Support Vector Machine (contd) and Kernel Methods

CS771: Introduction to Machine Learning

Announcements

- Mid-sem grading almost finished. Hope to release marks by tomorrow if not today
- Quiz 1 and Quiz 2 marks already released
  - We will clear remaining regrading requests in 1-2 days
- Will have 2 make-up lectures (dates announced later)

Plan for today

- Solving the SVM optimization problem efficiency using co-ordinate ascent
- SVM for multi-class classification
- SVM for regression
- Kernel methods for learning nonlinear models

A Co-ordinate Ascent Algorithm for SVM

- Recall the dual objective of soft-margin SVM (assuming no bias b)

$$\underset{\mathbf{0}\leq\boldsymbol{\alpha}\leq C}{\operatorname{argmax}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{m,n=1}^{N} \alpha_m \alpha_n y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$$

- Focusing on just one of the components of $\boldsymbol{\alpha}$ (say α_n), the objective becomes

Note that $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$

Can compute these in the beginning itself

Can efficiently compute it if we also store $\boldsymbol{w}$. It is equal to $\boldsymbol{w}^\top \boldsymbol{x}_n - \alpha_n y_n \|\boldsymbol{x}_n\|^2$

$$\underset{\mathbf{0}\leq\alpha_n\leq C}{\operatorname{argmax}} \ \alpha_n - \frac{1}{2}\alpha_n^2 \|\boldsymbol{x}_n\|^2 - \frac{1}{2}\alpha_n y_n \sum_{m\neq n} \alpha_m y_m \boldsymbol{x}_m^\top \boldsymbol{x}_n$$

- The above is a simple quadratic maximization of a concave function: Global maxima
- If constraint violated, project α_n in $[0, C]$: If $\alpha_n < 0$, set it to 0, if $\alpha_n > C$, set it to C
- Can cycle through each coordinate α_n in a random or cyclic fashion

Multi-class SVM

- Multiclass SVMs (assuming $K > 2$ classes) use K wt vectors $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$

  Prediction at test time: $\hat{y}_* = \text{argmax}_{k \in \{1,2,\ldots,K\}} \boldsymbol{w}_k^\top \boldsymbol{x}_*$

- Like binary SVM, can formulate a maximum-margin problem (without or with slacks)

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{k=1}^{K} \frac{||\boldsymbol{w}_k||^2}{2} \quad \text{s.t.} \quad \boldsymbol{w}_{y_n}^\top \boldsymbol{x}_n \geq \boldsymbol{w}_k^\top \boldsymbol{x}_n + 1 \quad \forall k \neq y_n$$

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{k=1}^{K} \frac{||\boldsymbol{w}_k||^2}{2} + C \sum_{n=1}^{N} \xi_n \quad \text{s.t.} \quad \boldsymbol{w}_{y_n}^\top \boldsymbol{x}_n \geq \boldsymbol{w}_k^\top \boldsymbol{x}_n + 1 - \xi_n \quad \forall k \neq y_n$$

Score on correct class

Score on an incorrect class $k \neq y_n$

- The version with slack corresponds to minimizing a multi-class hinge loss

$$\mathcal{L}(\boldsymbol{W}) = \sum_{n=1}^{N} \max\left\{0, 1 + \max_{k \neq y_n} \boldsymbol{w}_k^\top \boldsymbol{x}_n - \boldsymbol{w}_{y_n}^\top \boldsymbol{x}_n\right\} + \frac{\lambda}{2} \sum_{k=1}^{K} \|\boldsymbol{w}_k\|^2$$

Crammer-Singer Multi-class SVM

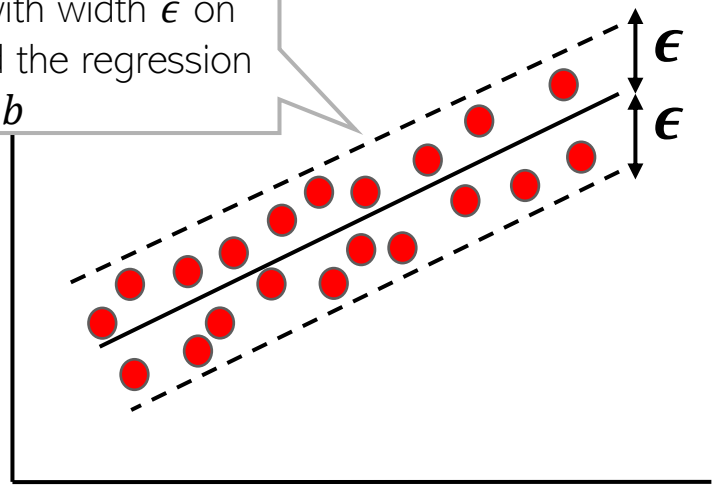
Loss=0 if score on correct class is at least 1 more than score on next best scoring class

Support Vector Regression (SVR)

