Machine Learning

CSE 142

Xin (Eric) Wang

Monday, November 8, 2021

Today

- Nonlinear kernel classifiers (Ch. 7.5)

Notes

- Midterm grades will be out tonight
  - "curve" your grades
  - midterm = Max(midterm, final)
  - TA Jing will discuss the midterm questions tomorrow
  - Go to the readers' office hours for questions about grading

- HW3 released last week, due by 11/17 (next Wed)
  - Start early!
  - TA Jing will provide some hints during the discussion sessions
  - Come to our office hours for help (TA, tutors and me)

Perceptron and SVM binary classifiers – summary

- In the perceptron model, we iteratively learn the linear discriminant $\boldsymbol{w}$, which is a linear combination of the misclassified input vectors $\boldsymbol{x}_i$

$$\boldsymbol{w} = \sum_{\boldsymbol{i}} \alpha_i y_i \boldsymbol{x}_i$$

α_i – # of times $\boldsymbol{x}_i$ was misclassified
y_i – class label of $\boldsymbol{x}_i$ {+1, –1}

  - After training, a new input is classified as a member of the positive class if $\boldsymbol{w}^T \boldsymbol{x} > 0$ (using homogeneous representation)

- In SVM learning, we solve a constrained optimization problem:

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1,\ldots,\alpha_n}{\arg\max} \left[-\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i \right]$$

$$\text{subject to } \alpha_i \geq 0, 1 \leq i \leq n \text{ and } \sum_{i=1}^{n} \alpha_i y_i = 0$$

which leads us to $\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$ where $\alpha_i = 0$ except for the support vectors

Non-homogeneous

Perceptron and SVM binary classifiers – summary

- In both perceptron and SVM learning, the linear decision boundary is a linear combination of the training data points
  - In the perceptron, just the ones that get misclassified in the iterative training
  - In the SVM, just the (few) support vectors
- Both learning methods have a dual form in which the dot product of training data points $\boldsymbol{x}_i^T \boldsymbol{x}_j$ is part of the main computation
  - All values of $\boldsymbol{x}_i^T \boldsymbol{x}_j$ are contained in the Gram matrix

$$\boldsymbol{G} = \boldsymbol{X}^T \boldsymbol{X} = [\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \dots \quad \boldsymbol{x}_k]^T [\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \dots \quad \boldsymbol{x}_k]$$

so it's often efficient to compute the Gram matrix in advance and index into it, rather than computing the dot products over and over again

Perceptron and SVM binary classifiers – summary

- Perceptron and (basic) SVM learning only converge to a solution if the training data is linearly separable
- If the data is not linearly separable, we can employ a soft margin SVM, where we introduce a *slack variable* ξ_i for each training data point, allowing for margin errors:

$$\boldsymbol{w}^T\boldsymbol{x}_i - t \geq 1 - \xi_i \qquad \xi_i > 0 \;\rightarrow\; \boldsymbol{x}_i \text{ is not a support vector}$$

and leading to this optimization problem:

