

# COMS30035, Machine learning: Kernels

James Cussens

`james.cussens@bristol.ac.uk`

School of Computer Science  
University of Bristol

27th September 2023

# Extracting features

- ▶ Let's have a look at [Mur22, p.370]

Suppose we're doing classification where each training data point is  $(\mathbf{x}, y)$ .

**Standard approach** Choose a function  $\phi$ . Map each raw data vector  $\mathbf{x}$  into a feature vector  $\phi(\mathbf{x})$ . Learn parameters using the feature vectors and class labels. When making a prediction for a test data vector  $\mathbf{x}'$  first encode as  $\phi(\mathbf{x}')$  and use learned model to make prediction.

**Kernel approach** Choose a kernel function  $k$  which (intuitively) measures similarity between raw data vectors  $k(\mathbf{x}_m, \mathbf{x}_n)$ . Learn parameters using the  $k(\mathbf{x}_m, \mathbf{x}_n)$  values and class labels. When making a prediction for a test data vector  $\mathbf{x}'$  use learned parameters and values of  $k(\mathbf{x}_n, \mathbf{x}')$  to make a prediction.

# Linear regression revisited

Consider a linear regression model:

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) \quad (1)$$

where

1.  $\mathbf{w}$  is the  $M$ -dimensional parameter vector to be learned from the data (which includes a component for the intercept)
2.  $\mathbf{x}$  is some datapoint, and
3.  $\phi(\mathbf{x})$  is the  $M$ -dimensional feature vector which  $\mathbf{x}$  gets mapped to by the *basis functions* [Bis06, §3.1].



# Dual representations

- ▶ Let  $N$  be the size of the data. Let  $\Phi$  be the *design matrix* whose  $n$ th row is just the feature vector for the  $n$ th datapoint (so it is basically ‘the data’). It turns out that we can reformulate in terms of an  $N$ -dimensional parameter vector  $\mathbf{a}$  as follows:

$$\mathbf{w} = \Phi^T \mathbf{a} \quad (2)$$

so that

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) = \mathbf{a}^T \Phi \phi(\mathbf{x}) \quad (3)$$

- ▶ This is known as a *dual representation*.
- ▶ So we have replaced an  $M$ -dimensional parameter vector with an  $N$ -dimensional one and moreover, to make a prediction for a new datapoint we have to use the entire training set (i.e.  $\Phi$ )
- ▶ On the face of it this does not seem such a great idea, (unless perhaps  $M$  is much bigger than  $N$ ).

# Kernel functions are scalar products in feature space

Suppose  $\mathbf{x}$  is some test datapoint. Let's have a look at  $\Phi\phi(\mathbf{x})$ . Suppose, for example, that we had 3 datapoints  $\mathbf{x}_1$ ,  $\mathbf{x}_2$  and  $\mathbf{x}_3$  and 2 features so

$$\Phi\phi(\mathbf{x}) = \begin{pmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) \\ \phi_1(\mathbf{x}_3) & \phi_2(\mathbf{x}_3) \end{pmatrix} \begin{pmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} \phi(\mathbf{x}_1)^T \phi(\mathbf{x}) \\ \phi(\mathbf{x}_2)^T \phi(\mathbf{x}) \\ \phi(\mathbf{x}_3)^T \phi(\mathbf{x}) \end{pmatrix} \quad (4)$$

If we define a *kernel function*

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \quad (5)$$

then we can write:

$$\Phi\phi(\mathbf{x}) = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}) \\ k(\mathbf{x}_2, \mathbf{x}) \\ k(\mathbf{x}_3, \mathbf{x}) \end{pmatrix} \quad (6)$$

# The kernel trick

So the prediction for datapoint  $\mathbf{x}$  in our example is:

$$\mathbf{a}^T \Phi \phi(\mathbf{x}) = \mathbf{a}^T \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}) \\ k(\mathbf{x}_2, \mathbf{x}) \\ k(\mathbf{x}_3, \mathbf{x}) \end{pmatrix} \quad (7)$$

- ▶ The key point is that **we just need the kernel function** to make the prediction.
- ▶ The ‘kernel trick’ is to evaluate the kernel function values, e.g.  $k(\mathbf{x}_1, \mathbf{x})$  without first computing  $\phi(\mathbf{x}_1)$  and  $\phi(\mathbf{x})$  and then computing their scalar product.
- ▶ This allows us to use very high-dimensional (even infinite dimensional!) feature spaces since features are never directly computed.

