03 - Know your Classics

Supervised Learning: k-NN, decision trees, SVM and kernel trick

François Pitié

Ussher Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

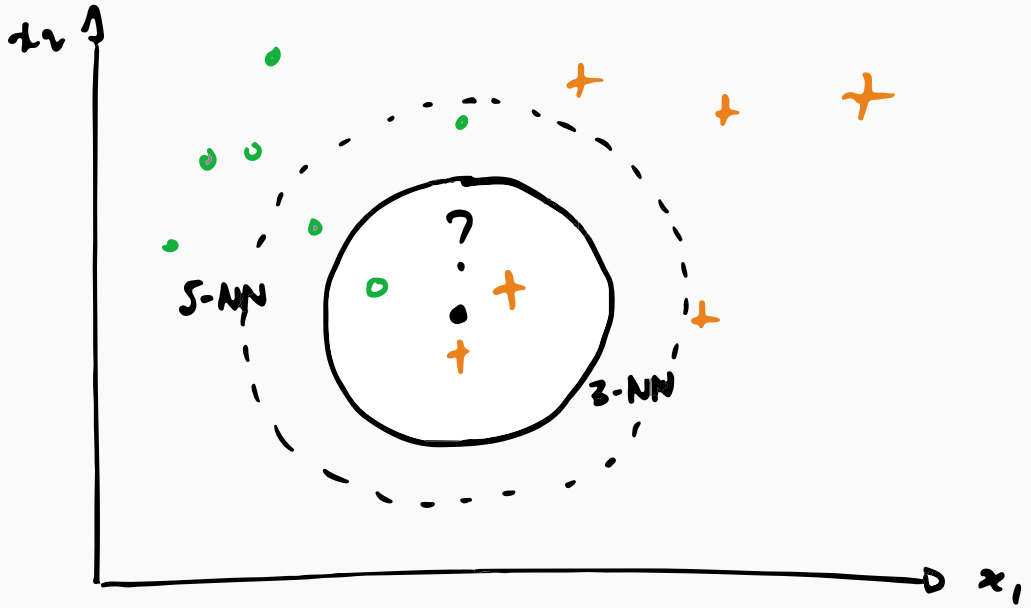
Before we dive deeper into Neural Networks, keep in mind that Neural Nets have been around for a while and, until recently, they were not the method of choice for Machine Learning.

A zoo of algorithms exits out there, and we'll briefly introduce here some of the **classic methods** for supervised learning.

