Large Scale Learning

Dr. Fayyaz ul Amir Afsar Minhas
PIEAS Biomedical Informatics Research Lab
Department of Computer and Information Sciences
Pakistan Institute of Engineering & Applied Sciences
PO Nilore, Islamabad, Pakistan
http://faculty.pieas.edu.pk/fayyaz/

Topics Covered

- Gradients and Subgradients
- Gradient descent based learning
  - PEGASOS
  - SARAH
  - Coordinate Descent
- Kernel Approximation
  - Fourier Approximation
- Randomized Algorithms
  - Random Kitchen Sinks
  - Random Projections
  - Extreme Learning Machine
  - Doubly Stochastic Learning

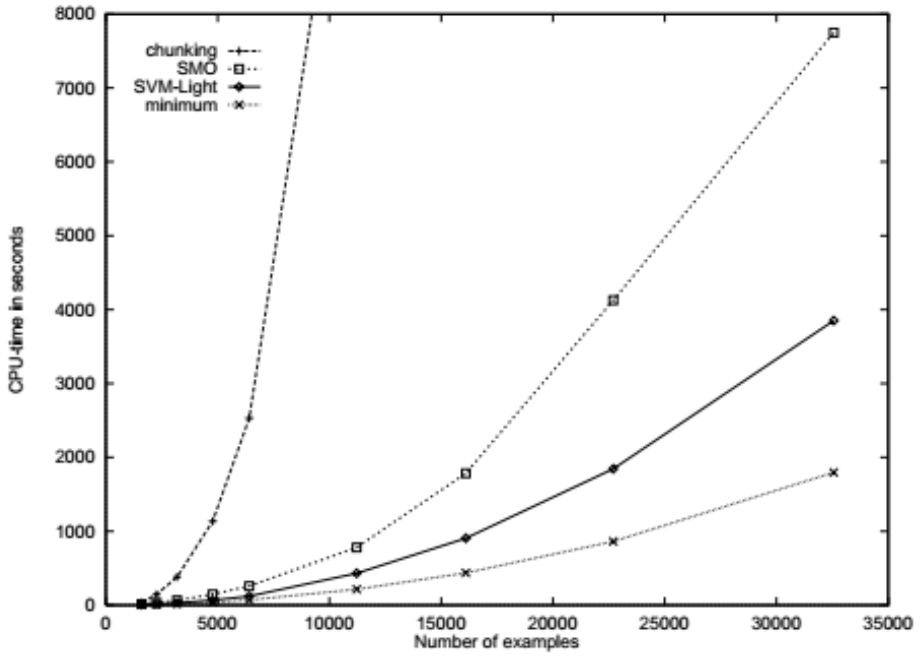
Learning from Big Data

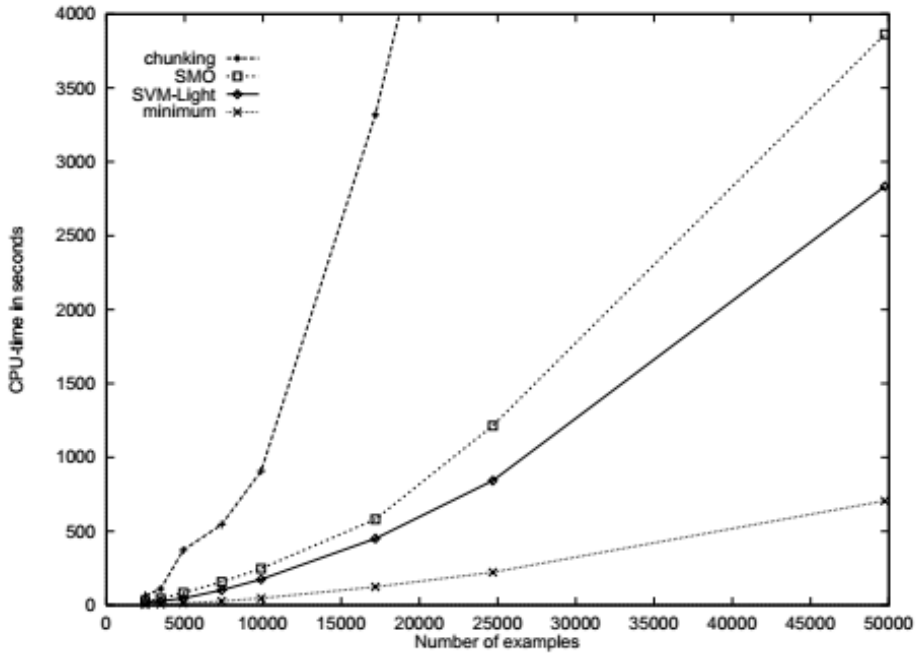
- Classification of large data sets
- Requires super-efficient learning algorithms
  - These problems are constrained by the total computation time
  - In small scale learning problems, the error is constrained by the number of training examples
- QP solver for SVM
  - Given n examples, the computational cost is $O(n^2)$ to $O(n^3)$ given a precomputed kernel matrix for small and large values of C respectively
    - Bottou and Lin, “Support Vector Machine Solvers”, JMLR

