Lecture 8

DJM

13 November 2018

Motivation

Overview

Representation learning is the idea that performance of ML methods is highly dependent on the choice of representation

For this reason, much of ML is geared towards transforming the data into the relevant features and then using these as inputs

This idea is as old as statistics itself, really,

(e.g. Pearson (1901), where PCA was first introduced)

However, the idea is constantly revisited in a variety of fields and contexts

Commonly, these learned representations capture 'low level' information like overall shape types

Other sharp features, such as images, aren't captured

It is possible to quantify this intuition for PCA at least

PCA

Principal components analysis (PCA) is an (unsupervised) dimension reduction technique

It solves various equivalent optimization problems

(Maximize variance, minimize L_2 distortions, find closest subspace of a given rank,...)

At its core, we are finding linear combinations of the original (centered) covariates

$$Z_{ij} = \alpha_j^\top X_i$$

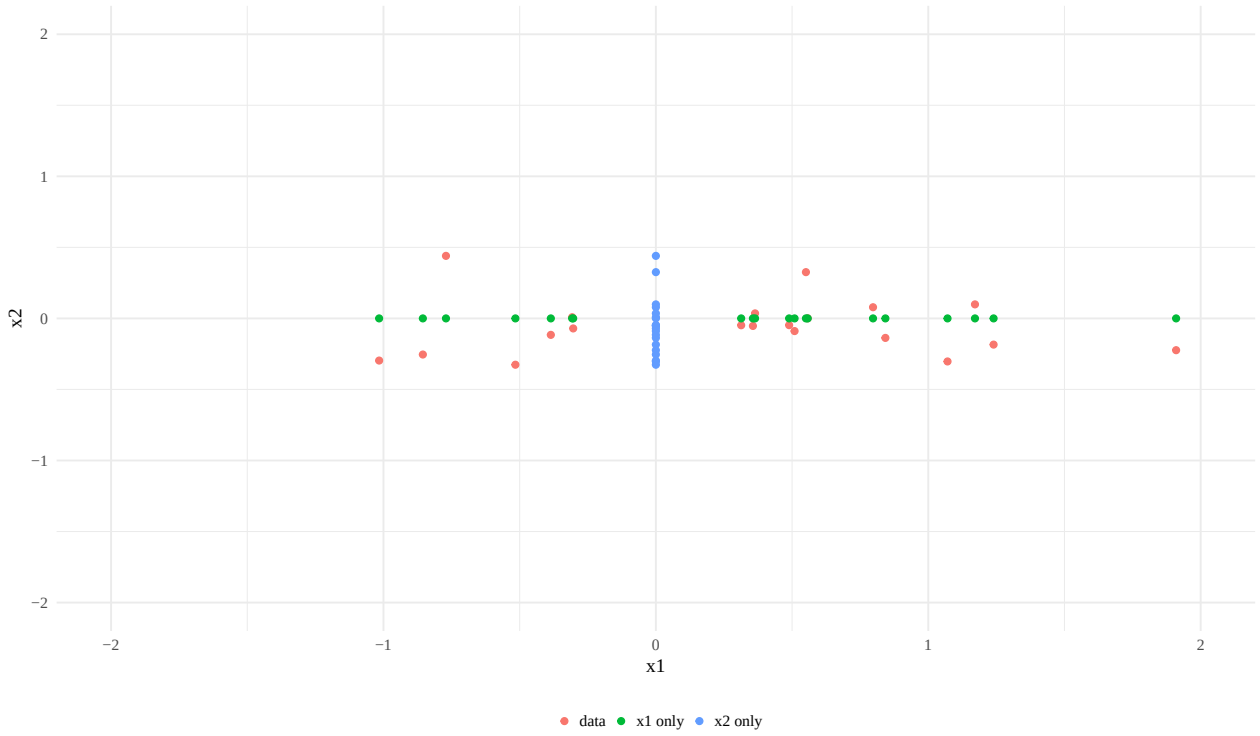
This is expressed via the SVD: $X - \overline{X} = UDV^\top$ as

$$Z = XV = UD$$

Lower dimensional embeddings

Suppose we have predictors x_1 and x_2

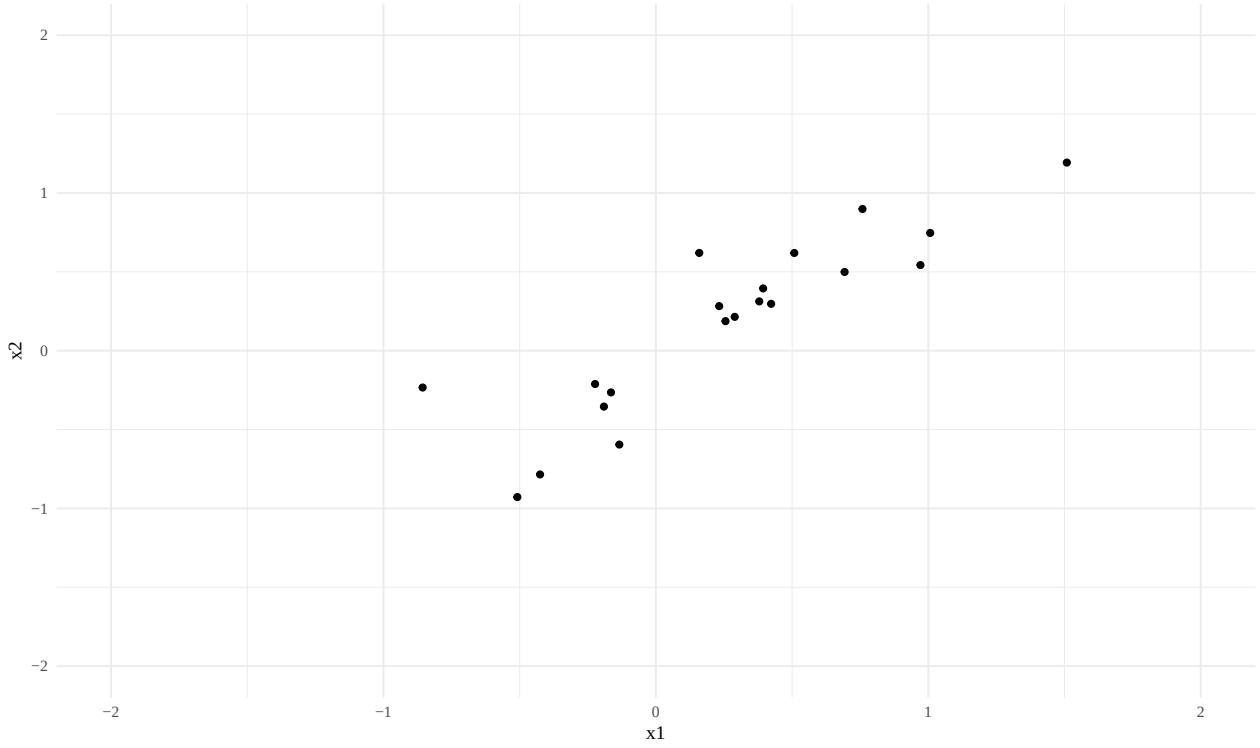
- We more faithfully preserve the structure of the data by keeping x_1 and setting x_2 to zero than the opposite



Lower dimensional embeddings

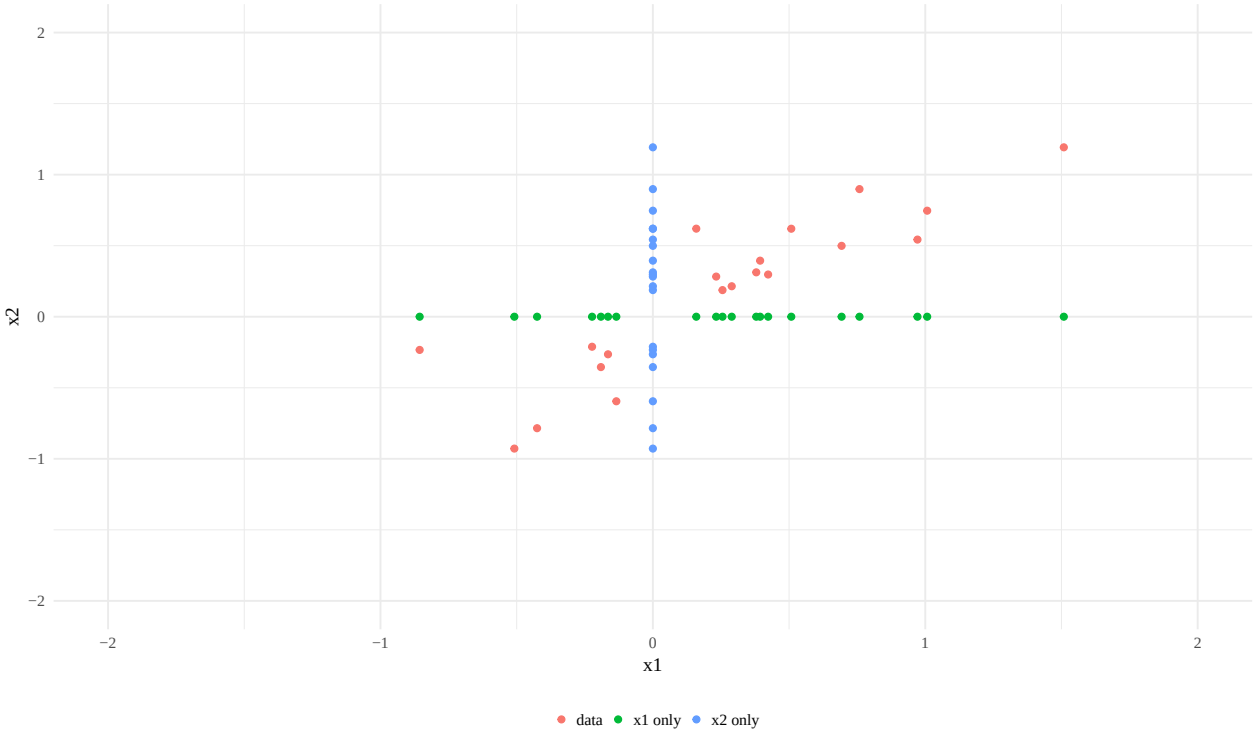
An important feature of the previous example is that x_1 and x_2 aren't correlated

What if they are?



