Kernel Learning IV

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

Assignment 2 deadline extended to 02 Nov (Saturday) 9:59PM IST

- *Applies to both PDF submission (Gradescope) and code submission (website)*

No cascading extensions to other portions of the course

- *Assignment 3 will be released early next week as scheduled originally*
  - Cannot delay this as we are nearing end of course as is
- *Quiz 4 will be held on 01 Nov as scheduled originally*
  - Cannot delay this as we have our endsem exam scheduled by DoAA on 18 Nov

Recap of Last Lecture

Saw how several ML algos can be kernelized

Easy for some e.g. LwP, kNN, k-means, not so easy for others e.g. RR, PCA

In all cases, we essentially showed that these algos perform identically if instead of feat vecs they are given dot/inner products b/w these feat vecs

Having realized this, we pretended to use $\phi(\mathbf{x}^i) \in \mathcal{H}$ ($\mathbf{x}^i \in \mathbb{R}^d$ are orig feats) and when asked for inner products, gave $K(\mathbf{x}^i, \mathbf{x}^j) = \langle \phi(\mathbf{x}^i), \phi(\mathbf{x}^j) \rangle_{\mathcal{H}}$

Useful since $K(\mathbf{x}^i, \mathbf{x}^j)$ usually computable in $\mathcal{O}(d)$ time even if $\mathbf{dim}(\phi) = \infty$

Poly kernels $K(\mathbf{x}^i, \mathbf{x}^j) = \left(\langle \mathbf{x}^i, \mathbf{x}^j \rangle + c\right)^p$ called homogeneous if $c = 0$

These only use features of the form $\prod_{k=1}^d x_k^{p_k}$ where $p_k \geq 0$ and $\sum_{k=1}^d p_k = p$

If we have $c > 0$ ($c < 0$ makes the kernel non-Mercer) then the kernels use all features of the form $\prod_{k=1}^d x_k^{p_k}$ where $p_k \geq 0$, $\sum_{k=1}^d p_k = q$ where $0 \leq q \leq p$

Non-homogeneous poly kernels have more expressive feature maps

Accelerated Kernel Learning

- Kernelized versions of algos can be much slower than linear versions
  - *Training typically done in a "dual" form since "primal" forms inaccessible*
    - Updates slower e.g. each step of SDCA takes $\mathcal{O}(n)$ kernel computations i.e. $\mathcal{O}(nd)$ time
  - *Model size typically grows as training dataset size n increases*
    - Kernel SVM: #SV can be as large as n in which case all train data has to be stored
    - Kernel RR/PCA: entire training data n has to be stored in general anyway
  - *Prediction time also goes up – upto $\mathcal{O}(n)$ kernel computations i.e. $\mathcal{O}(nd)$ time*
- Kernel methods are examples of *non-parametric ML algorithms*
  - *Name is misleading: an ML algo is called non-parametric if the #params in its model can grow in an unbounded manner as train set sizes increases*
    - Contrast this with linear SVM/RR/LwP where the model has only $\mathcal{O}(d)$ parameters (d-dim vectors) irrespective of how large (or small) n is – these are called *parametric ML algos*
    - Even non kernel algos can be non-parametric e.g. vanilla (non-kernel) kNN

Accelerated Kernel Learning

Several techniques exist – can be grouped into three main families

Presented using kernel SVM as example – can be applied to kernel RR/PCA too

Post-processing: learn the kernel SVM model (will be a bit costly) but then compress the model to make storage/prediction cheaper

Approximate training: directly learn a kernel SVM model that is cheap to store and predict

Kernel approximation: use a kernel different than the one we originally wanted but choose this new kernel so that it

Provides approximately the same kernel values as the original one

Results in models that are cheap to store and predict

Post Processing

Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

Burges and Scholkopf, ***Improving the Speed and Accuracy of SVMs****, NIPS 1996.*

Cossalter et al. ***Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction****, ICML 2011.*