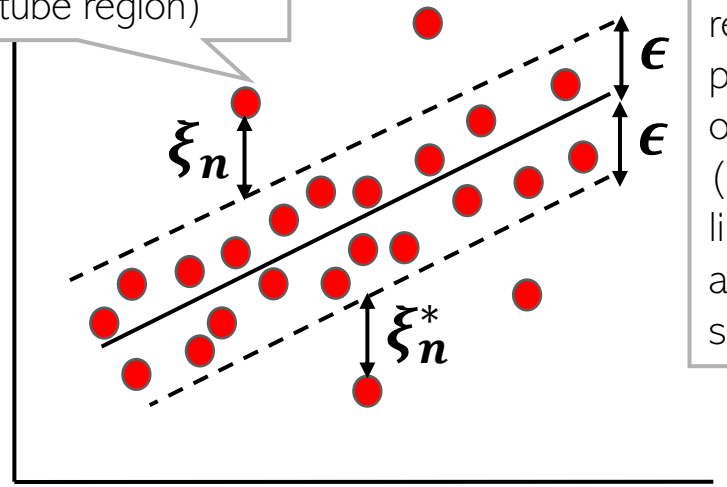
No loss if the prediction is within ϵ of the true output

- SVR is a variant of SVM for regression problems. Uses ϵ-insensitive loss

Want all the examples to fall inside the tube with width ϵ on each side around the regression line $y = \boldsymbol{w}^\top \boldsymbol{x} + b$



Also allow some slacks (some points may fall outside the tube region)



SVR is more robust than linear regression in the presence of outliers (regression line/curve is not affected much by such points)

SVR can also be made nonlinear using kernels

$$\text{Minimize} \quad \frac{\|\boldsymbol{w}\|^2}{2}$$

$$\text{Subject to} \quad y_n - \boldsymbol{w}^\top \boldsymbol{x}_n - b \leq \epsilon$$

$$\boldsymbol{w}^\top \boldsymbol{x}_n + b - y_n \leq \epsilon$$

$$\text{Minimize} \quad \frac{\|\boldsymbol{w}\|^2}{2} + C \sum_{n=1}^{N} (\xi_n + \xi_n^*)$$

$$\text{Subject to} \quad y_n - \boldsymbol{w}^\top \boldsymbol{x}_n - b \leq \epsilon + \xi_n$$

$$\boldsymbol{w}^\top \boldsymbol{x}_n + b - y_n \leq \epsilon + \xi_n^*$$

$$\xi_n, \xi_n^* \geq 0$$

Nonlinear Models using Kernels

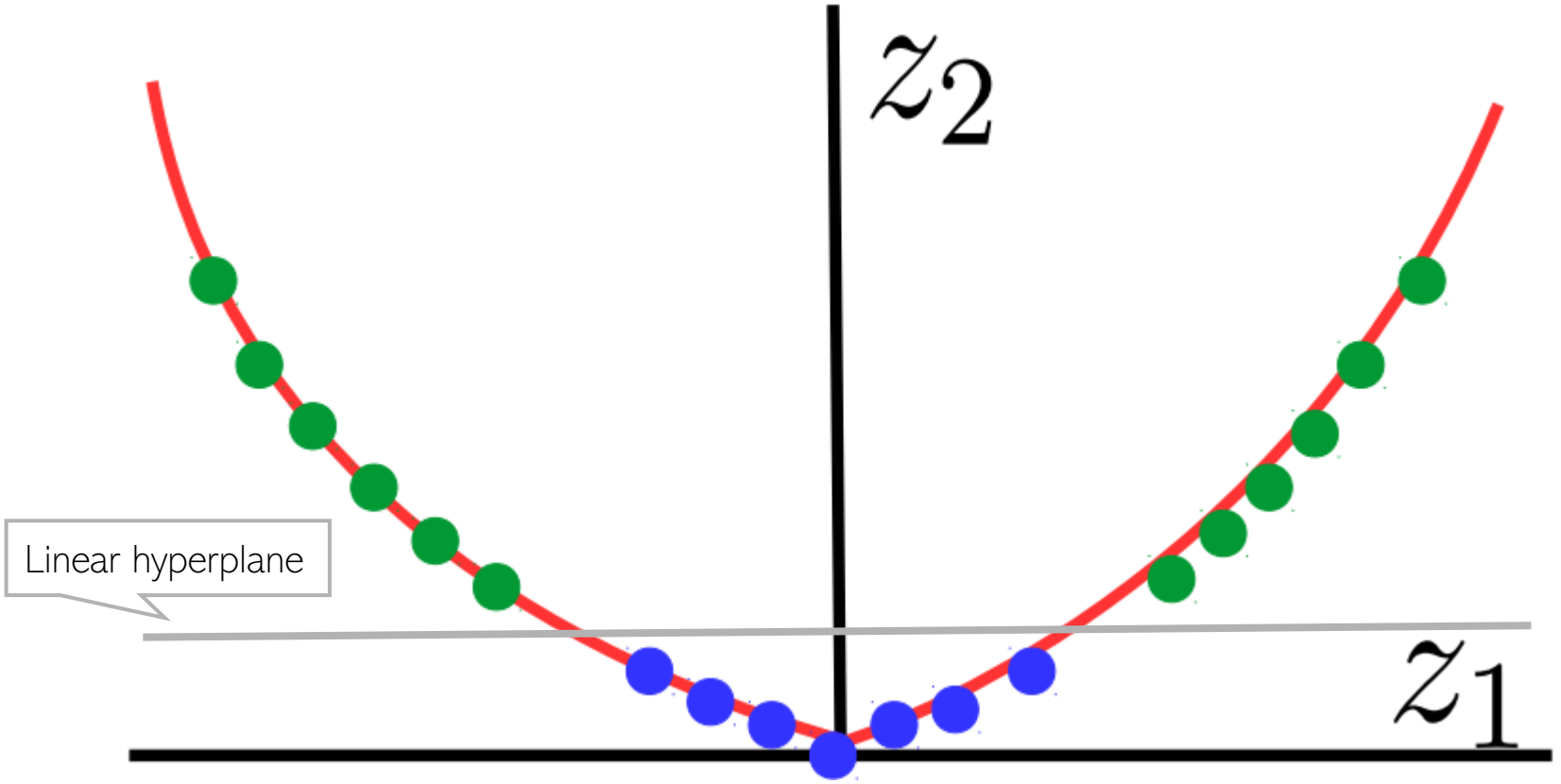
Linear Models for Nonlinear Problems?

