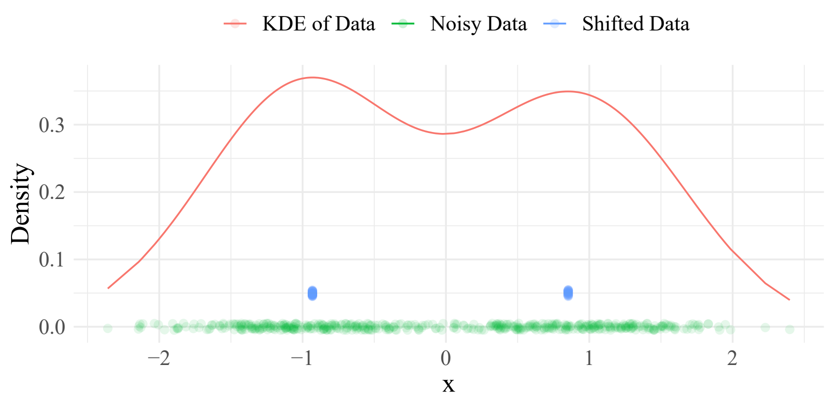
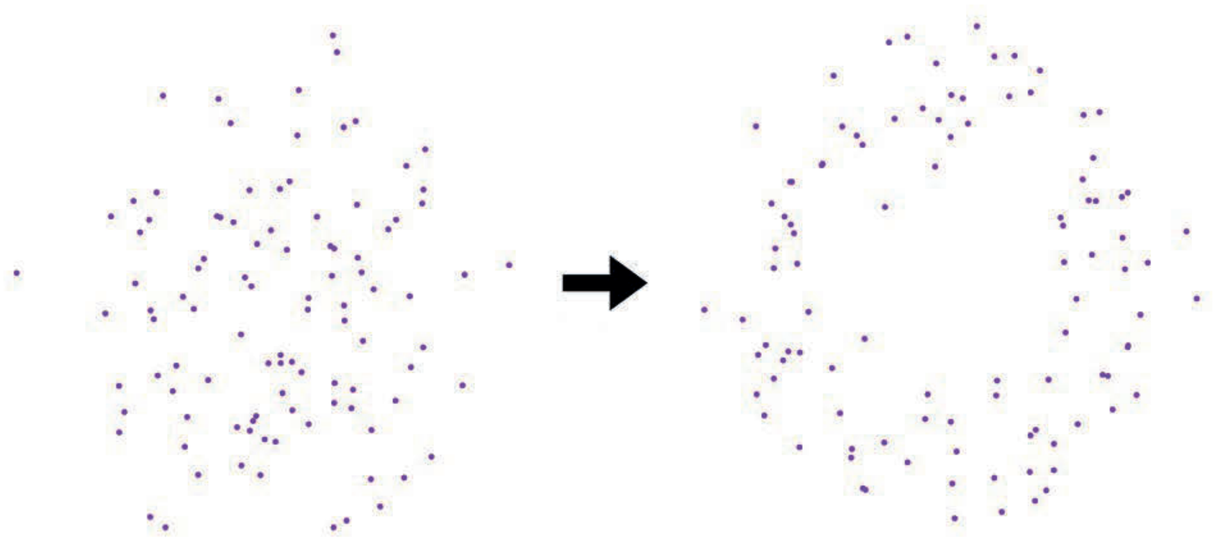


Figure 4: Kloke's Algorithm over 'dataNoisy'- the shifted points are in the blue color, which results in less noisy data. It did a better job of denoising !



Circle

How to use common functions in the R package denoiseTDA, and denoising two-dimensional noisy circle data set.

I. Creating the Toy Data : Circle

This time we will be generating noisy circle.

You can generate circle points using function `CircleGen(init=c(0,0), R=1, N=100)` from `denoiseTDA` package.

1. Generating Circle

```
circle<-CircleGen(init=c(1,1), R=1, N=1000)
```

2. Applying noise on it

```
sigma<-0.5
circleNoisy<-NoisyGen(circle, var = sigma)
```

3. Plotting our data

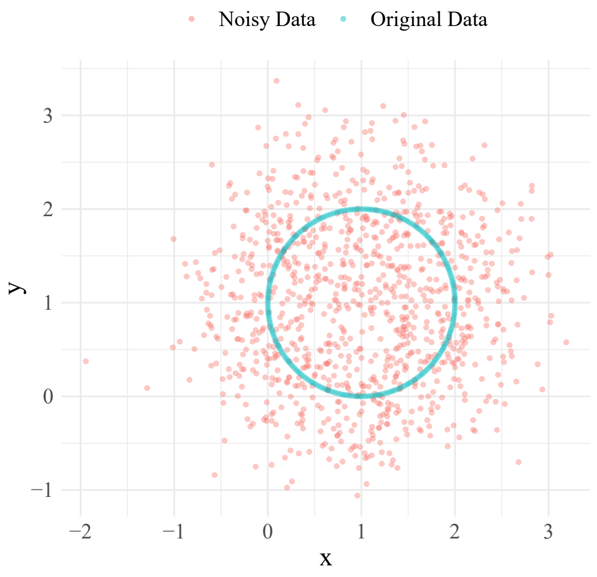


Figure 5: Circle Data - blue points shows the initial data, and red shows a noisy version of it.