Lower dimensional embeddings

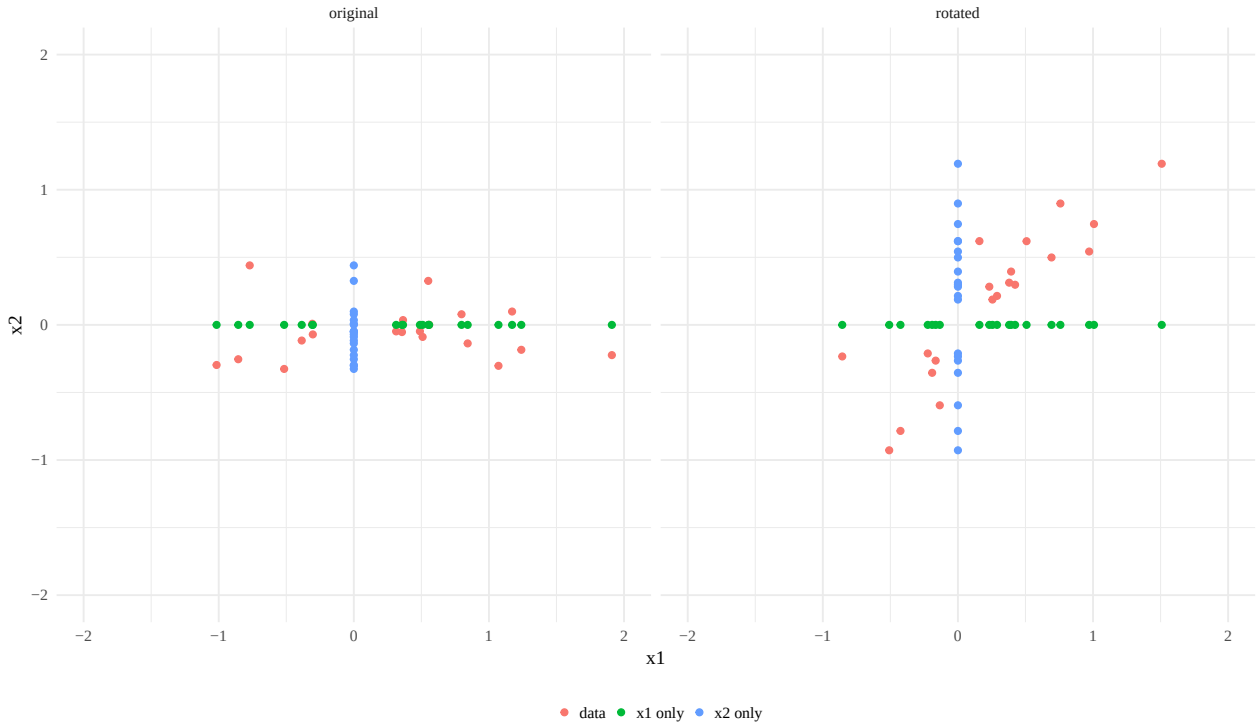
We lose a lot of structure by setting either x_1 or x_2 to zero



Lower dimensional embeddings

There isn't that much structurally different between the examples

One is just a rotation of the other



PCA

If we knew how to rotate our data, then we could more easily retain the structure.

PCA gives us exactly this rotation

Optimization

If we want to find the first K principal components, the relevant optimization program is:

$$\min_{\mu,(\lambda_i),V_K} \sum_{i=1}^{n} \|X_i - \mu - V_K\lambda_i\|^2$$

This representation is important

It shows that we are trying to reconstruct lower dimensional representations of the covariates

PCA

$$\min_{\mu,(\lambda_i),V_K} \sum_{i=1}^{n} \|X_i - \mu - V_K\lambda_i\|^2$$

We can partially optimize for μ and (λ_i) to find

- $\widehat{\mu} = \overline{X}$
- $\widehat{\lambda}_i = V_K^\top (X_i - \widehat{\mu})$

We can find

$$\min_V \sum_{i=1}^{n} \left\| (X_i - \widehat{\mu}) - VV^\top (X_i - \widehat{\mu}) \right\|^2$$

where V is constrained to be orthogonal

This is the so called Steifel manifold of rank-K orthogonal matrices

The solution is given by the singular vectors V

(General) spectral connectivity analysis

Metric embeddings

Spectral connectivity analysis (SCA)

- Linear and nonlinear
- Dimension reduction or feature creation
- Examples: PCA, Locally linear embeddings, Hessian maps, ~~Laplacian eigenmaps~~
- Useful tools in classification, clustering, (regression)

PCA (review)

Collect data: $X_1, \ldots, X_n$ where $X_i \in \mathbb{R}^p$.

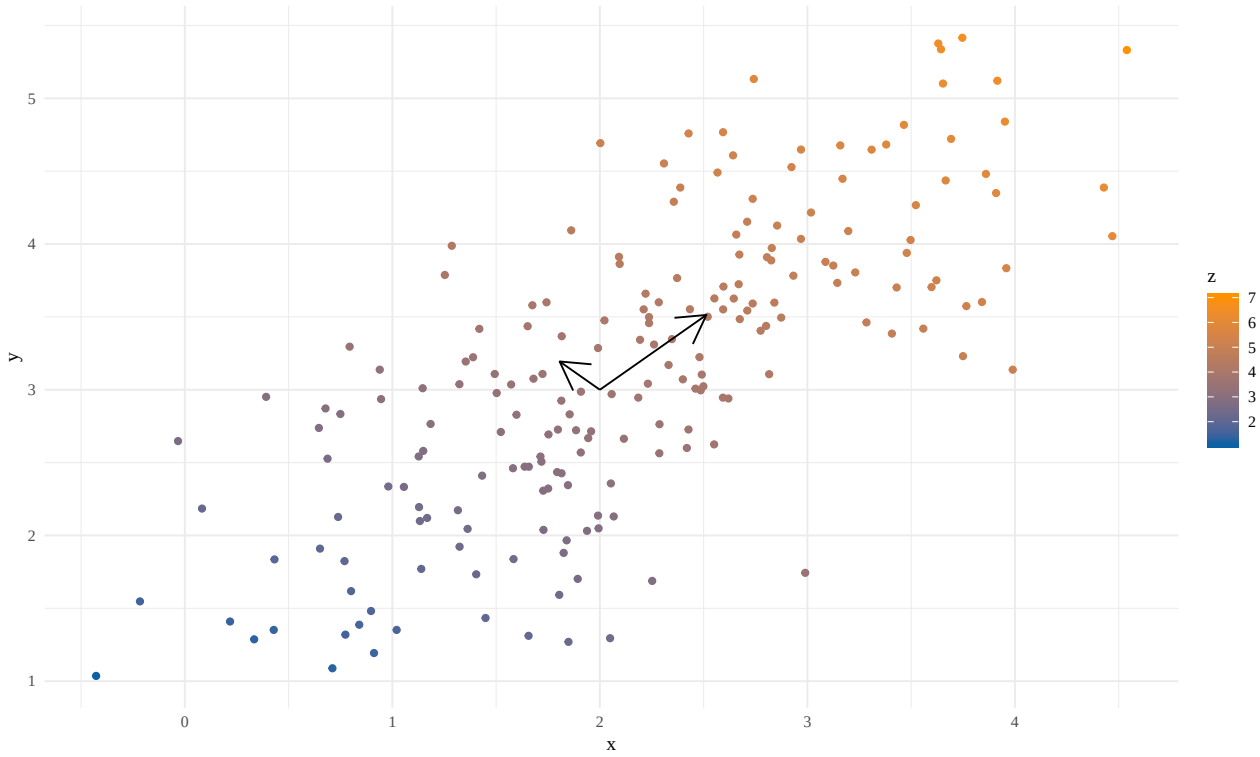
1. Center and (scale) the data matrix X
2. Compute the SVD of $X = U\Sigma V^\top$ or $XX^\top = U\Sigma^2 U^\top$ or $X^\top X = V\Sigma^2 V^\top$
3. Return $U_d\Sigma_d$, where Σ_d is the largest d eigenvalues of X

- You need to compute SVD of $XX^\top$ [or X]

When PCA works well

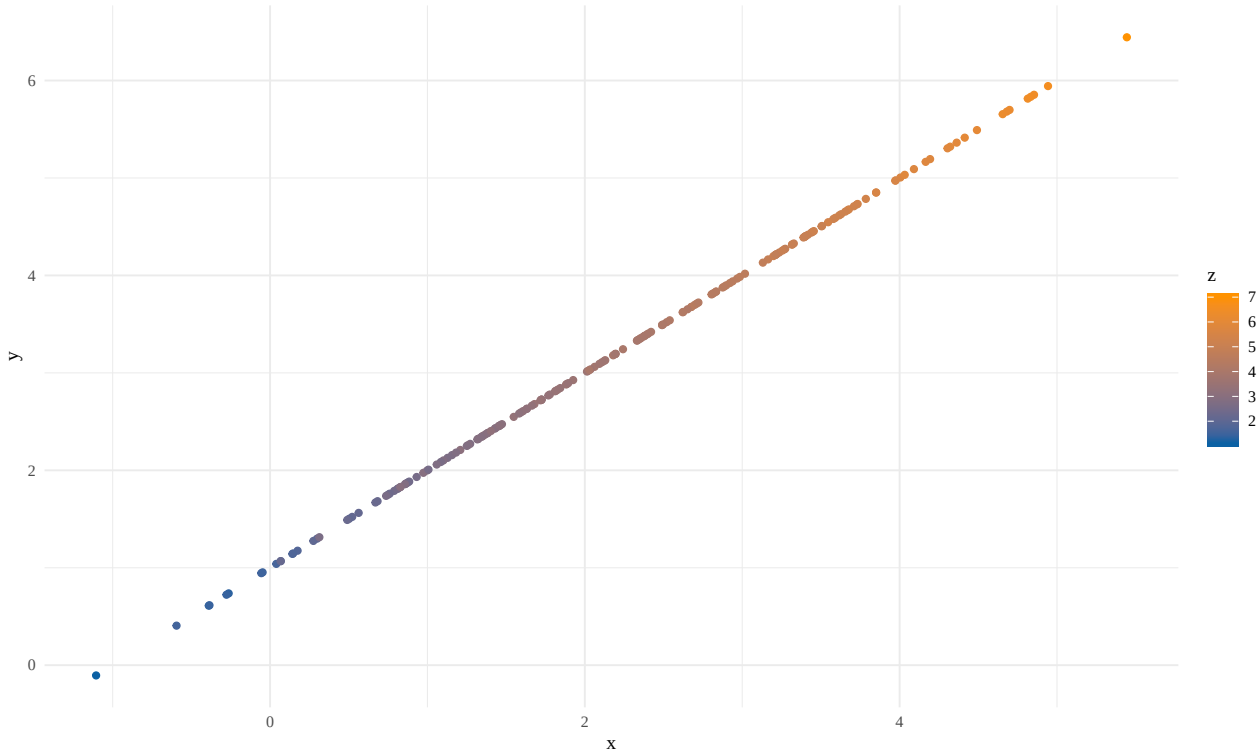
PCA "works" when the data can be represented (in a lower dimension) as 'lines' (or planes, or hyperplanes).

