M5MS10
Machine Learning

Spring 2018

Lecture 5

Dr Ben Calderhead

b.calderhead@imperial.ac.uk

M5MS10
Machine Learning

Spring 2018

Lecture 5

Dr Ben Calderhead

b.calderhead@imperial.ac.uk

IS10
Learning

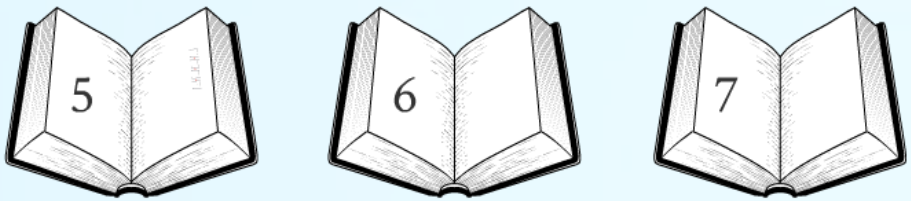
2018

re 5

lderhead

mperial.ac.uk

Non-parametric Methods



K-Nearest Neighbours

Non-parametric Classification

Until now we have considered *parametric* supervised learning approaches for both regression and classification, mainly based on linear models within a probabilistic framework.

In this lecture we shall now consider non-parametric approaches to *classification*.

In particular, we will investigate a model that is **not** motivated by the construction of a probabilistic model, K-Nearest Neighbours.

Non-parametric Approach

The term non-parametric has been used in many slightly different ways throughout the machine learning and statistics literature.

The terminology is perhaps slightly misleading however, since both parametric and non-parametric models do indeed have parameters!

It is useful to use the following definition from Prof. Yoshua Bengio (University of Montreal):

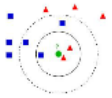
"We say that a learning algorithm is non-parametric if the complexity of the functions it can learn is allowed to grow as the amount of training data is increased"

K-Nearest Neighbours

A very simple method of classification that is not based on any model.

Instead it relies on measuring *distances* between each data point and a number of its neighbours.

Despite its simplicity, it works very well in many practical settings. We note that a probabilistic version of this is possible.

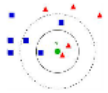


KNN Algorithm

Consider the case where each input is represented by a D dimensional feature vector, $\mathbf{x} \in \mathbb{R}$.

We can compute a *distance* between any two vectors in this space using a distance function, $d(\mathbf{x}_m, \mathbf{x}_n)$.

For a test input $\mathbf{x}_{new}$, we compute the distances to all other points and base our classification decision for y_{new} on a simple majority decision using the K smallest values in $\{d(\mathbf{x}_{new}, \mathbf{x}_n)\}_{n=\{1,\ldots,N\}}$.



KNN Comments

There is no model in KNN and so no training is needed!

All computational effort is expended when making a classification, and computational complexity is dominated by calculating distances from each training point.

When we use a Euclidean distance the computational scaling is *linear* in the dimensionality of the feature space and the number of training points, while the sorting procedure scales with order $N \log N$. (Remember the definition of non-parametric!)

Tunable Parameters for KNN

Although KNN is *non-parametric* and model-free, there are still parameters that need to be chosen somehow.

Clearly we need to decide upon which value K should take and we can do this by once again using cross validation.

The other variable we have is the choice of *distance function* - the choice of metric employed here may have a great effect on the performance of the algorithm.

Non-parametric Classification

Until now we have considered *parametric* supervised learning approaches for both regression and classification, mainly based on linear models within a probabilistic framework.

In this lecture we shall now consider non-parametric approaches to *classification*.

In particular, we will investigate a model that is **not** motivated by the construction of a probabilistic model, K-Nearest Neighbours.

Non-parametric Approach

The term non-parametric has been used in many slightly different ways throughout the machine learning and statistics literature.

The terminology is perhaps slightly misleading however, since both parametric and non-parametric models do indeed have parameters!

It is useful to use the following definition from Prof. Yoshua Bengio (University of Montreal):

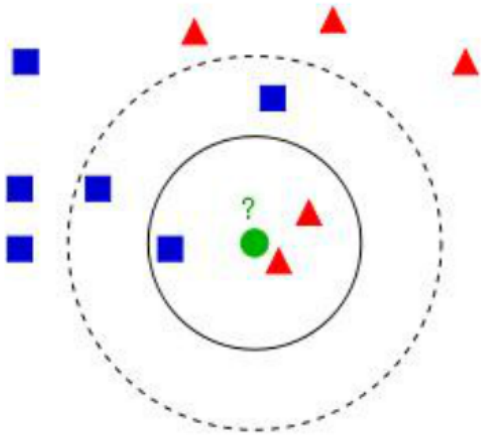
"We say that a learning algorithm is non-parametric if the complexity of the functions it can learn is allowed to grow as the amount of training data is increased"

K-Nearest Neighbours

A very simple method of classification that is not based on any model.

Instead it relies on measuring *distances* between each data point and a number of its neighbours.

Despite its simplicity, it works very well in many practical settings. We note that a probabilistic version of this is possible.

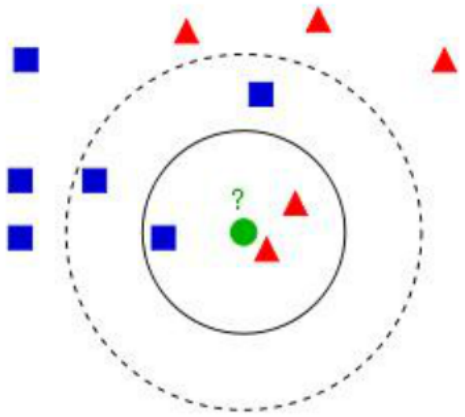


KNN Algorithm

Consider the case where each input is represented by a D dimensional feature vector, $\mathbf{x} \in \mathbb{R}$.

We can compute a *distance* between any two vectors in this space using a distance function, $d(\mathbf{x}_m, \mathbf{x}_n)$.

For a test input $\mathbf{x}_{\text{new}}$, we compute the distances to all other points and base our classification decision for y_{new} on a simple majority decision using the K smallest values in $[d(\mathbf{x}_{\text{new}}, \mathbf{x}_n)]_{n=[1,\ldots,N]}$.



KNN Comments

There is no model in KNN and so no training is needed!

All computational effort is expended when making a classification, and computational complexity is dominated by calculating distances from each training point.

When we use a Euclidean distance the computational scaling is *linear* in the dimensionality of the feature space and the number of training points, while the sorting procedure scales with order $N \log N$. (Remember the definition of non-parametric!)

Tunable Parameters for KNN

Although KNN is *non-parametric* and model-free, there are still parameters that need to be chosen somehow.