II. De-Noising Toy Data : Circle

Mean Shift Algorithm

Implementing function `MeanShiftAlg( set, sigma, iter, step)`.

1. De-noising

```
circleShifted <- MeanShiftAlg(set=circleNoisy, sigma=sigma, iter=200, step=0.3 )
```

2. Plotting `circleShifted`

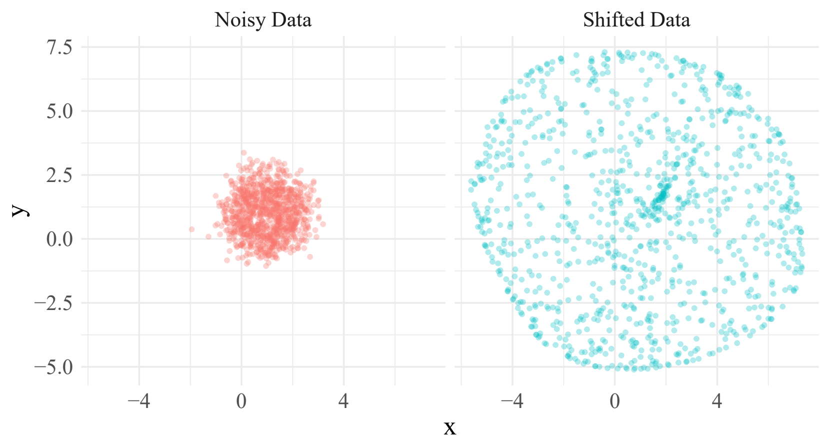


Figure 6: Circle Data - blue points shows the initial data, and red shows a noisy version of it.

From the plot above you can see how poorly did Mean Shift Algorithm shifted points. Play with parameters, maybe you can get a clearer picture.

Kloke's Algorithm

Implimenting function `KlokeAlg(sample, set, sigma, omega, iter, step)` [6]

[6] Play with parameters `step`, `sigma`, `omega`, see how it affects the shifted data.

1. Sampling the set

```
circleSample<-MatSample(circleNoisy, N=200)
```

2. Shifting points by **Kloke's Algorithm**

```
circleShifted <- KlokeAlg(sample=circleSample, set=circleNoisy,
                          sigma=sigma, omega=0.1, iter=200, step=0.3 )
```

3. Plot

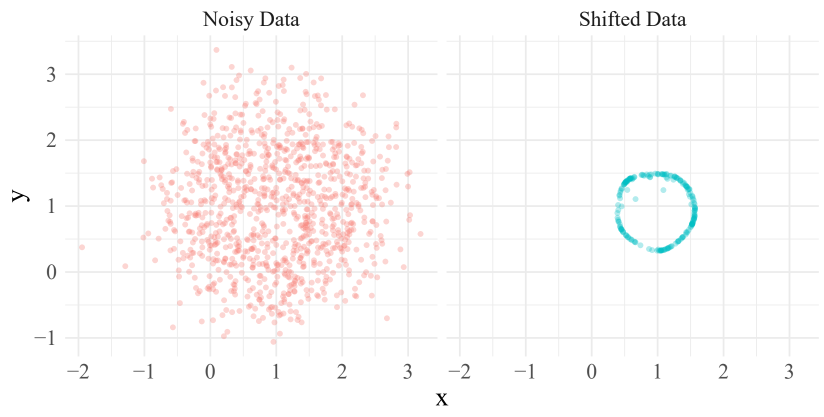


Figure 7: Circle Data - blue points shows the initial data, and red shows a noisy version of it.

It succsessfully return the underlying shape of the data!

Other Shapes

Now try to de-noise this !

```
Circ1<-CircleGen(R=1.5, N=1000) #default initial point is (0,0)
Circ2<-CircleGen(init=c(1.8, 1.8), N=1000)
Circ3<-CircleGen(init=c(-1.8, 1.8), N=1000)
Point1<-matrix(rep(c(0,0), each=1000), ncol=2)
circAll<-rbind(Circ1, Circ2, Circ3, Point1)
shapeNoisy<- NoisyGen(circAll, var=0.6)  # <- THIS TO DENOISE
```