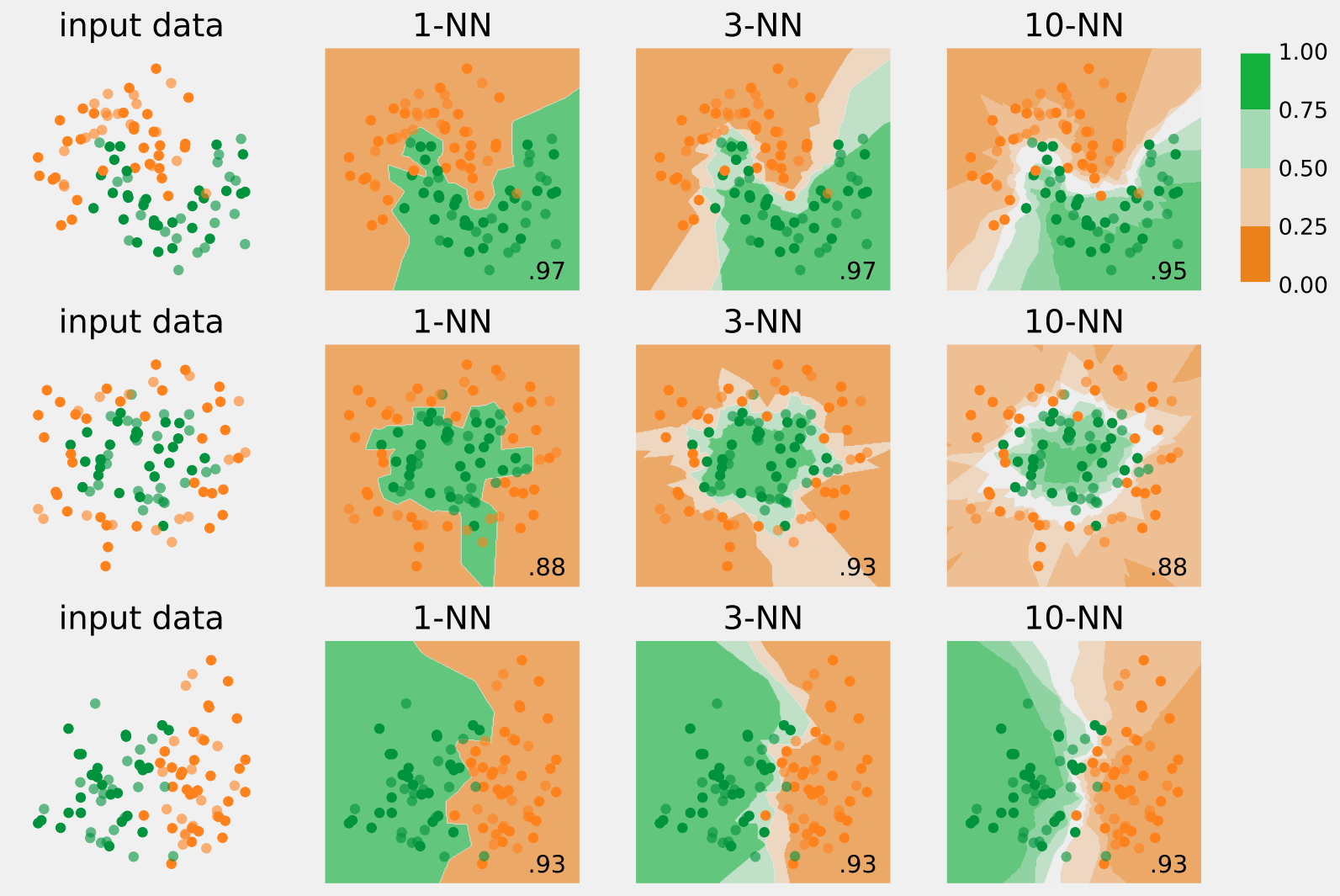
k-nearest neighbours

k-nearest neighbours is a very simple yet powerful technique:

For an input $\mathbf{x}$, you retrieve the k-nearest neighbours in the training data, then return the average of the corresponding y values.



k-nearest neighbours



Decision boundaries on 3 problems. Darker shades of orange or green correspond to more certain decisions.

k-nearest neighbours

pros:

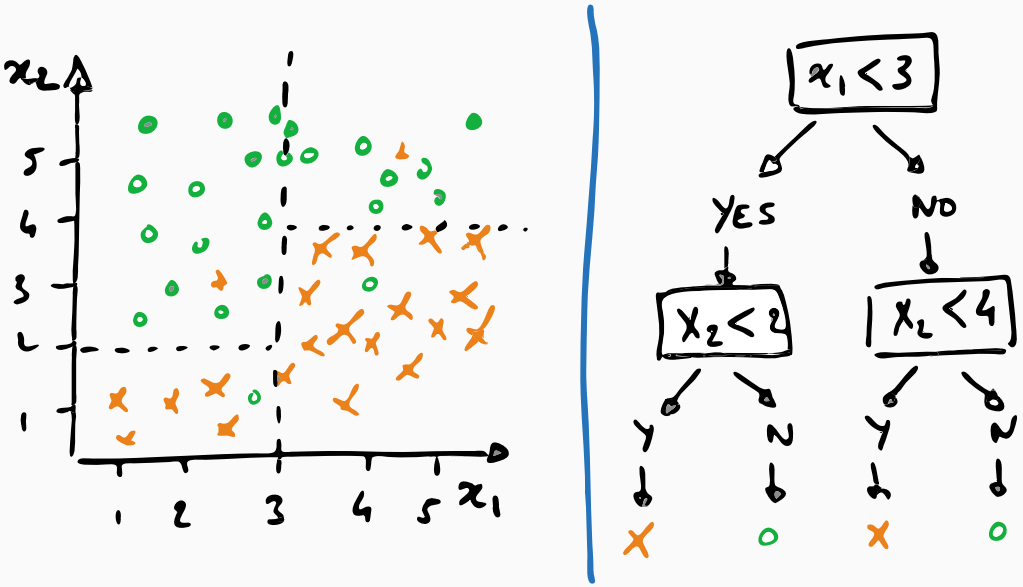
- It is a non-parametric technique.
- It works surprisingly well and you can obtain high accuracy if the training set is large enough.