- Consider the following one-dimensional inputs from two classes

- Can't separate using a linear hyperplane

Linear Models for Nonlinear Problems?

- Consider mapping each $\boldsymbol{x}$ to two-dimensions as $\boldsymbol{x} \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, x^2]$



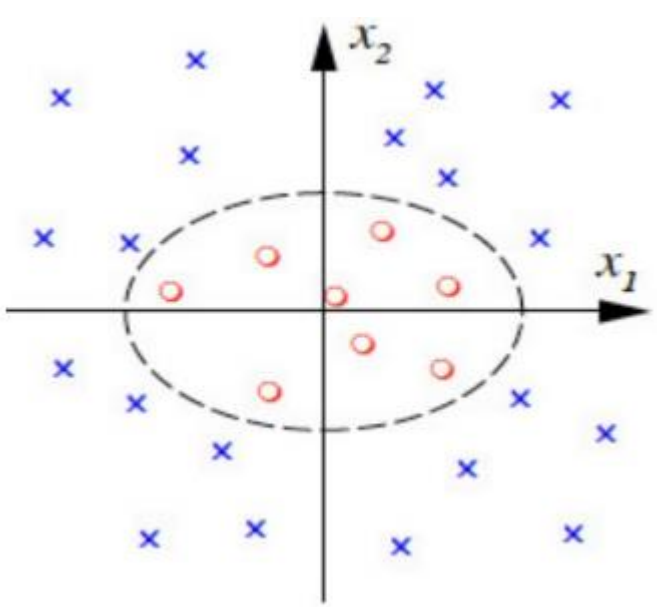
- Classes are now linearly separable in the two-dimensional space

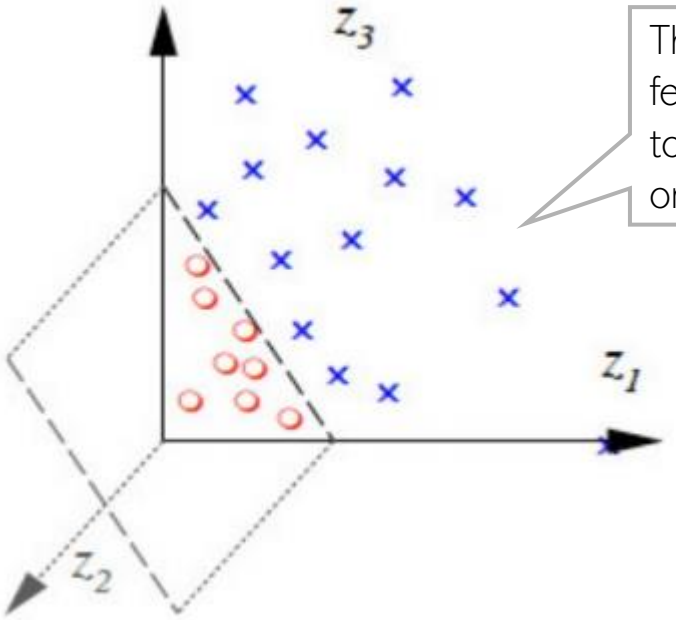
Linear Models for Nonlinear Problems

- Can assume a feature mapping ϕ that maps/transforms the inputs to a "nice" space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)} x_1 x_2, x_2^2)$$