Big Data issues

- Kernel computation
  - Computing the kernel is expensive
  - Only a few kernel values have any impact on the solution (support vectors!)
  - The kernel does not fit in the memory

Computation time





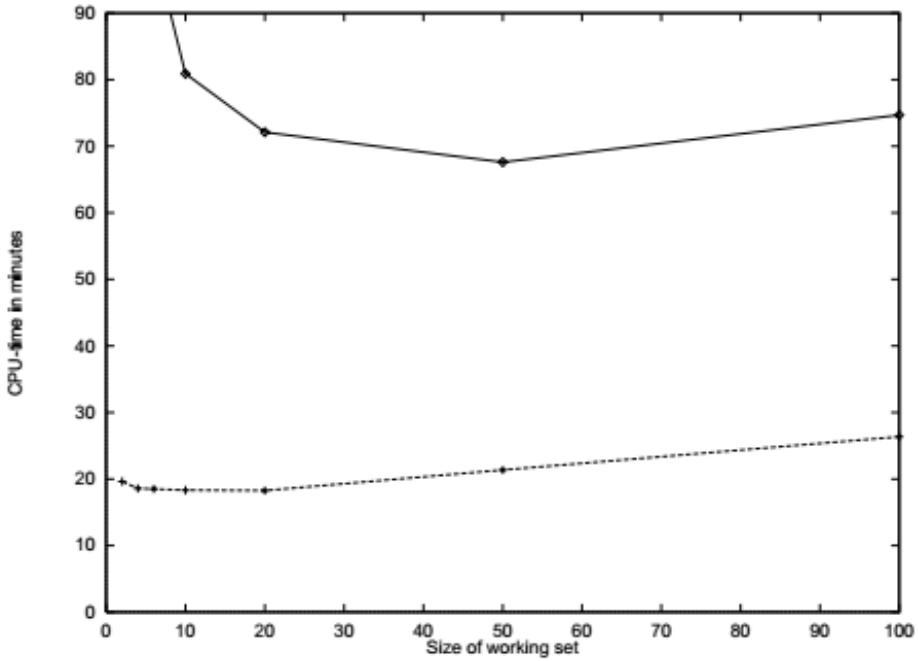
- Joachims, “Making Large-Scale SVM Learning Practical” in: 'Advances in Kernel Methods - Support Vector Learning', Bernhard Scholkopf, Christopher J. C. Burges, and Alexander J. Smola (eds.),
MIT Press, Cambridge, USA, 1998