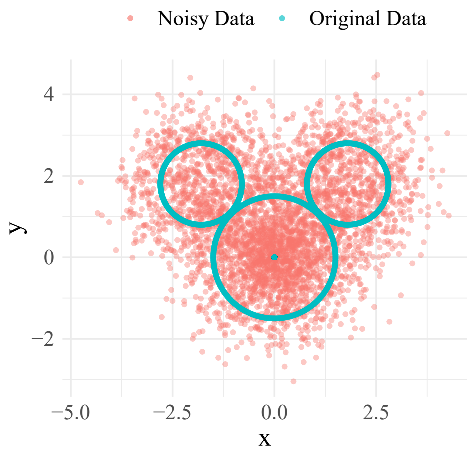
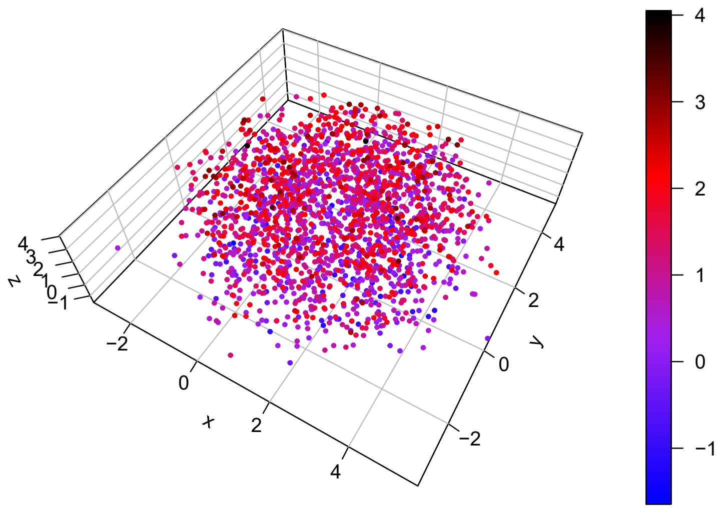


Figure 8: Change variable ‘var‘ to experiment and create your own data with shape!

As long as you input matrix you can de-noise at any dimension. For example try to denoise this data

```
library(plot3D)
N<-200 # Number of points in the each circle
       # so it doesn't generate evenly spaced torus -_-
       # there will be more points on circles closer to the center of torus
Num<-200 # Number of the circles
sigma<-0.6 # Noisyness of data
R1<-2 # Radius of outer circle
R2<-1 # Radius of inner circle
deg<-seq(0,360, length.out = Num)
CircMain<-CircleGen(R=R1+R2*cos(deg[1]), N=N)
Z<-rep(R2*sin(deg[1]), N)
bin<-cbind(CircMain, matrix(Z, ncol=1))
data<-bin
for(i in 2:Num){
  CircMain<-CircleGen(R=R1+R2*cos(deg[i]), N=N)
  Z<-rep(R2*sin(deg[i]), N)
  bin<-cbind(CircMain, matrix(Z, ncol=1))
  data<-rbind(data, bin)
}
sampleSize<-2000
row<-sample(1:(N*Num), sampleSize)
dataSam<-matrix(data[row[1],], ncol=3)
for(i in 2:sampleSize){
  dataSam<-rbind(dataSam, matrix(data[row[i],], ncol=3))
}
# DENOISE THIS TorusNoisy
TorusNoisy<-NoisyGen(dataSam,
                     var=sigma*R2, mean=R2)
sampleSize<-400   # <- PLAY WITH THE SIZE OF THE SAMPLE
TorusSam<-MatSample(dataNoisy, sampleSize) # <- Selecting Sample
#Ploting
x<-TorusNoisy[,1]
y<-TorusNoisy[,2]
z<-TorusNoisy[,3]
scatter3D(x, y, z, phi = 50, theta= 30, bty = "b2",
          col = ramp.col(c("blue","purple","red", "black")),
        pch = 20, cex = 0.7, ticktype = "detailed", expand=0.3)
```

Figure 9: Generating Noisy Torus



Insight and Future Work

The main time was spent on creating shown functions and testing them over different data. Kloke's Algorithm has more adjustable control, and require fewer iterations which result in the more efficient de-noising algorithm than Mean Shift Algorithm.

A little time was put on applying the algorithms over the real data. Some not trivial issues occurred, which resulted in the absence of shifting of the real or uneven shift. Possible solution for those issues proposed.

Mean Shift and Kloke's Algorithm

One of the problems occurring with Mean Shift Algorithm is, that when the data is highly noisy, the algorithm fails to recover the underlying shape of data. Possibly after adjusting all of the parameters, it could recover the shape. However, it is very likely it will also require more iterations. Which is especially inefficient when data has a higher dimension than one.

In the plot below you can see that algorithm fails to recover the underlying shape (which is two points, generated in the same way as in the tutorial from above, just noisier).

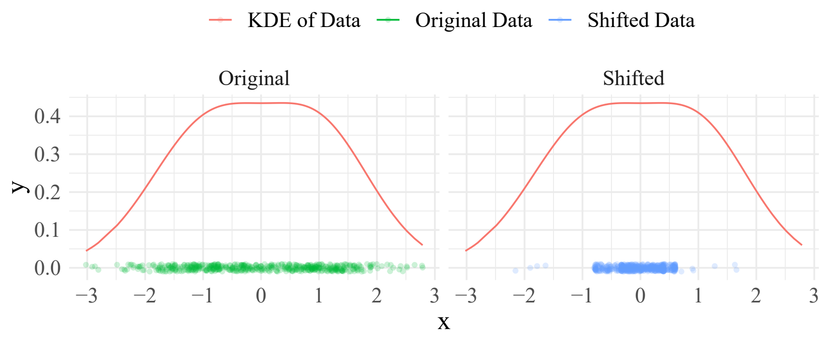


Figure 10: Before and after shift by **Mean Shift Algorithm**, underlying shape is two points at coordinates -1 and 1.