Clearly we need to decide upon which value K should take and we can do this by once again using cross validation.

The other variable we have is the choice of *distance function* - the choice of metric employed here may have a great effect on the performance of the algorithm.

K-Nearest Neighbours

Non-parametric Classification

Until now we have considered *parametric* supervised learning approaches for both regression and classification, mainly based on linear models within a probabilistic framework.

In this lecture we shall now consider non-parametric approaches to *classification*.

In particular, we will investigate a model that is **not** motivated by the construction of a probabilistic model, K-Nearest Neighbours.

Non-parametric Approach

The term non-parametric has been used in many slightly different ways throughout the machine learning and statistics literature.

The terminology is perhaps slightly misleading however, since both parametric and non-parametric models do indeed have parameters!

It is useful to use the following definition from Prof. Yoshua Bengio (University of Montreal):

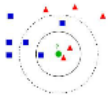
"We say that a learning algorithm is non-parametric if the complexity of the functions it can learn is allowed to grow as the amount of training data is increased"

K-Nearest Neighbours

A very simple method of classification that is not based on any model.

Instead it relies on measuring *distances* between each data point and a number of its neighbours.

Despite its simplicity, it works very well in many practical settings. We note that a probabilistic version of this is possible.

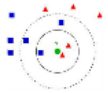


KNN Algorithm

Consider the case where each input is represented by a D dimensional feature vector, $\mathbf{x} \in \mathbb{R}$.

We can compute a *distance* between any two vectors in this space using a distance function, $d(\mathbf{x}_m, \mathbf{x}_n)$.

For a test input $\mathbf{x}_{new}$, we compute the distances to all other points and base our classification decision for y_{new} on a simple majority decision using the K smallest values in $\{d(\mathbf{x}_{new}, \mathbf{x}_n)\}_{n=\{1,\ldots,N\}}$.



KNN Comments

There is no model in KNN and so no training is needed!

All computational effort is expended when making a classification, and computational complexity is dominated by calculating distances from each training point.

When we use a Euclidean distance the computational scaling is *linear* in the dimensionality of the feature space and the number of training points, while the sorting procedure scales with order $N \log N$. (Remember the definition of non-parametric!)

Tunable Parameters for KNN

Although KNN is *non-parametric* and model-free, there are still parameters that need to be chosen somehow.

Clearly we need to decide upon which value K should take and we can do this by once again using cross validation.

The other variable we have is the choice of *distance function* - the choice of metric employed here may have a great effect on the performance of the algorithm.

Measures of Distance

Distance Functions

We need an appropriate way of measuring distance between input vectors that defines how close (or similar) two input are to each other.

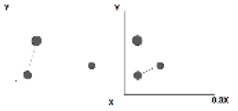
Our distance function must satisfy the following mathematical properties, for all vectors $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}$:

- **Non-negativity:** $d(\mathbf{x}, \mathbf{y}) \geq 0$
- **Reflexivity:** $d(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} = \mathbf{y}$
- **Symmetry:** $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$
- **Triangle Inequality:** $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \geq d(\mathbf{x}, \mathbf{z})$

Distance Functions

Euclidean distance $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{d=1}^{D} (x_d - y_d)^2}$ satisfies all of these properties and so can be used as a distance metric in $\mathbb{R}^D$.

If however we consider a non-Euclidean metric, such that we scale the measures of distance in different dimensions, then the relations between points can be very different.



Rescaling the Data

Since each dimension can have different scales, it is common practice to normalise the vectors. If there are N data points, we can normalise them to have zero mean simply by subtracting the sample mean,

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n$$

Similarly, we can rescale the data set such that each axis also has a common variance of 1,

$$\hat{\mathbf{x}}_i = \frac{\tilde{\mathbf{x}}_i}{\sqrt{\frac{1}{N} \sum_{n=1}^{N} \tilde{\mathbf{x}}_{ni}^2}}$$

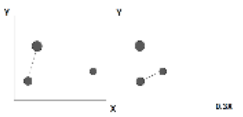
Consider the use of distance in PCA - different scalings will affect the variance of projections...

Distance Measures

Instead of explicitly rescaling the data, we can incorporate that directly into the distance metric, for example,

$$d(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})^T \Sigma (\mathbf{x} - \mathbf{y})$$

where Σ is some local linear transformation, for example based on a number of nearest neighbours, that distorts the local Euclidean space.



Distance Measures

Another commonly used measure of distance is the Minkowski family of metrics, defined by

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{d=1}^{D} (x_d - y_d)^p \right)^{\frac{1}{p}}$$

We note that when $p = 1$ this metric is known as the $L1$ norm, or the *Manhattan block* distance, since it measures distance in terms of the shortest paths that run parallel to the axes.

When $p = \infty$ then the L_∞ norm defines the distance between $\mathbf{x}$ and $\mathbf{y}$ as the maximum distance out of each of the D axes.

Distance Functions

We need an appropriate way of measuring distance between input vectors that defines how close (or similar) two input are to each other.

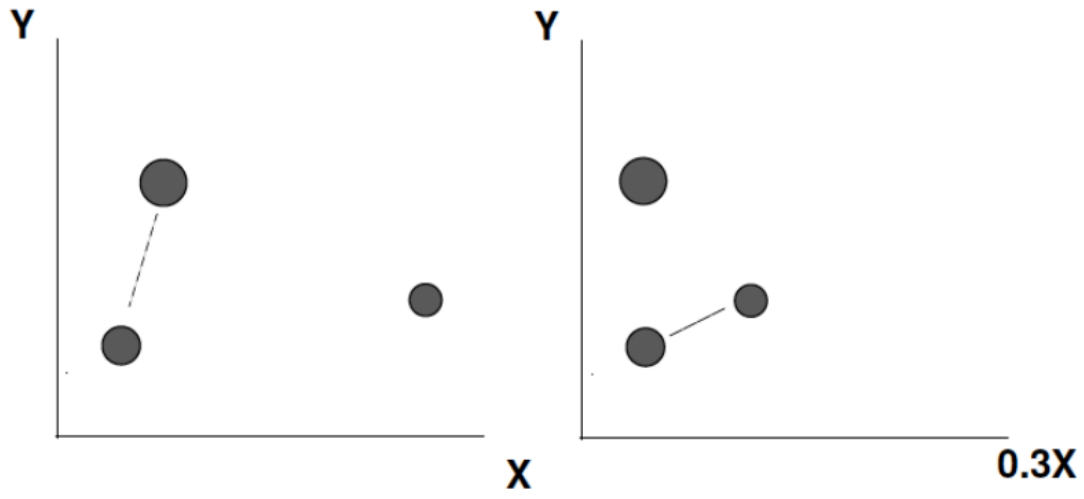
Our distance function must satisfy the following mathematical properties, for all vectors $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}$:

- **Non-negativity:** $d(\mathbf{x}, \mathbf{y}) \geq 0$
- **Reflexivity:** $d(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} = \mathbf{y}$
- **Symmetry:** $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$
- **Triangle Inequality:** $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \geq d(\mathbf{x}, \mathbf{z})$

Distance Functions

Euclidean distance $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{d=1}^{D}(x_d - y_d)^2}$ satisfies all of these properties and so can be used as a distance metric in $\mathbb{R}^D$.

