Learning with Kernels II

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

Gymkhana holiday on October 18 – no class this Friday

Next class on Monday, Oct 21 2019

Recap of Last Lecture

Using non-linear maps to project data onto high dimensional spaces can improve separability of data and allow linear models to do well

We prefer linear models since they are simple and well-understood, we have nice algorithms to learn linear models

Non-linearity of the map is essential but not sufficient – a bad non-linear map need not improve performance of linear models and may worsen performance

E.g. consider a map which maps every vector $\mathbf{u} \in \mathbb{R}^d$ onto the all-ones vector i.e. $\mathbf{1} = [1,1,\ldots,1] \in \mathbb{R}^{d^2}$. Sure, this map is non-linear but a fantastically useless one

Kernels: notions of similarity

***Mercer kernels**: map data to $\mathcal{H}$ and return the dot product as similarity value*

Examples of kernels over vectors: linear, Gaussian, polynomial, Laplacian

Examples of domain-specific kernels: n-gram, substring (NLP), intersection, pyramid (vision), convolutional, random walk (graphs/trees)

Recap of Last Lecture

Using non-linear maps to project data onto high dimensional spaces can improve

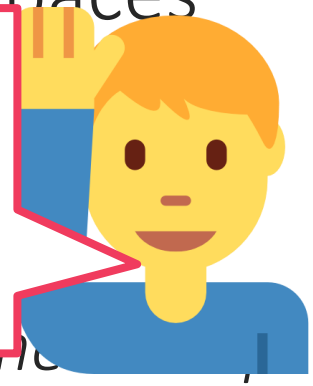
We prefer
nice algori

Non-linear

need not improve performance of linear models and may worsen performance

E.g. consider a map which maps every vector $\mathbf{u} \in \mathbb{R}^d$ onto the all-ones vector i.e. $\mathbf{1} = [1,1,\ldots,1] \in \mathbb{R}^{d^2}$. Sure, this map is non-linear but a fantastically useless one

Often, the term RKHS (*Reproducing Kernel Hilbert Space*) is used to describe the space $\mathcal{H}$ onto which the map ϕ maps objects. The name has significance (there is something called a *reproducing kernel*) but those details are beyond the scope of CS771



Kernels: notions of similarity

***Mercer kernels**: map data to $\mathcal{H}$ and return the dot product as similarity value*

Examples of kernels over vectors: linear, Gaussian, polynomial, Laplacian

Examples of domain-specific kernels: n-gram, substring (NLP), intersection, pyramid (vision), convolutional, random walk (graphs/trees)

The Kernel Trick

An algorithmically effective way of using linear models on non-lin maps

Every kernel K is associated with a map ϕ such that $K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$

The map ϕ is usually (very) non-linear and (very) high dimensional i.e. good candidate for our overall goal of using linear models over non-linear maps

Peculiar property of several ML algos (last class – LwP, kNN)

So far we have seen ML algos work with feature vectors of train/test points

However, many of them work even if feature vectors are not provided directly but instead pairwise dot/inner products b/w feature vectors is provided!

For training, pairwise dot products between train points needed

For testing, dot products between the test point and all train points needed

Thus, we can say we want to work with high-dim feature vectors $\phi(\mathbf{x})$ and when the ML algo asks us for dot products, give it $K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ ☺

Would get same result as working directly with $\phi(\mathbf{x})$ but without having to compute ϕ

The Kernel Trick

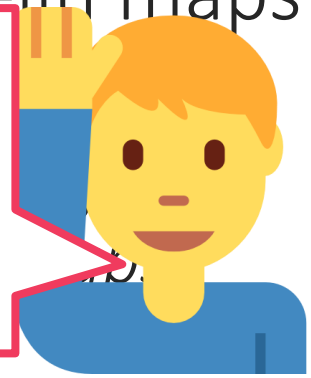
An algorithmically effective way of using linear models on non-lin maps

Every kernel K is

The map ϕ is usu

candidate for ou

This peculiar property is often called *kernelizability*. An ML algo is said to be kernelizable if we can show that it works identically if, instead of feature vectors, we supply pairwise train-train and test-train dot products of feature vectors



Peculiar property of several ML algos (last class – LwP, kNN)

So far we have seen ML algos work with feature vectors of train/test points

However, many of them work even if feature vectors are not provided directly but instead pairwise dot/inner products b/w feature vectors is provided!