Post Processing

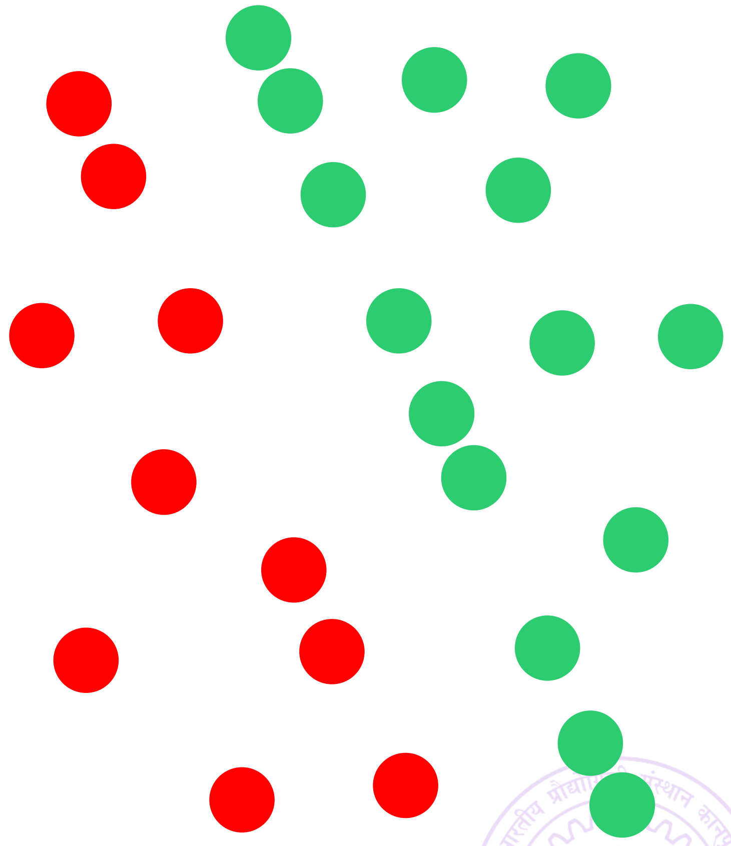
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Post Processing

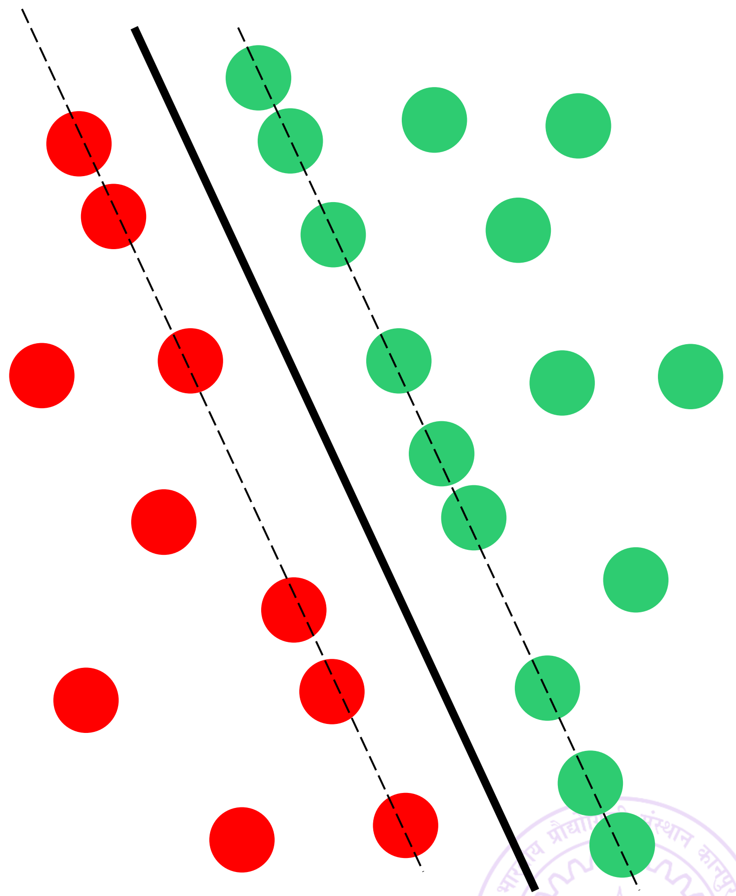
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

Burges and Scholkopf, ***Improving the Speed and Accuracy of SVMs****, NIPS 1996.*

Cossalter et al. ***Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction****, ICML 2011.*