An analogous example is a data with an underlying circular shape which is shown in the tutorial, in Circle section. Where also the Mean Shift Algorithm fails to recover the underlying shape of the circle.

While applying Kloke's Algorithm, in the plot below, results in recovering the shape. Since Kloke's Algorithm has more adjustable parameters, it is more plausible to recover the underlying shape with fewer iterations, which is especially important for the higher dimensions of data. In general, even with the same amount of iterations, Kloke's Algorithm will require less time to compute.

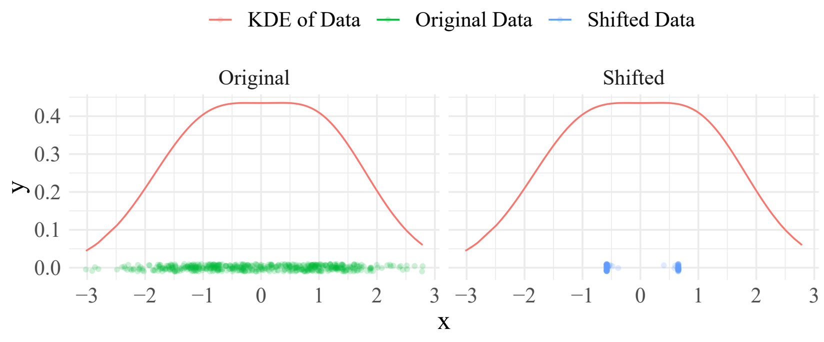


Figure 11: Before and after shift by **Kloke's Algorithm**, underlying shape is two points at coordinates -1 and 1.

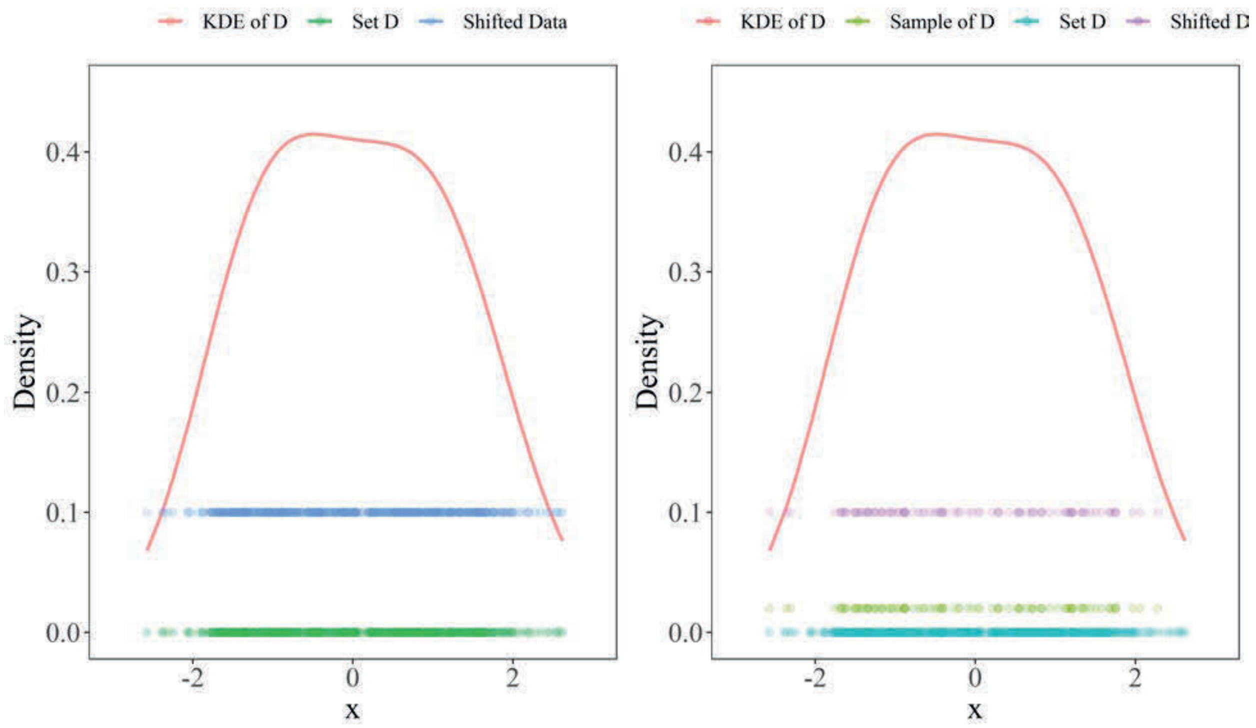
Kloke's Algorithm also successfully recovers the shape of the data in an example with an underlying circular shape which is shown in the tutorial, in Circle section.