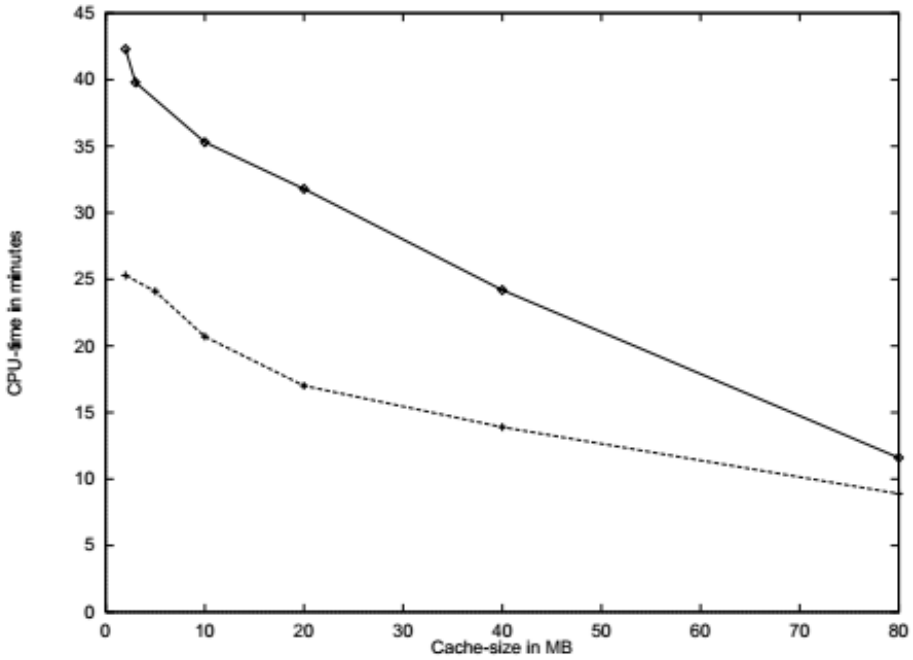
Computation time

- Scikit-SVM has complexity of $O(dn^2)$ to $O(dn^3)$ and uses libsvm
  - Can benefit from sparse data
  - Can benefit from caching the kernel
  - Linear SVM is faster (sklearn.svm.LinearSVC) and uses liblinear for larger number of examples
- Check out: Tips on practical use
  - Data Copying
  - Kernel Caching
  - Data Scaling and Preprocessing (badly scaled data will cause issues!)
  - Uses Joblib (built-in support for parallelization!)
  - Use Stochastic Gradient Descent (sklearn.linear_model.SGDClassifier)

http://scikit-learn.org/stable/modules/svm.html#tips-on-practical-use

Effect of Kernel Caching





Representation

- SVM style algorithms can be represented as:

$$\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{(\mathbf{x},y)\in S} \ell(\mathbf{w};(\mathbf{x},y))$$

$$\ell(\mathbf{w};(\mathbf{x},y)) \;=\; \max\{0, 1 - y\,\langle \mathbf{w}, \mathbf{x}\rangle\}$$

  - The loss functions and the regularizers can be changed
  - We solved the dual of the above formulation using a QP Solver
  - However, to develop new classifiers, we need to know about some “easy” and fast optimization methods

Types of SVM Solvers

- Interior Point Methods: $O(n^2)$, reaches an ϵ-accurate solution in $O(\log(\log(1/\epsilon))$
- Decomposition based methods: SMO, SVM-Light, typically super-linear in the number of examples
- Cutting plane methods: reach an ϵ-accurate solution in $O(nd / \lambda\epsilon)$ time
- Primal solvers: Use the representation theorem $\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$ to solve the nonlinear classification proble in the dual with conjugate gradient or newton optimization methods
- Stochastic gradient descent
  - http://scikit-learn.org/stable/modules/sgd.html

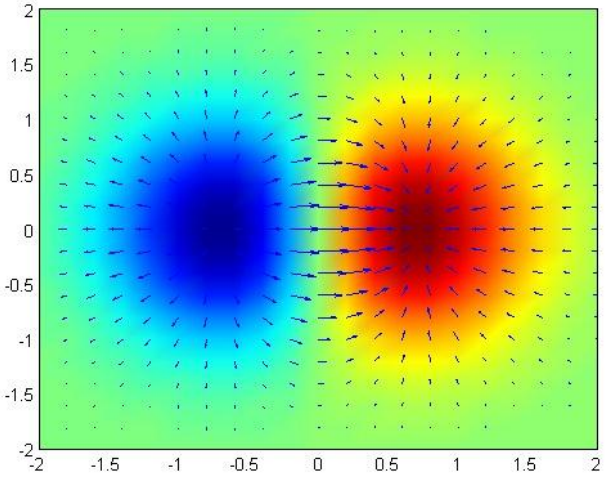
Efficient Algorithms for your own learning machines