# Kernel functions and similarity

- ▶ A kernel function represents the degree of ‘similarity’ between its two arguments (so kernel functions are always symmetric).
- ▶ A high kernel value represents a high degree of similarity.
- ▶ Returning to our example, the prediction for  $\mathbf{x}$  is:

$$y(\mathbf{x}) = \mathbf{a}^T \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}) \\ k(\mathbf{x}_2, \mathbf{x}) \\ k(\mathbf{x}_3, \mathbf{x}) \end{pmatrix} = a_1 k(\mathbf{x}_1, \mathbf{x}) + a_2 k(\mathbf{x}_2, \mathbf{x}) + a_3 k(\mathbf{x}_3, \mathbf{x}) \quad (8)$$

- ▶ So the prediction for  $\mathbf{x}$  is a linear function of the ‘similarities’ between  $\mathbf{x}$  and each element of the training data.
- ▶ So unlike, say, linear regression it looks like we have to keep the entire training set around to make predictions.

# Kernel functions and similarity

- ▶ A kernel function represents the degree of ‘similarity’ between its two arguments (so kernel functions are always symmetric).
- ▶ A high kernel value represents a high degree of similarity.
- ▶ Returning to our example, the prediction for  $\mathbf{x}$  is:

$$y(\mathbf{x}) = \mathbf{a}^T \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}) \\ k(\mathbf{x}_2, \mathbf{x}) \\ k(\mathbf{x}_3, \mathbf{x}) \end{pmatrix} = a_1 k(\mathbf{x}_1, \mathbf{x}) + a_2 k(\mathbf{x}_2, \mathbf{x}) + a_3 k(\mathbf{x}_3, \mathbf{x}) \quad (8)$$

- ▶ So the prediction for  $\mathbf{x}$  is a linear function of the ‘similarities’ between  $\mathbf{x}$  and each element of the training data.
- ▶ So unlike, say, linear regression it looks like we have to keep the entire training set around to make predictions.
- ▶ But in fact we only need those  $\mathbf{x}_i$  where  $a_i \neq 0$ . (See later on *support vector machines*.)

# Learning with kernels

- ▶ So far we have focused on making predictions using a learned value of the dual parameter vector  $\mathbf{a}$ .
- ▶ If we needed to compute feature values  $\phi(\mathbf{x})$  to learn  $\mathbf{a}$ , then the advantage of using kernels would disappear.
- ▶ But the good news is that (for many models) we can learn  $\mathbf{a}$  just using kernels.
- ▶ So both learning and predicting just require evaluating kernel functions.

# The Gram matrix

- ▶ The *Gram matrix*  $\mathbf{K}$  is defined to be  $\Phi\Phi^T$ .
- ▶ So  $K_{nm} = \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m)$  is the ‘similarity’ between the  $n$ th and  $m$ th datapoint.
- ▶ Given data (the  $\mathbf{x}_n$ ) and a particular choice of kernel  $k$ , we can compute the Gram matrix  $\mathbf{K}$ .
- ▶  $\mathbf{K}$  is what we need for learning.
- ▶ Note that  $\mathbf{K}$  is symmetric.

# Learning with kernels example (1)

- ▶ Suppose we want to add a quadratic regulariser (aka *weight decay*) term when minimising the squared error on the training set  $\mathbf{w}$  [Bis06, §3.1.4].
- ▶ Then, if we were not using kernels, our goal [Bis06, (6.2)] is to minimise  $J(\mathbf{w})$  where:

$$J(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{ \mathbf{w}^T \phi(\mathbf{x}_n) - t_n \}^2 + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w} \quad (9)$$

- ▶  $N$  is the number of training datapoints,  $\lambda$  is the regularisation parameter and  $t_n$  is the observed target value of the  $n$ th training datapoint.