I suppose the main reason for such improvement, is that when points are getting highly dense, the Mean Shift Algorithm, starts to overshoot, which results in collapsing the data into one point. While in Kloke's Algorithm, when points get highly dense, opposite happens, the shift becomes less, due to the negative term of the function which Jennifer Novak Kloke implied in algorithm[7]. Eventually it results into the repulsive behavior, and retriving the underlying shape.

[7] Function is shown in Appendix A, in the `KlokeAlg()` function description.

Below is an animation[8] where you can see how points are getting shifted. Points on both plots are consistently being shifted by the same amount of iterations. But with different algorithms. The left one - **Mean Shift Algorithm** and the right one - **Kloke's Algorithm**. In this animation, you can observe the behavior described above.

[8] To see animation open this document in the Adobe Acrobat, and press the play button in the control menu underneath the image.



There should be a better description and reasoning about how Kloke's algorithm improves the de-noising property over the Mean Shift Algorithm.

Parameters

Some parameters in the function might not be clear in the use, possibly will be useful current perception of how each parameter affects the shifting process. Most of the descriptions are about Kloke's algorithm since this algorithm appeared to be more practical.

`iter` - $[n]$ - is a number of iterations. More useful to have it low (below 500). Especially with data of higher dimensions. Since in the process you will likely be adjusting parameters to eventually result in recovering the shape. Keep it low (below 200) to get to try to adjust other parameters for a desirable result.

`step` - $[c]$ - a selected percentage of the maximum initail shift, from 0 to 1.[9] With less noisy data, it is usefull to have `step`~ 0.5. When working with higly noisy data, it becomes useful to have the step higher (>0.7). However it will start to overshoot, to prevent that, you can also increase ω.

[9] *Possible Adjustment to the algorithm, is to define a better suit of the step since a properly chosen step will result in the fewer iterations and better denoising

`sigma`- $[\sigma]$ - it indicates how narrowly you want to bond points. Very low values (nearly 0) will result in the absence of shift since you will eventually be trying to shift points into itself. To see the reasoning look at the Kernel Density Estimator plot below. When sigma gets extremely low, density evaluated as the same across the whole data set.

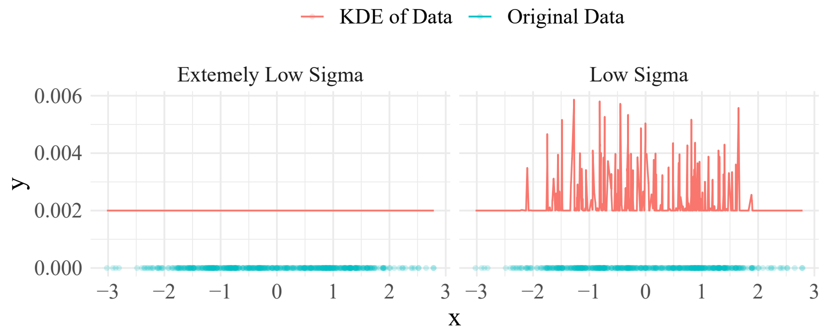


Figure 12: Low 'sigma' in the Kernel Density Estimator, underlying shape is two points at coordinates -1 and 1.

`omega` - $[\omega]$ - only used in the Kloke's Algorithm. Higher values reduce shift. Normally from 0 to 1. It can be nearly 0 when data has low noise ($omega$<0.2) It is very useful to have it higher(ω>0.2) adjust ω parameter when you are trying to de-noise very noisy data (σ>0.8).

`sample` - $[S_0]$ - sample of the set, be carefully selecting the size of the sample since the too small size of the sample might not retain the shape of the data. Too high size of the sample might increase the time for computing the shift.

Application

There was not much done on applying shifting algorithms over the real data. Since the toy data generated is commonly symmetric in the shape, and the maximum distance between two points (where hole must be) in the initial shape (except torus) is 2. In torus the smaller circle, which revolve around center, has analogous relationship (the radius of that circle is 1). It results in more clear recover of the underlying shape.

When you apply the algorithm over real unchanged data, you will see that the data did not shift, and increasing σ does not change the result.

1. Need to normalize the real data.

Bring it to lower scale, for example, where the mean distance between two points is 0.5. Bascally you need to shrink data, since the algorithm cluster points which has distance between each other lower than selected σ in the algorithm.

In this example if take data from `CO1.cvs`, which is shown below[10]

[10] The code used for plots below is in the Appendix B - Application.

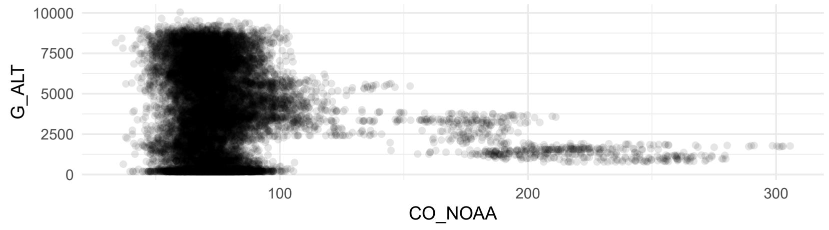


Figure 13: Original real data

Try to apply Kloke's Algorithm over the raw data, no shift happens. You can try adjust all of the parameters and see what what happens.