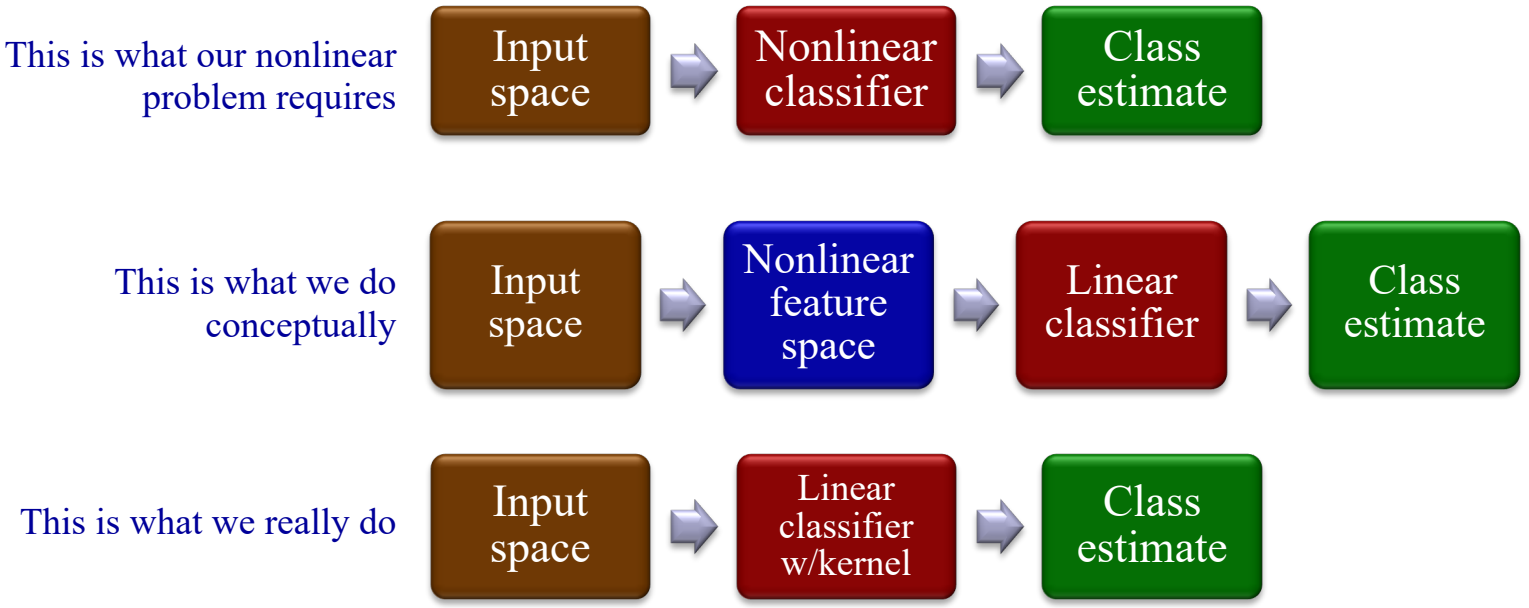
$$\mathbf{w}^*, t^*, \xi_i^* = \arg\min_{\mathbf{w},t,\xi_i}\left[\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i\right]$$

$$\text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1-\xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n$$

where the complexity parameter C is a user-defined parameter that allows for a tradeoff between maximizing the margin (lower C) and minimizing the margin errors (higher C)

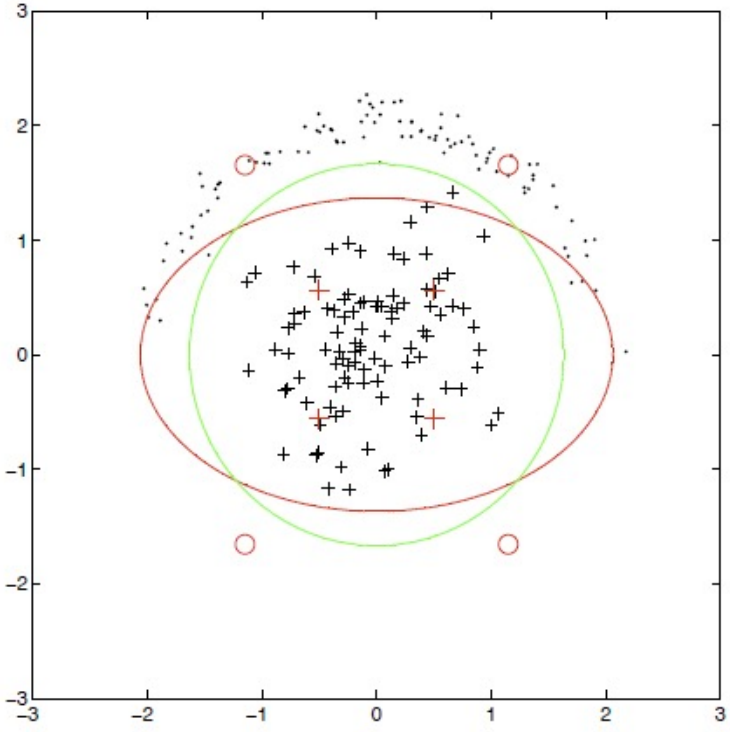
Nonlinear kernel classifiers

- In many problems, linear decision boundaries just won't do the job.
- We can adapt our linear methods to learn (some) nonlinear decision boundaries by transforming the data nonlinearly to a feature space in which linear classification can be applied
  - These are kernel methods – the ***kernel trick***!