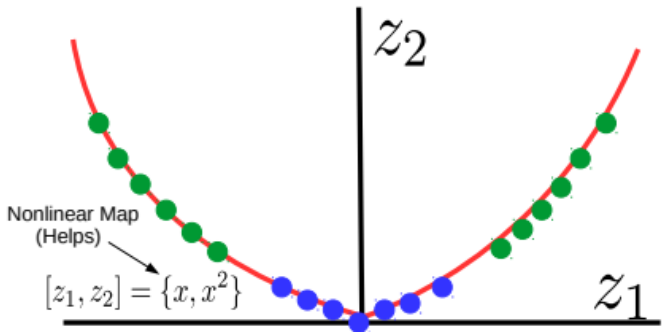


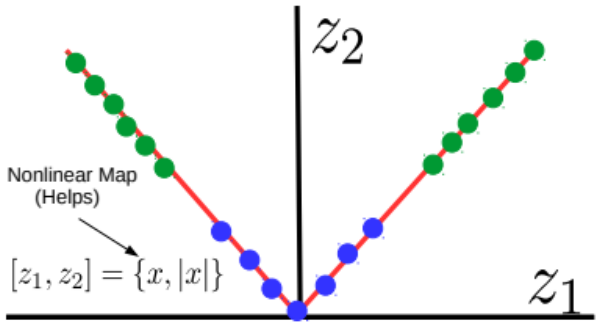
The linear model in the new feature space corresponds to a nonlinear model in the original feature space

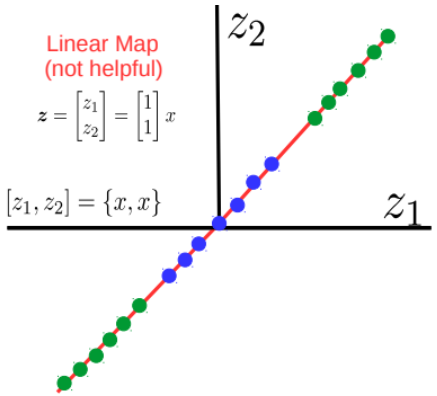
- .. and then happily apply a linear model in the new space!

Not Every Mapping is Helpful

- Not every higher-dim mapping helps in learning nonlinear patterns
- Must be a nonlinear mapping
- For the nonlin classfn problem we saw earlier, consider some possible mappings







How to get these "good" (nonlinear) mappings?

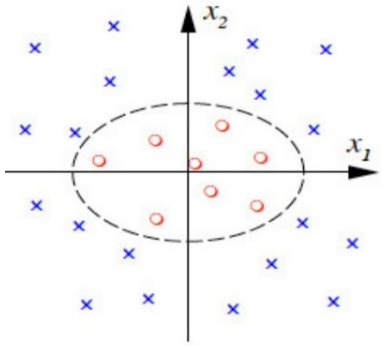
- Learn good mappings from data itself (e.g., deep learning or distance metric learning)
- Use pre-defined "good" mappings (e.g., defined by kernel functions - today's topic)

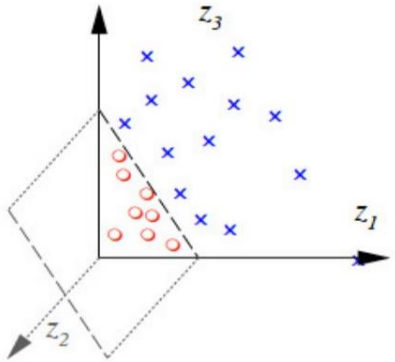
Even if I knew a good mapping, it seems I need to apply it for every input. Won't this be computationally expensive?

Also, the number of features will increase? Will it not slow down the learning algorithm?

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)}x_1x_2, x_2^2)$$





Thankfully, using kernels, you don't need to compute these mappings explicitly

The kernel will define an "implicit" feature mapping

Important: The idea can be applied to any ML algo in which training and test stage only require computing distances/similarities b/w inputs

In a high-dim space implicitly defined by an underlying mapping ϕ associated this this kernel function $k(.,.)$

- Kernel: A function $k(.,.)$ that gives dot product similarity b/w two inputs, say x_n and x_m

$$k(x_n, x_m) = \phi(x_n)^\top \phi(x_m)$$

Important: As we will see, computing $k(.,.)$ does not require computing the mapping ϕ

Some Pre-defined Kernel Functions

- Linear kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z}$
- Quadratic Kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ or $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$
- Polynomial Kernel (of degree d): $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d$ or $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$
- Radial Basis Function (RBF) or "Gaussian" Kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma \|\boldsymbol{x} - \boldsymbol{z}\|^2]$
  - Gaussian kernel gives a similarity score between 0 and 1
  - $\gamma > 0$ is a hyperparameter (called the kernel bandwidth parameter)
  - The RBF kernel corresponds to an infinite dim. feature space $\mathcal{F}$ (i.e., you can't actually write down or store the map $\boldsymbol{\phi}(\boldsymbol{x})$ explicitly – but we don't need to do that anyway ☺)
  - Also called "stationary kernel": only depends on the distance between $\boldsymbol{x}$ and $\boldsymbol{z}$ (translating both by the same amount won't change the value of $k(\boldsymbol{x}, \boldsymbol{z})$)
- Which kernel to use or its hyperparams (e.g., d, γ) values can be set via cross-val.