For training, pairwise dot products between train points needed

For testing, dot products between the test point and all train points needed

Thus, we can say we want to work with high-dim feature vectors $\phi(\mathbf{x})$ and when the ML algo asks us for dot products, give it $K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$ ☺

Would get same result as working directly with $\phi(\mathbf{x})$ but without having to compute ϕ

Kernel kNN

Step 1: choose a kernel K with map ϕ

Gaussian kernel is popular but domain specific kernels may do better

Step 2: training

Receive training points $(\mathbf{x}^i, y^i), i \in [n], \mathbf{x}^i \in \mathbb{R}^d$

Store them

Step 3: prediction

Receive a test point $\mathbf{x}^t \in \mathbb{R}^d$

Find the nearest neighbour using $i^t = \arg\min_{i \in [n]} \left\| \phi(\mathbf{x}^t) - \phi(\mathbf{x}^i) \right\|_{\mathcal{H}}^2$

Too expensive so instead use $i^t = \arg\min_{i \in [n]} K(\mathbf{x}^i, \mathbf{x}^i) - 2K(\mathbf{x}^i, \mathbf{x}^t)$

Predict y^{i^t}

Kernel LwP

Step 1: choose a kernel K with map ϕ

Gaussian kernel is popular but domain specific kernels may do better

Step 2: training

Receive training points $(\mathbf{x}^i, y^i), i \in [n], \mathbf{x}^i \in \mathbb{R}^d, y^i \in \{-1,1\}$

Store them (cannot compute prototypes explicitly as ϕ *may be* ∞ *dim ☹)*

Step 3: prediction

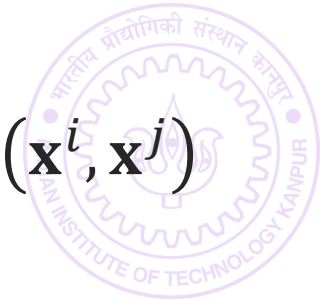
Receive a test point $\mathbf{x}^t \in \mathbb{R}^d$

Find distance to +ve prototype $\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2$ *where* $\tilde{\mathbf{p}}^+ = \frac{1}{n_+} \cdot \sum_{y^i=1} \phi(\mathbf{x}^i)$

Too expensive so instead use the formula

$$\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2 = K(\mathbf{x}^t, \mathbf{x}^t) - \frac{2}{n_+} \cdot \sum_{y^i=1} K(\mathbf{x}^t, \mathbf{x}^i) + \frac{1}{n_+^2} \cdot \sum_{y^i=1} \sum_{y^j=1} K(\mathbf{x}^i, \mathbf{x}^j)$$

Predict $\text{sign}(\|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^-\|_2^2 - \|\phi(\mathbf{x}^t) - \tilde{\mathbf{p}}^+\|_2^2)$



Kernel SVM

PRIMAL FORMULATION

$$\min_{\mathbf{w},\xi_i} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

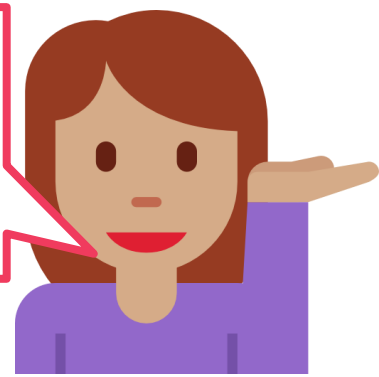
s.t. $y^i \cdot \mathbf{w}^\top \mathbf{x}^i \geq 1 - \xi_i$ and $\xi_i \geq 0$

DUAL FORMULATION

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^i y^j \langle \mathbf{x}^i, \mathbf{x}^j \rangle$$

s.t. $\alpha_i \in [0, C]$

Lets see what happens if we execute the SVM after applying a (nonlinear) feature map ϕ



Kernel SVM

PRIMAL FORMULATION

$$\min_{\mathbf{w},\xi_i} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

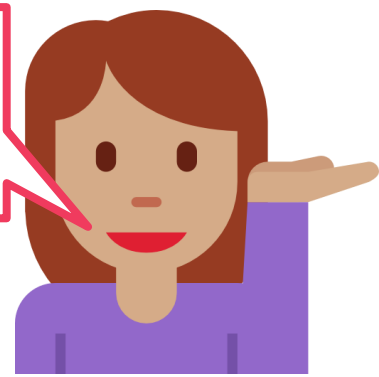
s.t. $y^i \cdot \mathbf{w}^\top \phi(\mathbf{x}^i) \geq 1 - \xi_i$ and $\xi_i \geq 0$