## Learning with kernels example (2)

- ▶ Let  $\mathbf{t} = (t_1, \dots, t_N)^T$  and let  $\mathbf{I}_N$  be the  $N \times N$  identity matrix then it turns out that setting:

$$\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{t} \quad (10)$$

- ▶ is equivalent to minimising  $J(\mathbf{w})$  (see [Bis06, §6.1] for the necessary algebra).
- ▶ The key point is that the dual parameters can be learned just using kernels and without computing any feature values.

# Recap

- ▶ Earlier we saw an example of:

1. learning:  $\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{t}$
2. and computing predicted values:

$$y(\mathbf{x}) = a_1 k(\mathbf{x}_1, \mathbf{x}) + a_2 k(\mathbf{x}_2, \mathbf{x}) + a_3 k(\mathbf{x}_3, \mathbf{x})$$

where both were done using only a kernels

- ▶ But for learning we needed to compute the kernel value for every pair of training datapoints, and for prediction we needed the entire training set.
- ▶ *Support vector machines* are a kernel-based method for classification which avoids this excessive computation.
- ▶ We still also need to address the question of which kernel function to use, more on this later ...

# Linear classification

Consider a simple linear model for two class classification:

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \quad (11)$$

where the *bias*  $b$  has been made explicit and where the class label is either -1 or 1.

- ▶ Let's assume (rather optimistically!) that the training dataset is linearly separable, so there is some  $\mathbf{w}$  and  $b$  such that  $y(\mathbf{x}_n) > 0$  if  $t_n = 1$  and  $y(\mathbf{x}_n) < 0$  if  $t_n = -1$ . (So  $t_n y(\mathbf{x}) > 0$  for all  $\mathbf{x}_n$ .)
- ▶ Typically there will be more than one hyperplane that separates the classes, so which one to choose?

# Maximum margin classifiers

- ▶ A natural choice (which has a theoretical justification) is to choose the hyperplane which maximises the *margin*: the distance from the hyperplane to the closest training datapoint.
- ▶ Let's look at this using a scikit-learn Jupyter notebook
- ▶ The training data points closest to the separating hyperplane are the *support vectors*.
- ▶ In a sense, they are the training datapoints 'that matter'.

# Maximising the margin

The learning (=optimisation) problem we have to solve is:

$$\arg \max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_n [t_n(\mathbf{w}^T \phi(\mathbf{x}_n) + b)] \right\} \quad (12)$$

But we can rescale  $\mathbf{w}$  and  $b$  so that for a point  $\mathbf{x}_n$  that is closest to the separating hyperplane

$$t_n(\mathbf{w}^T \phi(\mathbf{x}_n) + b) = 1 \quad (13)$$

and for all datapoints:

$$t_n(\mathbf{w}^T \phi(\mathbf{x}_n) + b) \geq 1 \quad n = 1, \dots, N \quad (14)$$

Plugging back into (12) we now just need to maximise  $\frac{1}{\|\mathbf{w}\|}$  which is the same as minimising:

$$\arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \quad (15)$$

subject to the linear inequality constraints (14).

This is a *quadratic programming* problem.

# Dual representation

The dual representation of the maximum margin problem is to maximise:

$$\tilde{L}(\mathbf{a}) = \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \quad (16)$$

subject to the constraints:

$$a_n \geq 0, \quad n = 1, \dots, N \quad (17)$$

$$\sum_{n=1}^N a_n t_n = 0 \quad (18)$$

- ▶ where, of course,  $k(\mathbf{x}_n, \mathbf{x}_m) = \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_m)$ .
- ▶ This is another quadratic program.
- ▶ This dual representation can be derived from the original one by using Lagrange multipliers (which are the  $a_n$ ).