cons:

- Finding the nearest neighbours is computationally expensive.
- It may generalise very badly if your training set is small.
- You don't learn much about the features themselves.

Decision Trees

In **decision trees** (Breiman et al., 1984) and its many variants, each node of the decision tree is associated with a region in the input space, and internal nodes partition that region into sub-regions (in a divide and conquer fashion).



Typically the regions are split along the axes of the input space (eg. at each node you take a decision according to a binary test such as $x_2 < 3$).

Decision Trees

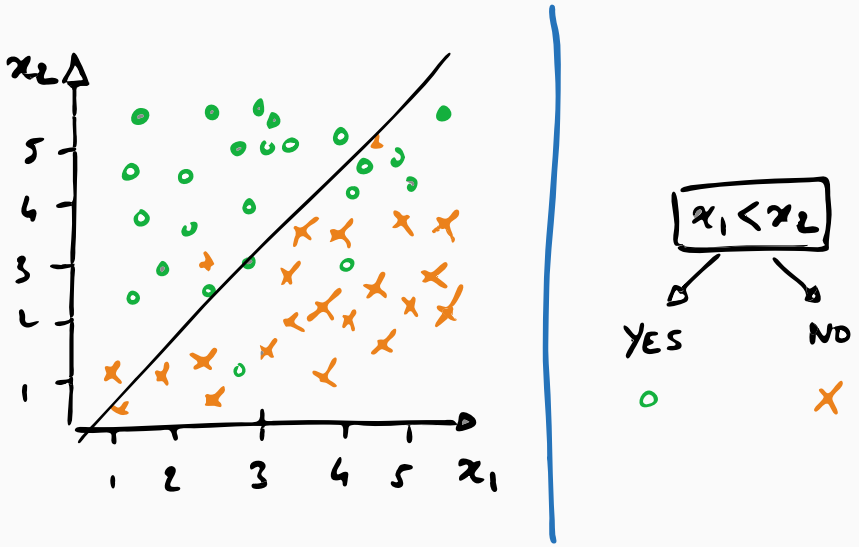
Decision Trees

pros:

- It is fast

cons:

- The decisions are taken along the original axes (eg. $x_1 < 3$). But in our previous example, it would have been more efficient to split the classes along a diagonal ($x_1 < x_2$):



Decision Trees

See also:

Ada Boost, Random Forests.

Links:

https://www.youtube.com/watch?v=p17C9q2M00Q

SVM

Until recently Support Vector Machines were the most popular technique around.

Like in Logistic Regression, SVM starts as a linear classifier:

$$y = [\mathbf{x}^\mathsf{T}\mathbf{w} > 0]$$

The difference with logistic regression lies in the choice of the loss function.