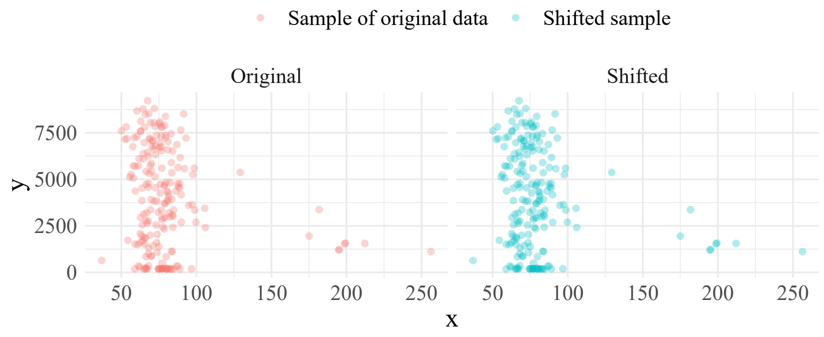


Figure 14: The sample was taken from the original data, and Kloke's algorithm applied. No shift occurs. Try to change parameters and see what happens. It is possible that the reason is that with bigger values, the computation overflows.

And then when the data is scaled, the shift occurs.

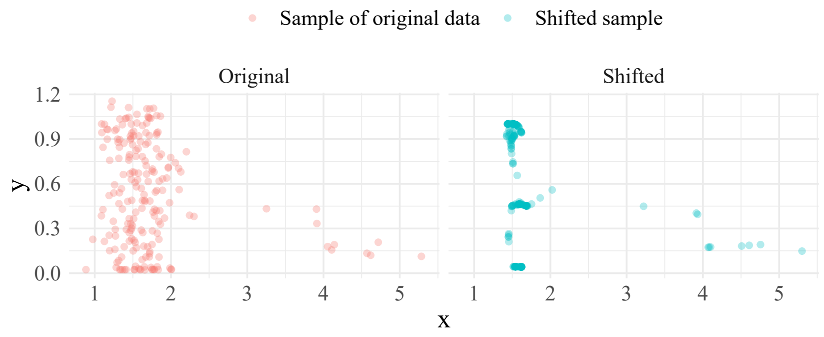


Figure 15: Data was sampled, then normalized, as the result Kloke's Algorithm shifts points.

Since the algorithm does not account for different variables, and shifts symmetrically, assuming the same scale on both axis. Which leads to one of the aspects to pay attention to that **the data variables can have a different noise**. As a result, it unevenly shifts the data, resulting in shifting more in one axis than the other, or it could be intentional, to retain the aspect ratio. Which arises the other concern

2. How to normalize the data with variables of different scale, noise?

Possibly, by accounting the real noise, and scaling it down proportionally. Which is requiring a good knowledge of the data you are working with.

Appendix A | Function List

Algorithm Functions

Some important functions for de-noising.

```
KDE(set, sigma)
MeanShiftAlg(set, sigma, iter, step)
KlokeKDE(sample, set, sigma, omega)
KlokeAlg(sample, set, sigma, omega, iter, step)
```

Edit Functions

Some useful function to generate and modify data. They are optional to use.

```
NoisyGen(Data, var, mean=0)
MatSample(mat, N)
MaToData(Mat)
CircleGen(init, R, N)
```

Algorithm Functions

`KDE(set, sigma)`

The function used for Kernel Density Estimator

$$f(x) = \frac{1}{|D|} \sum_{p \in D} e^{\frac{-||x-p||^2}{2\sigma^2}}$$

`set` - (matrix)[D] - data set

`sigma` - (numeric)[σ] - smoothing parameter.

returns - (matrix) returns the matrix with first columns are from input `set` and aditional column, which is **KDE** values. In other words returns a matrix with one dimension higher than the input `set`.

`MeanShiftAlg(set, sigma, iter, step)`

Mean Shift Algorithm is based on the kernel density estimator, which is

$$f(x) = \frac{1}{|D|} \sum_{p \in D} e^{\frac{-||x-p||^2}{2\sigma^2}}$$

The algorithm itself

$$D_{n+1} = \left\{ p + c\frac{\nabla f_n(p)}{M} | p \in D_n \right\}$$

`set` - (matrix)[D] - data set.

`sigma`- (numeric)[σ] - an indicater of *how tight* you want to shift points.

`step` - (numeric)[c] - a value in percentage (0 to 1), which indicates by which percent of the initial maximum shift you want shift points.