Post Processing

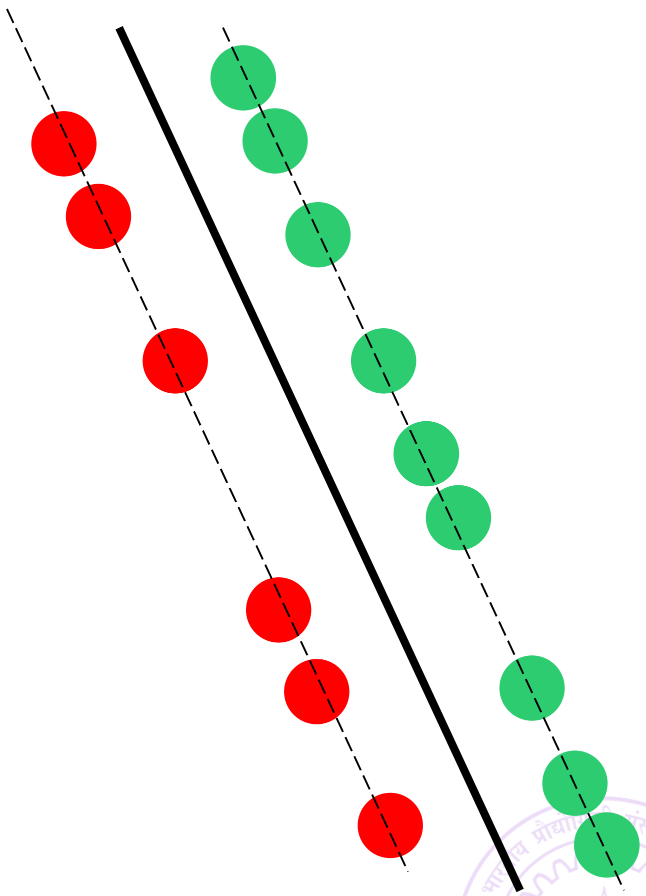
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

Burges and Scholkopf, ***Improving the Speed and Accuracy of SVMs****, NIPS 1996.*

Cossalter et al. ***Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction****, ICML 2011.*



Post Processing

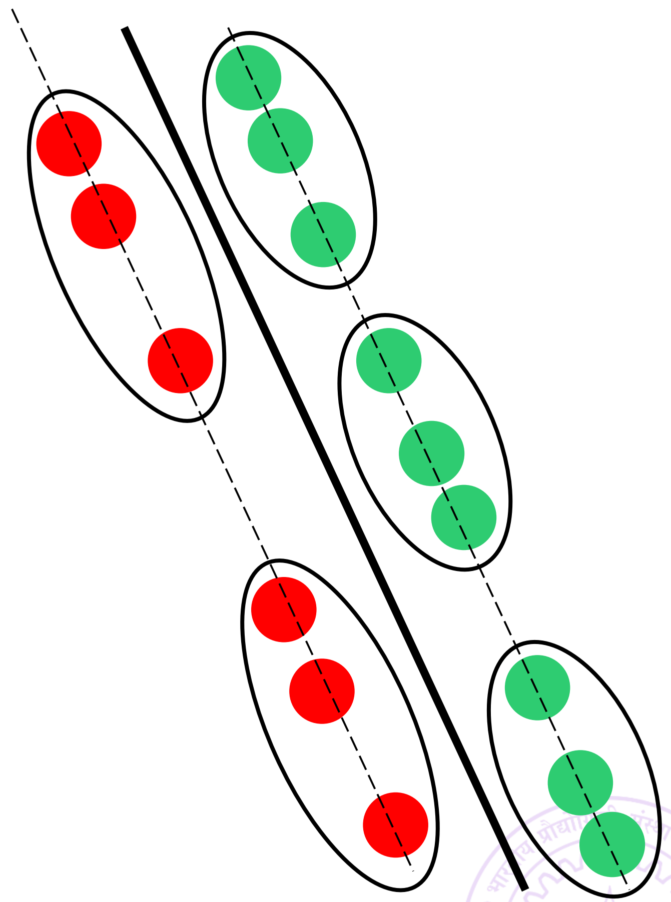
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\left\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\right\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\left\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\right\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Post Processing