So, in two dimensions:



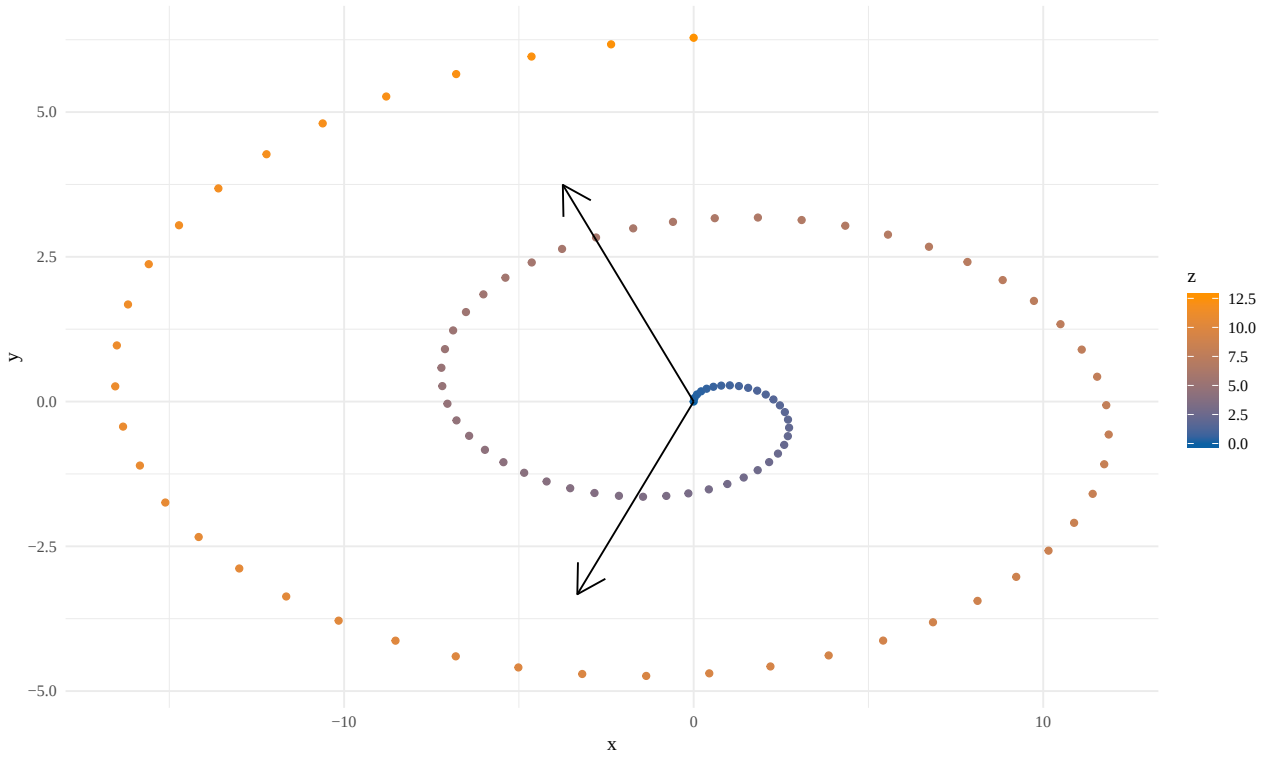
PCA reduced

Here, we can capture a lot of the variation and underlying 'structure' with just 1 dimension, instead of the original 2 (the coloring is for visualizing).



PCA bad

What about other data structures? Again in two dimensions

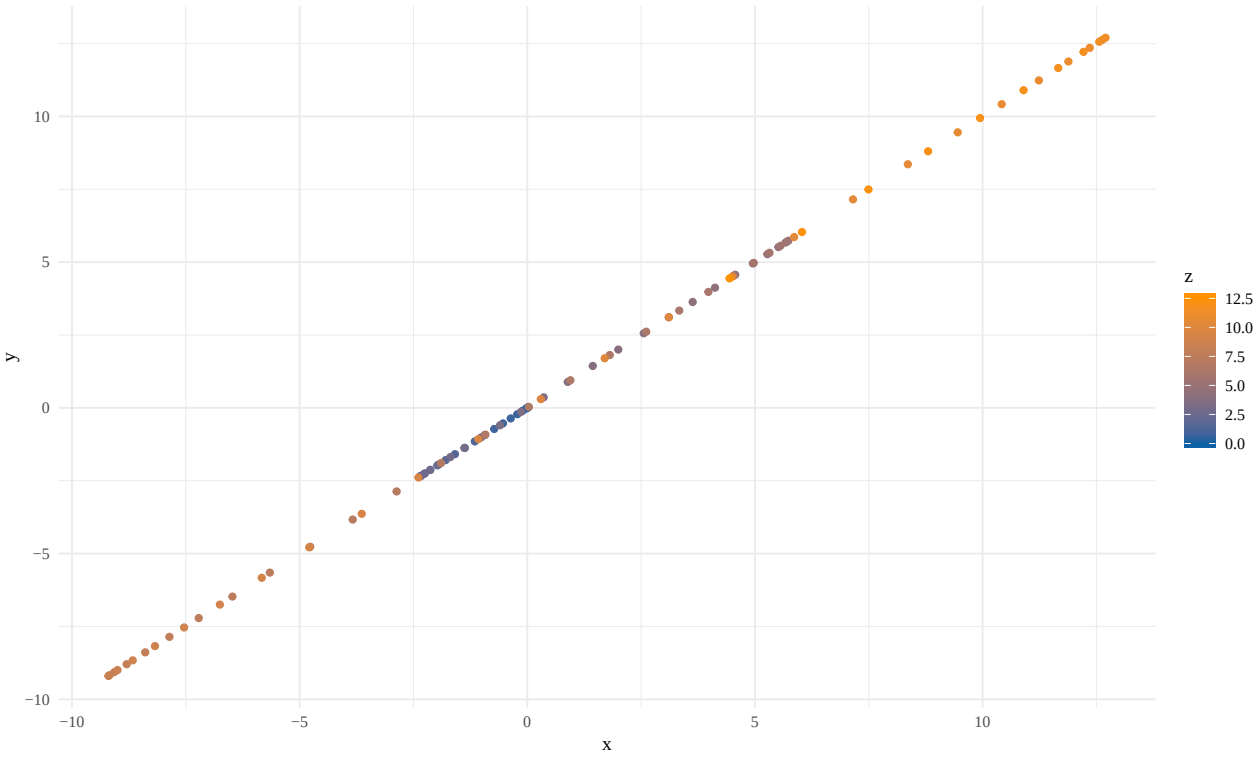


PCA bad (2)

Here, we have failed miserably.

There is actually only 1 dimension to this data (imagine walking up the spiral going from blue to orange).

However, when we write it as 1 PCA dimension, all the points are all 'mixed up'.



Explanation

- PCA wants to minimize distances (equivalently maximize variance). This means it 'slices' through the data at the 'meatiest' point, and then the next one, and so on. If the data are 'curved' this is going to induce artifacts.
- PCA also looks at things as being 'close' if they are near each other in a Euclidean sense.
- On the spiral, our intuition says that things are 'close' only if the distance is constrained to go along the curve. In other words, purple and blue are close, blue and orange are not.

Nonlinearity and CMDS

Kernel PCA (KPCA)

Classical PCA comes from $\tilde{X} = X - MX = UDV^\top$, where $M = \mathbf{1}\mathbf{1}^\top/n$ and $\mathbf{1} = (1, 1, \ldots, 1)^\top$

However, we can just as easily get it from the outer product

$$\mathbb{K} = \tilde{X}\tilde{X}^\top = (I - M)XX^\top(I - M) = UD^2U^\top$$

The intuition behind KPCA is that $\mathbb{K}$ is an expansion into a kernel space, where

$$\mathbb{K}_{i,i'} = k(\tilde{X}_i, \tilde{X}_{i'}) = \langle \tilde{X}_i, \tilde{X}_{i'} \rangle$$

Reminder: Anytime we see an inner product, we can kernelize it

Kernel PCA

Following this intuition, the approach is simple:

1. Specify a kernel k

   (e.g. $k(X, X') = \exp\{-\gamma^{-1} ||X - X'||_2^2\}$)
2. Form $K_{i,i'} = k(X_i, X_{i'})$
3. Standardize and get eigenvector decomposition

$$\mathbb{K} = (I - M)K(I - M) = UD^2U^\top$$

This implicitly finds the inner product:

$$k(X_i, X_{i'}) = \langle \phi(X_i), \phi(X_{i'}) \rangle$$

However, we need only specify the kernel

Kernel PCA

The scores are still $Z = UD$

The q^{th} KPCA score is (up to centering)

$$Z_{iq} = \sum_{i'=1}^{n} \beta_{i'q} k(X_i, X_{i'})$$

where $\beta_{i',q} = u_{i'q}/d_q$

Note: As we don't explicitly generate the feature map, there are no loadings

Kernel PCA

Reminder: To get the first PC in classical PCA, we want to solve

$$\max_{\alpha} \mathbb{V}\alpha^\top X \quad \text{subject to} \quad ||\alpha||_2^2 = 1$$

Translate this into the kernel setting, and we are trying to solve

$$\max_{g \in \mathcal{H}_k} \mathbb{V}g(X) \text{ subject to } ||g||_{\mathcal{H}_k} = 1$$

The representer theorem states that a solution to this problem is

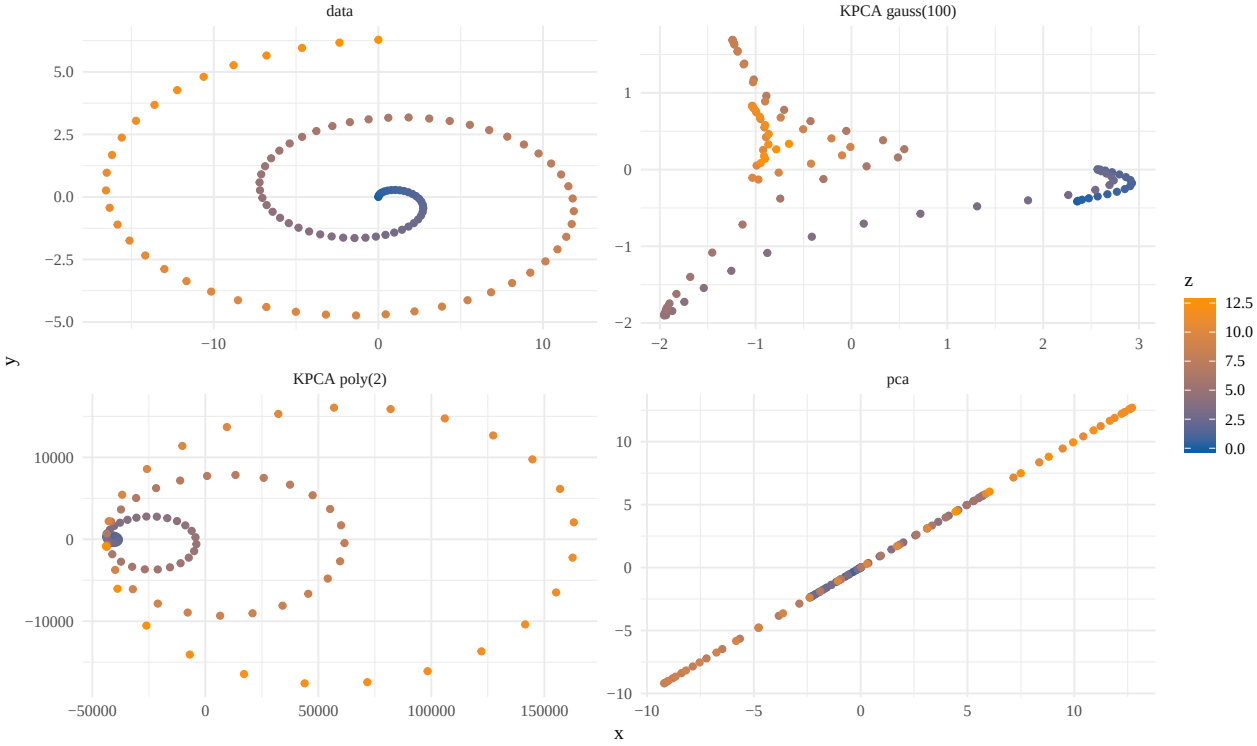
$$\widehat{g}(X) = \sum_{i=1}^{n} \beta_i k(X, X_i)$$

