Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

03 - Know your Classics

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

François Pitié

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

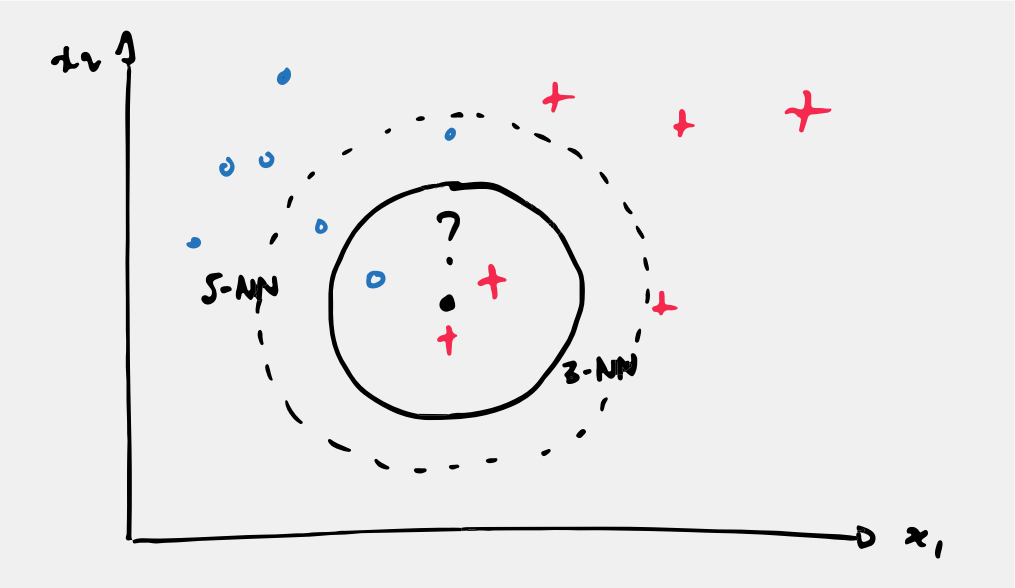
[4C16/5C16] Deep Learning and its Applications — 2023/2024

Before we dive 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.

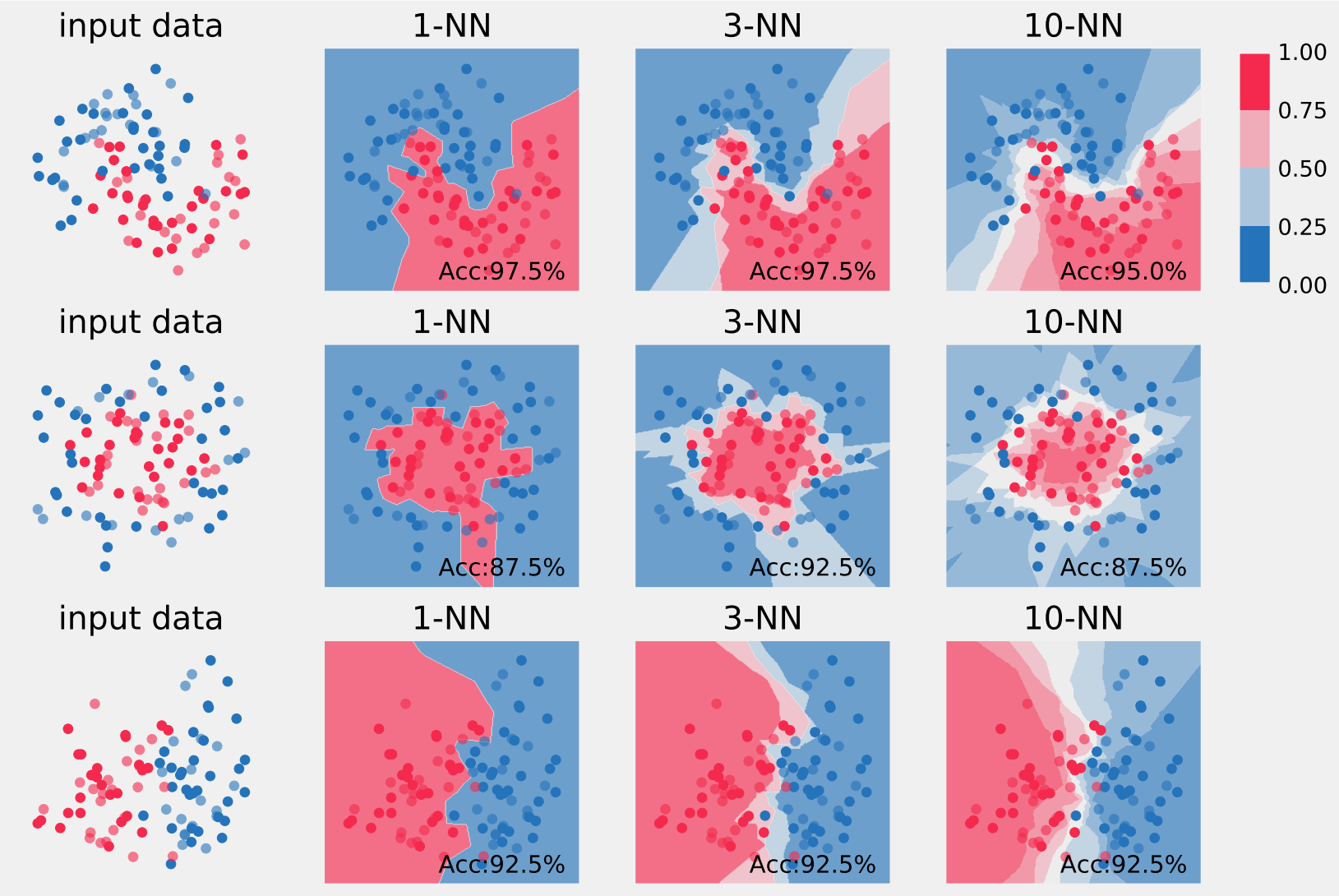
A zoo of algorithms exits out there, and we'll briefly introduce here some of the **classic methods** for supervised learning.