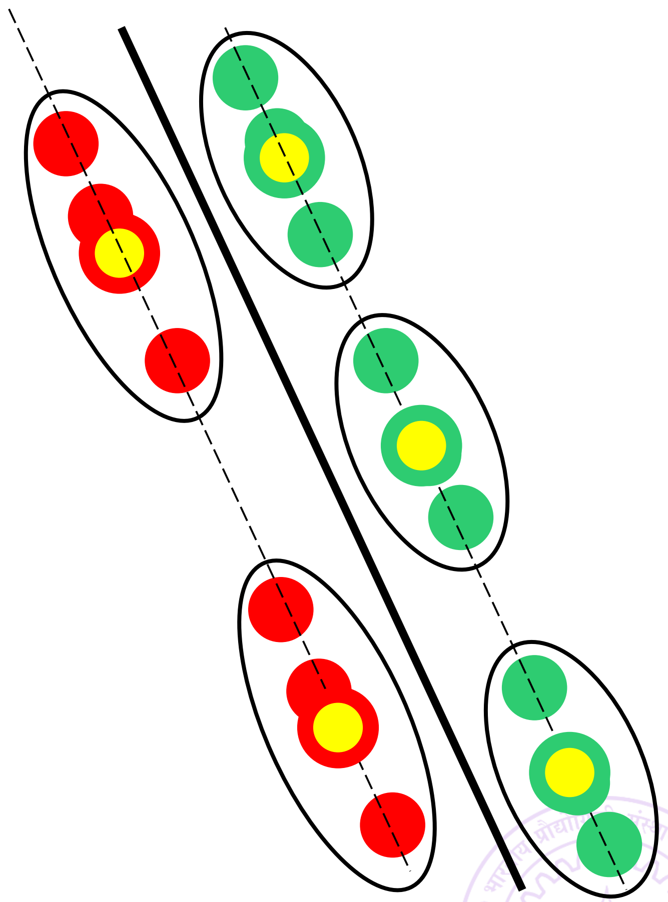
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\left\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\right\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\left\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\right\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Post Processing

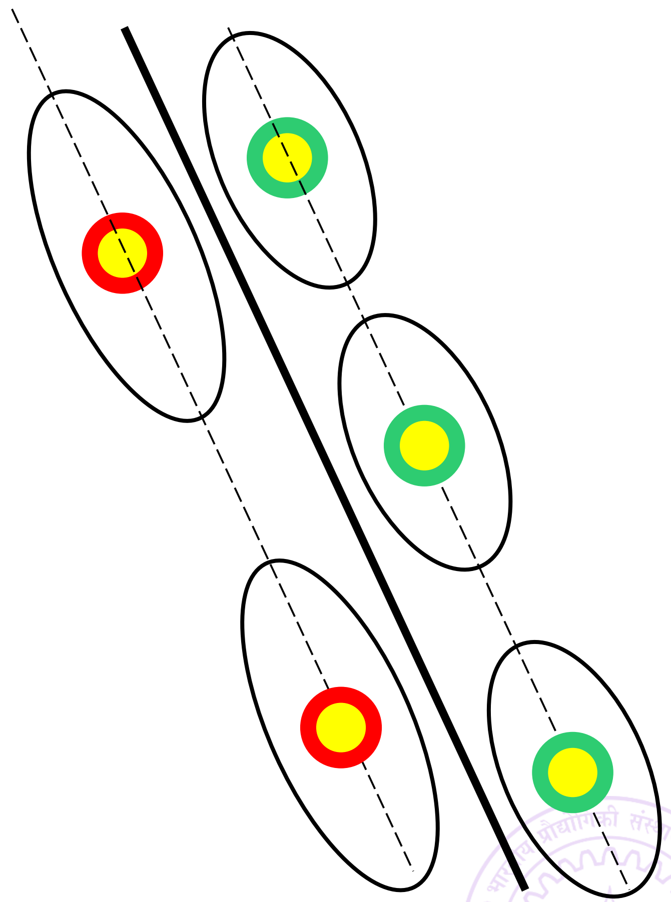
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Post Processing