If however we consider a non-Euclidean metric, such that we scale the measures of distance in different dimensions, then the relations between points can be very different.



Rescaling the Data

Since each dimension can have different scales, it is common practice to normalise the vectors. If there are N data points, we can normalise them to have zero mean simply by subtracting the sample mean,

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$

Similarly, we can rescale the data set such that each axis also has a common variance of 1,

$$\tilde{\tilde{\mathbf{x}}}_i = \frac{\tilde{\mathbf{x}}_i}{\sqrt{\frac{1}{N}\sum_{n=1}^{N}\tilde{\mathbf{x}}_n^2}}$$

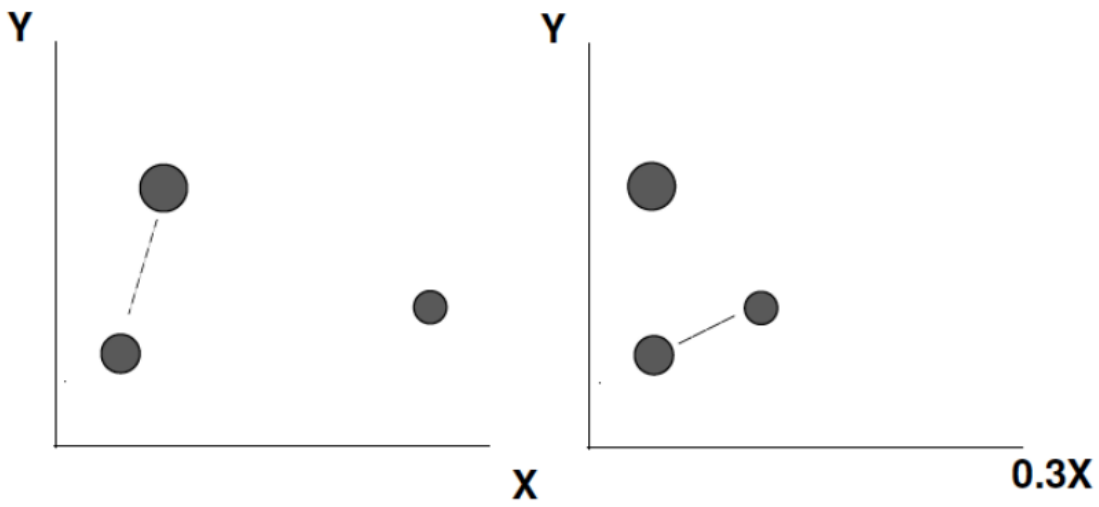
Consider the use of distance in PCA - different scalings will affect the variance of projections...

Distance Measures

Instead of explicitly rescaling the data, we can incorporate that directly into the distance metric, for example,

$$d(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})^T \Sigma (\mathbf{x} - \mathbf{y})$$

where Σ is some local linear transformation, for example based on a number of nearest neighbours, that distorts the local Euclidean space.



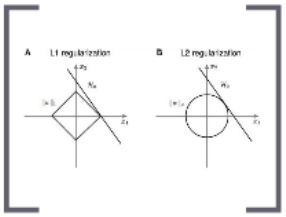
Distance Measures

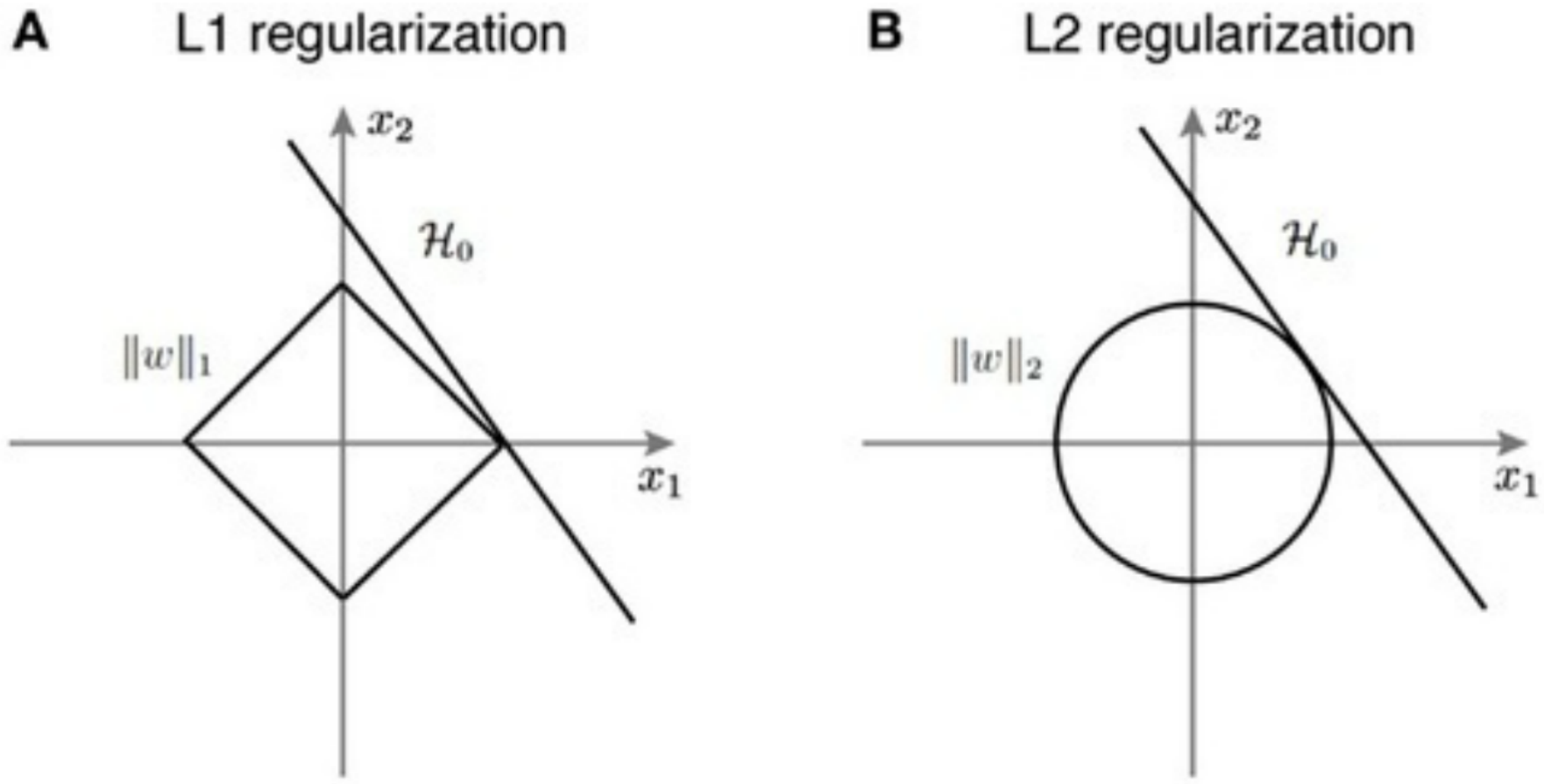
Another commonly used measure of distance is the Minkowski family of metrics, defined by

