Kernels

Yuelin Wu

School of Mathematics
Sun Yat-sen University

October 22, 2020

Table of Contents

1 Introduction

2 Mercer's Theorem

3 The kernel trick

Table of Contents

1 Introduction

2 Mercer's Theorem

3 The kernel trick

Introduction

We have been assuming that each object can be represented as a fixed-size feature vector, typically of the form $\mathbf{x}_i \in \mathbb{R}^D$. However, for certain kinds of objects, it is not clear how to best represent them as fixed-sized feature vectors.

For example:

- A text document or protein sequence, which can be of variable length?
- A molecular structure, which has complex 3d geometry?
- An evolutionary tree, which has variable size and shape?

Introduction

One approach to such problems is to define a generative model for the data, and use the inferred latent representation and/or the parameters of the model as features, and then to plug these features in to standard methods.(eg. deep learning).

Another approach is to assume that we have some way of measuring the similarity between objects, that doesn't require preprocessing them into feature vector format. (For example, when comparing strings, we can compute the edit distance between them.)

Introduction

Let $\kappa(\mathbf{x}, \mathbf{x}') \geq 0$ be some measure of similarity between objects $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, where $\mathcal{X}$ is some abstract space; we will call κ a kernel function.

Kernel as the frequent used method to measure the similarity, should possess certain proporty:

- Symmetric: It's reasonable to assume the exchange invariance of the kernel function.
- Non-negative: We discover the equivalence between kernel function and the Gaussian process. We can define $\kappa(\mathbf{x}, \mathbf{x}')$ by the random process on $\mathcal{X}$, i.e.

$$\kappa\left(\mathbf{x}, \mathbf{x}'\right) := \text{Cov}(Y_{\mathbf{x}}, Y_{\mathbf{x}'}).$$

On the contrary, given the kernel κ, we can define a mean zero Gaussian process $(Y_x)_{x \in \mathcal{X}}$ uniquely s.t.

$$\text{Cov}(Y_{\mathbf{x}}, Y_{\mathbf{x}'}) = \kappa\left(\mathbf{x}, \mathbf{x}'\right) \Rightarrow \sum_{i,j} \kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) \geq 0.$$

Introduction

- Reproducing proporty: If the kernel function is symmetric and non-negative, the Moore-Aronszajn theorem indicated that there exists a unique Hilbert space $\mathcal{H}$ and a map $\phi : \mathcal{X} \to \mathcal{H}$ s.t.

$$\kappa(x, y) = \langle \phi(x), \phi(y) \rangle,$$

where $\mathcal{H}$ is a HIlbert space of functions valued in $\mathcal{X}$ (called reproducing kernel HIlbert space) and $\phi(x) = K(\cdot, x)$. ϕ is also called the feature map if we replace $\mathcal{H}$ by its distance preserving isomorphism representation:

$$f\left(x\right) \to \left(a_1, ..., a_k, ...\right),$$

where $f\left(x\right) = \sum_k a_k \phi_k(x)$ and (ϕ_k) are the eigen-functions of κ mentioned below.

Introduction

- Bochner theorem: Let ψ is continous on $\mathcal{X} = \mathbb{R}^n$ and:

$$\kappa(x, y) := \psi(x - y),$$

κ is non-negative if and only if there exists a Borel measure μ on $\mathcal{X}$ s.t.

$$\psi(x) = \int e^{itx} \mu(dt).$$

RBF kernels

The squared exponential kernel (SE kernel) or Gaussian kernel is defined by:

$$\kappa\left(\mathbf{x}, \mathbf{x}'\right) = \exp\left(-\frac{1}{2}\left(\mathbf{x}-\mathbf{x}'\right)^T \Sigma^{-1}\left(\mathbf{x}-\mathbf{x}'\right)\right)$$

If Σ is diagonal, this can be written as ARD kernel:

$$\kappa\left(\mathbf{x}, \mathbf{x}'\right) = \exp\left(-\frac{1}{2}\sum_{j=1}^{D}\frac{1}{\sigma_j^2}\left(x_j - x_j'\right)^2\right)$$

If Σ is spherical, we get the radial basis function or RBF kernel:

$$\kappa\left(\mathbf{x}, \mathbf{x}'\right) = \exp\left(-\frac{\left\|\mathbf{x}-\mathbf{x}'\right\|^2}{2\sigma^2}\right)$$

Here σ^2 is known as the bandwidth, it is only a function of $\|\mathbf{x}-\mathbf{x}'\|$.