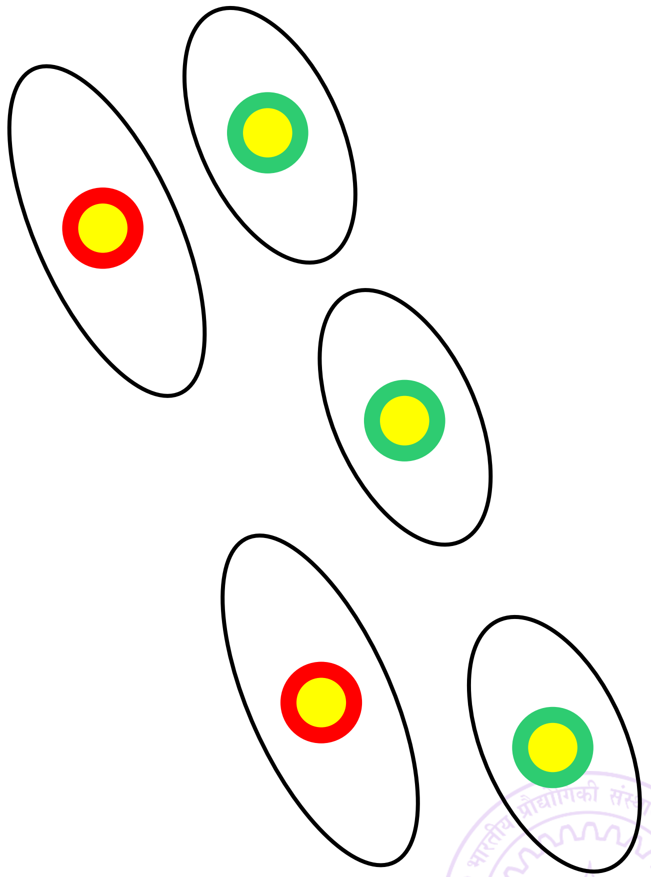
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \dots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \dots, \tilde{\mathbf{x}}_{i_k}\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Post Processing

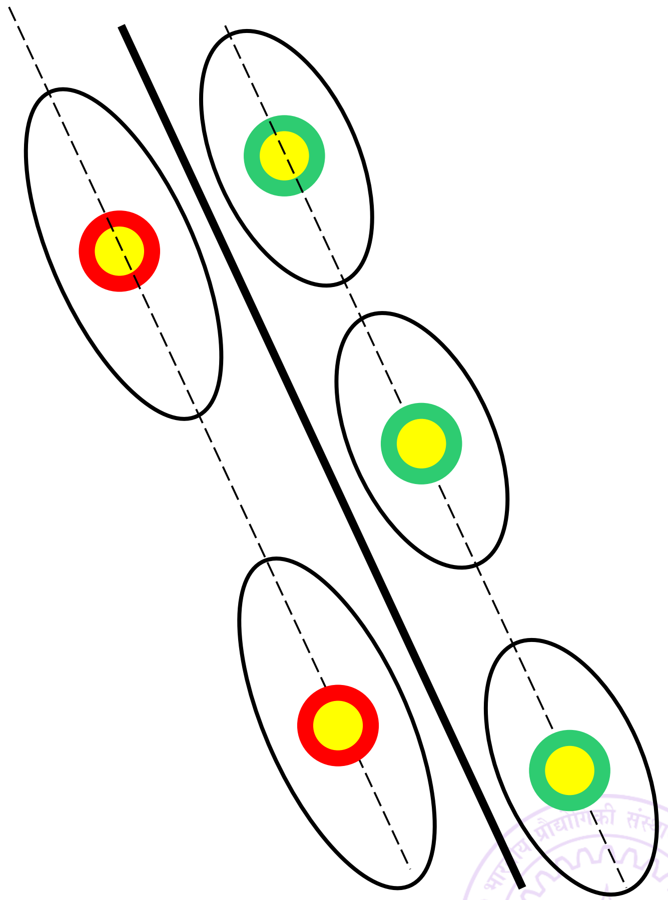
Learn kernel SVM, support vectors $\{\mathbf{x}_{i_j}, \alpha_{i_j}\}$

Find a *reduced set* of $k \ll \tilde{n}$ support vectors $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$ e.g. by using k-means clustering in original space i.e. $\mathbb{R}^d$

Re-compute α values for these reduced set support vectors e.g. by running SVM again on $\{\tilde{\mathbf{x}}_{i_1}, \ldots, \tilde{\mathbf{x}}_{i_k}\}$

*Burges and Scholkopf, **Improving the Speed and Accuracy of SVMs**, NIPS 1996.*

*Cossalter et al. **Adaptive Kernel Approximation for Large-Scale Non-Linear SVM Prediction**, ICML 2011.*



Approximate Training

Kernel SVMs always use support vectors that are a subset of train data

Maybe removing this restriction can reduce their number?