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{d=1}^{D} (x_d - y_d)^p \right)^{\frac{1}{p}}$$

We note that when $p = 1$ this metric is known as the $L1$ norm, or the *Manhattan block* distance, since it measures distance in terms of the shortest paths that run parallel to the axes.

When $p = \infty$ then the L_∞ norm defines the distance between $\mathbf{x}$ and $\mathbf{y}$ as the maximum distance out of each of the D axes.




A
L1 regularization
x_2
$\mathcal{H}_0$
$\|w\|_1$
x_1
B
L2 regularization
x_2
$\mathcal{H}_0$
$\|w\|_2$
x_1

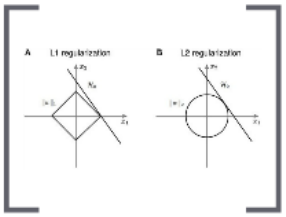
Distance Measures

Another commonly used measure of distance is the Minkowski family of metrics, defined by

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{d=1}^{D} (x_d - y_d)^p \right)^{\frac{1}{p}}$$

We note that when $p = 1$ this metric is known as the $L1$ norm, or the *Manhattan block* distance, since it measures distance in terms of the shortest paths that run parallel to the axes.

When $p = \infty$ then the L_∞ norm defines the distance between $\mathbf{x}$ and $\mathbf{y}$ as the maximum distance out of each of the D axes.



Measures of Distance

Distance Functions

We need an appropriate way of measuring distance between input vectors that defines how close (or similar) two input are to each other.

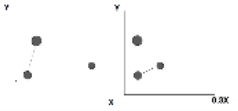
Our distance function must satisfy the following mathematical properties, for all vectors $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}$:

- **Non-negativity:** $d(\mathbf{x}, \mathbf{y}) \geq 0$
- **Reflexivity:** $d(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} = \mathbf{y}$
- **Symmetry:** $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$
- **Triangle Inequality:** $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \geq d(\mathbf{x}, \mathbf{z})$

Distance Functions

Euclidean distance $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{d=1}^{D} (x_d - y_d)^2}$ satisfies all of these properties and so can be used as a distance metric in $\mathbb{R}^D$.

If however we consider a non-Euclidean metric, such that we scale the measures of distance in different dimensions, then the relations between points can be very different.



Rescaling the Data

Since each dimension can have different scales, it is common practice to normalise the vectors. If there are N data points, we can normalise them to have zero mean simply by subtracting the sample mean,

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n$$

Similarly, we can rescale the data set such that each axis also has a common variance of 1,

$$\hat{\mathbf{x}}_i = \frac{\tilde{\mathbf{x}}_i}{\sqrt{\frac{1}{N} \sum_{n=1}^{N} \tilde{\mathbf{x}}_{ni}^2}}$$

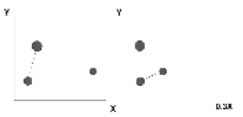
Consider the use of distance in PCA - different scalings will affect the variance of projections...

Distance Measures

Instead of explicitly rescaling the data, we can incorporate that directly into the distance metric, for example,

$$d(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})^T \Sigma (\mathbf{x} - \mathbf{y})$$

where Σ is some local linear transformation, for example based on a number of nearest neighbours, that distorts the local Euclidean space.



Distance Measures

Another commonly used measure of distance is the Minkowski family of metrics, defined by

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{d=1}^{D} (x_d - y_d)^p \right)^{\frac{1}{p}}$$

We note that when $p = 1$ this metric is known as the $L1$ norm, or the *Manhattan block* distance, since it measures distance in terms of the shortest paths that run parallel to the axes.

When $p = \infty$ then the L_∞ norm defines the distance between $\mathbf{x}$ and $\mathbf{y}$ as the maximum distance out of each of the D axes.

The Kernel Trick

The Kernel Trick

The so called kernel trick is an incredibly useful concept for building machine learning algorithms, and we shall see it is fundamental for many different machine learning algorithms.

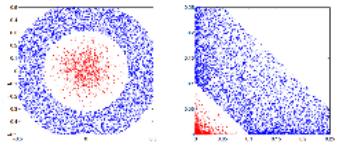
We have already come across the idea of working with some nonlinear transformation of the input vectors (or features), rather than trying to classify with them directly.

For example, we could use a polynomial (or other indeed other nonlinear transformation) such that

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

The Kernel Trick

Here we see that if we have a complicated data set, with one class forming an annular ring around the other, we may employ a *quadratic transformation* using a second order polynomial such that the two classes become *linearly separable*.



Inner Products

Let's consider what happens to the distance function when we employ some nonlinear transformation of the data.

The standard Euclidean squared distance follows simply as,

$$d(\mathbf{x}_1, \mathbf{x}_2)^2 = (\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$$

If however we first transform the data, we see that the squared distance measure in the transformed space will be,

$$\begin{aligned} d(\phi(\mathbf{x}_1), \phi(\mathbf{x}_2))^2 &= (\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2))^T(\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2)) \\ &= \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) + \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) - 2\phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) \end{aligned}$$

which can be calculated using only *inner products* in the transformed space.

Kernel Functions

A kernel function is defined for all $\mathbf{x}_1$ and $\mathbf{x}_2$ in some feature space, e.g. $\mathbb{R}^D$, such that

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2)$$

ϕ defines a mapping from the original feature space to an *inner product feature space* $\mathcal{F}$, i.e. $\phi : \mathbf{x} \rightarrow \phi(\mathbf{x}) \in \mathcal{F}$.

