Kernelizing ML algorithms

CS771: Introduction to Machine Learning

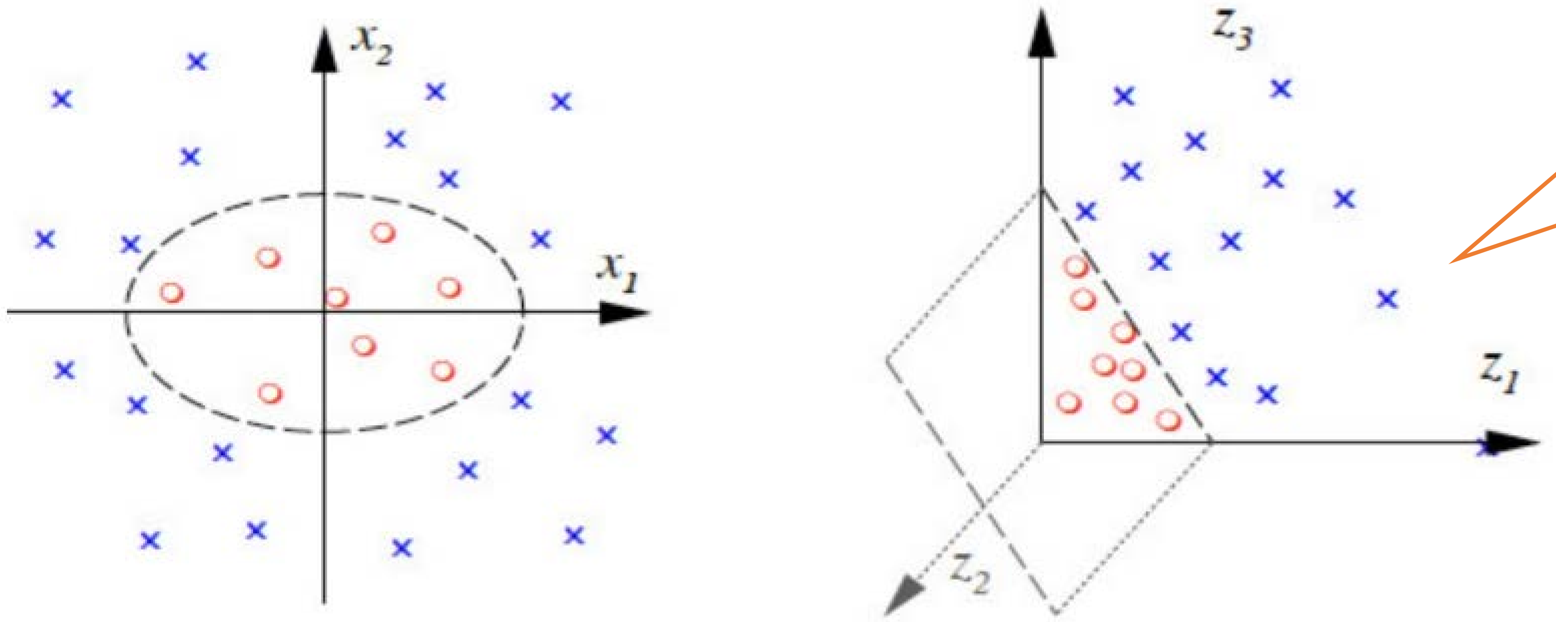
Nisheeth

Recap: Kernel functions

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

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

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



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!

Using Kernels

- Kernels can turn many linear models into nonlinear models
- Recall that $\boldsymbol{k(x,z)}$ 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 can be "kernelized
- Just replace each term of the form $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ by $\phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = K_{ij}$
- Most ML models/algos can be easily kernelized, e.g.,
  - Distance based methods, Perceptron, SVM, linear regression, etc.
  - Many of the unsupervised learning algorithms too can be kernelized (e.g., K-means clustering, Principal Component Analysis, etc. - will see later)
  - Let's look at two examples: Kernelized SVM and Kernelized Ridge Regression

Nonlinear SVM using Kernels

Solving Soft-Margin SVM

- Recall the soft-margin SVM optimization problem

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad f(\mathbf{w},b,\boldsymbol{\xi}) = \frac{||\mathbf{w}||^2}{2} + C\sum_{n=1}^{N}\xi_n$$

$$\text{subject to} \quad 1 \le y_n(\mathbf{w}^T\mathbf{x}_n + b) + \xi_n, \quad -\xi_n \le 0 \qquad n = 1,\ldots,N$$

- Here $\boldsymbol{\xi} = [\xi_1, \xi_2, \ldots, \xi_N]$ is the vector of slack variables
- Introduce Lagrange multipliers α_n, β_n for each constraint and solve Lagrangian