Table of Contents

1 Introduction

2 Mercer's Theorem

3 The kernel trick

Mercer (positive definite) kernels

The kernel function satisfy the requirement that the Gram matrix, defined by

$$\mathbf{K} = \begin{pmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ & \vdots & \\ \kappa(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix}$$

be positive definite for any set of inputs $\{\mathbf{x}_i\}_{i=1}^{N}$. We call such a kernel a Mercer kernel, or positive definite kernel.

- The Gaussian kernel is a Mercer kernel by Bochner theorem.

Mercer (positive definite) kernels

If the Gram matrix is positive definite, we can compute an eigenvector decomposition of it as follows: $\mathbf{K} = \mathbf{U}^T \Lambda \mathbf{U}$, where Λ is a diagonal matrix of eigenvalues $\lambda_i > 0$.
Consider an element of K :

$$k_{ij} = \left(\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,i} \right)^T \left(\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,j} \right)$$

Let us define $\phi(\mathbf{x}_i) = \Lambda^{\frac{1}{2}} \mathbf{U}_{:,i}$. Then we can write

$$k_{ij} = \boldsymbol{\phi}(\mathbf{x}_i)^T \boldsymbol{\phi}(\mathbf{x}_j)$$

Mercer (positive definite) kernels

In general, if the kernel is Mercer, then there exists a function ϕ mapping $\mathbf{x} \in \mathcal{X}$ to $\mathbb{R}^D$ such that:

$$\kappa\left(\mathbf{x}, \mathbf{x}'\right) = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}\left(\mathbf{x}'\right)$$

where $\boldsymbol{\phi}$ depends on the eigen functions of κ (so D is a potentially infinite dimensional space).

Mercer (positive definite) kernels

For example, consider the (non-stationary) polynomial kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = \left(\gamma \mathbf{x}^T \mathbf{x}' + r\right)^M \text{ where } r > 0.$$

If $M = 2, \gamma = r = 1$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$, we have

$$\begin{aligned}\left(1 + \mathbf{x}^T \mathbf{x}'\right)^2 &= \left(1 + x_1 x_1' + x_2 x_2'\right)^2 \\ &= 1 + 2x_1 x_1' + 2x_2 x_2' + (x_1 x_1)^2 + \left(x_2 x_2'\right)^2 + 2x_1 x_1' x_2 x_2'\end{aligned}$$

This can be written as $\phi(\mathbf{x})^T \phi(\mathbf{x}')$, where
$\phi(\mathbf{x}) = \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2\right]^T$

So using this kernel is equivalent to working in a 6 dimensional feature space.
An example of a kernel that is not a Mercer kernel is the so-called sigmoid kernel, defined by:

$$\kappa(\mathbf{x}, \mathbf{x}') = \tanh\left(\gamma \mathbf{x}^T \mathbf{x}' + r\right)$$

Table of Contents

1 Introduction

2 Mercer's Theorem

3 The kernel trick

Kernel trick

- It's hard to define our feature vector in terms of kernels, $\phi(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_N)]$.
- Kernel trick: We can work with the original feature vectors x, but modify the algorithm so that it replaces all inner products of the form $\langle \mathbf{x}, \mathbf{x}' \rangle$ with a call to the kernel function, $\kappa(\mathbf{x}, \mathbf{x}')$.
- Note that we require that the kernel be a Mercer kernel for this trick to work.

Kernelized nearest neighbor classification

Recall that in a 1NN classifier (Section 1.4.2), we just need to compute the Euclidean distance of a test vector to all the training points, find the closest one, and look up its label. This can be kernelized by observing that:

$$\|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2 = \langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_{i'}, \mathbf{x}_{i'} \rangle - 2 \langle \mathbf{x}_i, \mathbf{x}_{i'} \rangle$$

This allows us to apply the nearest neighbor classifier to structured data objects.

