Turning Linear Models into Nonlinear Models using Kernel Methods (Contd)

CS771: Introduction to Machine Learning

Piyush Rai

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 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 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

An Aside: Kernelizing a Euclidean Distance

- 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

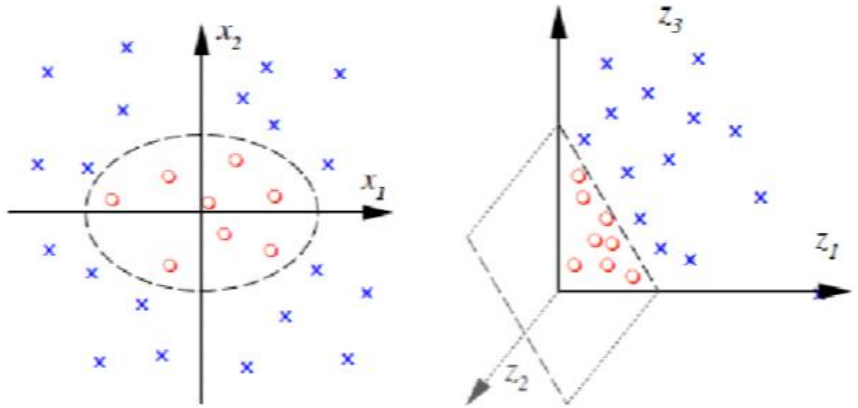
- Recall the soft-margin linear SVM objective (with no bias term)

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

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 $\mathcal{X}$

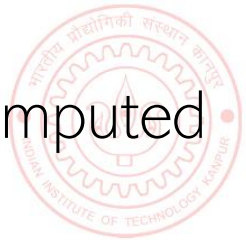


Kernelized SVM Prediction

- SVM weight vector for the kernelized case will be $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{\phi}(\boldsymbol{x}_n)$
- Note: 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 (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

They do; with a bit of algebra ☺

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

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

- 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

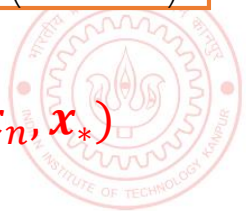
$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n \quad \text{where} \quad \boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}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

$N \times 1$ vector of dual variables

Note: Not sparse unlike SVM

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

Prediction cost is also linear in N (like KNN)



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 $\boldsymbol{\phi(x)}$ associated with kernel $\boldsymbol{k}$
- Then we could apply linear models directly on $\boldsymbol{\phi(x)}$ without having to kernelize
- But this is in general not possible since $\boldsymbol{\phi(x)}$ is very high/infinite dimensional
- An alternative: Get a good set of low-dim features $\boldsymbol{\psi(x) \in \mathbb{R}^L}$ using the kernel $\boldsymbol{k}$
- If $\boldsymbol{\psi(x)}$ is a good approximation to $\boldsymbol{\phi(x)}$ then we can use $\boldsymbol{\psi(x)}$ in a linear model

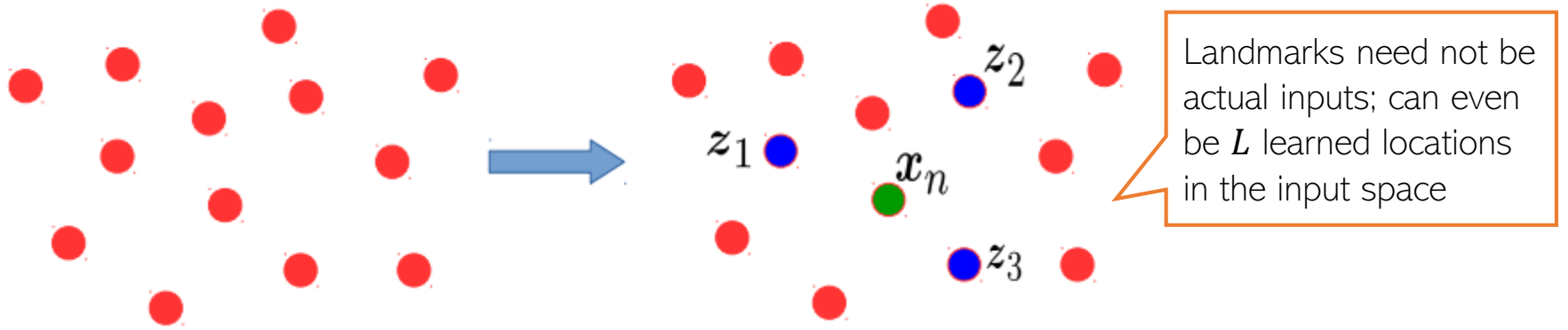
$$\textbf{Goodness Criterion:} \quad \boldsymbol{\psi(x_i)^\top \psi(x_j) \approx \phi(x_i)^\top \phi(x_j)}$$

. . . which also means $\boldsymbol{\psi(x_i)^\top \psi(x_j) \approx k(x_i, x_j)}$

- 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 $\boldsymbol{z_1, z_2, \ldots, z_L}$ 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 $\boldsymbol{w} \in \mathbb{R}^L$ 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

Coming up next

- Unsupervised learning