`iter` - (integer)[n] - a number of iterations (shifts) you want to make.

returns - (matrix)[D_{n+1}] - a shifted `set` by Mean Shift Algorithm

KlokeKDE(sample, set, sigma, omega)

Modyfied Kernel Density by Jennifer Novak Kloke

$$F_n(x) = \frac{1}{|D|}\sum_{p\in D} e^{\frac{-||x-p||^2}{2\sigma^2}} - \frac{\omega}{|S_n|}\sum_{p\in S_n} e^{\frac{-||x-p||^2}{2\sigma^2}}$$

`sample` - (matrix)[S_n] - a sample of the set.

`set` - (matrix)[D] - a data set.

`sigma`- (numeric)[σ] - smoothing parameter.

`omega` - (numeric)[ω] - how significantly you want to see an impact from the second term of the equation.

returns - (matrix)[$F_n(S_n)$] - returns the matrix with first columns are from input `sample` and aditional column, which is **KlokeKDE** values. In other words returns a matrix with one dimension higher than the input `sample`.

KlokeAlg(sample, set, sigma, omega, iter, step)

Kloke Algorithm is based on the modyfied kernel density estimator, which is shown below

$$F_n(x) = \frac{1}{|D|}\sum_{p\in D} e^{\frac{-||x-p||^2}{2\sigma^2}} - \frac{\omega}{|S_n|}\sum_{p\in S_n} e^{\frac{-||x-p||^2}{2\sigma^2}}$$

The algorithm itself

$$S_{n+1} = \left\{p + c\frac{\nabla F_n(p)}{M} | p \in S_n\right\}$$

`sample` - (matrix) [S_0] - a sample of the set.

`set` - (matrix) [D] - a data set.

`sigma`- (numeric) [σ]- an indicater of *how tight* you want to shift points.

`omega` - (numeric) [ω] - higher value reduces the shift (from 0 to 1).

`step` - (numeric) [c]- a value in percentage (from 0 to 1), which indicates which percent of the maximum shift you want shift points.

`iter` - (integer) [n] - a number iteration (shifts) you want to make.

returns -(matrix) [S_{n+1}] - a shifted `sample` by Kloke's Algorithm.

Edit Functions

`NoisyGen(Data, var, mean=0)`

Generates noise over the input matrix.

`Data` - (matrix) the matrix which you want to make noisier

`var` - (numeric) variance of how noisy you want make data

`mean` - (numeric) the mean of the noise you want to apply to the data

returns - (matrix) noisy version of the input matrix `Data`

`MatSample(Mat, N)`

Samples input matrix by row.

`Mat` - (matrix) is a matrix you want to sample

`N` - (integer) a size of the sample

returns - (matrix) a shifted `sample` by Kloke's Algorithm

`MatToData(Mat)`

Which transforms matrix with columns up to 3 into the data frame. Where the 1st column is evaluated as `x` , 2nd as `y` and 3rd as `z`.

`Mat` - (matrix) matrix which you want to convert into data frame

returns - (data frame) with column names `x, y, z`.

`CircleGen(init=c(0,0), R=1, N=100)`

Generates evenly spaced points which form circle with defined properties.

`init` - (vector) initial center of the circle in a form of vector with default `init=c(0,0)`.

`R` - (numeric) radius of the circle with default `R=1`.

`N` - (integer) number of points in the circle, default `N=100`.

returns - (matrix) coordinates of points which form circle, in a form of 2-dimensional matrix of row length `N`.

Appendix B | Code

You are very welcome into looking into code, and check me :D

`denoiseTDA` package

Link to the github package:
`https://github.com/URiyaxD/denoiseTDA`

Edit Functions

You can see the function by typing in console `?+function` to see the code in the **Example** section. For example

```
?MatSample
```

Edit functions source code:
`https://github.com/URiyaxD/denoiseTDA/blob/master/R/func_edit.R`

Algorithm Functions

However you **can not see code** for the shifting algorithms. Since it was writen in `C++`. To see the code go directly to the source code.

Shifting algorithm source code:
`https://github.com/URiyaxD/denoiseTDA/blob/master/src/func_alg.cpp`

Might be useful to look at this course to learn how to work with C++ in R
`https://www.datacamp.com/courses/optimizing-r-code-with-rcpp`

Application

```
library(dplyr)
dataC01<-read.csv("data/C01.csv")
data<-filter(dataC01, CO_NOAA!=-99999, G_ALT !=-99999)
ggplot()+
  geom_point(data,mapping= aes(x=CO_NOAA , y=G_ALT ), alpha=0.1)+theme_minimal()
```