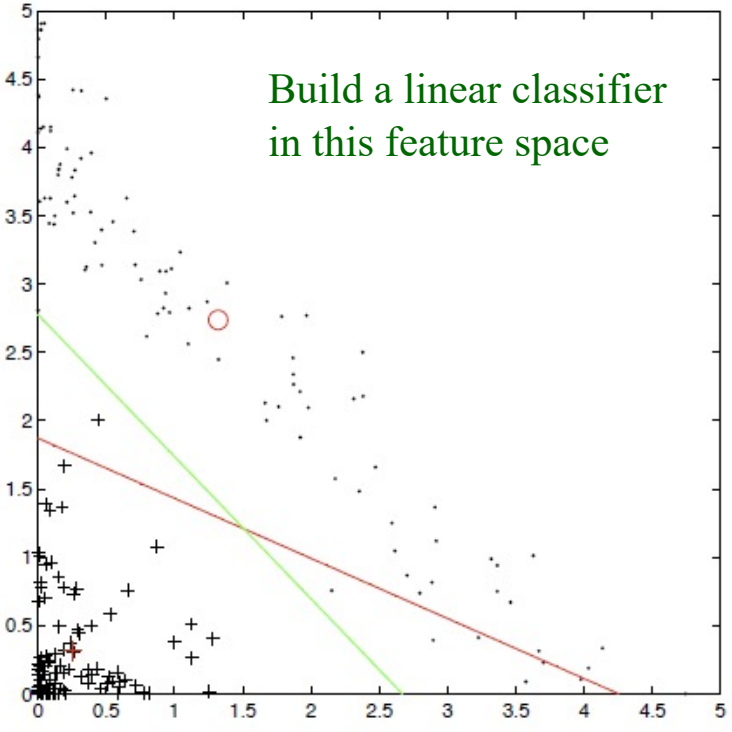


Example: a quadratic decision boundary

Input space



Feature space – square the input features



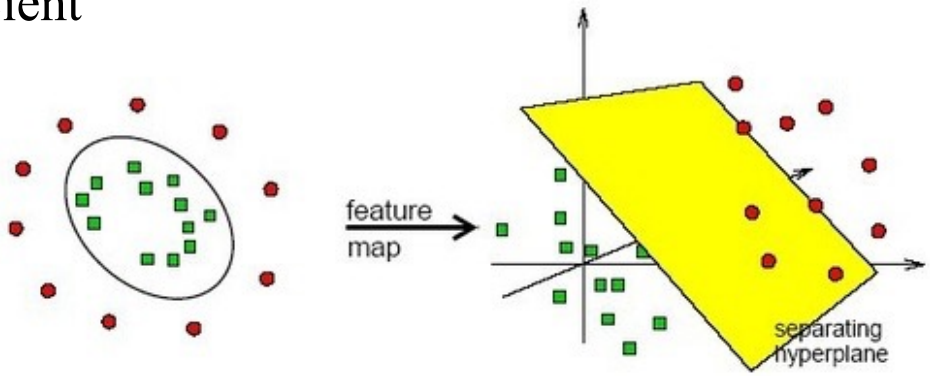
Red – Basic linear classifier decision boundary

Green – Perceptron decision boundary

In kernel methods we don't actually construct the feature space – rather, we perform all operations in the input space

The kernel trick

- In machine learning, the "kernel trick" is a way of mapping features into another (often higher dimensional) space to make the data linearly separable, *without having to compute the mapping explicitly*.
- The dot product operation in a linear classifier $\boldsymbol{x}_1 \cdot \boldsymbol{x}_2$ is replaced by a kernel function $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2)$ that computes the dot product of the values $(\boldsymbol{x}_1', \boldsymbol{x}_2')$ in the new (linearly separable) space.
  - Again, without having to compute the mapping from $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ to $(\boldsymbol{x}_1', \boldsymbol{x}_2')$
  - So it's both effective and efficient
- Let's see an example….