DUAL FORMULATION

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^i y^j \quad \langle \phi(\mathbf{x}^i), \phi(\mathbf{x}^j) \rangle$$

s.t. $\alpha_i \in [0, C]$

Note that if $\phi: \mathbb{R}^d \rightarrow \mathcal{H}$ then the model itself is $\mathbf{w} \in \mathcal{H}$



Kernel SVM

PRIMAL FORMULATION

$$\min_{\mathbf{w},\xi_i} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

s.t. $y^i \cdot \mathbf{w}^\top \phi(\mathbf{x}^i) \geq 1 - \xi_i$ and $\xi_i \geq 0$

Solving the primal is infeasible

If $\dim(\mathcal{H}) = \infty$ a single SGD step would take infinitely long ☹

DUAL FORMULATION

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^i y^j K(\mathbf{x}^i, \mathbf{x}^j)$$

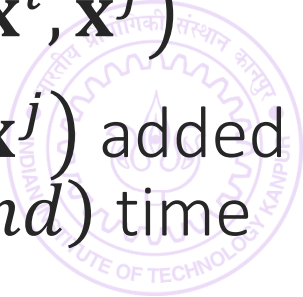
s.t. $\alpha_i \in [0, C]$

Computing $K(\mathbf{x}^i, \mathbf{x}^j)$ usually $\mathcal{O}(d)$ even if $\dim(\mathcal{H}) = \infty$ e.g. Gaussian kernel

Can still solve this problem using SDCA

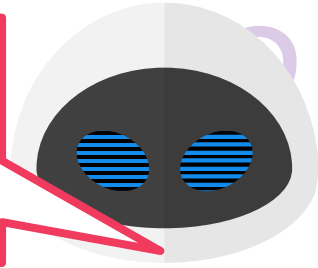
Each step of SDCA still takes $\mathcal{O}(n)$ time apart from time to compute $K(\mathbf{x}^i, \mathbf{x}^j)$

If time taken to compute $K(\mathbf{x}^i, \mathbf{x}^j)$ added then each SDCA takes about $\mathcal{O}(nd)$ time



Ker

Finding/storing the model explicitly is not feasible even if we solve the dual problem perfectly since $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y^i \phi(\mathbf{x}^i) \in \mathcal{H}$ and $\dim(\mathcal{H}) \gg 1$



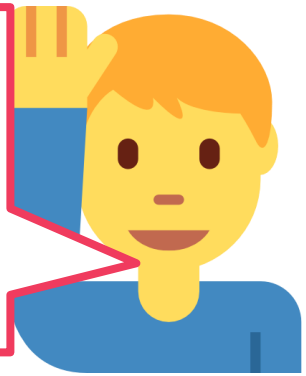
PRIMAL FO

$$\min_{\mathbf{w},\xi_i} \frac{1}{2} \|\mathbf{w}\|$$

s.t. $y^i \cdot \mathbf{w}^\top$

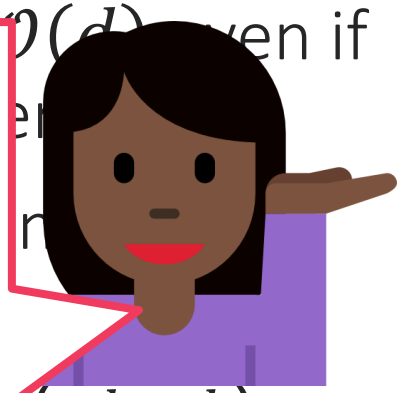
So instead we can store all the α_i values (only n of them). At test time, given a test point $\mathbf{x}^t$, we can predict using

$$\mathbf{w}^\top \phi(\mathbf{x}^t) = \sum_{i=1}^{n} \alpha_i y^i \langle \phi(\mathbf{x}^i), \phi(\mathbf{x}^t) \rangle = \sum_{i=1}^{n} \alpha_i y^i K(\mathbf{x}^i, \mathbf{x}^t)$$



$\mathcal{O}(d)$ even if

Note that if the test data point is very similar to one of the training points i.e. $K(\mathbf{x}^i, \mathbf{x}^t)$ is large, then that label i.e. y^i influences the prediction much more. If we think this way, kernel SVM almost looks like a "soft" form of kNN. If there are $\tilde{n}$ support vectors, then prediction requires $\tilde{n}$ kernel computations i.e. roughly $\mathcal{O}(\tilde{n}d)$ time since each kernel computation takes roughly $\mathcal{O}(d)$ time



Training is more expensive, model size is larger, prediction time is more for kernel SVM than was for linear SVM – very typical of non-linear models