Kernelized K-medoids clustering

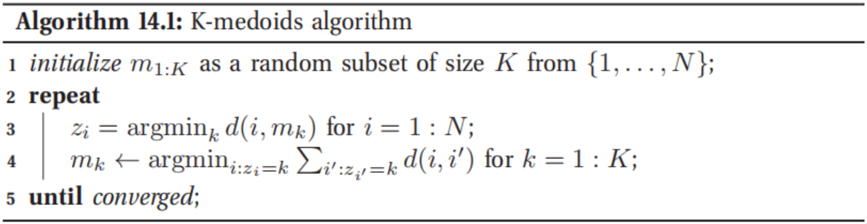
K-means clustering uses Euclidean distance to measure dissimilarity, which is not always appropriate for structured objects.

- The first step is to replace the K-means algorithm with the K-medoids algorothm.
- In K-means, we represent each cluster's centroid by the mean of all data vectors assigned to this cluster.
- In K-medoids, we we look at each object that belongs to the cluster, and measure the sum of its distances to all the others in the same cluster, and then pick the one which has the smallest sum:

$$m_k = \underset{i:z_i=k}{\operatorname{argmin}} \textstyle\sum_{i':z_{i'}=k} d\left(i, i'\right)$$
$$\text{where } d\left(i, i'\right) \triangleq \left\|\mathbf{x}_i - \mathbf{x}_{i'}\right\|_2^2$$

Kernelized K-medoids clustering

This takes $O\left(n_k^2\right)$ work per cluster, whereas K-means takes $O\left(n_k D\right)$ to update each cluster. The pseudo-code is given as follow:



Algorithm 14.1: K-medoids algorithm

```
1 initialize $m_{1:K}$ as a random subset of size $K$ from $\{1, \ldots, N\}$;
2 repeat
3     $z_i = \operatorname{argmin}_k d(i, m_k)$ for $i = 1 : N$;
4     $m_k \leftarrow \operatorname{argmin}_{i:z_i=k} \sum_{i':z_{i'}=k} d(i, i')$ for $k = 1 : K$;
5 until converged;
```

This algorithm can be kernelized by replacing the distance computation $d\left(i, i'\right)$.

Kernelized ridge regression

Applying the kernel trick to distance-based methods was straightforward. It is not so obvious how to apply it to parametric models such as ridge regression. However, it can be done, as we now explain. This will serve as a good “warm up” for studying SVMs.

The primal problem

Let $\mathbf{x} \in \mathbb{R}^D$ be some feature vector, and X be the corresponding $N \times D$ design matrix. We want to minimize

$$J(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda\|\mathbf{w}\|^2$$

The optimal solution is given by

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_D\right)^{-1}\mathbf{X}^T\mathbf{y} = \left(\sum \mathbf{x}_i\mathbf{x}_i^T + \lambda\mathbf{I}_D\right)^{-1}\mathbf{X}^T\mathbf{y}$$

How can we kernel without $\mathbf{x}_i^T\mathbf{x}_i$?

Matrix inversion lemma

Matrix inversion lemma(Corollary 4.3.1): Consider a general partitioned matrix $\mathbf{M} = \begin{pmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{pmatrix}$, assume E and H are invertible. We have

$$\left(\mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G}\right)^{-1} = \mathbf{E}^{-1} + \mathbf{E}^{-1}\mathbf{F}\left(\mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F}\right)^{-1}\mathbf{G}\mathbf{E}^{-1}$$
$$\left(\mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G}\right)^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}\left(\mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F}\right)^{-1}$$
$$\left|\mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G}\right| = \left|\mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F}\right|\left|\mathbf{H}^{-1}\right|\left|\mathbf{E}\right|$$

Use the second formula, let $\mathbf{H}^{-1} \triangleq \lambda^{-1}\mathbf{I}_N, \mathbf{F} \triangleq \mathbf{X}^{\mathrm{T}}, \mathbf{G} \triangleq -\mathbf{X}, \mathbf{E} \triangleq \mathbf{I}_D$ so

$$\begin{aligned}\left(\mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G}\right)^{-1}\mathbf{F}\mathbf{H}^{-1} &= \left(\mathbf{I}_D + \mathbf{X}^{\mathsf{T}}\lambda^{-1}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\lambda^{-1} \\ &= \left(\lambda\mathbf{I}_D + \mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathsf{T}}\end{aligned}$$