Several other kernels proposed for non-vector data, such as trees, strings, etc

Remember that kernels are a notion of similarity between pairs of inputs

Kernels can have a pre-defined form or can be learned from data (a bit advanced for this course)

Controls how the distance between two inputs should be converted into a similarity

Kernels as (Implicit) Feature Maps

- Consider two inputs (in the same two-dim feature space): $\boldsymbol{x} = [x_1, x_2], \boldsymbol{z} = [z_1, z_2]$
- Suppose we have a function $k(.,.)$ which takes two inputs $\boldsymbol{x}$ and $\boldsymbol{z}$ and computes

Called the "kernel function"

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$$

Can think of this as a notion of similarity b/w $\boldsymbol{x}$ and $\boldsymbol{z}$

This is not a dot/inner product similarity but similarity using a more general function of $\boldsymbol{x}$ and $\boldsymbol{z}$ (square of dot product)

$$= (x_1 z_1 + x_2 z_2)^2$$

$$= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2$$

$$= \left(x_1^2, \sqrt{2} x_1 x_2, x_2^2\right)^\top \left(z_1^2, \sqrt{2} z_1 z_2, z_2^2\right)$$

$$= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$$

Thus kernel function $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ implicitly defined a feature mapping $\boldsymbol{\phi}$ such that for $\boldsymbol{x} = [x_1, x_2]$, $\boldsymbol{\phi}(\boldsymbol{x}) = \left(x_1^2, \sqrt{2} x_1 x_2, x_2^2\right)$

Dot product similarity in the new feature space defined by the mapping $\boldsymbol{\phi}$

Remember that a kernel does two things: Maps the data implicitly into a new feature space (feature transformation) and computes pairwise similarity between any two inputs under the new feature representation

- Also didn't have to compute $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$. Defn $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ gives that

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as $k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma \|\boldsymbol{x} - \boldsymbol{z}\|^2]$
- Using this kernel corresponds to mapping data to infinite dimensional space

$$k(x, z) = \exp[-(x - z)^2] \quad \text{(assuming } \gamma = 1 \text{ and } x \text{ and } z \text{ to be scalars)}$$

$$= \exp(-x^2)\exp(-z^2)\exp(2xz)$$

$$= \exp(-x^2)\exp(-z^2)\sum_{0=1}^{\infty}\frac{2^k x^k z^k}{k!}$$

$$= \phi(x)^\top \phi(z)$$

Thus an infinite-dim vector (ignoring the constants coming from the 2^k and $k!$ terms

- Here $\phi(\boldsymbol{x}) = [\exp(-x^2)x^0, \exp(-x^2)x^1, \exp(-x^2)x^2, \exp(-x^2)x^3, \ldots, \exp(-x^2)x^\infty]$
- But again, note that we never need to compute $\phi(\boldsymbol{x})$ to compute $k(x, z)$
  - $k(x, z)$ is easily computable from its definition itself ($\exp[-(x - z)^2]$ in this case)

Kernel Function: Some Other Aspects

- Not every function of the form $k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$ is a kernel function
- k must satisfy Mercer's Condition
  - k must define a dot product for some Hilbert Space
  - Above is true if k is symmetric and positive semi-definite (p.s.d.) function (though there are exceptions; there are also "indefinite" kernels)

$$k(\boldsymbol{x}, \boldsymbol{z}) = k(\boldsymbol{z}, \boldsymbol{x})$$

$$\iint f(\boldsymbol{x}) k(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) d\boldsymbol{x} d\boldsymbol{z} \geq 0$$

For all "square integrable" functions f (such functions satisfy $\int f(\boldsymbol{x})^2 d\boldsymbol{x} < \infty$

Loosely speaking a PSD function here means that if we evaluate this function for N inputs (N^2 pairs) then the $N \times N$ matrix will be PSD (also called a kernel matrix)

Can easily verify that the Mercer's Condition holds for these

- Let k_1, k_2 be two kernel functions then the following are as well
  - $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) + k_2(\boldsymbol{x}, \boldsymbol{z})$: simple sum
  - $k(\boldsymbol{x}, \boldsymbol{z}) = \alpha k_1(\boldsymbol{x}, \boldsymbol{z})$: scalar product with $\alpha > 0$
  - $k(\boldsymbol{x}, \boldsymbol{z}) = k_1(\boldsymbol{x}, \boldsymbol{z}) k_2(\boldsymbol{x}, \boldsymbol{z})$: direct product of two kernels