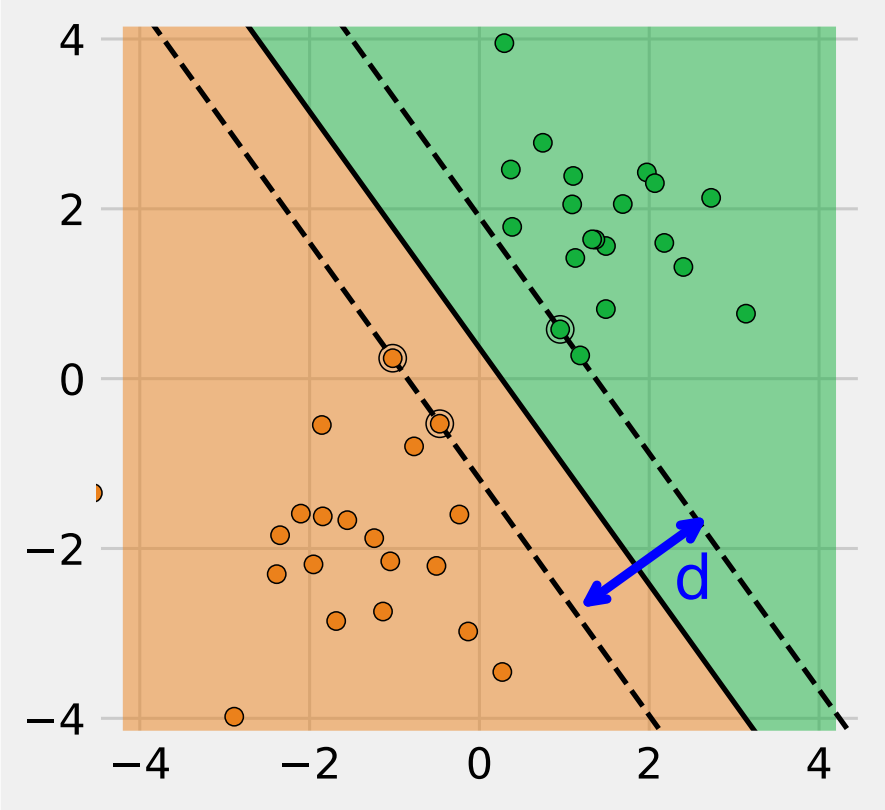
SVM

Whereas in logistic regression the loss function was based on the cross-entropy, the loss function in SVM is based on the **Hinge loss** function:

$$L_{SVM}(\mathbf{w}) = \sum_{i=1}^{N} [y_i = 0] \max(0, 1 + \mathbf{x}_i^\top \mathbf{w}) + [y_i = 1] \max(0, 1 - \mathbf{x}_i^\top \mathbf{w})$$

SVM

From a geometrical point of view, SVM seeks to find the hyperplane that maximises the separation between the two classes.



SVM

There is a lot more to SVM, but this will be not coverd in this course.

Note that there is a priori no advantage of using linear SVM over logistic regression in terms of performance alone. It all depends on the type of data you have. (see the "no free lunch" theorem (Wolpert and Macready, 1997))

SVM

SVM became very popular when it began to be associated with the **kernel trick**.