We note that K is a positive, semi-definite function.

Kernel Function Example

Let's have a look at an example of the kernel trick. We'll assume $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^2$ and define the mapping

$$\phi : \mathbf{a} = (a_1, a_2)^T \rightarrow \phi(\mathbf{a}) = (a_1^2, a_2^2, \sqrt{2}a_1a_2)^T \in \mathcal{F} = \mathbb{R}^3$$

So we can evaluate the *inner-product* in the transformed input space $\mathcal{F} = \mathbb{R}^3$ as

$$\begin{aligned} \phi(\mathbf{a})^T\phi(\mathbf{b}) &= (a_1^2, a_2^2, \sqrt{2}a_1a_2)(b_1^2, b_2^2, \sqrt{2}b_1b_2)^T \\ &= a_1^2b_1^2 + a_2^2b_2^2 + 2a_1a_2b_1b_2 \\ &= (a_1b_1 + a_2b_2)^2 \\ &= (\mathbf{a}^T\mathbf{b})^2 \end{aligned}$$

The Kernel Trick

We can therefore define the kernel function in terms of inputs in the original space,

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

We therefore have no need to explicitly compute the mapping every time we want to compute the inner product in the transformed space!

Furthermore, we can even compute distances in the transformed without the need to explicitly transform each of the variables, since

$$\begin{aligned} d(\phi(\mathbf{a}), \phi(\mathbf{b}))^2 &= (\phi(\mathbf{a}) - \phi(\mathbf{b}))^T(\phi(\mathbf{a}) - \phi(\mathbf{b})) \\ &= K(\mathbf{a}, \mathbf{a}) + K(\mathbf{b}, \mathbf{b}) - 2K(\mathbf{a}, \mathbf{b}) \end{aligned}$$

The Kernel Trick

The so called kernel trick is an incredibly useful concept for building machine learning algorithms, and we shall see it is fundamental for many different machine learning algorithms.

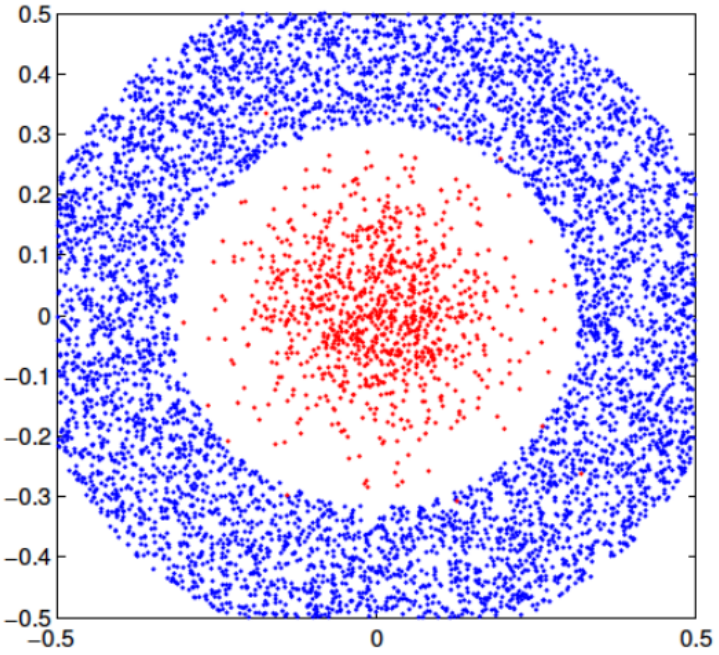
We have already come across the idea of working with some nonlinear transformation of the input vectors (or features), rather than trying to classify with them directly.

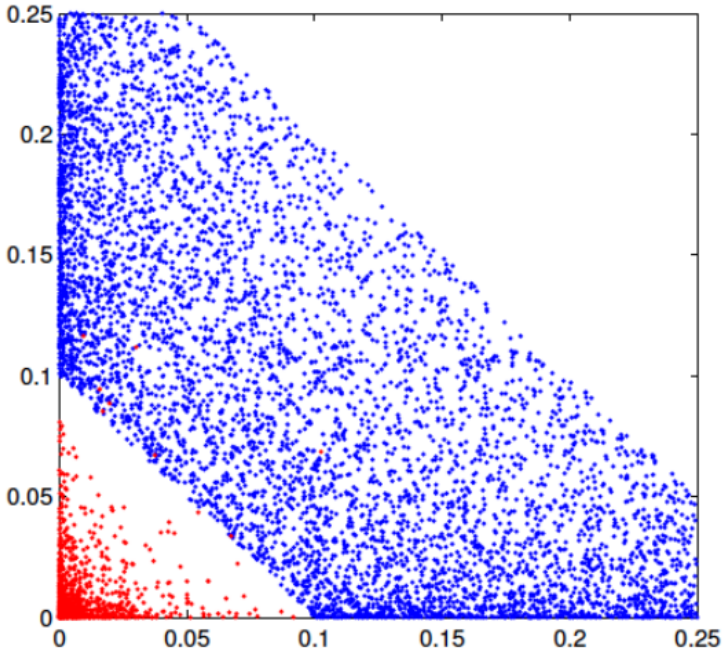
For example, we could use a polynomial (or other indeed other nonlinear transformation) such that

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

The Kernel Trick

Here we see that if we have a complicated data set, with one class forming an annular ring around the other, we may employ a *quadratic transformation* using a second order polynomial such that the two classes become *linearly separable*.