Can also combine these rules and the resulting function will also be a kernel function

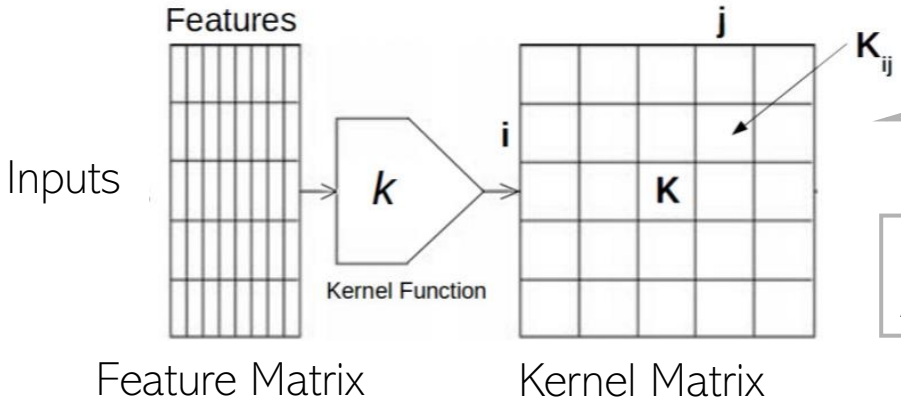
Kernel Matrix

- Kernel based ML algos work with kernel matrices rather than feature vectors
- Given N inputs, the kernel function k can be used to construct a Kernel Matrix $\boldsymbol{K}$
- The kernel matrix $\boldsymbol{K}$ is of size $N \times N$ with each entry defined as

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

Note again that we don't need to compute ϕ and this dot product explicitly

- K_{ij} : Similarity between the i^{th} and j^{th} inputs in the kernel induced feature space ϕ



K is a symmetric and positive semi-definite matrix

$z^\top Kz \geq 0 \; \forall z \in \mathbb{R}^N$

Also, all eigenvalues of K are non-negative

Using Kernels in ML algorithms

Using Kernels

- Kernels can turn many linear models into nonlinear models
- Recall that $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high-dim feature space $\mathcal{F}$
- Important: Any ML model/algo in which, during training and test, inputs only appear as dot product (pairwise similarity) can be "kernelized
- Just replace each term of the form $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ by $\boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{\phi}(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = K_{ij}$
- Most ML models/algos can be kernelized
- Will look at an example: Kernelized SVM
  - Perhaps the most popular/natural example of kernelization

A Side-note: Kernelizing a Euclidean Distance

- Not just dot products but Euclean distance can be kernelized too
- Many algorithms, e.g., LwP, KNN, etc. use Euclidean distances, e.g.,

$$d(a,b) = \|a-b\|^2 \quad = \|a\|^2 + \|b\|^2 - 2a^\top b \; = a^\top a + b^\top b - 2a^\top b$$

- This can be kernelized as well by replacing the above norms and inner products by their kernelized versions, assuming a kernel k with feature map ϕ

$$\begin{aligned} d(\phi(a),\phi(b)) &= \|\phi(a)-\phi(b)\|^2 \\ &= \phi(a)^\top\phi(a) + \phi(b)^\top\phi(b) - 2\phi(a)^\top\phi(b) \\ &= k(a,a) + k(b,b) - 2k(a,b) \end{aligned}$$

Nonlinear SVM using Kernels

Kernelized SVM Training

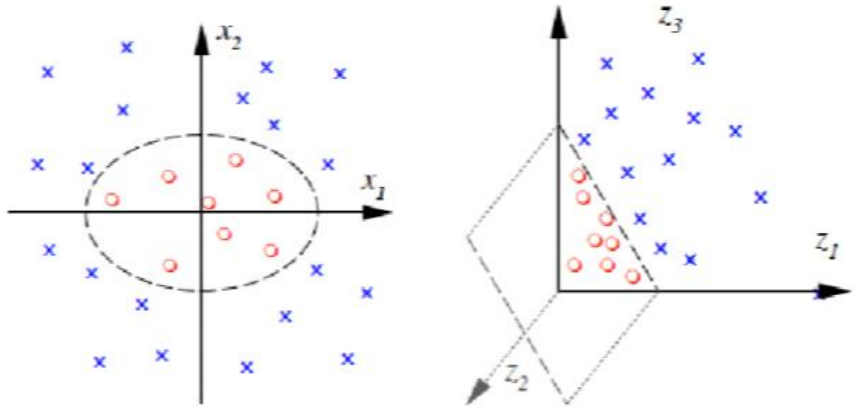
- Recall the soft-margin linear SVM objective (with no bias term)

$$\underset{\mathbf{0}\leq\boldsymbol{\alpha}\leq\boldsymbol{C}}{\operatorname{argmax}} \ \boldsymbol{\alpha}^{\top}\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^{\top}\mathbf{G}\boldsymbol{\alpha}$$