Compare

$$Z_{iq} = \sum_{i'=1}^{n} \beta_{i'q} k(X_i, X_{i'})$$

where $\beta_{i',q} = u_{i'q}/d_q$

KPCA example



KPCA: summary

Kernel PCA seeks to generalize the notion of similarity using a kernel map

This can be interpreted as finding smooth, orthogonal directions in a RKHS

This can allow us to start picking up nonlinear (in the original feature space) aspects of our data

This new representation can be passed to a supervised method to form a semisupervised learner

Semi-supervised detour

Basic semi-supervised

1. You get data $\mathcal{D}_{train} = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$.
2. You do something unsupervised on the X's to create features (like PCA).
3. You use the learned features to find a predictor $\widehat{f}$ using the new features.

Semisupervised learning in practice

Looking at:

$$Z_{iq} = \sum_{i'=1}^{n} \beta_{i'q} k(X_i, X_{i'})$$

this is only defined at our observed features

Write

- $\mathcal{D}_{train} = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$
- $\mathcal{D}_{test} = \{(X_1^*, Y_1^*), \ldots, (X_{n^*}^*, Y_{n^*}^*)\}$

Two common scenarios are

1. We are given $\mathcal{D}_{train}$ and $X_1^*, \ldots, X_{n^*}^*$ to build $\widehat{f}$
2. We are given only $\mathcal{D}_{train}$ to build $\widehat{f}$

Case 1

We are given $\mathcal{D}_{train}$ and $X_1^*, \ldots, X_{n^*}^*$ to build $\widehat{f}$

Then we can just use straight forward KPCA

(Or any unsupervised learning step)

1. Form $\mathbb{K}$ on $\mathcal{D}_{train}$ and $X_1^*, \ldots, X_{n^*}^*$
2. Get UD
3. Pass $Z_q = UD[, 1:q]$ to train $\widehat{f}$
4. Get $\widehat{Y} = \widehat{f}(Z_q)$

Case 2

We are given only $\mathcal{D}_{train}$ to build $\widehat{f}$

Now, we don't know the coordinates of $X_1^*, \ldots, X_{n^*}^*$ in the representation space

To get a new observation X^* embedded into this representation:

$$Z_0 = D^{-1}U^\top (I - M)[k^* - K\mathbf{1}/n]$$

where $k^* = [k(X^*, X_1), \ldots, k(X^*, X_n)]^\top$

Then we compute:

1. Form $\mathbb{K}$ on $\mathcal{D}_{train}$
2. Get UD
3. Pass $Z_q = UD[, 1:q]$ to train $\widehat{f}$
4. Form Z_q^* for all $X_1^*, \ldots, X_{n^*}^*$
5. Get $\widehat{Y}_{test} = \widehat{f}(Z_q^*)$

Classical multidimensional scaling

CMDS

A broader class comes from the following procedure (see Figure 4.2 in the text):

1. Calculate a matrix of distances (or dissimilarities) between data points (Δ)

2. Choose some function $\tau : \mathbb{R} \to \mathbb{R}$ and set $B = \tau(\Delta^2)$.
3. Write $B = U\Sigma U^\top$ (find the eigendecomposition)
4. Approximate your data with $U_{[d]}\Sigma_{[d]}^{1/2}$ where $[d]$ means "the first d columns"

CMDS (I think) is steps 2-4: embedding "dissimilarities" by using the eigendecomposition.

What I'll call Laplacian Eigenmaps

We can get an estimate of the distance in the unknown geometry that the data come from (known as a 'nonlinear' manifold) by altering the usual Euclidean distance.

In this case, I'll use the heat kernel (like most everyone) but this is not necessary.

Distance between x and y:

- PCA: $\|x - y\|_2$
- Laplacian eigenmaps : $\exp\left(-\frac{\|x-y\|_2^2}{\epsilon}\right)$

Some notes:

- The name 'Laplacian Eigenmaps' comes from getting the eigenvector decomposition of the Laplacian restricted to the manifold (which is the second derivative version of the gradient). This gives crucial information about its geometry.
- The form of the distance says that if $\|x - y\|_2^2$ is much bigger than ϵ, then the distance is effectively zero. This encodes the idea that things that are 'close' in Euclidean distance are 'close' on the manifold as well.
- If the manifold is smooth, then local Euclidean distance is an approximation to the distance on the manifold.

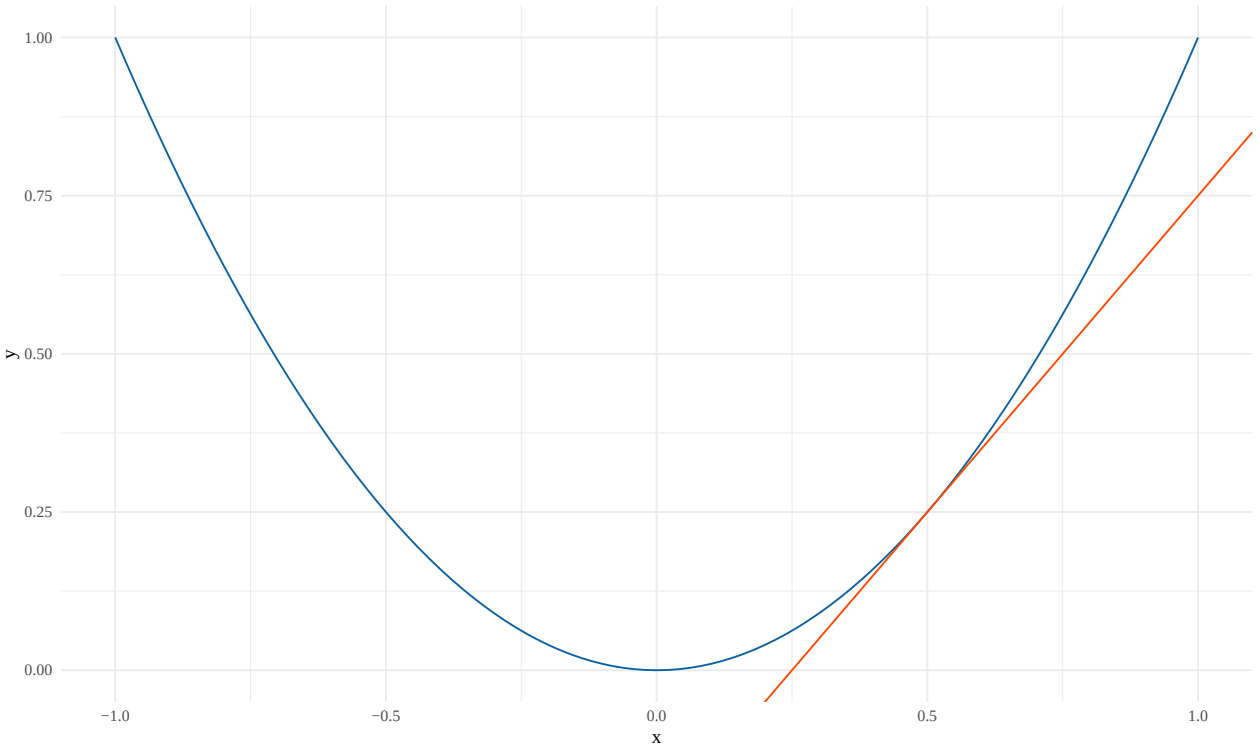
What is a manifold?

How good of an approximation is Euclidean distance?

This question is equivalent to how asking: how quickly does the tangent (space) change?

In 1-D, the tangent space is just the first derivative at that point:

$$f(x) = x^2 \Rightarrow f'(x) = 2x.$$



What is a manifold

Therefore, the quality of the (local) Euclidean distance, depends on the second derivative

(ie: how fast does the first derivative change?)

In higher dimensions, the second derivative is known as the **Laplacian**:

$$\sum_j \frac{\partial^2 f}{\partial x_j^2}$$

(Note: This is also known as the divergence of the gradient)

What are Laplacian Eigenmaps, then?

Imagine the operator $\mathbb{L}$ that performs this operation:

$$\mathbb{L}f = \sum_j \frac{\partial^2 f}{\partial x_j^2}$$

Then $\mathbb{L}$ is the **Laplacian**, mapping a function to the divergence of its gradient

- **Key Idea:** We can get the eigenvectors/eigenvalues of $\mathbb{L}$. Analogously to PCA, we can now do inference with these eigenvectors.

Note: There is a substantial overlap with KPCA, the difference being the centering of K and the row sum versus column sum normalization

Laplacian Eigenmaps (procedure)

Collect data: $X_1, \ldots, X_n$ where $X_i \in \mathbb{R}^p$.