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \ \max_{\alpha\ge 0,\beta\ge 0} \ \mathcal{L}(\mathbf{w},b,\xi,\alpha,\beta) = \frac{||\mathbf{w}||^2}{2} + +C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n\{1 - y_n(\mathbf{w}^T\mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n$$

- The terms in red color above were not present in the hard-margin SVM
- Two set of dual variables $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_N]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \ldots, \beta_N]$
- Will eliminate the primal var $\mathbf{w}$, b, $\boldsymbol{\xi}$ to get dual problem containing the dual variables

Solving Soft-Margin SVM

Note: if we ignore the bias term b then we don't need to handle the constraint $\sum_{n=1}^{N} \alpha_n y_n = 0$ (problem becomes a bit more easy to solve)

Otherwise, the α_n's are coupled and some opt. techniques such as co-ordinate aspect can't easily applied

- The Lagrangian problem to solve

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha}\geq 0,\boldsymbol{\beta}\geq 0} \mathcal{L}(\boldsymbol{w}, b, \xi, \alpha, \beta) = \frac{||\boldsymbol{w}||^2}{2} + +C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n\{1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - \xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n$$

- Take (partial) derivatives of $\mathcal{L}$ w.r.t. $\boldsymbol{w}, b$, and ξ_n and setting to zero gives

Weighted sum of training inputs

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \Rightarrow \boxed{\boldsymbol{w} = \sum_{n=1}^{N}\alpha_n y_n \boldsymbol{x}_n}, \quad \frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^{N}\alpha_n y_n = 0, \quad \frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Rightarrow C - \alpha_n - \beta_n = 0$$

- Using $C - \alpha_n - \beta_n = 0$ and $\beta_n \geq 0$, we have $\alpha_n \leq C$ (for hard-margin, $\alpha_n \geq 0$)
- Substituting these in the Lagrangian $\mathcal{L}$ gives the Dual problem

The dual variables β don't appear in the dual problem!

Given $\boldsymbol{\alpha}$, $\boldsymbol{w}$ and b can be found just like the hard-margin SVM case

$$\max_{\boldsymbol{\alpha}\leq C,\boldsymbol{\beta}\geq 0} \mathcal{L}_D(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{m,n=1}^{N}\alpha_m\alpha_n y_m y_n(\boldsymbol{x}_m^T\boldsymbol{x}_n) \quad \text{s.t.} \quad \sum_{n=1}^{N}\alpha_n y_n = 0$$

Maximizing a concave function (or minimizing a convex function) s.t. $\boldsymbol{\alpha} \leq \boldsymbol{C}$ and $\sum_{n=1}^{N}\alpha_n y_n = 0$. Many methods to solve it.

$$\max_{\boldsymbol{\alpha}\leq C} \mathcal{L}_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top\mathbf{G}\boldsymbol{\alpha}$$

In the solution, $\boldsymbol{\alpha}$ will still be sparse just like the hard-margin SVM case. Nonzero α_n correspond to the support vectors

(Note: For various SVM solvers, can see "Support Vector Machine Solvers" by Bottou and Lin)

Kernelized SVM Training

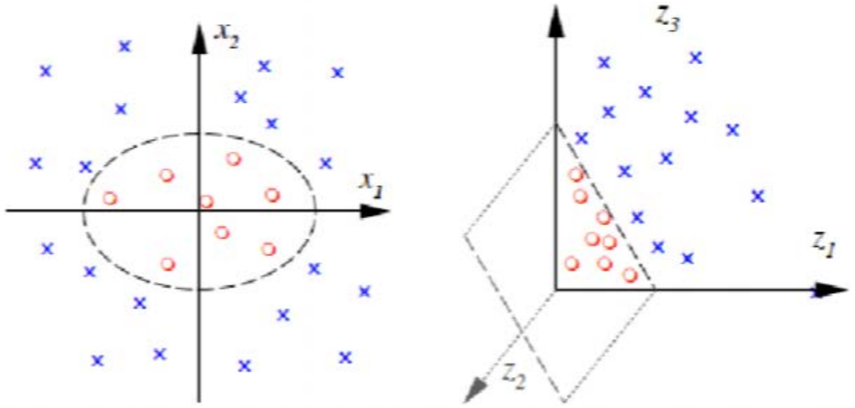
- Recall the soft-margin linear SVM objective (with no bias term)

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

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

- 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 $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \phi(\boldsymbol{x}_n)$
- Note: We can't store $\boldsymbol{w}$ unless the feature mapping $\phi(\boldsymbol{x}_n)$ 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

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

- Note that the prediction cost also scales linearly with N (unlike a linear model where we only need to compute $\boldsymbol{w}^\top \boldsymbol{x}_*$, whose cost only depends on D, not N)
- 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

Nonlinear Ridge Regression using Kernels

Kernelized Ridge Regression

- Recall the ridge regression problem: $\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$
- The solution to this problem was

Inputs don't appear to be as inner product. No hope of kernelization?