The kernel trick

- In the original feature space, the two classes (o's and x's) are not linearly separable
- So let's map $\boldsymbol{p} = (x_1, x_2)$ to a new space $\boldsymbol{q} = (z_1, z_2, z_3)$ via the transformation $\varphi(\boldsymbol{p})$:

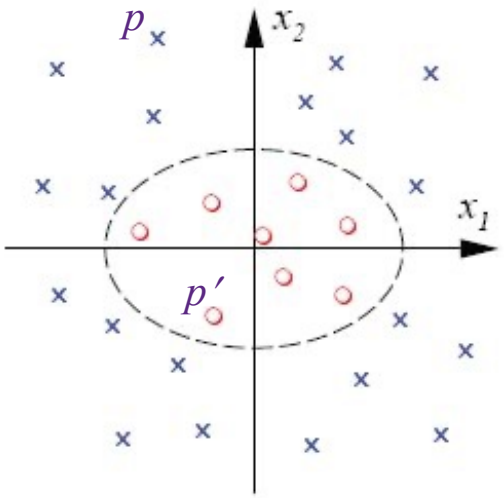
$$z_1 = x_1^2$$
$$z_2 = x_2^2$$
$$z_3 = \sqrt{2}x_1x_2$$

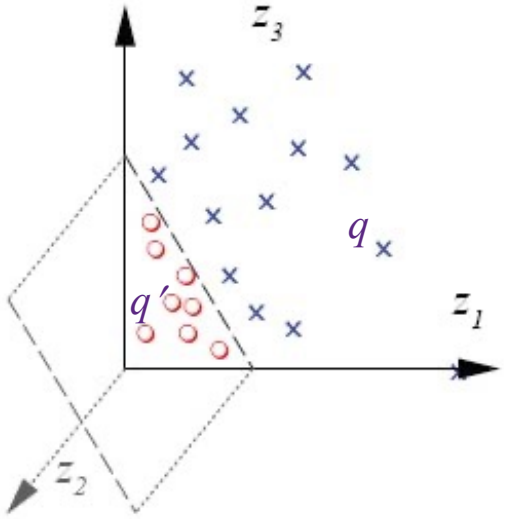
  where, it turns out, the o's and x's are linearly separable.

- A dot product in the new space:

$$\begin{aligned}\boldsymbol{q} \cdot \boldsymbol{q}' &= z_1 z_1' + z_2 z_2' + z_3 z_3' \\ &= x_1^2 x_1'^2 + x_2^2 x_2'^2 + \sqrt{2}x_1x_2\sqrt{2}x_1' x_2' \\ &= (x_1 x_1' + x_2 x_2')^2 \\ &= (\boldsymbol{p} \cdot \boldsymbol{p}')^2 = \kappa(\boldsymbol{p_1}, \boldsymbol{p_2})\end{aligned}$$

  is merely the square of the original dot product!





Feature transformation and the kernel trick

- The kernel trick is widely used in machine learning
- Assumption: achieving linear separation is worth the effort
  - There are non-linear classifiers, but linear classification tends to be simple and fast
- Assumption: the dot product is the key computation
  - Yes, for a linear classifier
  - So we just replace the dot product with the kernel function
- How do we find the mapping that will make the data linearly separable?
  - Good question!
  - Insight into the data, trial and error, …
  - Are there principled ways to determine such a transformation?