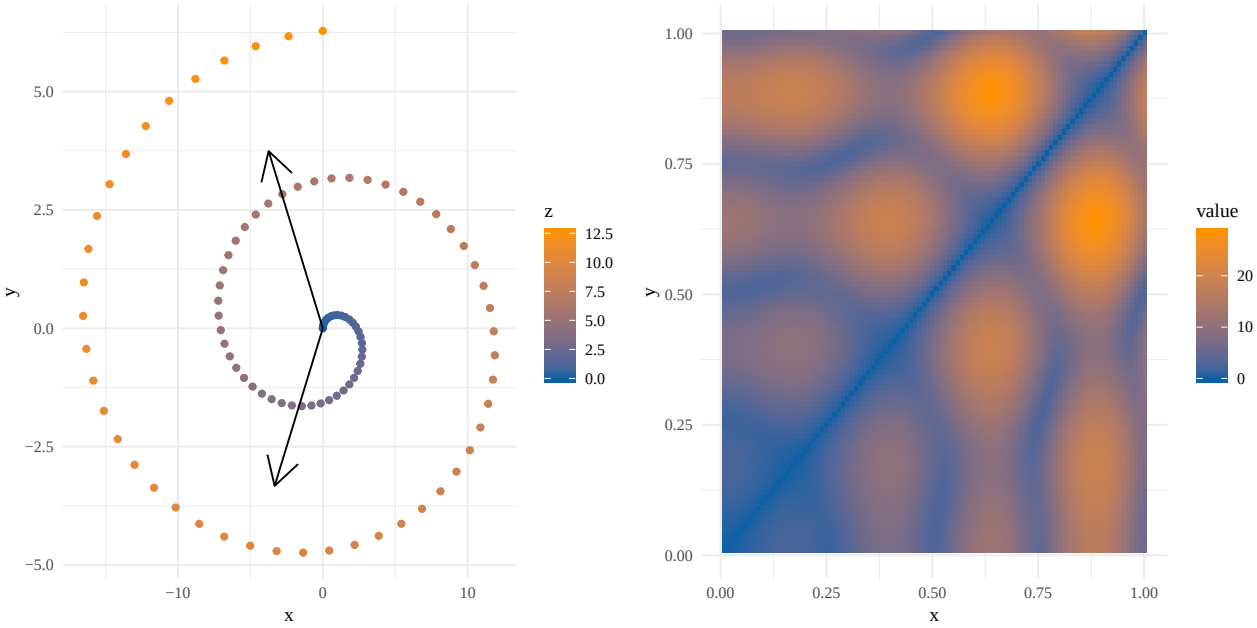
1. Center and scale the data matrix X
2. Compute $\mathbb{K}$ where
$$\mathbb{K}_{ij} = \exp\left(-\frac{\|X_i - X_j\|_2^2}{\epsilon}\right)$$
3. Form the Laplacian $\mathbb{L} = \mathbb{I} - \mathbb{M}^{-1}\mathbb{K}$ where $\mathbb{M} = \texttt{diag(rowSums(}\mathbb{K}\texttt{))}$
4. Compute the SVD of $\mathbb{L} = U\Sigma U^\top$.
5. Return $U_d\Sigma_d^{-1}$, where Σ_d contains the smallest d nonzero eigenvalues of $\mathbb{L}$

   (Note that the eigenvectors of $\mathbb{L}$ and $\mathbb{M}^{-1}\mathbb{K}$ are the same, but $\Sigma(\mathbb{L}) = \mathbb{I} - \Sigma(\mathbb{M}^{-1}\mathbb{K})$)

- This is basically just CMDS on $\tau(\Delta)$.
- ~~You need to compute SVD of $\mathbb{L}$.~~

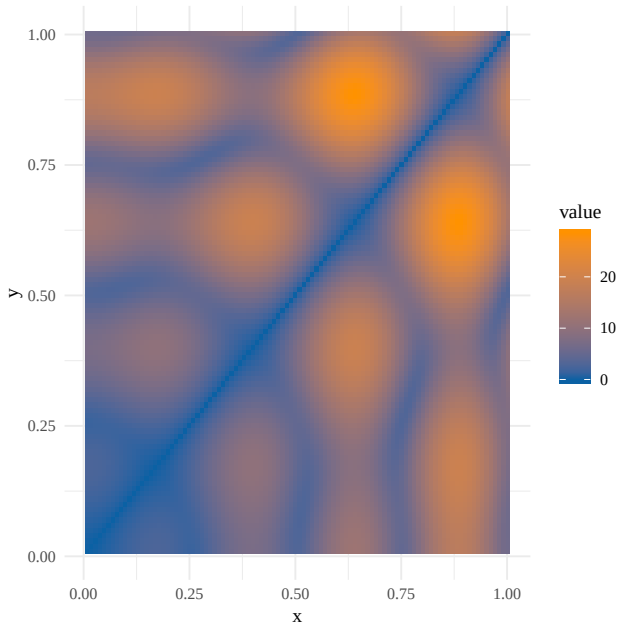
Deeper investigation 1.

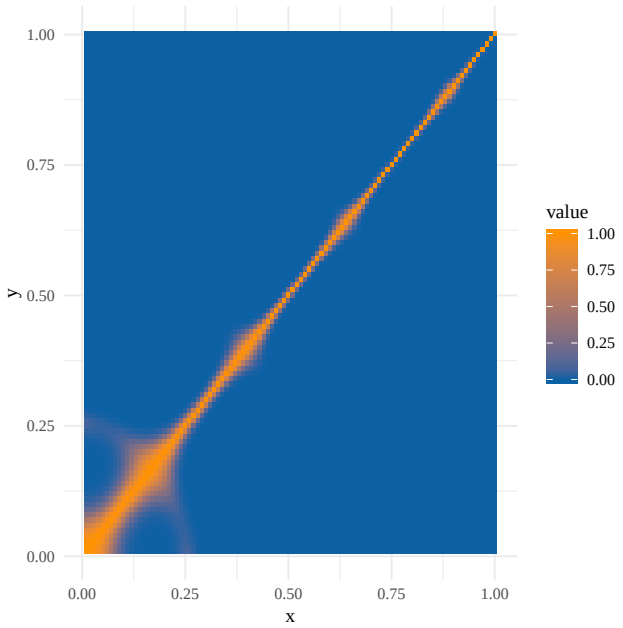
The distance matrix



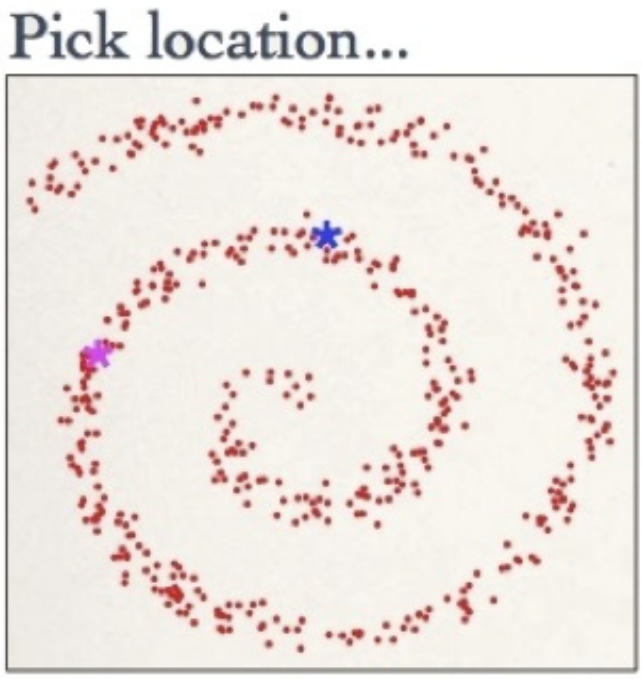
More deeper

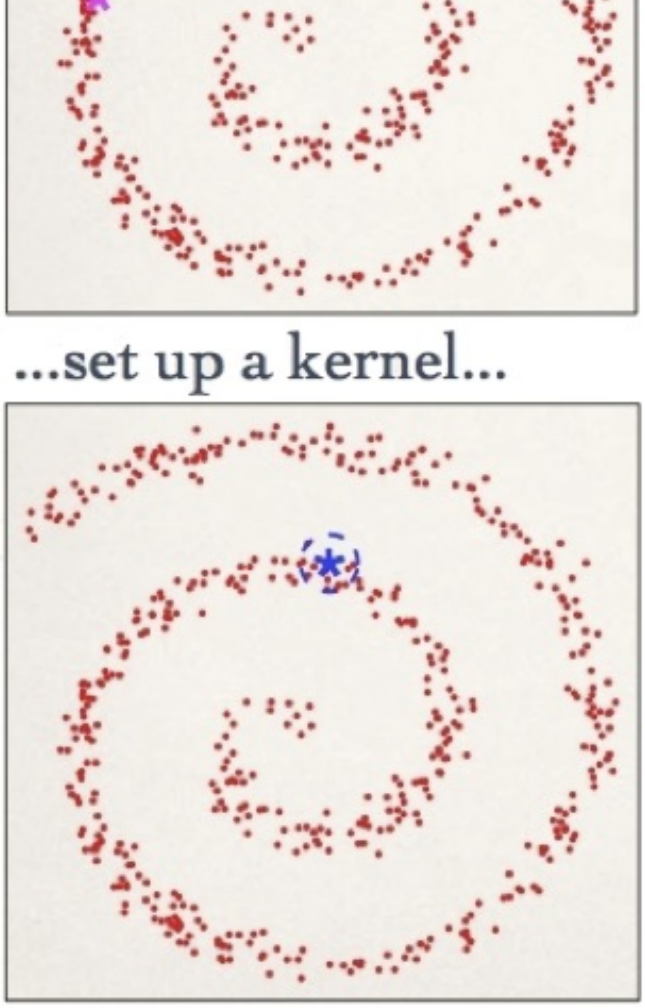
Exponentiate $-\Delta/\gamma$

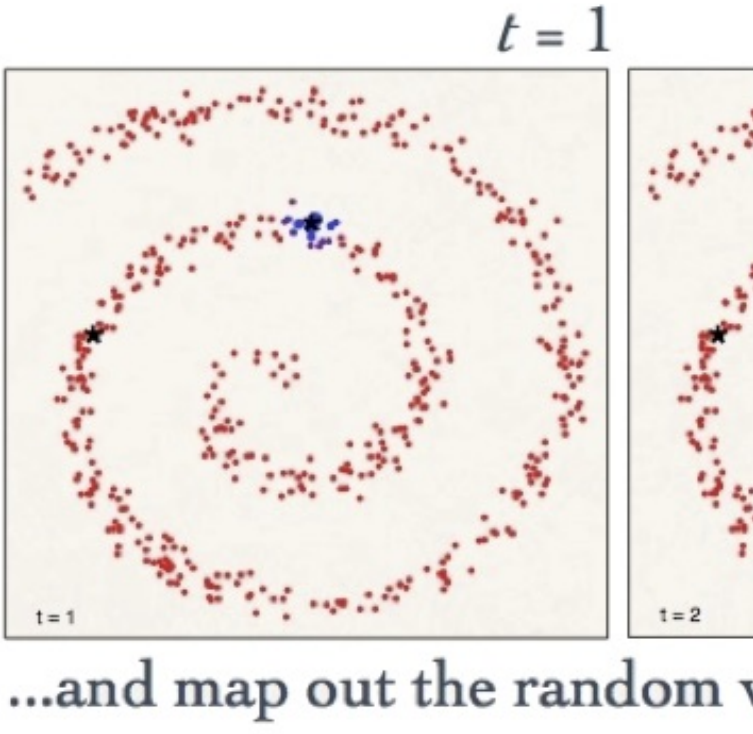

1.00
0.75
0.50
0.25
0.00
y
0.00
0.25
0.50
0.75
1.00
x
value
20
10
0


1.00
0.75
0.50
0.25
0.00
y
0.00
0.25
0.50
0.75
1.00
x
value
1.00
0.75
0.50
0.25
0.00