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 majority class amoung the k values. You can also return the confidence as a proportion of the majority class.



k-nearest neighbours



Decision boundaries on 3 problems. The intensity of the shades indicates the certainty we have about the prediction.

k-nearest neighbours

pros:

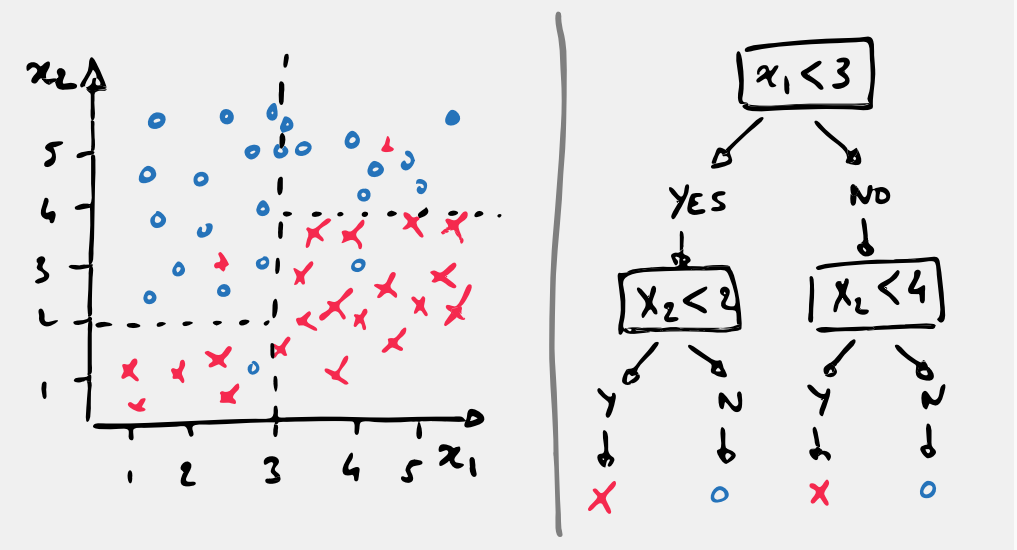
- It is a non-parametric technique.
- It works surprisingly well and you can obtain high accuracy if the training set is large enough.

cons:

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

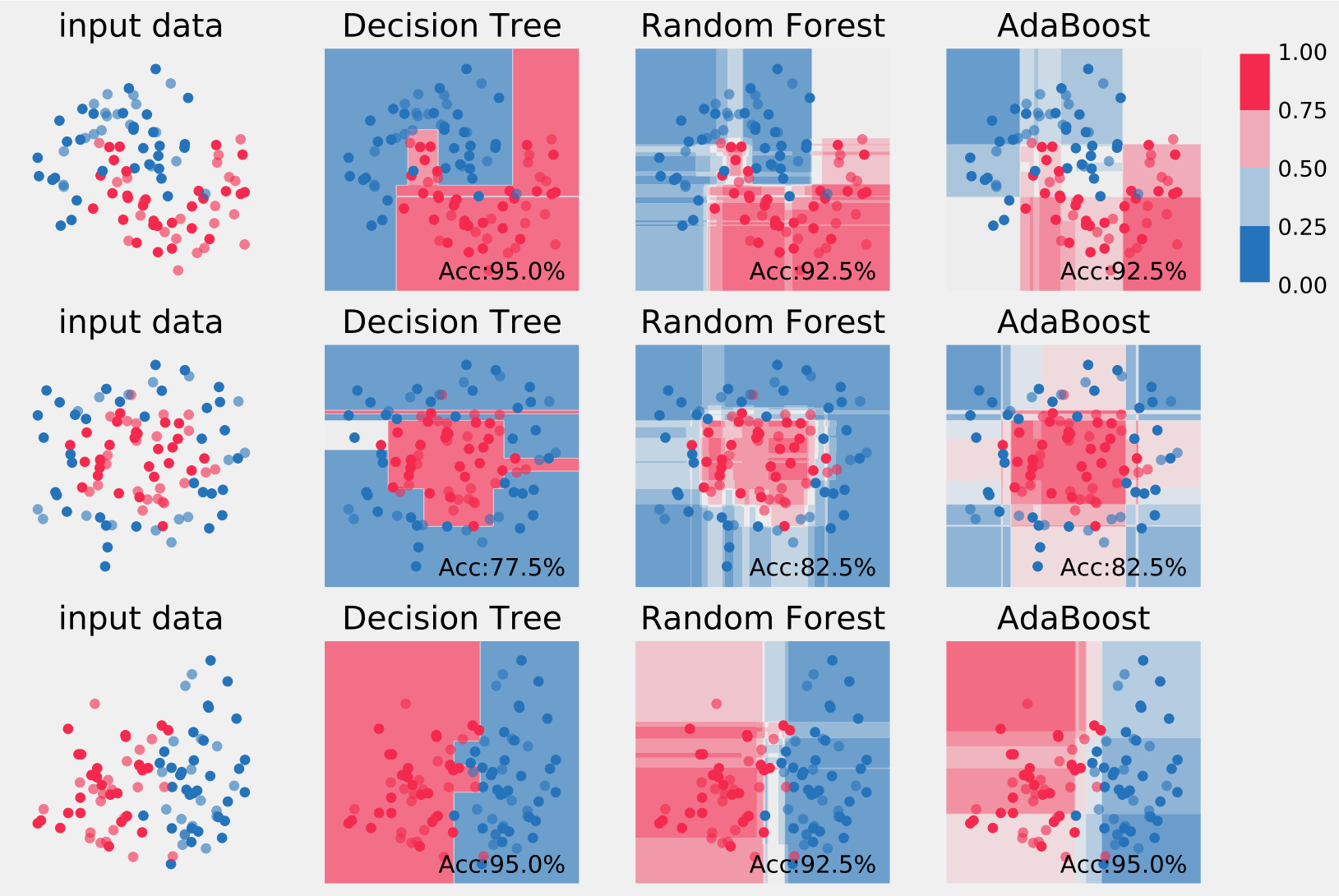
Decision Trees

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



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



In Ada Boost and Random Forests multiple decision trees are used to aggregate a probability on the prediction.

Decision Trees

Random Forests gained a lot of popularity before the rise of Neural Nets as they can be very efficiently computed.

For instance they where used for the body part identification in the Microsoft Kinect.

[1] Real-Time Human Pose Recognition in Parts from a Single Depth Image
J. Shotton, A. Fitzgibbon, A. Blake, A. Klpman, M. Finocchio, B. Moore, T. Sharp, 2011
[https://goo.gl/UTM6s1]

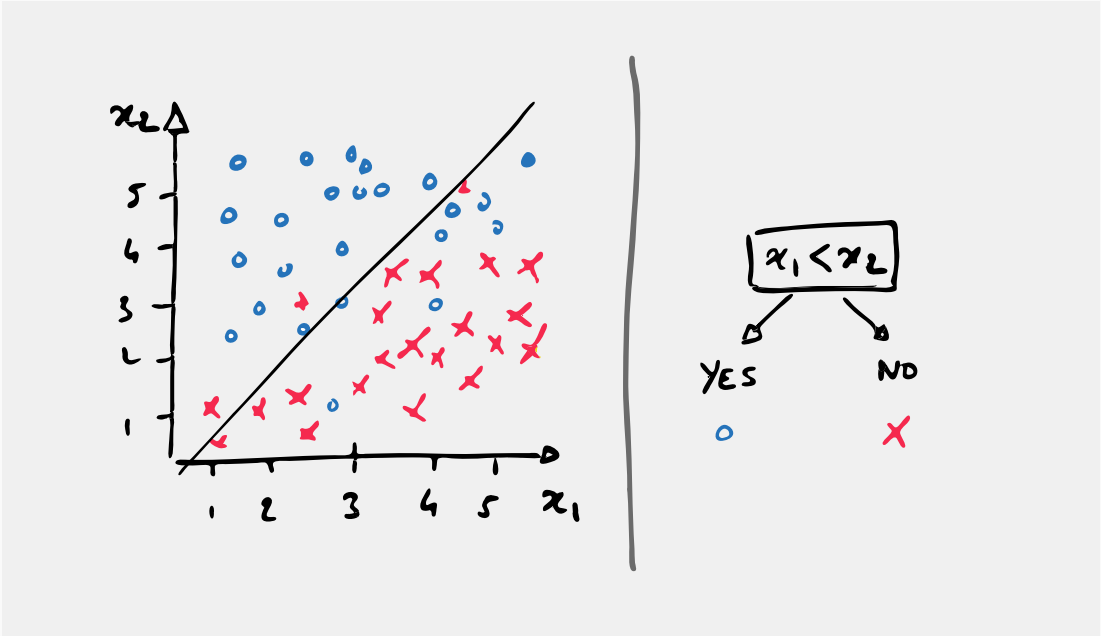
Decision Trees

pros:

- It is fast.

cons:

- Decisions are taken along axes (eg. $x_1 < 3$) but it could be more efficient to split the classes along a diagonal (eg. $x_1 < x_2$):



Decision Trees

See also:

Ada Boost, Random Forests, XGBoost.