- PEGASOS
  - Shalev-Shwartz, Shai, Yoram Singer, Nathan Srebro, and Andrew Cotter. "Pegasos: Primal Estimated Sub-Gradient Solver for SVM." *Mathematical Programming* 127, no. 1 (March 1, 2011): 3–30. doi:10.1007/s10107-010-0420-4.
  - Gradient based method
  - Solves the SVM in the primal
  - The number of iterations is not dependent upon the number of examples
  - Easy implementation!
  - Supports out of core learning
    - Learning from data that doesn't fit into main memory
    - Online classifier, i.e., the learning can be done in batches

Gradient

- a generalization of the usual concept of derivative of a function in one dimension to a function in several dimensions.
- Given a function $f(\boldsymbol{x}) = f(x_1, x_2, \ldots, x_n)$, its gradient is given by

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

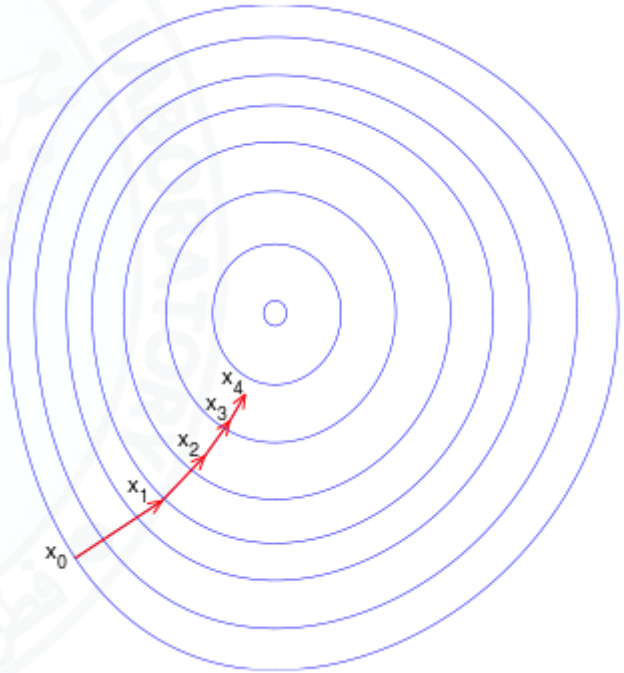


$$f(\boldsymbol{x}) = x_1 e^{{x_1}^2 + {x_2}^2}$$

Gradient Descent

- A derivative based optimization method to find local minimum

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

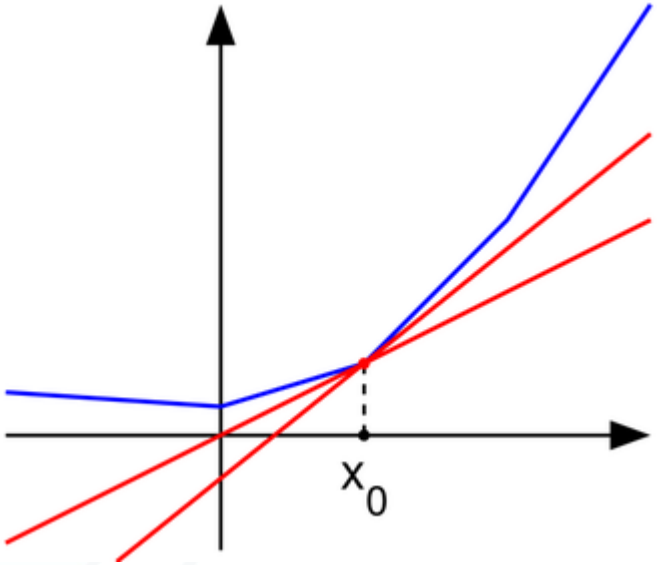


Subgradient

– Generalizes the gradient to functions which are not differentiable.