# Support vector machines

- ▶ So to learn a max margin classifier we just need the  $k(\mathbf{x}_n, \mathbf{x}_m)$  values (i.e. the Gram matrix).
- ▶ We do not need to compute  $\phi(\mathbf{x}_n)$ , so  $\phi(\mathbf{x}_n)$  can be as high-dimensional as we like!
- ▶ To classify a new datapoint we compute (the sign of)

$$y(\mathbf{x}) = \sum_{n=1}^N a_n t_n k(\mathbf{x}, \mathbf{x}_n) + b \quad (19)$$

- ▶ So again only the kernel function is needed.
- ▶ Crucially, typically for most training datapoints  $\mathbf{x}_n$  we have  $a_n = 0$  and they are not needed for making predictions.
- ▶ The ones that are needed are called *support vectors*.

# Choosing a kernel

- ▶ You have already seen an SVM with a particular choice of kernel: the *linear kernel*  $k(\mathbf{x}_n, \mathbf{x}_m) = \mathbf{x}_n^T \mathbf{x}_m$ .
- ▶ Let's look at some more interesting kernels.
- ▶ We will use this useful Jupyter notebook
- ▶ The default kernel for NuSVC is the popular RBF kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2) \quad (20)$$

- ▶ The (implicit) feature space for the RBF kernel is infinite dimensional.



# Kernelise everything!

- ▶ So far, we have implicitly assumed that the original form of the data  $\mathbf{x}$  is a vector of real numbers.
- ▶ But data can also be: graphs, text documents, images, websites, whatever.
- ▶ For any sort of  $\mathbf{x}$  as long as we have a kernel function  $k(\mathbf{x}, \mathbf{x}')$ , measuring the similarity between  $\mathbf{x}$  and  $\mathbf{x}'$  we can apply kernel-based machine learning such as SVMs.
- ▶ This paper [KJM20] has an interesting flowchart which helps one choose an appropriate graph kernel.

# Choosing/Constructing kernels

- ▶ Suppose you have some, say, classification task and you decide to use an SVM approach.
- ▶ How do you decide which kernel to use?

# Choosing/Constructing kernels

- ▶ Suppose you have some, say, classification task and you decide to use an SVM approach.
- ▶ How do you decide which kernel to use?
- ▶ In practice, people typically use existing, known kernels.
- ▶ RBF is a popular choice.
- ▶ One can also construct a new kernel function from existing known kernels. See [Bis06, §6.2].
- ▶ If 'rolling your own' kernel, the function you define must be symmetric and also any Gram matrix  $\mathbf{K}$  must be a *positive semidefinite* matrix.
- ▶ scikit-learn lets you use your own Python functions as kernels, but does not check that your function is a valid kernel!

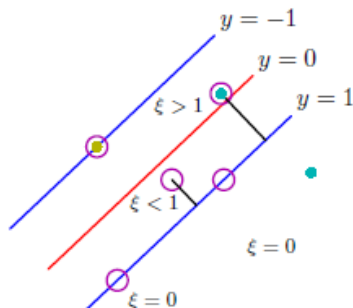
# SVMs in practice

- ▶ So far we have assumed that:
  1. the training data is linearly separable (in the implicit feature space)
  2. and that we have only two classes.
- ▶ Let's now remove these assumptions.

# Soft margins

- ▶ If we want a nice wide margin then we might have to allow training points to be inside the margin,
- ▶ or even on the wrong side of the decision boundary.
- ▶ Figure from [Bis06, p.332].
- ▶ Note: In the figure below three of the purple circles are missing their cyan dots. This is corrected in the hard copy book.

Illustration of the slack variables  $\xi_n \geq 0$ .  
Data points with circles around them are support vectors.