They do; with a bit of algebra ☺

$$\boldsymbol{w} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D\right)\left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Can use matrix inversion lemma $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- Using the lemma, can rewrite $\boldsymbol{w}$ as

$N \times 1$ vector of dual variables

Note: Not sparse unlike SVM

$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n \quad \text{where} \quad \boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$$

- Kernelized weight vector will be $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n)$

Prediction cost is also linear in N (like KNN)

- Prediction for a test input $\boldsymbol{x}_*$ will be $\boldsymbol{w}^\top \phi(\boldsymbol{x}_*) = \left(\sum_{n=1}^{N} \alpha_n \phi(\boldsymbol{x}_n)\right)^\top \phi(\boldsymbol{x}_*) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x}_*)$

Speeding-up Kernel Methods

Speeding-up Kernel Methods

- Kernel methods, unlike linear models are slow at training and test time
- Would be nice if we could easily compute mapping $\boldsymbol{\phi}(\boldsymbol{x})$ associated with kernel $\boldsymbol{k}$
- Then we could apply linear models directly on $\boldsymbol{\phi}(\boldsymbol{x})$ without having to kernelize
- But this is in general not possible since $\boldsymbol{\phi}(\boldsymbol{x})$ is very high/infinite dimensional
- An alternative: Get a good set of low-dim features $\psi(x) \in \mathbb{R}^L$ using the kernel $\boldsymbol{k}$
- If $\psi(x)$ is a good approximation to $\boldsymbol{\phi}(\boldsymbol{x})$ then we can use $\boldsymbol{\psi}(\boldsymbol{x})$ in a linear model

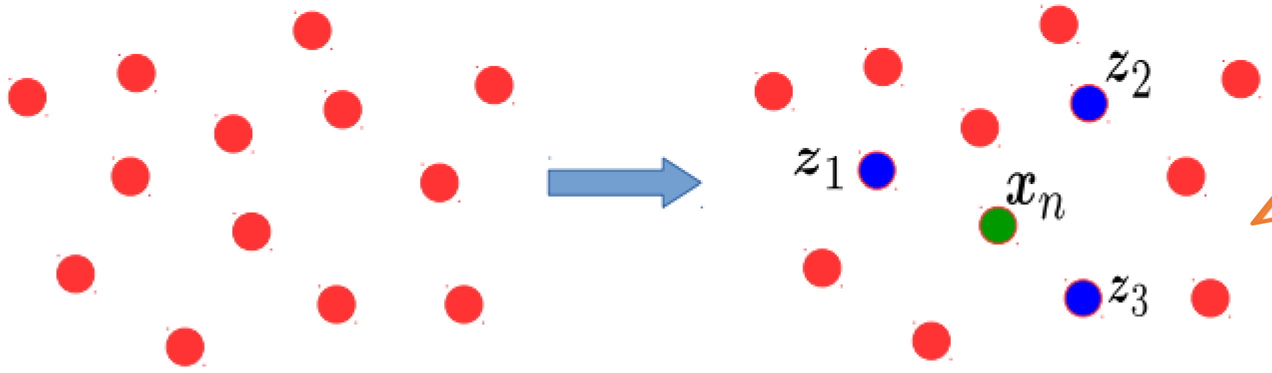
$$\textbf{Goodness Criterion:} \quad \psi(\boldsymbol{x}_i)^\top \psi(\boldsymbol{x}_j) \approx \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

… which also means $\psi(\boldsymbol{x}_i)^\top \psi(\boldsymbol{x}_j) \approx k(\boldsymbol{x}_i, \boldsymbol{x}_j)$

- Will look at two popular approaches: Landmarks and Random Features

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



Landmarks need not be actual inputs; can even be L learned locations in the input space

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

- 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 Features 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 $t_{\boldsymbol{w}}(.)$ is a function with params $\boldsymbol{w} \in \mathbb{R}^L$ with $\boldsymbol{w}$ drawn from some distr. $p(\boldsymbol{w})$

- Example: For the RBF kernel, $t_{\boldsymbol{w}}(.)$ is cosine func. and $p(\boldsymbol{w})$ 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 $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_L$ from $p(\boldsymbol{w})$, 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 (Recht & Rahimi, NIPS 2007. Remember the ML alchemy talk? That was these guys)

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 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)