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:

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

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

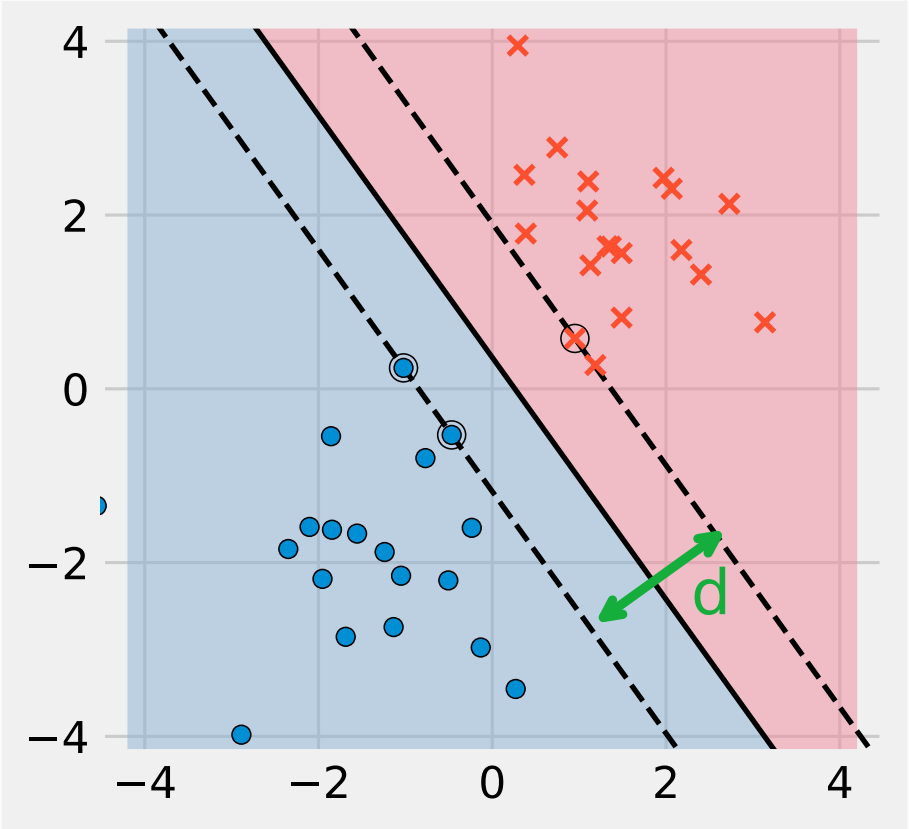
SVM

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:

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

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.

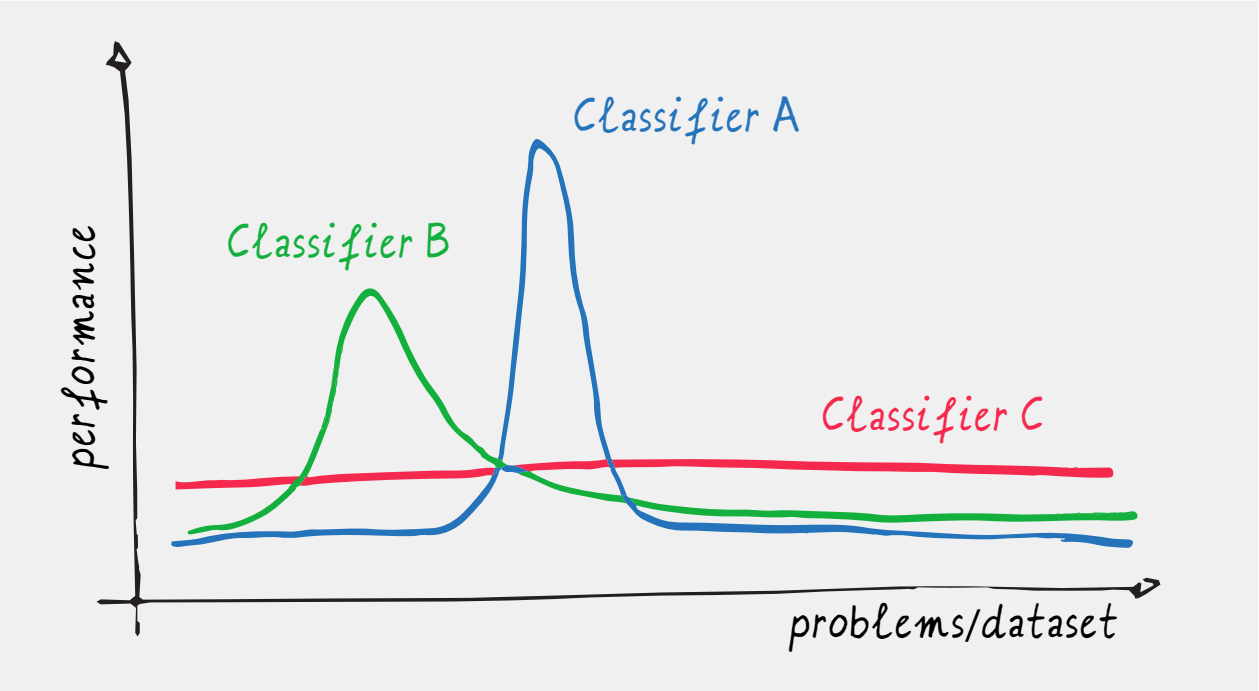
No Free Lunch Theorem

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.