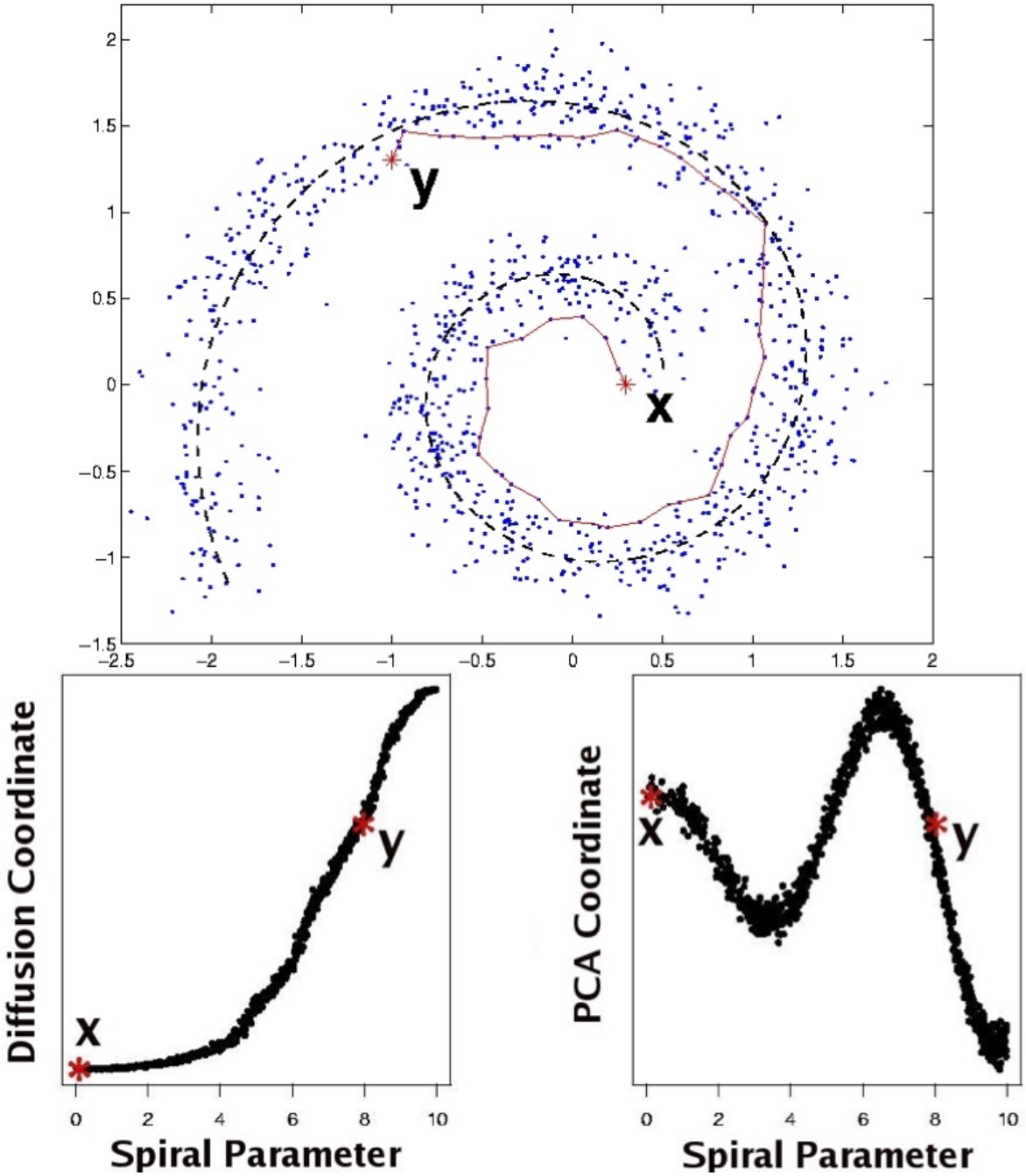
Diffusion distance







Projection


y
x
Diffusion Coordinate
Spiral Parameter
PCA Coordinate
Spiral Parameter

Another example

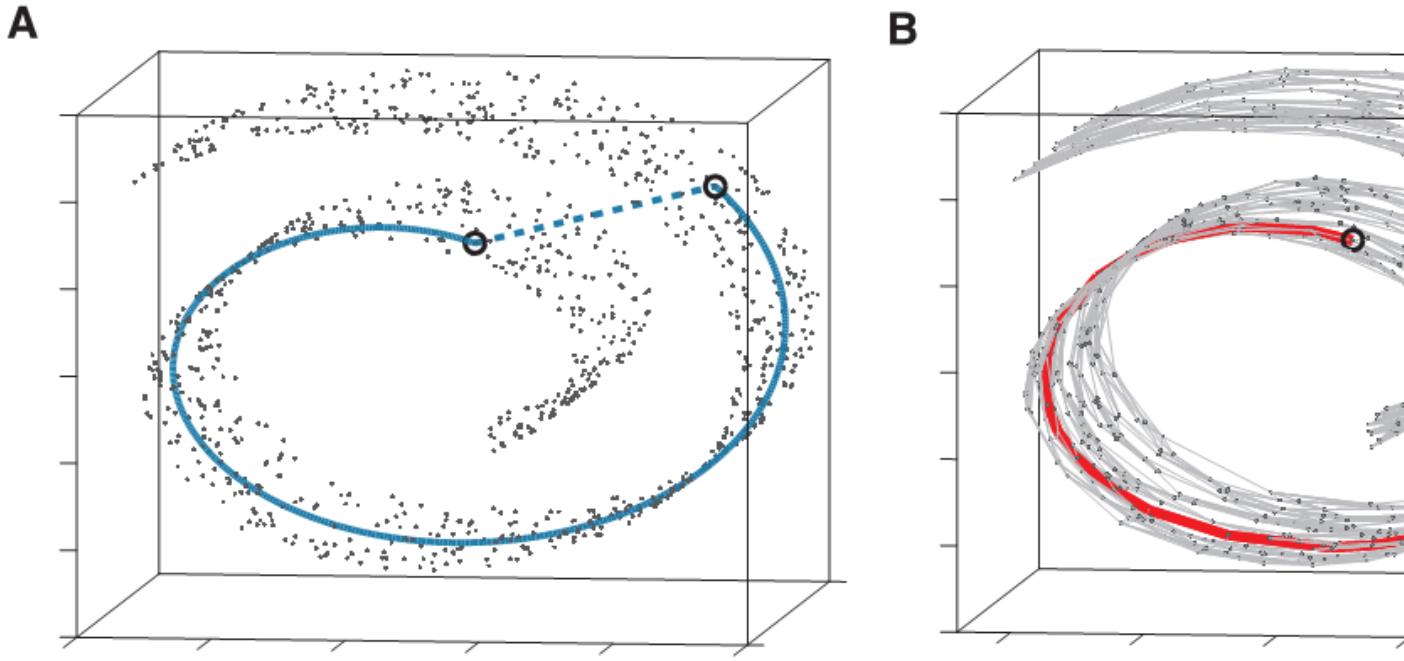


Fig. 3. The "Swiss roll" data set, illustrating how Isomap exploits geodesic paths for nonlinear dimensionality reduction. (**A**) For two arbitrary points (circled) on a nonlinear manifold, their Euclidean distance in the high-dimensional input space (length of dashed line) may not accurately reflect their intrinsic similarity, as measured by geodesic distance along the low-dimensional manifold (length of solid curve). (**B**) The neighborhood graph G constructed in step one of Isomap (with $K = 7$ and $N =$ 1000

What goes wrong?

Collect data: $X_1, \ldots, X_n$ where $X_i \in \mathbb{R}^p$.

You want to perform PCA / Laplacian eigenmaps / LLE / dimension reduction / manifold learning

n, p are Big, like really ~~BIG~~

You have some problems:

1. The matrices $XX^\top$ or $\mathbb{L}$ may be hard to store
2. Computing the SVD of either is an $O(n^3)$ operation (for PCA, it is $O(\min\{np^2, p^3\})$, but both are big)

Approximation: Can we compute the SVD approximately and still do a good job?

Nyström extension

Based on a technique for finding a numerical solution for integral equations

Very similar to out-of-sample embedding

Algorithm (briefly)

1. Subsample Ł: choose $M \subset \{1, \ldots, n\}$ such that $|M| = m$, $m \ll n$

$$\widetilde{Ł} := Ł_{M,M}$$

2. Compute eigendecomposition of $\widetilde{Ł} = \widetilde{U}\widetilde{\Sigma}\widetilde{U}^\top$
3. "Extend" $\widetilde{U}$ via simple formula to create U^{nys}

   Write

$$Ł = \begin{bmatrix} Ł_{M,M} & Ł_{12} \\ Ł_{21} & Ł_{22} \end{bmatrix} \quad U^{nys} = \begin{bmatrix} \widetilde{U} \\ Ł_{21}\widetilde{U}\widetilde{\Sigma}^{-1} \end{bmatrix}$$

Requires only $O(nm^2)$ computations

Gaussian projection

Produces an orthonormal matrix Q which approximates $\text{col}(W)$

1. Draw $n \times m$ Gaussian random matrix Ω.
2. Form $Y = Ł\Omega$.
3. Construct Q, an orthonormal matrix such that $\text{col}(Q) = \text{col}(Y)$
4. Form B such that B minimizes $||BQ^\top\Omega - Q^\top Y||_2$
5. Compute the eigenvector decomposition of B, ie: $B = \widehat{U}\widehat{\Sigma}\widehat{U}^\top$
6. Return $U^{gp} = Q\widehat{U}$.