The kernel function – summary

- We have linear methods that use the dot product among instances, $\boldsymbol{x}_1{}^T \boldsymbol{x}_2$ (also written $\boldsymbol{x}_1 \cdot \boldsymbol{x}_2$)
  - But if our data is not appropriate for a linear model, we can't use these methods!
- So… we find a transformation $\varphi(\boldsymbol{x})$ of the input space into a feature space that makes the data linearly separable
- Then, for training and subsequent classification, we conceptually transform inputs $\boldsymbol{x}$ into the feature space $\varphi(\boldsymbol{x})$ to learn a linear classifier and for classifying new instances
- But we don't actually have to do this. Instead, we define a kernel function $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2)$ that performs the dot product in the feature space – i.e., $\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2)$
  - $\kappa : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}$

Kernel perceptron

Learns a nonlinear decision boundary

Algorithm KernelPerceptron(D, κ) – perceptron training algorithm using a kernel.

Input : labelled training data D in homogeneous coordinates;
kernel function κ.

Output : coefficients α_i defining non-linear decision boundary.

$\alpha_i \leftarrow 0$ for $1 \leq i \leq |D|$;
converged←false;
while *converged* = false **do**
    converged←true;
    for $i = 1$ to $|D|$ **do**
        if $y_i \sum_{j=1}^{|D|} \alpha_j y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \leq 0$ **then**
            $\alpha_i \leftarrow \alpha_i + 1$;
            converged←false;
        end
    end
end

replaces $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$

Kernel perceptron

- The kernel perceptron doesn't learn a linear discriminant $\boldsymbol{w}$
  - It learns the α_i parameters (see the dual form of the learning algorithm)
- Classifying a new instance does not use $\boldsymbol{w}^T\boldsymbol{x} > 0$ – instead, it evaluates

$$\sum_{i=1}^{n} \alpha_i y_i \, \kappa(\boldsymbol{x}, \boldsymbol{x}_i) > 0$$

- This is $O(n)$, involving all training data with non-zero α_i
- This approach will be more efficient with SVMs, since $\alpha_i \neq 0$ only for the support vectors!
  - So let's look at kernel SVMs…

Kernel SVM

- The kernel SVM is the same basic idea as the kernel perceptron – replace the dot product $\boldsymbol{x}_i^T \boldsymbol{x}_j$ with a kernel function $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$ that captures the nonlinear mapping of the input space to the feature space, where the data are linearly separable

- We can then replace the Gram matrix **G** with the kernel matrix **K**, and use entries of **K** in the learning computation

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1, \ldots, \alpha_n}{\arg\max} -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} \alpha_i$$

$$\text{subject to } \alpha_i \geq 0, 1 \leq i \leq n \text{ and } \sum_{i=1}^{n} \alpha_i y_i = 0$$

Kernel SVM

- After learning the α_i parameters, we can then classify a new instance $\boldsymbol{x}$ using