Recall that the choice of loss function directly relates to assumptions you make about the distribution of the prediction errors, and thus about the dataset of your problem.

No Free Lunch Theorem

This is formalised in the "no free lunch" theorem (Wolpert, 1996), which tells us that classifiers perform equally well when averaged over all possible problems. In other words: your choice of classifier should depend on the problem at hand.



SVM

SVM gained popularity when it became associated with the **kernel trick**.

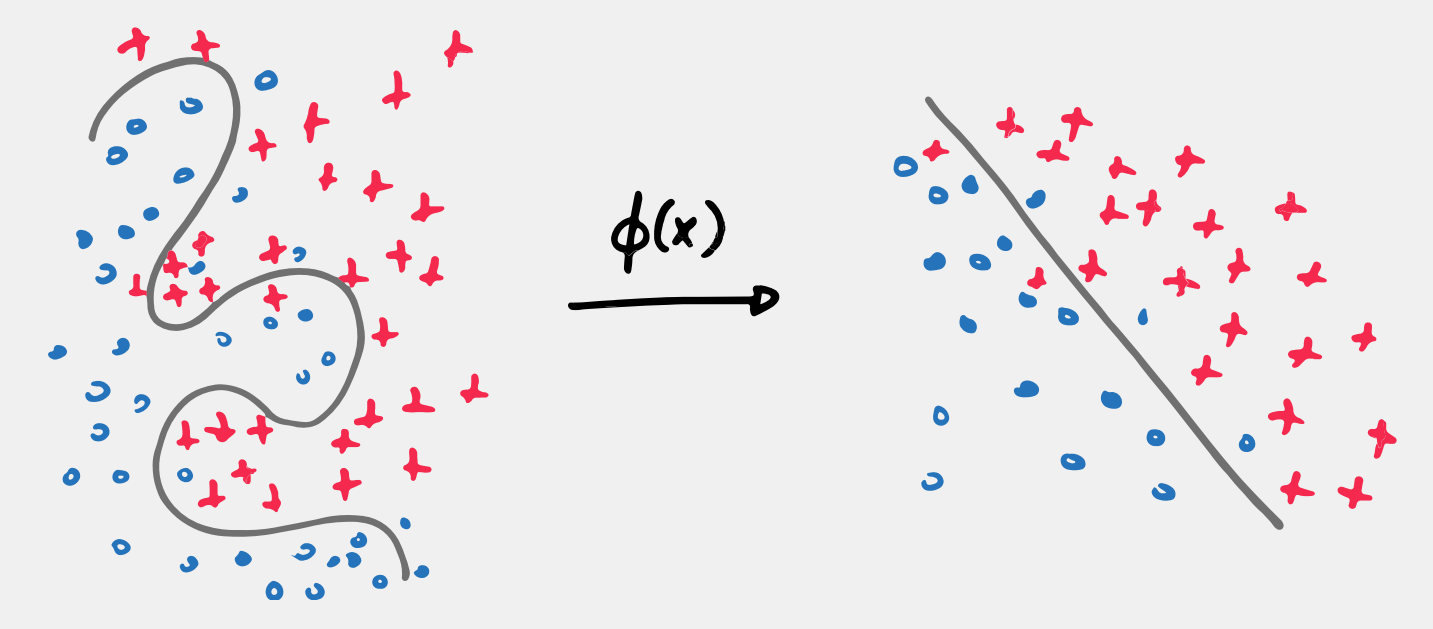
Kernel Trick

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

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

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

kernel Trick



The idea here is the same: we want to find a feature map $x \mapsto \phi(x)$ that transforms 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, and something that we'll see again with Deep Learning.

The collected features are usually not optimal for linearly separating the classes and it is often unclear how these should be transformed. We would like the machine learning technique to learn how to best recombine the features so as 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 $\phi(\mathbf{x})$ could potentially grow very large**.

Consider the following polynomial augmentations:

$$\phi\left([x_1\,,\,x_2]^\top\right) = [1\,,\,x_1\,,\,x_2\,,\,x_1x_2\,,\,x_1^2\,,\,x_2^2]^\top$$

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

The new feature vectors have significantly increased in size.

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

For example, if you have $p = 100$ features per observation and that you are looking at a polynomial of order 5, the resulting feature vector is of dimension about 100 millions!!

Now, recall that Least-Squares solutions are given by

$$\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{y}$$

if $\phi(\mathbf{x})$ is of dimension 100 millions, then $X^\top X$ is of size $10^8 \times 10^8$. This is totally impractical.

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

The Kernel trick offers an elegant solution to this problem and allows us to use very complex mapping functions ϕ without having to ever explicitly compute them.