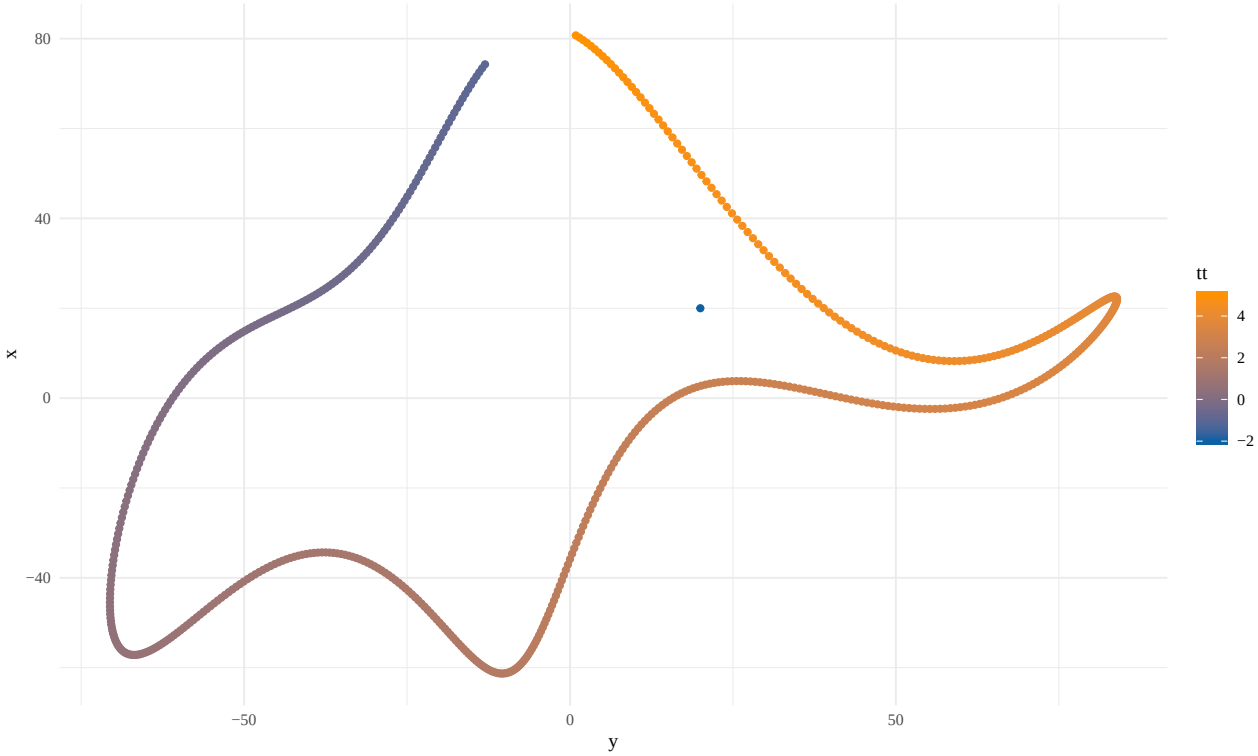
- Requires $O(n^2m)$ computations
- remains orthonormal

(Steps 3 and 4 with QR decomposition)

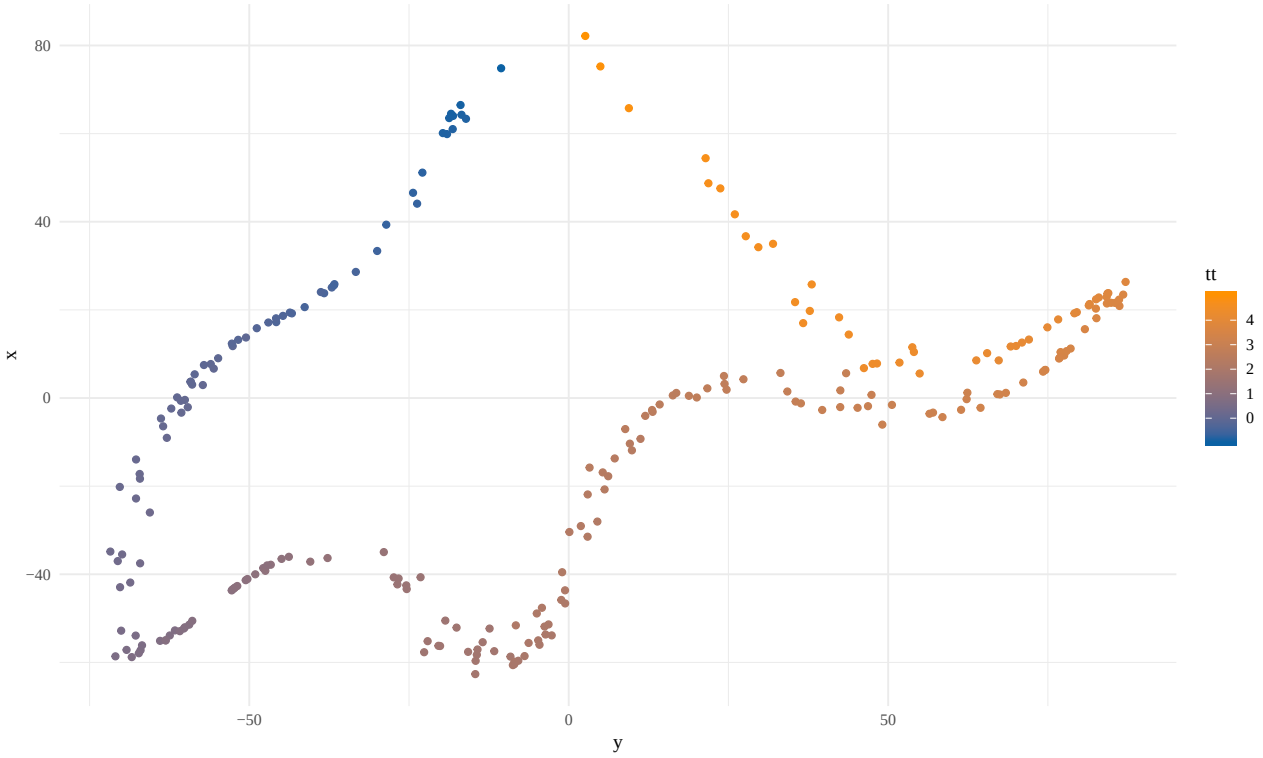
see Halko, Martinsson, and Tropp (2009)

Implementation

Here's our true manifold.

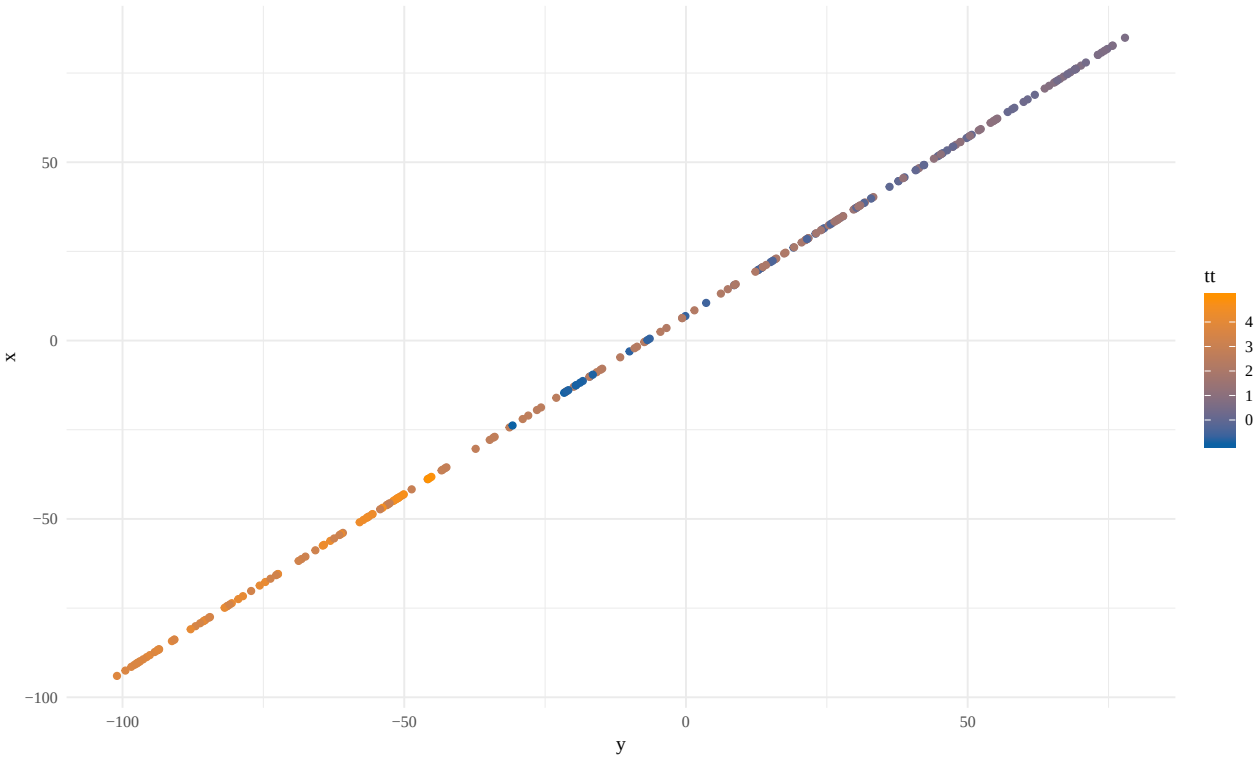


Data



PCA

```
Xbar = colMeans(X)
Xc = scale(X, center=Xbar, scale=FALSE) ## Decided not to scale
ee = eigen(tcrossprod(Xc))
Xhat = ee$vectors[,1] * sqrt(ee$values[1])
df_to_plot = data.frame(x=Xhat+Xbar[1], y=Xhat+Xbar[2], z=tt)
ggplot(df_to_plot, aes(x=y,y=x,col=tt)) + geom_point() +
  theme_minimal(base_family = 'serif') +
  scale_color_gradient(low=blue, high=orange)
```

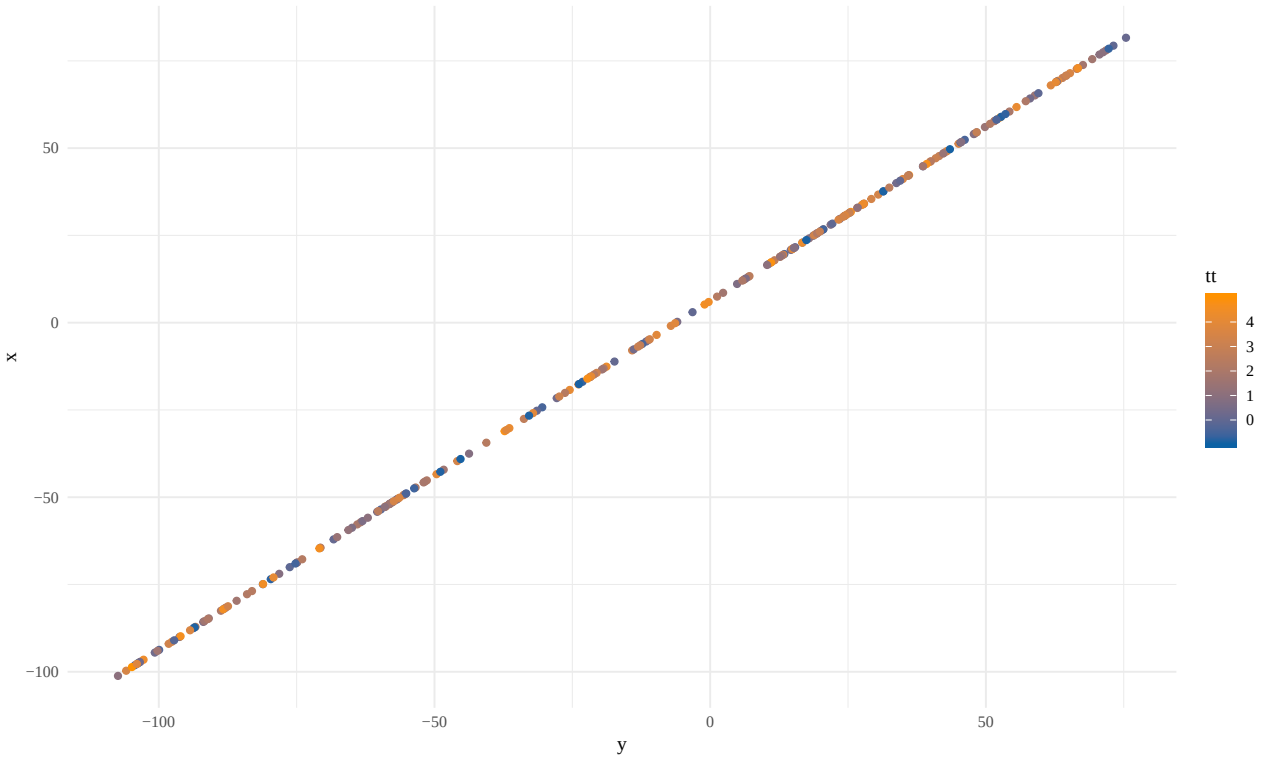


Not too good.