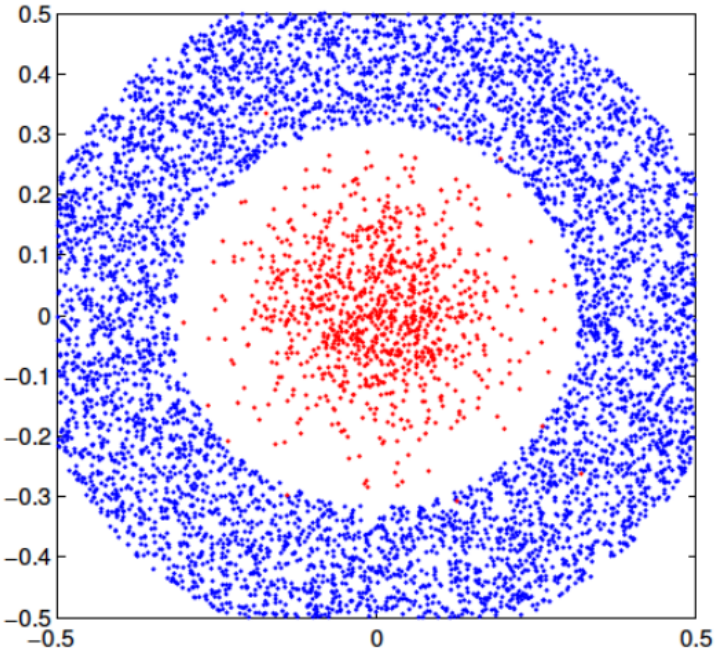


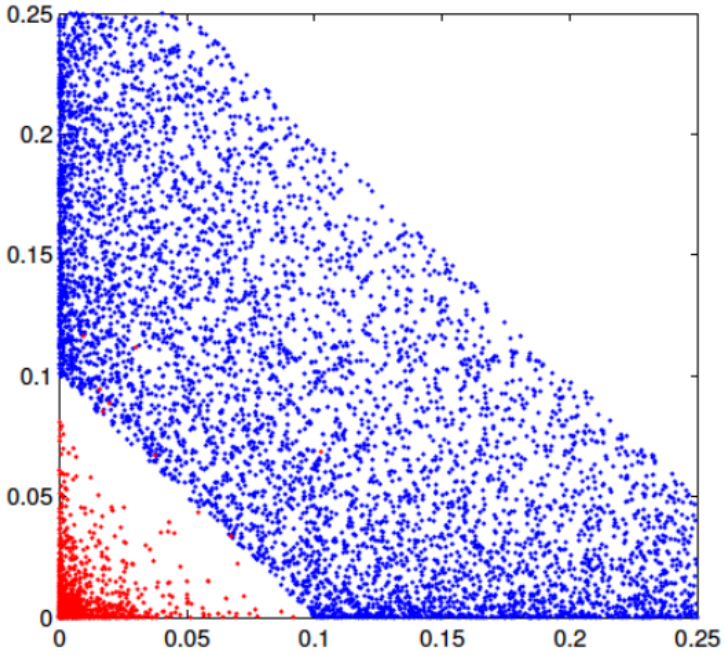


Missing video

The Kernel Trick

Here we see that if we have a complicated data set, with one class forming an annular ring around the other, we may employ a *quadratic transformation* using a second order polynomial such that the two classes become *linearly separable*.





Inner Products

Let's consider what happens to the distance function when we employ some nonlinear transformation of the data.

The standard Euclidean squared distance follows simply as,

$$d(\mathbf{x}_1, \mathbf{x}_2)^2 = (\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$$

If however we first transform the data, we see that the squared distance measure in the transformed space will be,

$$\begin{aligned} d(\phi(\mathbf{x}_1), \phi(\mathbf{x}_2))^2 &= (\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2))^T(\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2)) \\ &= \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) + \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) - 2\phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) \end{aligned}$$

which can be calculated using only *inner products* in the transformed space.

Kernel Functions

A kernel function is defined for all $\mathbf{x}_1$ and $\mathbf{x}_2$ in some feature space, e.g. $\mathbb{R}^D$, such that

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1)^T \phi(\mathbf{x}_2)$$

ϕ defines a mapping from the original feature space to an *inner product feature space* $\mathcal{F}$, i.e. $\phi : \mathbf{x} \rightarrow \phi(\mathbf{x}) \in \mathcal{F}$.

We note that K is a positive, semi-definite function.

Kernel Function Example

Let's have a look at an example of the kernel trick. We'll assume $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^2$ and define the mapping

$$\phi : \mathbf{a} = (a_1, a_2)^T \rightarrow \phi(\mathbf{a}) = (a_1^2, a_2^2, \sqrt{2}a_1 a_2)^T \in \mathcal{F} = \mathbb{R}^3$$

So we can evaluate the *inner-product* in the transformed input space $\mathcal{F} = \mathbb{R}^3$ as

$$\begin{aligned} \phi(\mathbf{a})^T \phi(\mathbf{b}) &= (a_1^2, a_2^2, \sqrt{2}a_1 a_2)(b_1^2, b_2^2, \sqrt{2}b_1 b_2)^T \\ &= a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1 a_2 b_1 b_2 \\ &= (a_1 b_1 + a_2 b_2)^2 \\ &= (\mathbf{a}^T \mathbf{b})^2 \end{aligned}$$

The Kernel Trick

We can therefore define the kernel function in terms of inputs in the original space,

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

We therefore have no need to explicitly compute the mapping every time we want to compute the inner product in the transformed space!

Furthermore, we can even compute distances in the transformed without the need to explicitly transform each of the variables, since

$$\begin{aligned} d(\phi(\mathbf{a}), \phi(\mathbf{b}))^2 &= (\phi(\mathbf{a}) - \phi(\mathbf{b}))^T(\phi(\mathbf{a}) - \phi(\mathbf{b})) \\ &= K(\mathbf{a}, \mathbf{a}) + K(\mathbf{b}, \mathbf{b}) - 2K(\mathbf{a}, \mathbf{b}) \end{aligned}$$

The Kernel Trick

The Kernel Trick

The so called kernel trick is an incredibly useful concept for building machine learning algorithms, and we shall see it is fundamental for many different machine learning algorithms.

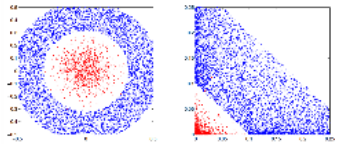
We have already come across the idea of working with some nonlinear transformation of the input vectors (or features), rather than trying to classify with them directly.

For example, we could use a polynomial (or other indeed other nonlinear transformation) such that

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

The Kernel Trick

Here we see that if we have a complicated data set, with one class forming an annular ring around the other, we may employ a *quadratic transformation* using a second order polynomial such that the two classes become *linearly separable*.



Inner Products

Let's consider what happens to the distance function when we employ some nonlinear transformation of the data.

The standard Euclidean squared distance follows simply as,

$$d(\mathbf{x}_1, \mathbf{x}_2)^2 = (\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$$

If however we first transform the data, we see that the squared distance measure in the transformed space will be,

$$\begin{aligned} d(\phi(\mathbf{x}_1), \phi(\mathbf{x}_2))^2 &= (\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2))^T(\phi(\mathbf{x}_1) - \phi(\mathbf{x}_2)) \\ &= \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) + \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) - 2\phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) \end{aligned}$$

which can be calculated using only *inner products* in the transformed space.

Kernel Functions

A kernel function is defined for all $\mathbf{x}_1$ and $\mathbf{x}_2$ in some feature space, e.g. $\mathbb{R}^D$, such that

$$K(\mathbf{x}_1, \mathbf{x}_2) = \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2)$$

ϕ defines a mapping from the original feature space to an *inner product feature space* $\mathcal{F}$, i.e. $\phi : \mathbf{x} \rightarrow \phi(\mathbf{x}) \in \mathcal{F}$.