Kernel Trick

We start from the observation that most loss functions only operates on the scores $\mathbf{x}^\top \mathbf{w}$, eg:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} E(\mathbf{w}) = \sum_i e(\mathbf{x}_i^\top \mathbf{w})$$

We can show that in these cases (see lecture notes), for any $\mathbf{x}$, the score at the optimum $\mathbf{x}^\top \hat{\mathbf{w}}$ can be re-expressed as:

$$\mathbf{x}^\top \hat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i \mathbf{x}^\top \mathbf{x}_i$$

where the scalars $\mathbf{x}^\top \mathbf{x}_i$ are dot-products between feature vectors.

The new weights $\boldsymbol{\alpha} = [\alpha_1, \cdots, \alpha_n]$ can be seen as a re-parametrisation into a $n \times 1$ vector of $\hat{\mathbf{w}}$, with $E(\mathbf{w})$ being re-expressed as $E(\boldsymbol{\alpha})$. These are often called the dual coefficients in SVM.

Kernel Trick

Things get interesting when using our expanded features:

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

To compute the score, we only ever need to know how to compute the dot products $\phi(\mathbf{x})^\top \phi(\mathbf{x}_i)$, not the actual high dimensional feature vector $\phi(\mathbf{x}_i)$.

Introducing the **kernel function**:

$$(\mathbf{u}, \mathbf{v}) \mapsto \kappa(\mathbf{u}, \mathbf{v}) = \phi(\mathbf{u})^\top \phi(\mathbf{v}),$$

which allows us to rewrite the score as:

$$\phi(\mathbf{x})^\top \hat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i).$$

Kernel Trick

The 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 $\kappa(\mathbf{u}, \mathbf{v}) = \phi(\mathbf{u})^\top \phi(\mathbf{v})$.

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

Kernel Trick

Many kernel functions are possible. For instance, the polynomial kernel is defined as:

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

and one can show that this is equivalent to using a polynomial mapping as proposed earlier. Except that instead of requiring 100's of millions of dimensions, we only need to take scalar products between vectors of dimension 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 induced mapping ϕ is infinitely dimensional, but that's OK because we never need to evaluate $\phi(\mathbf{x})$.

Kernel Trick: Intuition (pt1)

To have some intuition about these kernels, consider the kernel trick for a RBF kernel. The score for a particular observation $\mathbf{x}$ is:

$$\text{score}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i)$$

The kernel function $\kappa(\mathbf{u}, \mathbf{v}) = e^{-\gamma \|\mathbf{u}-\mathbf{v}\|^2}$ is a measure of similarity between observations. If both observations are similar, $\kappa(\mathbf{u}, \mathbf{v}) \approx 1$. If they are very different, $\kappa(\mathbf{u}, \mathbf{v}) \approx 0$. We can see it as a neighbourhood indicator function. If the observations are close, $\kappa(\mathbf{u}, \mathbf{v}) \approx 1$, else $\kappa(\mathbf{u}, \mathbf{v}) \approx 0$. The scale of this neighbourhood is controlled by γ.

(as you can imaging, this is less intuitive for other kernels)

Kernel Trick: Intuition (pt2)

Let's choose $\alpha_i = 1$ for positive observations and $\alpha_i = -1$ for negative observations. This is obviously not the optimal, but this is in fact close to what happens in SVM. We have now something resembling k-NN. Indeed, look at the score:

$$\begin{aligned}\text{score}(\mathbf{x}) &= \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i) \\ &\approx \sum_{i \in \text{neighbours of } \mathbf{x}} \begin{cases} 1 & \text{if } y_i \text{ positive} \\ -1 & \text{if } y_i \text{ negative} \end{cases} \\ &\approx \#\text{ positive neighbours of } \mathbf{x} - \#\text{ negative neighbours of } \mathbf{x}\end{aligned}$$

This makes sense: if $\mathbf{x}$ has more positive than negative neighbours in the dataset, then its score should be high, and its prediction positive.

Thus we have here something similar to k-NN. The main difference is that instead of finding a fixed number of the k closest neighbours, we consider all the neighbours within some radius (controlled by γ).

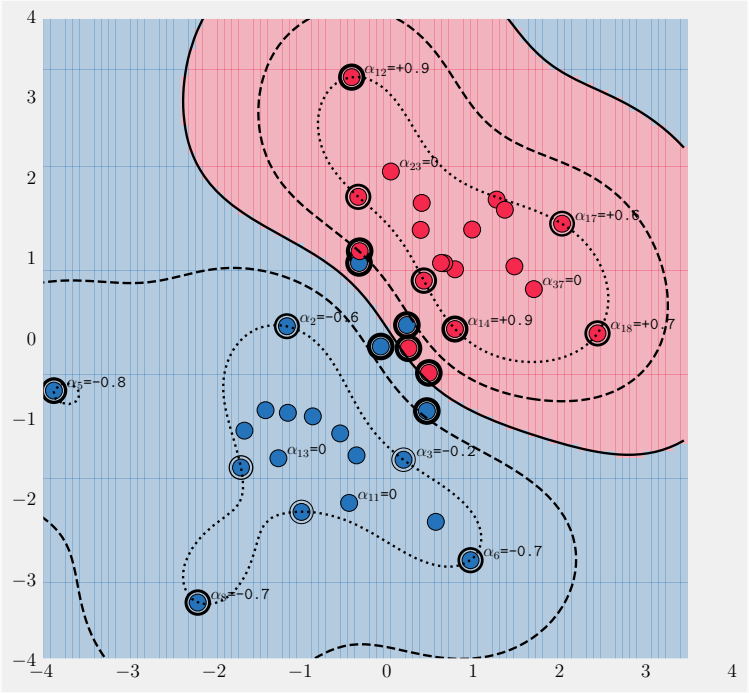
Kernel Trick: Intuition (pt3)

In SVM, the actual values of $\hat{\alpha}_i$ are estimated by ways of the minimisation of the Hinge loss.