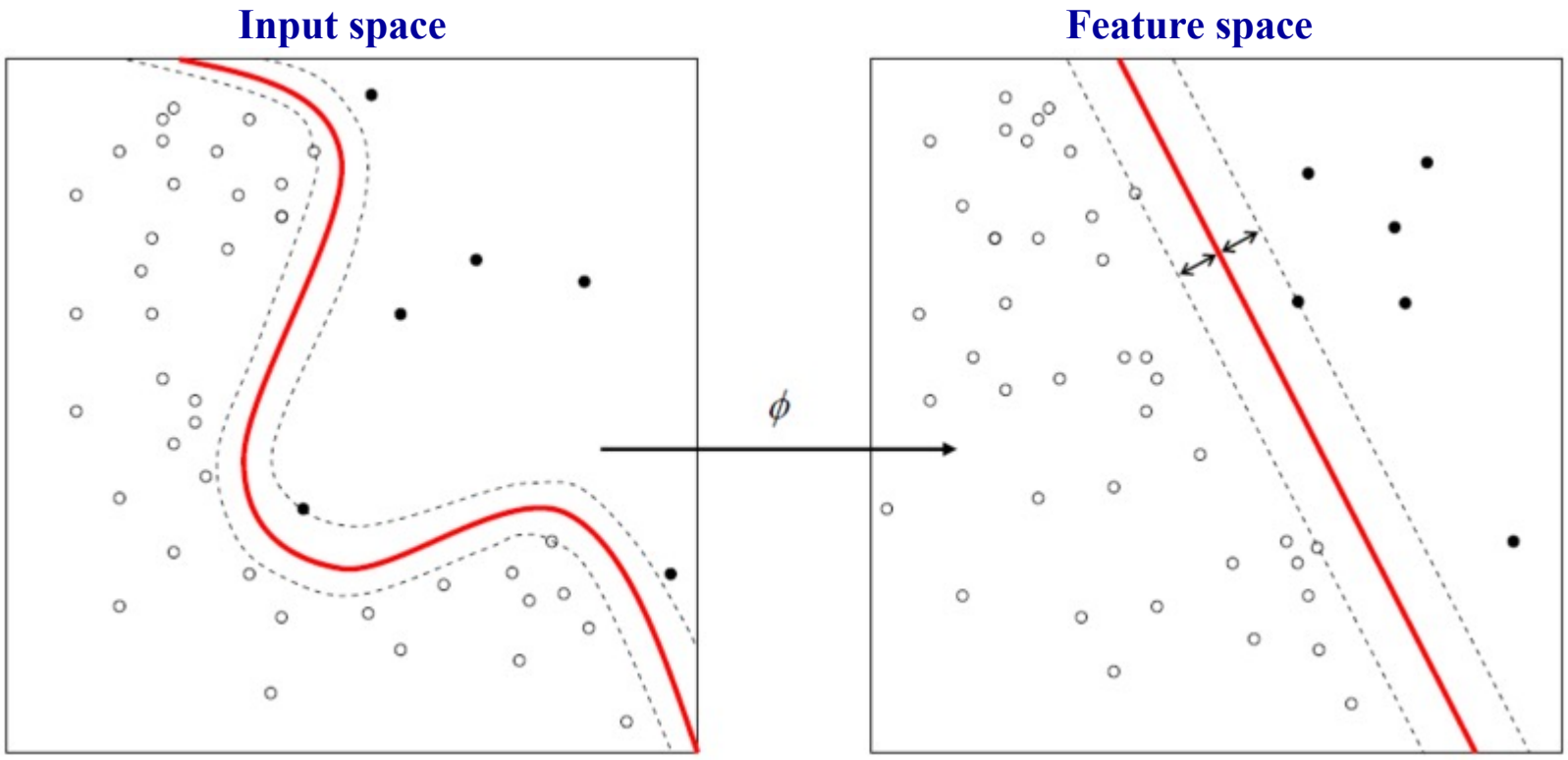
$$\sum_{i=1}^{n} \alpha_i y_i \, \kappa(\boldsymbol{x}, \boldsymbol{x}_i) > 0$$

- This sum is only over the support vectors (with non-zero α_i), so it's an efficient computation

- To learn a soft margin kernel SVM, we can include slack variables ξ_i and the complexity parameter C
  - Just like we did before

Kernel SVM

With kernel functions, SVMs can be used as non-linear classifiers



Some kernel functions

- The linear kernel:

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1{}^T \boldsymbol{x}_2$$

- The polynomial kernel:

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1{}^T \boldsymbol{x}_2 + c)^d$$

- The Gaussian kernel

$$\kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) = \exp\left(\frac{-\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{2\sigma^2}\right)$$

This is also known as a radial basis function (RBF) kernel

It is essentially a measure of similarity between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, scaled by σ

- The larger σ is, the more effect a distant point $\boldsymbol{x}_i$ will have

How to choose a kernel function

- Selecting a kernel function entails:
  - Choosing the function family (polynomial, RBF, etc.)
  - Determining the parameters of the function
    - (c, d) for polynomial
    - σ for RBF
    - Etc.
- Various optimization methods exist for making these choices, using cross-validation (randomly partitioning the experimental data into training and validation parts, repeatedly)
- Knowledge of the problem space can be helpful
  - Collected wisdom: "In cases like *this*, try *that* kernel function…"

Distance metrics and clustering

Chapter 8 in the textbook