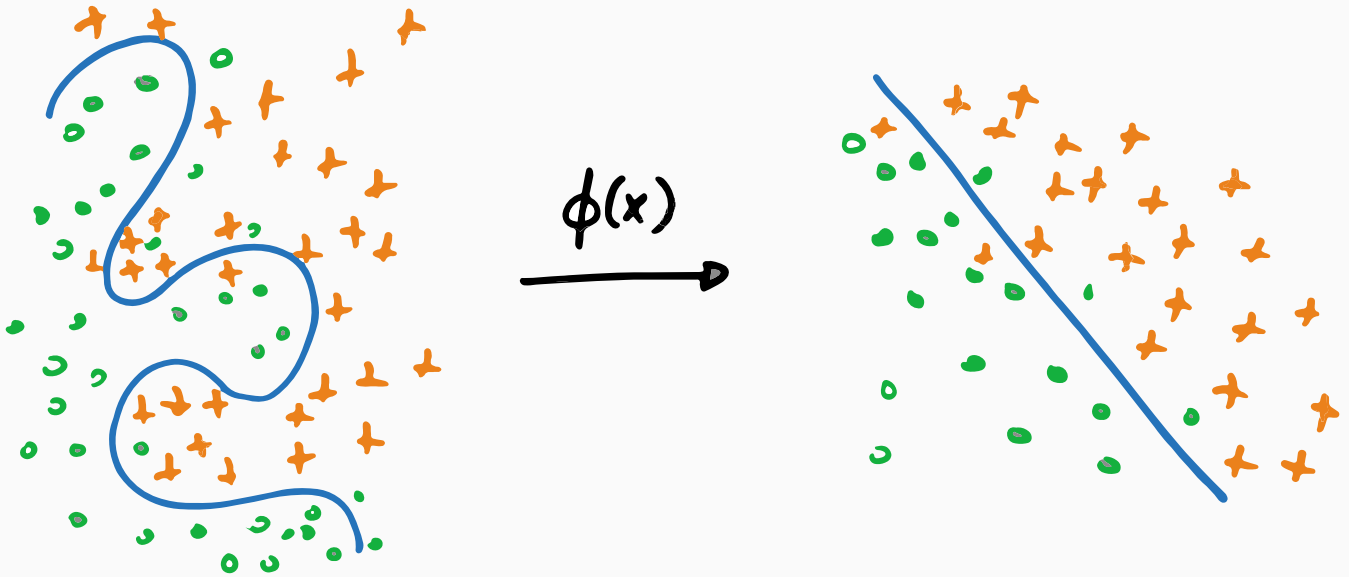
Kernel Trick (1)

Recall that in linear regression, we managed to fit non-linear functions by augmenting the feature space with higher order polynomials of each the observation, e.g, x, x^2, x^3, etc.

What we've done is to map the original features into a higher dimensional feature space: $\phi : \mathbf{x} \rightarrow \phi(\mathbf{x})$. In our case we had:

$$\phi(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \vdots \end{pmatrix}$$

kernel Trick



Feature mapping is used to transform the input data into a new dataset that can be solved using a linear classifier.

Transforming the original features into more complex ones is a key ingredient of machine learning.

The raw collected features are usually not optimal for linearly separating the classes, and it is often unclear how these should be transformed. Therfore we'd like the machine learning technique to learn how best recombine the features so as to to yield optimal class separation.

So our first problem is to find a useful feature transformation ϕ. Another problem is that the **size of the new feature vectors $\phi(\mathbf{x})$ can grow very quickly**.

Consider a degree 2 polynomial augmentation of features that were originally of dimension $p = 2$ or $p = 3$. The new feature vectors increase in size:

$$\phi\left([x_1\,,\,x_2]^\mathsf{T}\right) = [1\,,\,x_1\,,\,x_2\,,\,x_1x_2\,,\,x_1^2\,,\,x_2^2]^\mathsf{T}$$

$$\phi\left([x_1\,,\,x_2\,,\,x_3]^\mathsf{T}\right) = [1\,,\,x_1\,,\,x_2\,,\,x_3\,,\,x_1x_3\,,\,x_1x_2\,,\,x_2x_3\,,\,x_1^2\,,\,x_2^2\,,\,x_3^2]^\mathsf{T}$$

It can be shown that for input features of dimension d and a polynomial degree d, the transformed features are of dimension $\frac{(n+d)!}{n!\,d!}$.

Thus, if you have 100 features per observation and that you are looking at a polynomial of order 5, the resulting feature vector is of dimension 100 millions!!

Consider that a Least-Squares solution is given by

$$\mathbf{w} = (X^\mathsf{T} X)^{-1} X^\mathsf{T} \mathbf{y}$$

if $\boldsymbol{\phi}(\mathbf{x})$ is of dimension 10 millions, then $X^\mathsf{T} X$ is of size $\mathbf{10^8 \times 10^8}$. This is totally impractical.