Thus

$$\mathbf{w} = \left(\mathbf{X}^{\mathsf{T}}\mathbf{X} + \lambda\mathbf{I}_D\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y} = \mathbf{X}^{\mathsf{T}}\left(\lambda\mathbf{I}_N + \mathbf{X}\mathbf{X}^{\mathsf{T}}\right)^{-1}\mathbf{y}$$

The dual problem

We rewrite the ridge estimate as follows:

$$\mathbf{w} = \mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T + \lambda \mathbf{I}_N\right)^{-1} \mathbf{y}$$

we see that we can partially kernelize this, by replacing $\mathbf{X}\mathbf{X}^T$ with the Gram matrix K. So we can define the following dual variables:

$$\boldsymbol{\alpha} \triangleq (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{y}$$

Then we can rewrite the primal variables as follows:

$$\mathbf{w} = \mathbf{X}^T \boldsymbol{\alpha} = \sum_{i=1}^N \alpha_i \mathbf{x}_i$$
$$\hat{f}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} = \sum_{i=1}^N \alpha_i \mathbf{x}_i^T \mathbf{x} = \sum_{i=1}^N \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i)$$

Computational cost

- The cost of computing the dual variables α is $O\left(N^3\right)$, whereas the cost of computing the primal variables w is $O\left(D^3\right)$. Hence the kernel method can be useful in high dimensional settings, even if we only use a linear kernel.
- However, prediction using the dual variables takes $O\left(ND\right)$ time, while prediction using the primal variables only takes $O\left(D\right)$ time. We can speedup prediction by making α sparse, as we discuss in Section 14.5.

The primal problem

The primal problem of PCA is

- Compute the covariance matrix
$$\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T = (1/N)\mathbf{X}^T\mathbf{X}.$$
- Finding the normalized eigenvectors $\mathbf{V}_L$ of $\mathbf{S}$, where $\mathbf{V}_L$ contains the L eigenvectors with largest eigenvalues.
- Furthermore, the optimal low-dimensional encoding of the data is given by $\hat{\mathbf{z}}_i = \mathbf{x}_i^T \mathbf{V}_L$

As we show below, we will produce a nonlinear embedding, using the kernel trick, a method known as kernel PCA.

The primal problem

First, let $\mathbf{U}$ be an orthogonal matrix containing the eigenvectors of $\mathbf{X}\mathbf{X}^T$(Not $\mathbf{S}$!) with corresponding eigenvalues in $\mathbf{\Lambda}$. By definition we have

$$\left(\mathbf{X}\mathbf{X}^T\right)\mathbf{U} = \mathbf{U}\mathbf{\Lambda}.$$

Pre-multiplying by $\mathbf{X}^T$ gives

$$\left(\mathbf{X}^T\mathbf{X}\right)\left(\mathbf{X}^T\mathbf{U}\right) = \left(\mathbf{X}^T\mathbf{U}\right)\mathbf{\Lambda}$$

The eigenvectors of $\mathbf{X}\mathbf{X}^T$ (and hence of $\mathbf{S}$) is $\mathbf{V} = \mathbf{X}^T\mathbf{U}$
However, these eigenvectors are not normalized, since

$$\|\mathbf{v}_j\|^2 = \mathbf{u}_j^T\mathbf{X}\mathbf{X}^T\mathbf{u}_j = \lambda_j\mathbf{u}_j^T\mathbf{u}_j = \lambda_j$$

So the normalized eigenvectors are given by $\mathbf{V}_{pca} = \mathbf{X}^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$

Kernel PCA

- Now let $\mathbf{K} = \mathbf{X}\mathbf{X}^T$ be the Gram matrix. So we are implicitly replacing $\mathbf{x}_i$ with $\phi(\mathbf{x}_i) = \phi_i$
- Let $\mathbf{\Phi}$ be the corresponding (notional) design matrix, and $\mathbf{S}_\phi = \frac{1}{N}\sum_i \phi_i\phi_i^T$ be the corresponding (notional) covariance matrix in feature space.
- The eigenvectors are given by $\mathbf{V}_{kpca} = \mathbf{\Phi}^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$ where $\mathbf{U}$ and $\mathbf{\Lambda}$ contain the eigenvectors and eigenvalues of $\mathbf{K}$.
- Of course, we can't actually compute $\mathbf{V}_{kpca}$, since ϕ_i is potentially infinite dimensional. However, we can compute the projection of a test vector x_* onto the feature space as follows:

$$\hat{z}_* = \phi_*^T\mathbf{V}_{kpca} = \phi_*^T\mathbf{\Phi}\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}} = \mathbf{k}_*^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$$

where $\mathbf{k}_* = [\kappa(\mathbf{x}_*, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}_*, \mathbf{x}_N)]$