Inputs only appear as dot products ☺

$G_{ij} = y_i y_j \, \boldsymbol{x}_i^{\top}\boldsymbol{x}_j$

- To kernelize, we can simply replace $G_{ij} = y_i y_j \, \boldsymbol{x}_i^{\top}\boldsymbol{x}_j$ by $y_i y_j K_{ij}$
  - .. where $K_{ij} = k(x_i, x_j) = \phi(\boldsymbol{x}_i)^{\top}\phi(\boldsymbol{x}_j)$ for a suitable kernel function k
- The problem can now be solved just like the linear SVM case
- The new SVM learns a linear separator in kernel-induced feature space $\mathcal{F}$
  - This corresponds to a non-linear separator in the original feature space $\mathcal{X}$



Kernelized SVM Prediction

- SVM weight vector for the kernelized case will be $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{\phi}(\boldsymbol{x}_n)$
- Imp: We can't store $\boldsymbol{w}$ unless the feature mapping $\boldsymbol{\phi}(\boldsymbol{x}_n)$ is finite dimensional
  - In practice, we store the α_n's and the training data for test time (just like KNN)
  - In fact, need to store only training examples for which α_n is nonzero (i.e., the support vectors)
- Prediction for a new test input $\boldsymbol{x}_*$ (assuming hyperplane's bias $b = 0$) will be

$$y_* = \text{sign}(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_*)) = \text{sign}\left(\sum_{n=1}^{N} \alpha_n y_n \boldsymbol{\phi}(\boldsymbol{x}_n)^\top \boldsymbol{\phi}(\boldsymbol{x}_*)\right) = \text{sign}\left(\sum_{n=1}^{N} \alpha_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}_*)\right)$$

- Note that the prediction cost also scales linearly with N (actually in the number of support vectors, i.e., training inputs for which α_n is nonzero)
- Also note that, for unkernelized (i.e., linear) SVM, $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$ can be computed and stored as a $D \times 1$ vector and we can compute $\boldsymbol{w}^\top \boldsymbol{x}_*$ in $O(D)$ time

Kernel extensions of other ML models

Kernel extensions of other ML models

- Most of the models what have studied can be kernelized
  - Kernel based linear/ridge regression
  - Kernel based LwP
  - Kernel based nearest neighbors
  - Kernel logistic regression
  - Kernel Perceptron

But the extra price has to be paid in terms of storage cost and slower predictions

Kernel extension makes these approaches more powerful (nonlinear patterns can be learned)

- Some of these extensions are simple to obtain, some not so (but possible)

- Imp: In these models, just like kernel SVM, the model parameters (e.g., the weight vector) can't be stored as a finite-dim vector (unless ϕ is finite dim)
  - Thus the training inputs need to be stored at test time as well

- Also, just like kernel SVM, all of these will in general be slower at test time

Speeding-up Kernel Methods

Speeding-up Kernel Methods

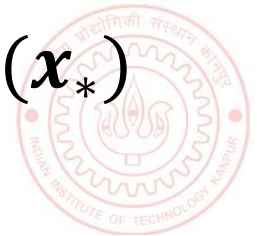
- Kernels assume that

$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m)$$

- Suppose for this kernel, we can get an L-dim feature vector $\psi(\boldsymbol{x})$ such that

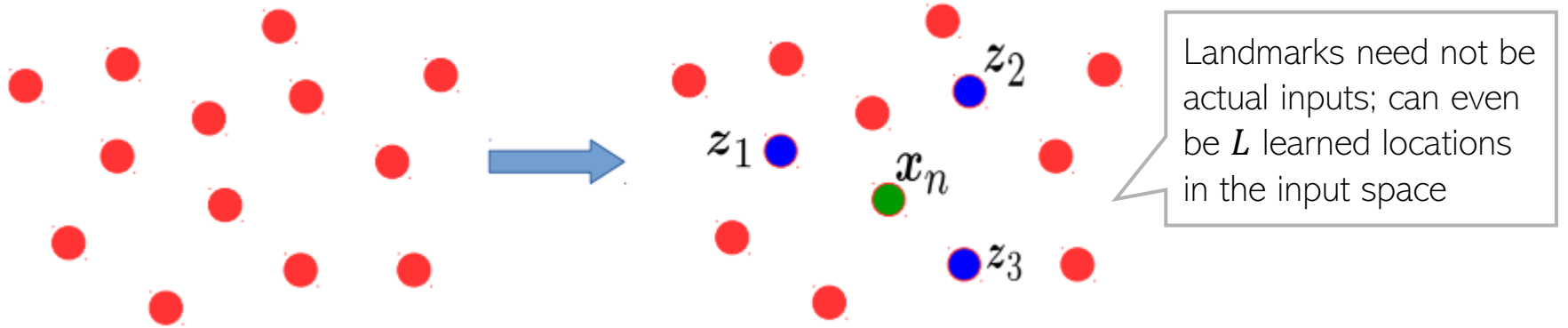
$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) \approx \psi(\boldsymbol{x}_n)^\top \psi(\boldsymbol{x}_m)$$