So, we want to transform the original features into higher level features but this comes at the cost of greatly increasing the dimension of the original problem.

The Kernel trick offers an elegant solution that allows us to introduce very complex mapping functions ϕ without having to ever explicitly compute them.

Kernel Trick

In most machine learning algorithms, we can show that (see later) that $\mathbf{w}$ can be re-expressed in terms of the existing input feature vectors:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \mathbf{x}^{(i)}$$

where α_i are new weights defining $\mathbf{w}$.

The loss functions usually depend on the score $\mathbf{w}^\mathsf{T}\mathbf{x}$, which can now be re-written as:

$$\mathbf{x}^\mathsf{T}\mathbf{w} = \sum_{i=1}^{n} \alpha_i \mathbf{x}^\mathsf{T}\mathbf{x}^{(i)}$$

The scalars $\mathbf{x}^\mathsf{T}\mathbf{x}^{(i)}$ are the dot-products between feature vectors.

Kernel Trick

Now look at what happens when we use augmented features:

$$\phi(\mathbf{x})^\mathsf{T}\mathbf{w} = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{x}^{(i)})$$

To compute $\phi(\mathbf{x})^\mathsf{T}\mathbf{w}$, we only ever need to know how to compute the dot products $\phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{x}^{(i)})$. Introducing the **kernel function**:

$$\kappa(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{z})$$

we can rewrite the scores as:

$$\phi(\mathbf{x})^\mathsf{T}\mathbf{w} = \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}, \mathbf{x}^{(i)})$$

Kernel Trick

Then kernel trick builds on the *Theory of Reproducing Kernels*, which we says that for a whole class of kernel functions κ we can find a mapping ϕ that is such that $\kappa(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\mathsf{T}\phi(\mathbf{z})$.

The key is that we can define κ without having to explicitly define ϕ.

Kernel Trick

Many kernel functions are possible. For instance polynomial kernel:

$$\kappa(\mathbf{u}, \mathbf{v}) = (r - \gamma \mathbf{u}^\top \mathbf{v})^d$$

one can show that this is equivalent to using a polynomial mapping as proposed earlier. Except instead of requiring 100's of millions of dimensions, we only need vectors of size n and to compute $\kappa(\mathbf{u}, \mathbf{v})$, which is linear in p.

Kernel Trick

The most commonly used kernel is probably the **Radial Basis Function (RBF)** kernel:

$$\kappa(\mathbf{u}, \mathbf{v}) = e^{-\gamma\|\mathbf{u}-\mathbf{v}\|^2}$$

The mapping ϕ implicitly defined by $\kappa(\mathbf{u}, \mathbf{v})$ is in fact infinitely dimensional! But that's fine because we never need to evaluate ϕ.

Let's come back to the claim that we can write $\mathbf{w} = \sum_{i=1}^{n} \alpha_i \mathbf{x}^{(i)}$

Many linear machine learning methods are based on minimising something like:

$$E(\mathbf{w}) = \mathcal{L}(X\mathbf{w}, y) + \lambda \|\mathbf{w}\|^2$$

For instance, in least squares,

$$\mathcal{L}(X\mathbf{w}, y) = \sum_{n=1}^{N} (y_i - \mathbf{x}_i^\top \mathbf{w})^2$$

and in SVM:

$$\mathcal{L}(X\mathbf{w}, y) = \sum_{i=1}^{N} [y_i = 0] \max(0, 1 + \mathbf{x}_i^\top \mathbf{w}) + [y_i = 1] \max(0, 1 - \mathbf{x}_i^\top \mathbf{w})$$

The term $\lambda \|\mathbf{w}\|^2$ is the regularisation term we already saw in linear regression.

When minimising $E(\mathbf{w})$, $\hat{\mathbf{w}}$ is necessarily of the form:

$$\hat{\mathbf{w}} = X^\top \alpha = \sum_{i=1}^{n} \alpha_i \mathbf{x}^{(i)}$$

Proof:

Consider $\hat{\mathbf{w}} = X^\top \alpha + \mathbf{v}$, with $\mathbf{v}$ such that $X\mathbf{v} = 0$.