We note that K is a positive, semi-definite function.

Kernel Function Example

Let's have a look at an example of the kernel trick. We'll assume $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^2$ and define the mapping

$$\phi : \mathbf{a} = (a_1, a_2)^T \rightarrow \phi(\mathbf{a}) = (a_1^2, a_2^2, \sqrt{2}a_1a_2)^T \in \mathcal{F} = \mathbb{R}^3$$

So we can evaluate the *inner-product* in the transformed input space $\mathcal{F} = \mathbb{R}^3$ as

$$\begin{aligned} \phi(\mathbf{a})^T\phi(\mathbf{b}) &= (a_1^2, a_2^2, \sqrt{2}a_1a_2)(b_1^2, b_2^2, \sqrt{2}b_1b_2)^T \\ &= a_1^2b_1^2 + a_2^2b_2^2 + 2a_1a_2b_1b_2 \\ &= (a_1b_1 + a_2b_2)^2 \\ &= (\mathbf{a}^T\mathbf{b})^2 \end{aligned}$$

The Kernel Trick

We can therefore define the kernel function in terms of inputs in the original space,

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

We therefore have no need to explicitly compute the mapping every time we want to compute the inner product in the transformed space!

Furthermore, we can even compute distances in the transformed without the need to explicitly transform each of the variables, since

$$\begin{aligned} d(\phi(\mathbf{a}), \phi(\mathbf{b}))^2 &= (\phi(\mathbf{a}) - \phi(\mathbf{b}))^T(\phi(\mathbf{a}) - \phi(\mathbf{b})) \\ &= K(\mathbf{a}, \mathbf{a}) + K(\mathbf{b}, \mathbf{b}) - 2K(\mathbf{a}, \mathbf{b}) \end{aligned}$$

Kernel PCA

PCA Overview

Strengths:

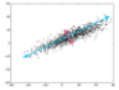
- Method based simply on calculating Eigenvectors
- The aren't any tuning of parameters required
- No local optima - unique solution

Weaknesses:

- Limited to second order statistics
- Limited to linear projections

PCA

Given a set of n centred observations, the first principal component is direction that maximises the variance of the the data.



$$X = (x_1, x_2, ..., x_n) \qquad \mathbf{u}_1 = \text{argmax}_{\|\mathbf{u}\|=1}\left[\frac{1}{N}\sum_{i=1}^{N}(\mathbf{u}^T\mathbf{x}_i)^2\right]$$

We can then calculate this direction by calculating the eigenvector corresponding to the largest eigenvector of this covariance matrix. $S_1 = \frac{1}{N}\mathbf{X}\mathbf{X}^T$

We note that the (i,j)th entry of the covariance matrix is the inner product of the i'th datapoint with the j'th datapoint.

Alternative Expression for PCA

The principal component lies in the span of the data, such that

$$\mathbf{u}_1 = \mathbf{X}\alpha$$

We can therefore rewrite the eigenevector equation in terms of the above alternative expression for the principal component,

$$\mathbf{S}_1\mathbf{u}_1 = \frac{1}{n}\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}\alpha$$

And left-multiplying both sides, we obtain

$$\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}^T\mathbf{X}\alpha$$

Kernel PCA

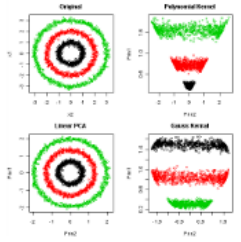
Main Idea: Replace the inner product matrix with the kernel matrix

PCA: $\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}^T\mathbf{X}\alpha$

Kernel PCA: $\frac{1}{n}K(\mathbf{X},\mathbf{X})\alpha = \lambda\alpha$

In other words, we can form the nxn kernel matrix K, then compute the eigendecomposition of K!

Kernel PCA Example



PCA Overview

Strengths:

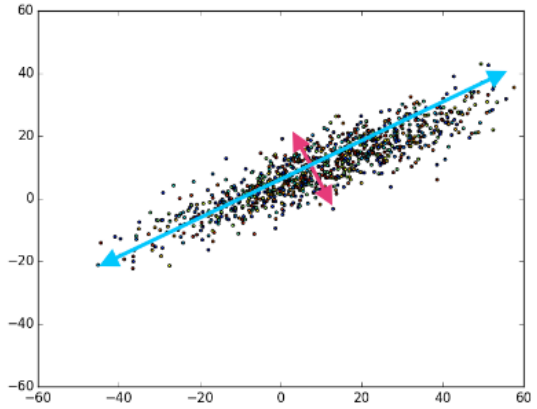
- Method based simply on calculating Eigenvectors
- The aren't any tuning of parameters required
- No local optima - unique solution

Weaknesses:

- Limited to second order statistics
- Limited to linear projections

PCA

Given a set of *n* centred observations, the first principal component is direction that maximises the variance of the the data.



$$X = (x_1, x_2, ..., x_n) \qquad \mathbf{u}_1 = \text{argmax}_{||\mathbf{u}_1||=1} \left[\frac{1}{N} \sum_{i=1}^{N} (\mathbf{u}_1^T \mathbf{x}_i)^2 \right]$$

We can then calculate this direction by calculating the eigenvector corresponding to the largest eigenvector of this covariance matrix.

$$S_1 = \frac{1}{N} \mathbf{X}\mathbf{X}^T$$

We note that the (*i*,*j*)th entry of the covariance matrix is the inner product of the *i*'th datapoint with the *j*'th datapoint.

Alternative Expression for PCA

The principal component lies in the span of the data, such that

$$\mathbf{u}_1 = \mathbf{X}\boldsymbol{\alpha}$$

We can therefore rewrite the eigenevector equation in terms of the above alternative expression for the principal component,

$$\mathbf{S}_1\mathbf{u}_1 = \frac{1}{n}\mathbf{X}\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha} = \lambda\mathbf{X}\boldsymbol{\alpha}$$

And left-multiplying both sides, we obtain

$$\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha} = \lambda\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha}$$