Learn support vectors as well (that are not necessarily training points)

Given budget k on how many SV we are allowed (based on space/time)

Learn $\mathbf{z}^1, \ldots, \mathbf{z}^k \in \mathbb{R}^d$ *and* $\alpha_1, \ldots, \alpha_k \in \mathbb{R}$ *so that* $\mathbf{w} = \sum_{i=1}^{k} \alpha_i \cdot \phi(\mathbf{z}^i) \in \mathcal{H}$ *is a good model for our learning task*

Reduces model size and prediction time to $O(kd)$ *where we can control* k

No free lunch: training problem becomes much more difficult – not a convex problem anymore ☹

Joachims and Yu. ***Sparse Kernel SVMs via Cutting-Plane Training****, Machine Learning 76(2):179-193, 2009*

Tsang et al. ***Core Vector Machines****, JMLR 6:363-392, 2005*

Kernel Approximation - Landmarking

Given a training set $S = \{\mathbf{x}^1, \ldots, \mathbf{x}^n\} \subset \mathbb{R}^d$ and a kernel K

Based on budget, Identify k train points $\{\hat{\mathbf{x}}^1, \ldots, \hat{\mathbf{x}}^k\}$ (e.g. clustering, randomly)

Represent every other point (train/test) in terms of their similarity to these prototypes where similarity is given by the kernel K

$$\hat{\phi}(\mathbf{x}) = \left[K(\mathbf{x}, \hat{\mathbf{x}}^1), \ldots, K\left(\mathbf{x}, \hat{\mathbf{x}}^k\right)\right] \in \mathbb{R}^k$$

Treat $\hat{\phi}$ as a new feature representation and use it in ML algos

If k is reasonably small, can use linear SVM/RR/PCA on new features directly

Offers $O(k)$ model size and $O(dk)$ prediction time where we can control k

Can be shown mathematically that if K was nice for the problem (e.g. gave us small classification/regression error), then so will $\hat{\phi}$

*Balcan and Blum. **On a Theory of Learning with Similarity Functions**, ICML 2006.*

*K. and Jain. **Similarity-based Learning via Data driven Embeddings**, NIPS 2011.*

Kernel Approximation – Nystrom Method

Close cousin of landmarking – removes redundancy in landmarks

Suppose we have chosen k *landmarks* $\{\hat{\mathbf{x}}^1, \ldots, \hat{\mathbf{x}}^k\} \subset S$. *Let* $G = U\Lambda U^\top$ *be the Gram matrix of these landmarks w.r.t kernel* K *and its (thin) eigendecomp.*

As before, define the landmarked feature map $\hat{\phi}(\mathbf{x}) = \left[K(\mathbf{x}, \hat{\mathbf{x}}^1), \ldots, K\left(\mathbf{x}, \hat{\mathbf{x}}^k\right)\right]$ *but instead use* $\tilde{\phi}(\mathbf{x}) = \sqrt{\Lambda^{-1}} U^\top \hat{\phi}(\mathbf{x})$ *(*Λ^{-1} *exists since we chose thin ED)*

Another interpretation – landmarking effectively asks us to use the kernel $\hat{K}(\mathbf{x}, \mathbf{y}) = \hat{\phi}(\mathbf{x})^\top \hat{\phi}(\mathbf{y})$ *whereas Nystrom effectively asks us to use the kernel*

$$\tilde{K}(\mathbf{x}, \mathbf{y}) = \tilde{\phi}(\mathbf{x})^\top \tilde{\phi}(\mathbf{y}) = \hat{\phi}(\mathbf{x})^\top G^\dagger \hat{\phi}(\mathbf{y})$$

Suppose we cheated and chose $\hat{\mathbf{x}}^1 = \cdots = \hat{\mathbf{x}}^k$ (i.e. same landmark)

Nystrom method will detect this since G *will be rank 1 and so* $\tilde{\phi}(\mathbf{x}) \in \mathbb{R}^1$

Williams et al. ***Using Nystrom Method to Speed Up Kernel Machines****, NIPS 2000*

Yang et al. ***Nystrom Method vs Random Fourier Features****, NIPS 2012*

Kernel Approximation – Explicit Features

Kernels use high/infinite dim feature maps – root of all problems

Can we get finite (small) dim feature maps that approx. the kernel value?