# Allowing data on the wrong side of the margin

Earlier we had the following optimisation problem:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} w^T w \\ \text{subject to} \quad & t_n(w^T \phi(x_n) + b) \geq 1 \end{aligned} \tag{21}$$

scikit learn actually solves the following optimisation (primal version)

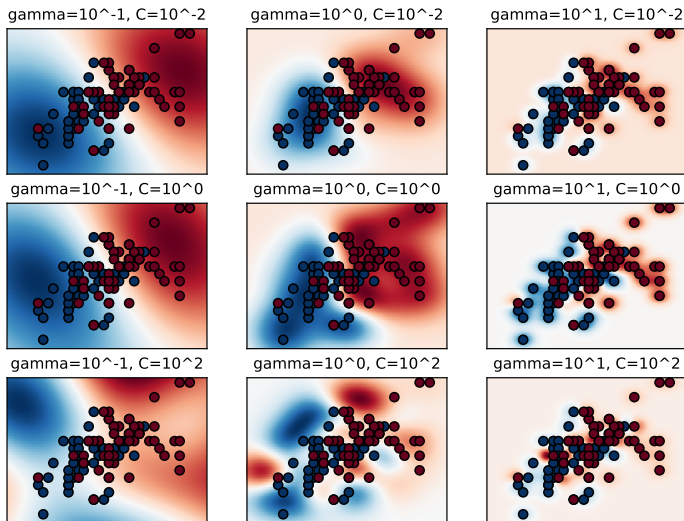
$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} w^T w + C \sum_{n=1}^N \xi_n \\ \text{subject to} \quad & t_n(w^T \phi(x_n) + b) \geq 1 - \xi_n, \\ & \xi_n \geq 0, n = 1, \dots, N \end{aligned} \tag{22}$$

# C as regularisation

$$\begin{aligned} \min_{w, b, \xi} \quad & \frac{1}{2} w^T w + C \sum_{n=1}^N \xi_n \\ \text{subject to} \quad & t_n(w^T \phi(x_n) + b) \geq 1 - \xi_n, \\ & \xi_n \geq 0, n = 1, \dots, N \end{aligned} \tag{23}$$

- ▶ “Points for which  $0 < \xi \leq 1$  lie inside the margin but on the correct side of the decision boundary ...”
- ▶ “... and those data points for which  $\xi_n > 1$  lie on the wrong side of the decision boundary and are misclassified” [Bis06, p.332]
- ▶ We now have a ‘soft margin’ where being on the wrong side of the margin is merely penalised and  $C$  is a regularisation parameter.
- ▶ Let’s look at part of the output of this Jupyter notebook to understand the effect of varying the value of  $C$ .

# RBF SVM classification





# scikit learn's decision function

The red/blue areas in the plots on the last slide represent values of the *decision function* which is defined in the scikit learn documentation.

$$\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b, \quad (24)$$

# RBF SVMs are non-parametric

- ▶ Recall that 'the' RBF kernel is:  $\exp(-\gamma\|x - x'\|^2)$
- ▶ SVMs with RBF kernels are *non-parametric*. The number of support vectors (and thus dual parameters) depends on the data (and value of  $\gamma$ ).

*Intuitively, the gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors. (scikit-learn docs)*

# Multiclass classification

- ▶ SVMs are fundamentally two class classifiers.
- ▶ If there are  $k$  classes, approaches include:
  - one-versus-one** where we train  $k(k - 1)/2$  SVM classifiers to distinguish between each pair of classes (and then take a 'vote' for the predicted class)
  - one-versus-the-rest** where we train  $k$  SVM classifiers to distinguish between each class and all other classes.
- ▶ In scikit-learn, SVC and NuSVC go for one-versus-one and LinearSVC does one-versus-the-rest.
- ▶ This Jupyter notebook provides an example of multi-class learning with various choices of kernel.



Christopher M. Bishop.

*Pattern Recognition and Machine Learning.*

Springer, 2006.



Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris.

A survey on graph kernels.

*Applied Network Science*, 5, 2020.



Kevin P. Murphy.

*Probabilistic Machine Learning: An introduction.*

MIT Press, 2022.