Kernel PCA

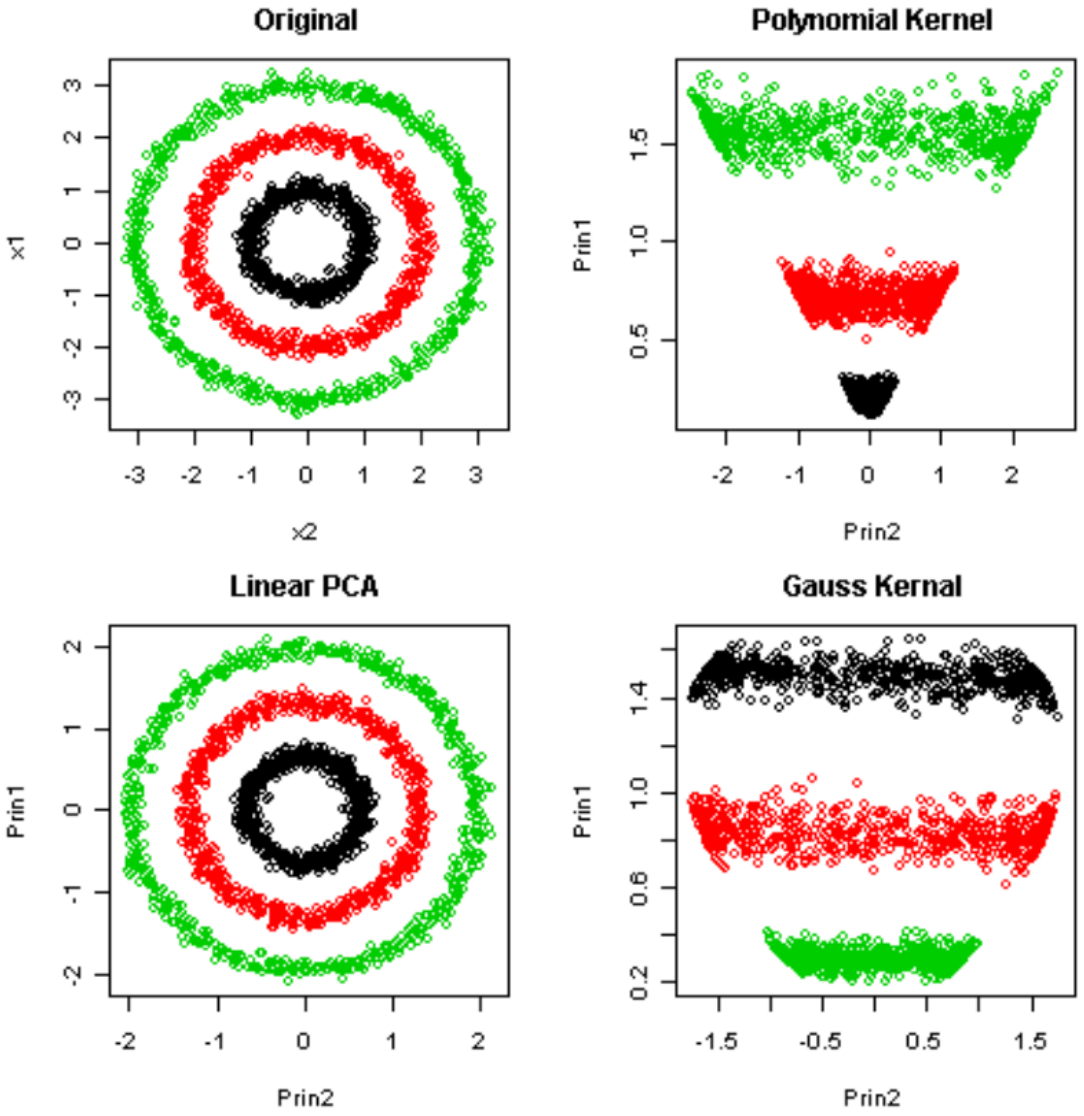
Main Idea: Replace the inner product matrix with the kernel matrix

PCA: $$\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha} = \lambda\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha}$$

Kernel PCA: $$\frac{1}{n}K(\mathbf{X},\mathbf{X})\boldsymbol{\alpha} = \lambda\boldsymbol{\alpha}$$

In other words, we can form the *n*x*n* kernel matrix *K*, then compute the eigendecomposition of *K!*

Kernel PCA Example



Kernel PCA

PCA Overview

Strengths:

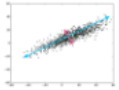
- Method based simply on calculating Eigenvectors
- The aren't any tuning of parameters required
- No local optima - unique solution

Weaknesses:

- Limited to second order statistics
- Limited to linear projections

PCA

Given a set of n centred observations, the first principal component is direction that maximises the variance of the the data.



$$X = (x_1, x_2, ..., x_n) \qquad \mathbf{u}_1 = \text{argmax}_{\|\mathbf{u}\|=1}\left[\frac{1}{N}\sum_{i=1}^{N}(\mathbf{u}^T\mathbf{x}_i)^2\right]$$

We can then calculate this direction by calculating the eigenvector corresponding to the largest eigenvector of this covariance matrix. $S_1 = \frac{1}{N}\mathbf{X}\mathbf{X}^T$

We note that the (i,j)th entry of the covariance matrix is the inner product of the i'th datapoint with the j'th datapoint.

Alternative Expression for PCA

The principal component lies in the span of the data, such that

$$\mathbf{u}_1 = \mathbf{X}\alpha$$

We can therefore rewrite the eigenevector equation in terms of the above alternative expression for the principal component,

$$\mathbf{S}_1\mathbf{u}_1 = \frac{1}{n}\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}\alpha$$

And left-multiplying both sides, we obtain

$$\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}^T\mathbf{X}\alpha$$

Kernel PCA

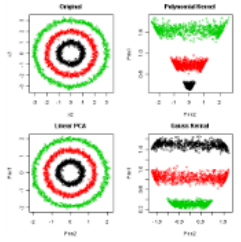
Main Idea: Replace the inner product matrix with the kernel matrix

PCA: $\frac{1}{n}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}\alpha = \lambda\mathbf{X}^T\mathbf{X}\alpha$

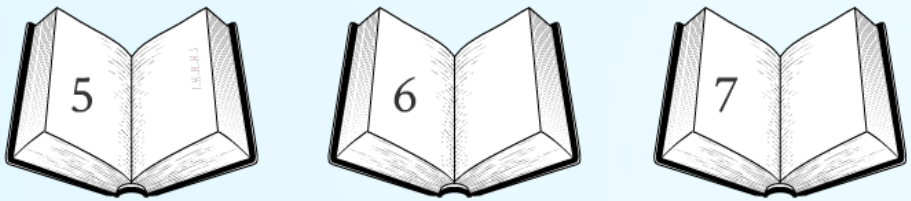
Kernel PCA: $\frac{1}{n}K(\mathbf{X},\mathbf{X})\alpha = \lambda\alpha$

In other words, we can form the nxn kernel matrix K, then compute the eigendecomposition of K!

Kernel PCA Example



Non-parametric Methods



M5MS10
Machine Learning

Spring 2018

Lecture 5

Dr Ben Calderhead

b.calderhead@imperial.ac.uk