The optimisation falls outside of the scope of this course material. We could use Gradient Descent, but, as it turns out, the Hinge loss makes this problem a quadratic programming problem and we can use a solver for that. The good news is that we can find the global minimum without having to worry about convergence issues.

We find after optimisation that, indeed, $-1 \leq \hat{\alpha}_i \leq 1$, with the sign of $\hat{\alpha}_i$ indicating the class membership, thus following a similar idea to what was proposed in the previous slide.

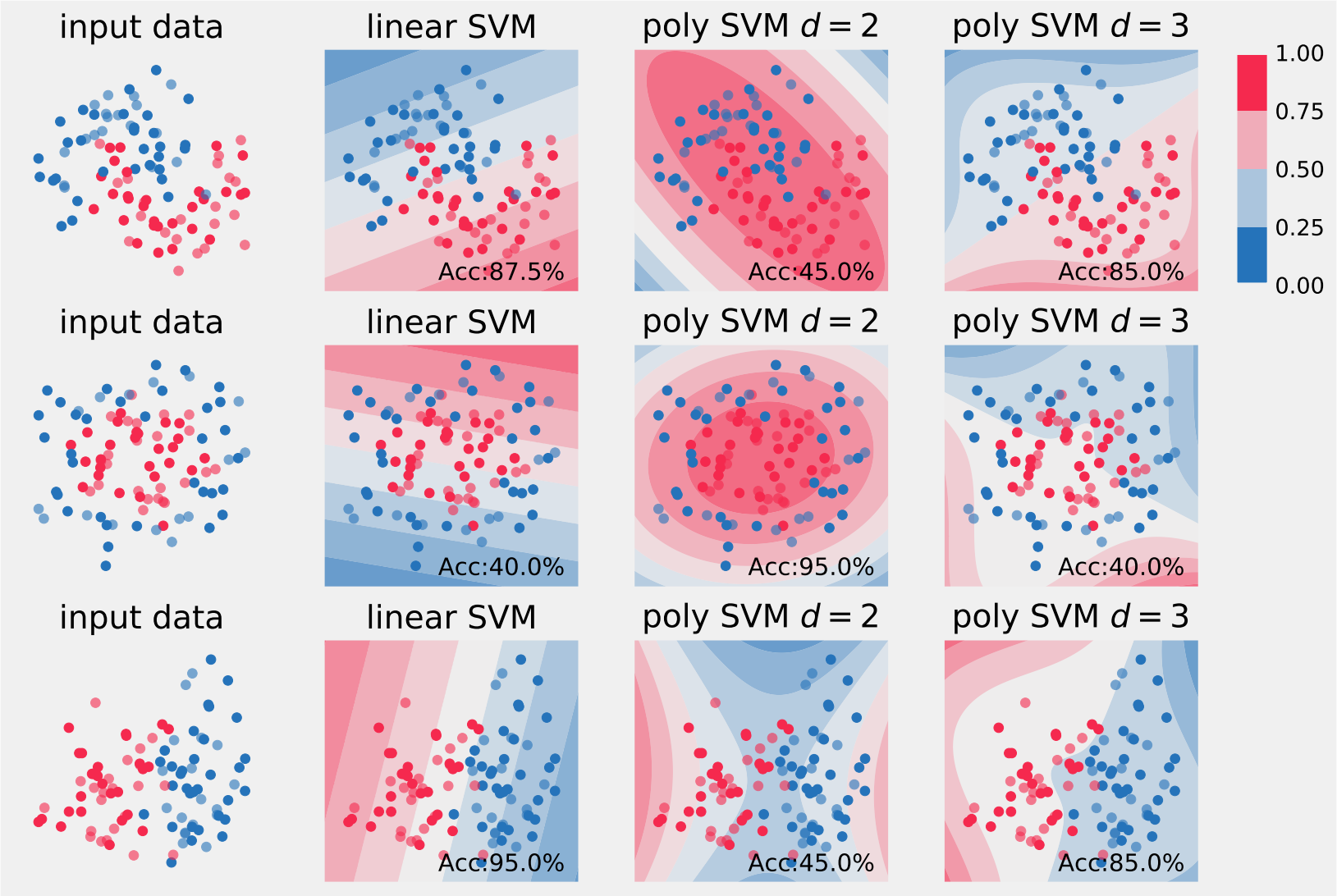
Kernel Trick: Intuition (pt4)



Here is an example using SVM-RBF, with contour lines for the score. The thickness of the outer circle for each observation $\mathbf{x}_i$ is proportional to $|\alpha_i| \leq 1$ (no black circle means $\alpha_i = 0$).