Kernel PCA

We have assumed the projected data has zero mean, but it isn't in general. Define the centered feature vector as

$$\tilde{\phi}_i = \phi\left(\mathbf{x}_i\right) - \tfrac{1}{N}\textstyle\sum_{j=1}^{N} \phi\left(\mathbf{x}_j\right)$$

The Gram matrix of the centered feature vectors is given by

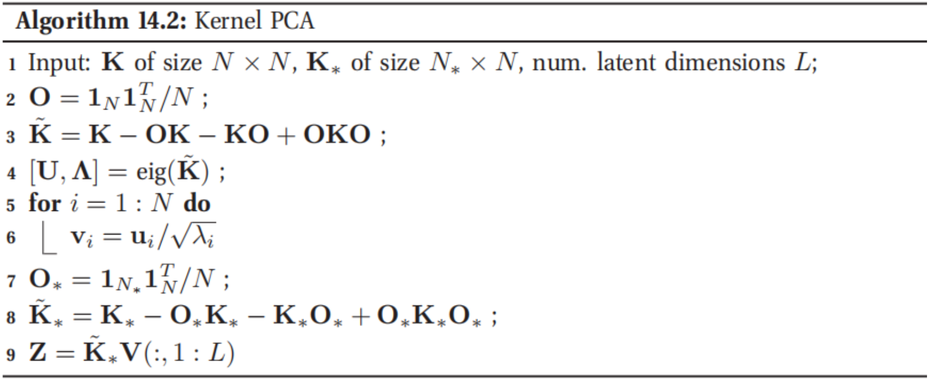
$$\begin{aligned}
\tilde{K}_{ij} &= \tilde{\phi}_i^T \tilde{\phi}_j \\
&= \phi_i^T \phi_j - \frac{1}{N}\sum_{k=1}^{N} \phi_i^T \phi_k - \frac{1}{N}\sum_{k=1}^{N} \phi_j^T \phi_k + \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{M} \phi_k^T \phi_l \\
&= \kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) - \frac{1}{N}\sum_{k=1}^{N} \kappa\left(\mathbf{x}_i, \mathbf{x}_k\right) - \frac{1}{N}\sum_{k=1}^{N} \kappa\left(\mathbf{x}_j, \mathbf{x}_k\right) + \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{M} \kappa\left(\mathbf{x}_k, \mathbf{x}_l\right)
\end{aligned}$$

This can be expressed in matrix notation as follows:

$$\mathbf{O} = \mathbf{1}_N \mathbf{1}_N^T / N$$

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{O}\mathbf{K} - \mathbf{K}\mathbf{O} + \mathbf{O}\mathbf{K}\mathbf{O}$$

Algorithm of KPCA



Algorithm 14.2: Kernel PCA

```
Input: $\mathbf{K}$ of size $N \times N$, $\mathbf{K}_*$ of size $N_* \times N$, num. latent dimensions $L$;
$\mathbf{O} = \mathbf{1}_N \mathbf{1}_N^T / N$ ;
$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{OK} - \mathbf{KO} + \mathbf{OKO}$ ;
$[\mathbf{U}, \mathbf{\Lambda}] = \mathrm{eig}(\tilde{\mathbf{K}})$ ;
for $i = 1 : N$ do
    $\mathbf{v}_i = \mathbf{u}_i / \sqrt{\lambda_i}$
$\mathbf{O}_* = \mathbf{1}_{N_*} \mathbf{1}_N^T / N$ ;
$\tilde{\mathbf{K}}_* = \mathbf{K}_* - \mathbf{O}_* \mathbf{K}_* - \mathbf{K}_* \mathbf{O}_* + \mathbf{O}_* \mathbf{K}_* \mathbf{O}_*$ ;
$\mathbf{Z} = \tilde{\mathbf{K}}_* \mathbf{V}(:, 1 : L)$
```

Discussion of KPCA

Whereas linear PCA is limited to using $L \leq D$ components, in kPCA, we can use up to N components, since the rank of $\mathbf{\Phi}$ is $N \times D^*$, where D^* is the (potentially infinite) dimensionality of embedded feature vectors.