We show that $E(X^\top \alpha + \mathbf{v}) > E(X^\top \alpha)$ if $\mathbf{v} \neq 0$:

$$\begin{aligned} E(X^\top \alpha + \mathbf{v}) &= \mathcal{L}(XX^\top \alpha + X\mathbf{v}, y) + \lambda \|X^\top \alpha + \mathbf{v}\|^2 \\ &= \mathcal{L}(XX^\top \alpha, y) + \lambda \left(\alpha^\top XX^\top \alpha + 2\alpha X\mathbf{v} + \mathbf{v}^\top \mathbf{v}\right) \\ &= \mathcal{L}(XX^\top \alpha, y) + \lambda \left(\alpha^\top XX^\top \alpha + \mathbf{v}^\top \mathbf{v}\right) \\ &> E(X^\top \alpha) \quad \text{if } \mathbf{v} \neq 0 \end{aligned}$$

now if $\mathbf{w} = X^\top \alpha$, then

$$E(\mathbf{w}) = E(\alpha) = \mathcal{L}(XX^\top \alpha, \mathbf{y}) + \lambda \alpha^\top XX^\top \alpha$$

We call $K = XX^\top$ the **Kernel Matrix**. It is a matrix of dimension $n \times n$ whose entries are the scalar products between observations:

$$K_{i,j} = \mathbf{x}_i^\top \mathbf{x}_j$$

Note that the expression to minimise

$$E(\alpha) = \mathcal{L}(K\alpha, \mathbf{y}) + \lambda \alpha^\top K \alpha$$

only contains matrices and vectors of dimension $n \times n$ or $n \times 1$. In fact, even if the features are of infinite dimension ($p = +\infty$), our reparamterised problem only depends on the number of observations n.

When we transform the features $\mathbf{x} \to \phi(\mathbf{x})$. The expression to minimise keeps the same form:

$$E(\alpha) = \mathcal{L}(K\alpha, \mathbf{y}) + \lambda \alpha^\top K \alpha$$

the only changes occur for K:

$$K_{i,j} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$$

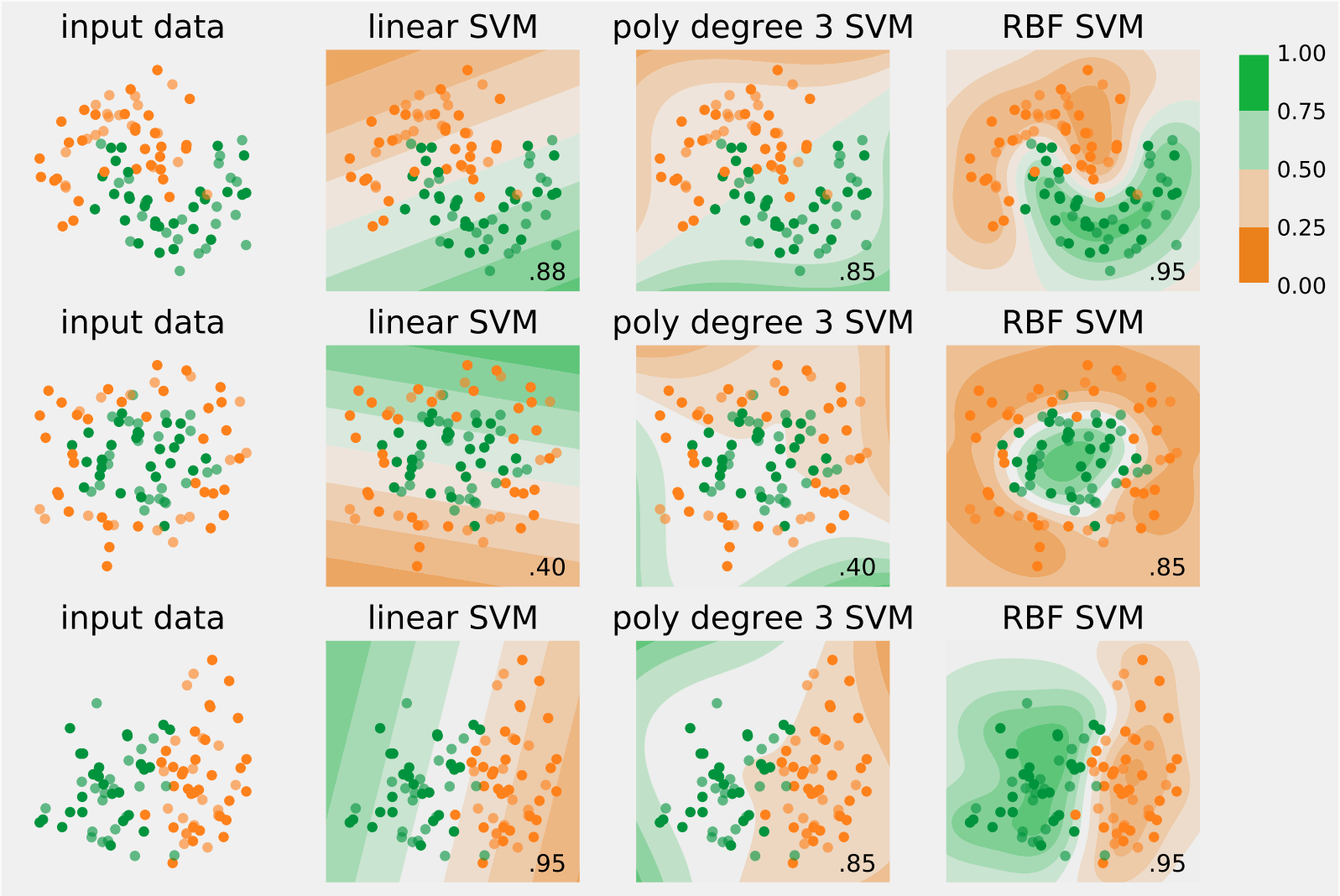
Thus we never really need to explicitly compute ϕ, we just need to know how to compute $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$.

Support vector machines **are not the only algorithm that can avail of the kernel trick**. Many other linear models (including logistic regression) can be enhanced in this way. They are known as kernel methods.

A major drawback to kernel methods is that the cost of evaluating the decision function is linear in the number of training examples, because the i^{th} example contributes a term $\alpha_i \kappa(\mathbf{x}, \mathbf{x}^{(i)})$ to the decision function. SVM somehow mitigates this by learning which examples contribute the most. These training examples are known as **support vectors**.

Kernel methods also suffer from a high computational cost of training when the dataset is large.

The current deep learning renaissance began when Hinton et al. (2006) demonstrated that a neural network could outperform kernel SVM on the MNIST benchmark.


input data
linear SVM
poly degree 3 SVM
RBF SVM
1.00
0.75
0.50
0.25
0.00
.88
.85
.95
input data
linear SVM
poly degree 3 SVM
RBF SVM
.40
.40
.85
input data
linear SVM
poly degree 3 SVM
RBF SVM
.95
.85
.95

Kernel Trick and SVM

See also:

Gaussian Processes,
Reproducing kernel Hilbert spaces,
kernel Logistic Regression

Links:

Laurent El Ghaoui's lecture at Berkeley: `https://goo.gl/hY1Bpn`
Eric Kim's python tutorial on SVM: `https://goo.gl/73iBdx`

Take Away

Neural Nets have existed for a while, but it is only recently (2006) that they have started to surpass all other techniques.

Kernel based techniques have been very popular up to recently as they offer an elegant way of transforming input features into more complex features that can then be linearly separated.

The problem with kernel techniques is that they cannot deal efficiently with large datasets (eg. more than 10's of thousands of observations)