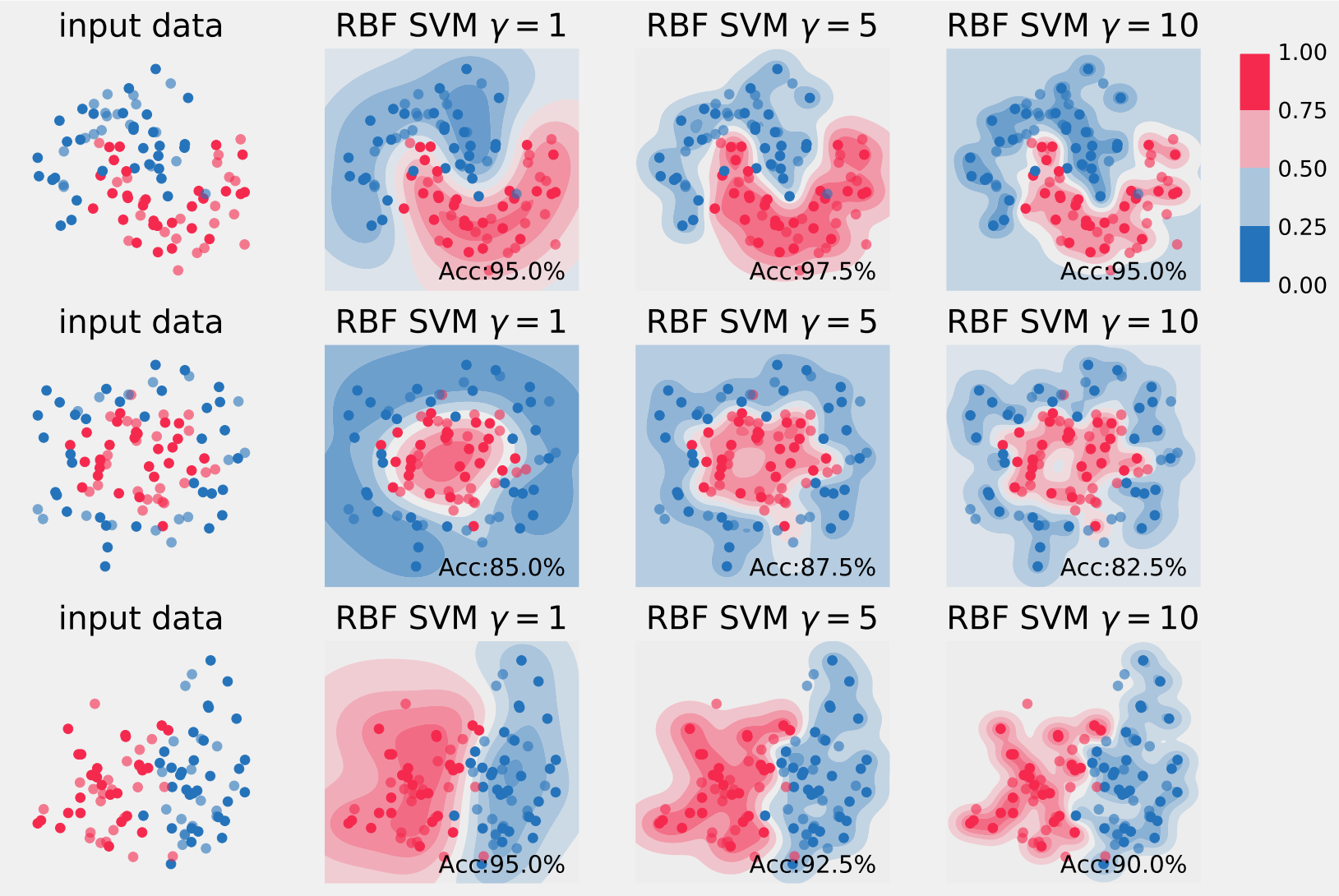
As you can see, only a fraction of the datapoints have non-null α_i. These are called **support vectors**. They typically lie near the class boundary. Only these datapoints are necessary to make predictions.

SVM results with polynomial kernel



Decision Boundaries for SVM using a linear and polynomial kernels.

SVM results with RBF kernel



Decision Boundaries for SVM using Gaussian kernels. The value of γ controls the smoothness of the boundary by setting the size of the neighbourhood.

Other Kernel Methods Exist

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**.

Kernel Methods Drawbacks

A major drawback to kernel methods is that the cost of evaluating the decision function is proportional to the number of training examples, because the i^{th} observation contributes a term $\alpha_i \kappa(\mathbf{x}, \mathbf{x}_i)$ to the decision function.

As we have seen, SVM somehow mitigates this by learning which examples contribute the most (the *support vectors*).

The cost of training is however still high for large datasets (*eg.* with tens of thousands of datapoints).

Kernel Methods and Neural Networks

Evidence that deep learning could outperform kernel SVM on large datasets emerged in 2006 when team lead by G. Hinton demonstrated that a neural network on the MNIST benchmark. The real tipping point occured with the 2012 paper by A. Krizhevsky, I. Sutskever and G. Hinton (see handout-00).

References

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 (2012) 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)