Figure 14.7 gives an example of the method applied to some $D = 2$ dimensional data using an RBF kernel. We project points in the unit grid onto the first 8 components and visualize the corresponding surfaces using a contour plot. We see that the first two component separate the three clusters, and following components split the clusters.

Figure 14.7

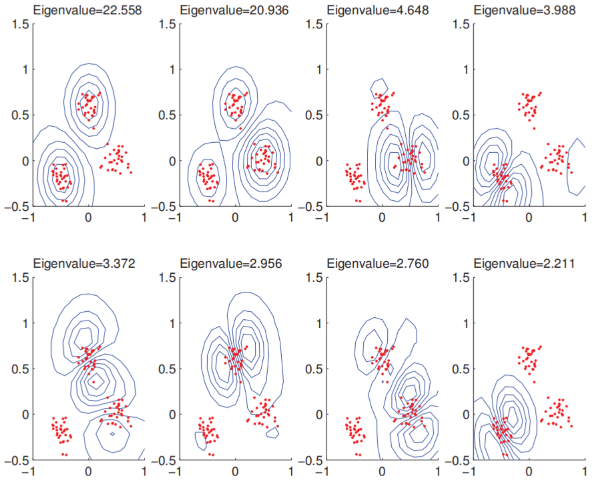


Figure 14.7 Visualization of the first 8 kernel principal component basis functions derived from some 2d data. We use an RBF kernel with $\sigma^2 = 0.1$. Figure generated by `kpcaScholkopf`, written by Bernhard Scholkopf.

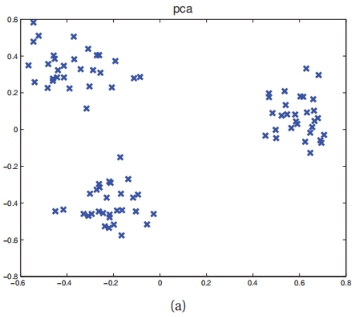
Discussion of KPCA

Although the features learned by kPCA can be useful for classification, they are not necessarily so useful for data visualization.

For example, Figure 14.8 shows the projection of the data from Figure 14.7 onto the first 2 principal bases computed using PCA and kPCA.

Obviously PCA perfectly represents the data. kPCA represents each cluster by a different line.

Figure 14.8



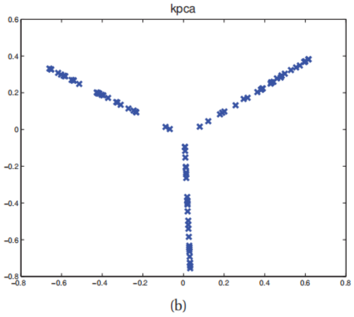


Figure 14.8 2d visualization of some 2d data. (a) PCA projection. (b) Kernel PCA projection. Figure generated by `kpcaDemo2`, based on code by L.J.P. van der Maaten.

Discussion of KPCA

Of course, there is no need to project 2d data back into 2d. So let us consider a different data-set.

We will use a 12 dimensional data set representing the three known phases of flow in an oil pipeline. (This data is widely used to compare data visualization methods)

We project this into 2d using PCA and kPCA (with an RBF kernel). The results are shown in Figure 14.9.

If we perform nearest neighbor classification in the low-dimensional space, kPCA makes 13 errors and PCA makes 20 (Lawrence.

Figure 14.9

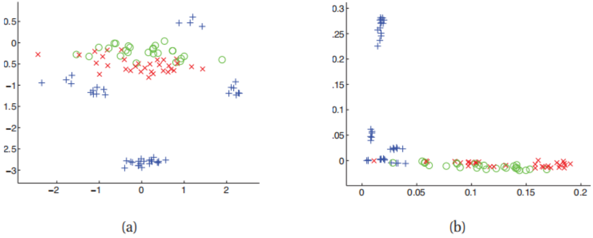


Figure 14.9 2d representation of 12 dimensional oil flow data. The different colors/symbols represent the 3 phases of oil flow. (a) PCA. (b) Kernel PCA with Gaussian kernel. Compare to Figure 15.10(b). From Figure 1 of (Lawrence 2005). Used with kind permission of Neil Lawrence.

Thanks !