Approximate PCA?

Not really necessary since p is small. But we do it anyway.

```
select = sample.int(nrow(X), floor(nrow(X)/2))
Xsmall = X[select,]
Xrest = X[-select,]
Xbarsmall = colMeans(Xsmall)
Xcsmall = scale(Xsmall, center=Xbar, scale=FALSE) ## Decided not to scale
ee = eigen(tcrossprod(Xcsmall))
L21 = tcrossprod(scale(Xrest, center=Xbarsmall, scale=FALSE), Xcsmall)
Xhat = c(ee$vectors[,1], L21 %*% ee$vectors[,1]/ee$values[1]) * sqrt(ee$values[1])
df_to_plot = data.frame(x=Xhat+Xbarsmall[1], y=Xhat+Xbarsmall[2], z=tt)
ggplot(df_to_plot, aes(x=y,y=x,col=tt)) + geom_point() +
  theme_minimal(base_family = 'serif') +
  scale_color_gradient(low=blue, high=orange)
```

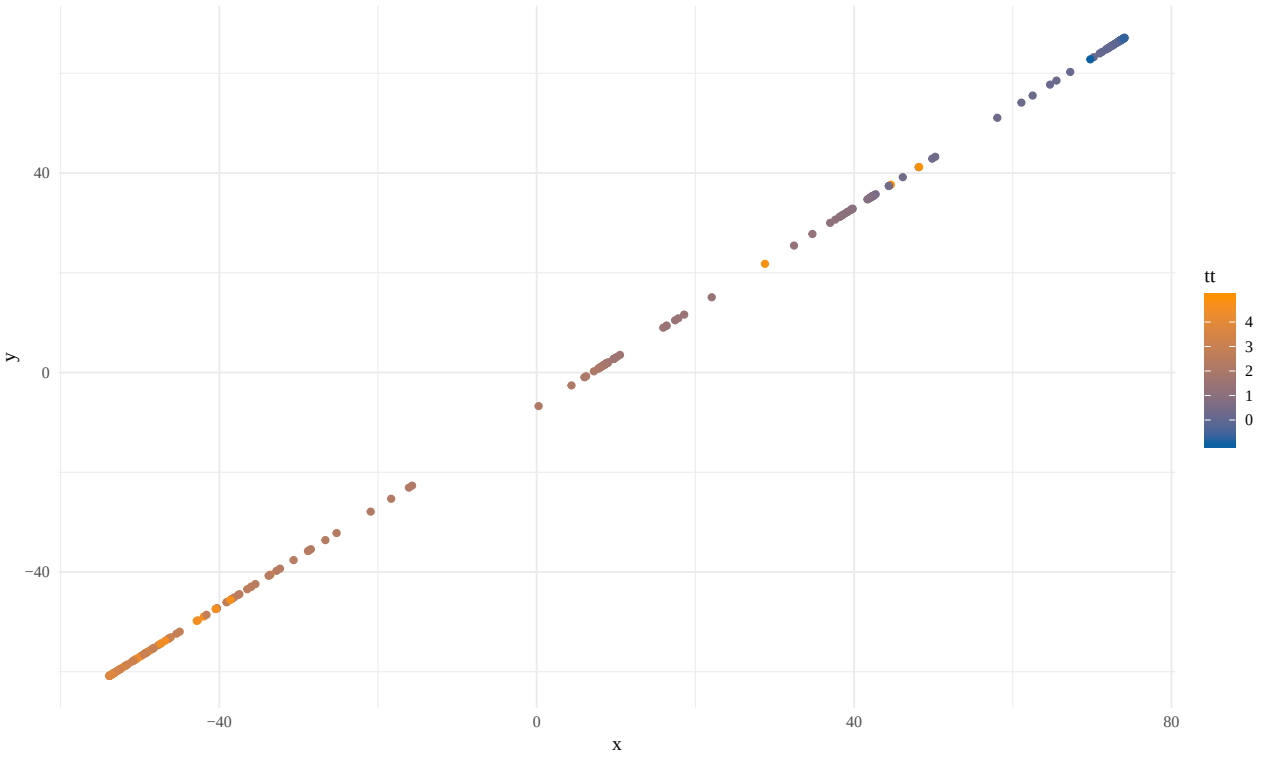


Laplacian eigenmaps

```
Dmat = as.matrix(dist(Xc))^2
K = exp(-Dmat/150) ## Why 150??
K[K<.1] = 0 ## sometimes this helps
L = diag(nrow(K)) - diag(1/rowSums(K)) %*% K
ee = eigen(L)
tail(ee$values,10)
```

```
##  [1]  1.258391e-01  9.575559e-02  8.554753e-02  4.080003e-02  2.914037e-02
##  [6]  1.345948e-02  1.052832e-02  4.125569e-03  1.274887e-03 -4.705437e-17
```

```
last = which.min(ee$values[ee$values>1e-12])
Xhat = ee$vectors[,last] / ee$values[last]
df_to_plot = data.frame(x=Xhat+Xbar[1], y=Xhat+Xbar[2], z=tt)
ggplot(df_to_plot, aes(x=x,y=y,col=tt)) + geom_point() +
  theme_minimal(base_family = 'serif') +
  scale_color_gradient(low=blue, high=orange)
```



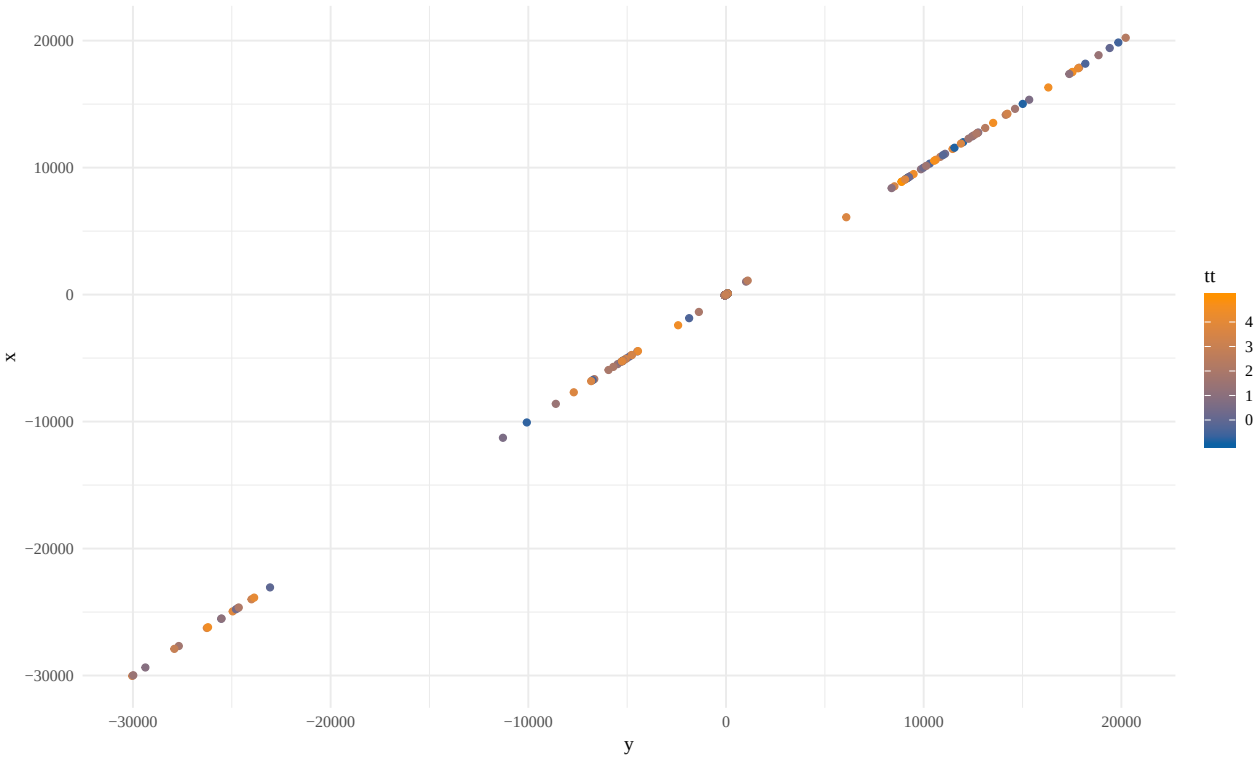
Still not great. (Try as I might)

Approximate LE?

```
Ks = K[select,select]
ls = length(select)
Ls = diag(ls) - diag(1/rowSums(Ks)) %*% Ks
ee = eigen(Ls)
ee$values = Re(ee$values) ##!
ee$vectors = Re(ee$vectors)
tail(ee$values)
```

```
## [1]  6.570249e-03  1.805218e-03  4.223347e-16 -1.636219e-16  3.432124e-17
## [6]  0.000000e+00
```

```
last = which.min(ee$values[ee$values>1e-12])
L21 = L[-select,select]
Xhat = c(ee$vectors[,last], L21 %*% ee$vectors[,last]/ee$values[last]) / ee$values[last]
df_to_plot = data.frame(x=Xhat+Xbar[1], y=Xhat+Xbar[2], z=tt)
ggplot(df_to_plot, aes(x=y,y=x,col=tt)) + geom_point() +
  theme_minimal(base_family = 'serif') +
  scale_color_gradient(low=blue, high=orange)
```



Pretty bad. I think I screwed something up.

Conclusions

1. Tuning parameter selection is critical
2. Construction of $\mathbb{K}$ is important: different similarity measures lead to drastically different solutions, often garbage
3. Gaussian projection gives orthogonal features (might work better in the LE result above)
4. Need some targeted theory