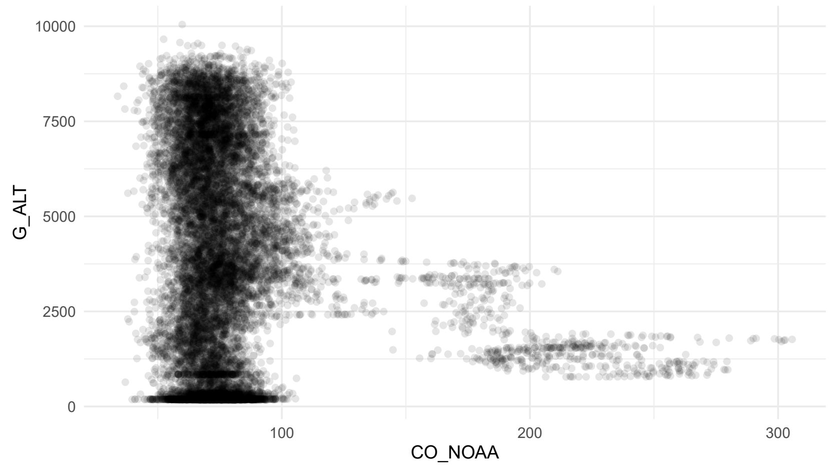


Figure 16: Original real data

```
MatData<-matrix(c(data$CO_NOAA, data$G_ALT), ncol=2)
MatSam<-MatSample(MatData, N=200)
Shifted<-KlokeAlg(MatSam , MatData, sigma=500, omega=0.04, iter=100, step=0.2)
cc<-MatToData(rbind(MatSam, Shifted))
cc$group<-c(rep("Original", each=length(MatSam[,1])),
            rep("Shifted", each=length(Shifted[,1])))
cc$col<-c(rep("Sample of original data", each=length(MatSam[,1])),
          rep("Shifted sample", each=length(Shifted[,1])))
p<-ggplot()+
  geom_point(cc,
             mapping=aes(x=x, y=y, group=factor(group), col=factor(col)),
             alpha=0.3, size=1.5)+
  theme_minimal()+
  scale_color_discrete(name="")+
  theme(text = element_text(family="serif", size = 16),
        legend.position = "top")

p+facet_grid(.~group)

#normalizing data, by dividing it by variance ->
MatData<-matrix(c(20*data$CO_NOAA/var(data$CO_NOAA),
```

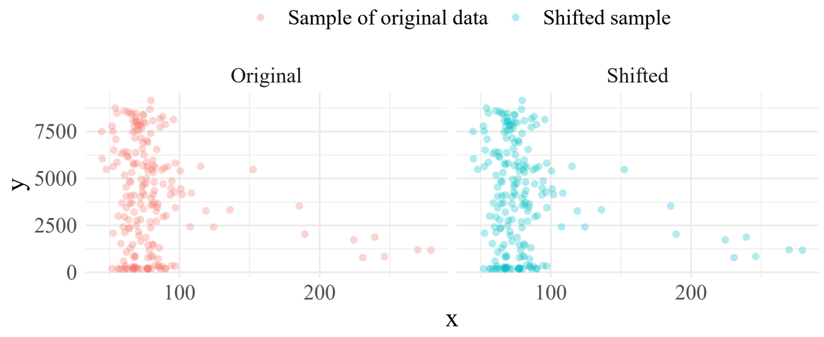


Figure 17: The sample was taken from the original data, and Kloke's algorithm applied. No shift occurs. Try to change parameters and see what happens. It is possible that the reason is that with bigger values, the computation overflows.

```
                    1000*data$G_ALT/var(data$G_ALT)), ncol=2)
MatSam<-MatSample(MatData, N=200)
Shifted<-KlokeAlg(MatSam , MatData, sigma=0.06, omega=0.04, iter=100, step=0.2)
cc<-MatToData(rbind(MatSam, Shifted))
cc$group<-c(rep("Original", each=length(MatSam[,1])),
            rep("Shifted", each=length(Shifted[,1])))
cc$col<-c(rep("Sample of original data", each=length(MatSam[,1])),
          rep("Shifted sample", each=length(Shifted[,1])))
p<-ggplot()+
  geom_point(cc,
             mapping=aes(x=x, y=y, group=factor(group), col=factor(col)),
             alpha=0.3, size=1.5)+
  theme_minimal()+
  scale_color_discrete(name="")+
  theme(text = element_text(family="serif", size = 16),
        legend.position = "top")
p+facet_grid(.~group)
```

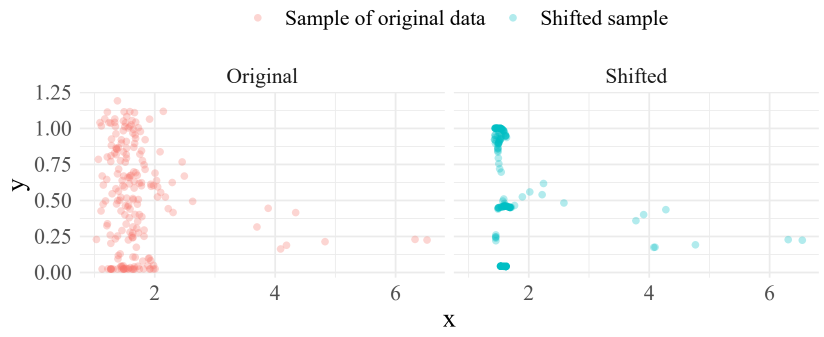


Figure 18: Data was sampled, then normalized, as the result Kloke's Algorithm shifts points.