- Using features $\psi(\boldsymbol{x})$, we can learn a linear model with weights $\boldsymbol{w} \in \mathbb{R}^L$
- This model will be a good approximation to the kernelized model
- Training will be faster because no need to store and work with kernel matrices
- Prediction at test time will also be faster - we just need to compute $\boldsymbol{w}^\top \psi(\boldsymbol{x}_*)$
- Many ways to get such features $\psi(\boldsymbol{x})$ for standard kernels



Extracting Features using Kernels: Landmarks

- Suppose we choose a small set of L "landmark" inputs $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_L$ in the training data



$$\psi(\boldsymbol{x}_n) = [k(\boldsymbol{z}_1, \boldsymbol{x}_n), k(\boldsymbol{z}_2, \boldsymbol{x}_n), k(\boldsymbol{z}_3, \boldsymbol{x}_n)] \in \mathbb{R}^3$$

- For each input $\boldsymbol{x}_n$, using a kernel k, define an L-dimensional feature vector as follows

$$\psi(\boldsymbol{x}_n) = [k(\boldsymbol{z}_1, \boldsymbol{x}_n), k(\boldsymbol{z}_2, \boldsymbol{x}_n), \ldots, k(\boldsymbol{z}_L, \boldsymbol{x}_n)] \in \mathbb{R}^L$$

- Can now apply a linear model on ψ representation (L-dimensional now) of the inputs
- This will be fast both at training as well as test time if L is small
- No need to kernelize the linear model while still reaping the benefits of kernels ☺

Extracting Feat. using Kernels: Random Features

- Many kernel functions* can be written as

$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m) = \mathbb{E}_{\boldsymbol{w}\sim p(\boldsymbol{w})}[t_{\boldsymbol{w}}(\boldsymbol{x}_n) t_{\boldsymbol{w}}(\boldsymbol{x}_m)]$$

.. where $\boldsymbol{t_w(.)}$ is a function with params $\boldsymbol{w} \in \mathbb{R}^D$ with $\boldsymbol{w}$ drawn from some distr. $\boldsymbol{p(w)}$

- Example: For the RBF kernel, $\boldsymbol{t_w(.)}$ is cosine func. and $\boldsymbol{p(w)}$ is zero mean Gaussian

$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) = \mathbb{E}_{\boldsymbol{w}\sim p(\boldsymbol{w})}[\cos(\boldsymbol{w}^\top \boldsymbol{x}_n) \cos(\boldsymbol{w}^\top \boldsymbol{x}_m)]$$

- Given $\boldsymbol{w_1, w_2, \ldots, w_L}$ from $\boldsymbol{p(w)}$, using Monte-Carlo approx. of above expectation

$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) \approx \frac{1}{L}\sum_{\ell=1}^{L} \cos(\boldsymbol{w}_\ell^\top \boldsymbol{x}_n) \cos(\boldsymbol{w}_\ell^\top \boldsymbol{x}_m) = \psi(\boldsymbol{x}_n)^\top \psi(\boldsymbol{x}_m)$$

.. where $\psi(\boldsymbol{x}_n) = \frac{1}{\sqrt{L}}[\cos(\boldsymbol{w}_1^\top \boldsymbol{x}_n), \ldots, \cos(\boldsymbol{w}_L^\top \boldsymbol{x}_n)]$ is an L-dim vector

- Can apply a linear model on this L-dim rep. of the inputs (no need to kernelize)

*Random Features for Large-Scale Kernel Machines (Ben and Retch, NIPS 2007. Note: This paper actually won the test-of-time award at NIPS 2017)

Learning with Kernels: Some Aspects

- Storage/computational efficiency can be a bottleneck when using kernels
- During training, need to compute and store the $N \times N$ kernel matrix $\mathbf{K}$ in memory
- Need to store training data (or at least support vectors in case of SVMs) at test time
- Test time can be slow: $O(N)$ cost to compute a quantity like $\sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x}_*)$
- Approaches like landmark and random features can be used to speed up
- Choice of the right kernel is also very important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- Quite a bit of research on learning the right kernel from data
  - Learning a combination of multiple kernels (Multiple Kernel Learning)
  - Bayesian kernel methods (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
  - Deep Learning can also be seen as learning the kernel from data (more on this later)

Also, a lot of recent work on connections between kernel methods and deep learning