Given a kernel K *over a domain* $\mathcal{X}$ *and budget* k*, find a map* $\bar{\phi}: \mathcal{X} \to \mathbb{R}^k$ *s.t.*

$$K(\mathbf{x}, \mathbf{y}) \approx \langle \bar{\phi}(\mathbf{x}), \bar{\phi}(\mathbf{y}) \rangle$$

If such maps exist then we can simply execute linear SVM/RR/PCA with $\bar{\phi}(\mathbf{x})$

Will get performance similar to what using K *directly with kernel SVM/RR/PCA would have given but now with only* $\mathcal{O}(k)$ *model size,* $\mathcal{O}(dk)$ *prediction time*

Note that landmarking/Nystrom do not seek to approximate kernel values

Why should such kernel approximating maps even exist?

We call a kernel K *a rank* r *kernel if its feature map* $\phi_K: \mathcal{X} \to \mathbb{R}^r$

Several popular kernels (poly, Gaussian, Laplacian) have a funny property that they can be expressed as an average (expectation) of rank 1 kernels

Several techniques exploit this property to construct such maps

Kernel Approximation – Explicit Features

Several popular kernels can be written as

$$K(\mathbf{x},\mathbf{y}) = \mathbb{E}_\omega[K_\omega(\mathbf{x},\mathbf{y})]$$

where $\omega \sim \mathcal{D}_K$ *where* $\mathcal{D}_K$ *is a distribution specific to* K *typically over* $\mathbb{N}, \mathbb{R}^d$

For every ω, $K_\omega(\mathbf{x},\mathbf{y})$ *is rank 1 i.e.* $\phi_\omega : \mathcal{X} \to \mathbb{R}$ *and* $K_\omega(\mathbf{x},\mathbf{y}) = \phi_\omega(\mathbf{x}) \cdot \phi_\omega(\mathbf{y})$

Such results are often classical e.g. Bochner's theorem, Schoenberg's theorem

Exploiting such a result is straightforward – sample $\omega_1, \ldots \omega_k \sim \mathcal{D}_K$ *and let*

$$\bar{\phi}(\mathbf{x}) = \frac{1}{\sqrt{k}} \cdot \left[\phi_{\omega_1}(\mathbf{x}), \ldots, \phi_{\omega_k}(\mathbf{x})\right] \in \mathbb{R}^k$$

For example, for the Gaussian kernel, Bochner's theorem tells us that

$$K(\mathbf{x},\mathbf{y}) = \mathbb{E}_{\boldsymbol{\omega}}[\cos(\boldsymbol{\omega}^\top \mathbf{x}) \cdot \cos(\boldsymbol{\omega}^\top \mathbf{y})]$$

where $\boldsymbol{\omega} \sim \mathcal{N}(\mathbf{0}, I_d)$ *i.e.* $\boldsymbol{\omega} \in \mathbb{R}^d$ *and* $\phi_{\boldsymbol{\omega}} : \mathbf{x} \mapsto \cos(\boldsymbol{\omega}^\top \mathbf{x})$

Several papers mathematically show that $K(\mathbf{x},\mathbf{y}) \approx \langle \bar{\phi}(\mathbf{x}), \bar{\phi}(\mathbf{y}) \rangle$

Kernel Approximation – Explicit Features

Feature constructions for Gaussian/Laplacian, intersection, poly kernels

Rahimi and Recht, Random Features for Large Scale Kernel Machines, NIPS '07

Maji and Berg, Max-margin Additive Classifiers for Detect, ICCV 2009

K. and Karnick. Random Feature Maps for Dot Product Kernels. AISTATS 2012

Other interesting approaches to kernel approximation exist too

Use decision trees to compute similarity between two points and use that as kernel – extremely fast prediction

Jose et al. Local Deep Kernel Learning, ICML 2013.

Learn these kernel approximations in a task-dependent manner

Perronnin et al. Large-scale Image Categorization with Explicit Data Embedding, CVPR 2010.