– Given a function f(x), its subderivative is any value of ‘c’ for which

$$f(x) - f(x_0) \geq c(x - x_0)$$



Subgradient

- the set of sub-derivatives at x_0 for a convex function is a nonempty closed interval [a, b]

$$a = \lim_{x \to x_0^-} \frac{f(x) - f(x_0)}{x - x_0}$$

$$b = \lim_{x \to x_0^+} \frac{f(x) - f(x_0)}{x - x_0}$$

- The set [*a*, *b*] of all subderivatives is called the subdifferential of the function *f* at x_0. If *f* is convex and its subdifferential at contains exactly one subderivative, then *f* is differentiable at x_0.

Subgradient

- The Taylor series approximation of a function at ‘a’ is given by

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots.$$

- The first order approximation is

$$f(a) + \frac{f'(a)}{1!}(x-a)$$

- Thus, if the first order approximation f(a)+c(x-a) of the function under-approximates the function, then ‘c’ is a subderivative of the function!

Examples

Loss function	Subgradient
$\ell(z, y_i) = \max\{0, 1 - y_i z\}$	$\mathbf{v}_t = \begin{cases} -y_i \mathbf{x}_i & \text{if } y_i z < 1 \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \log(1 + e^{-y_i z})$	$\mathbf{v}_t = -\frac{y_i}{1+e^{y_i z}} \mathbf{x}_i$
$\ell(z, y_i) = \max\{0, \lvert y_i - z \rvert - \epsilon\}$	$\mathbf{v}_t = \begin{cases} \mathbf{x}_i & \text{if } z - y_i > \epsilon \\ -\mathbf{x}_i & \text{if } y_i - z > \epsilon \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \max_{y \in \mathcal{Y}} \delta(y, y_i) - z_{y_i} + z_y$	$\mathbf{v}_t = \phi(\mathbf{x}_i, \hat{y}) - \phi(\mathbf{x}_i, y_i)$ where $\hat{y} = \arg\max_y \delta(y, y_i) - z_{y_i} + z_y$
$\ell(z, y_i) = \log\left(1 + \sum_{r \neq y_i} e^{z_r - z_{y_i}}\right)$	$\mathbf{v}_t = \sum_r p_r \phi(\mathbf{x}_i, r) - \phi(\mathbf{x}_i, y_i)$ where $p_r = e^{z_r} / \sum_j e^{z_j}$

Derivation of Pegasos

- The basic optimization problem of SVM can be written as:

$$\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{(\mathbf{x},y)\in S} \ell(\mathbf{w};(\mathbf{x},y)) \ ,$$

where

$$\ell(\mathbf{w};(\mathbf{x},y)) \ = \ \max\{0, 1 - y\,\langle \mathbf{w},\mathbf{x}\rangle\} \ ,$$

- Pegasos operates by picking one example randomly at each iteration and computing the value of the objective function with respect to that (Stochastic!)

$$f(\mathbf{w};i_t) \ = \ \frac{\lambda}{2}\|\mathbf{w}\|^2 + \ell(\mathbf{w};(\mathbf{x}_{i_t},y_{i_t}))$$

Pegasos Derviation

- The subgradient of this function is given by

$$\nabla_t = \lambda \mathbf{w}_t - \mathbb{1}[y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1] \, y_{i_t} \mathbf{x}_{i_t}$$

- The weight update equation can be written as

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla_t$$

- With the learning rate given by $\eta_t = 1/(\lambda t)$
  - The learning rate is gradually reduced ("annealing")
- The complete equation thus becomes

$$\mathbf{w}_{t+1} \leftarrow (1 - \frac{1}{t}) \mathbf{w}_t + \eta_t \mathbb{1}[y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1] \, y_{i_t} \mathbf{x}_{i_t}$$

Projection step (optional)

- Limit the weight vector to a ball of radius $1/\sqrt{\lambda}$.
  - Better control of generalization

- Done by

$$\mathbf{w}_{t+1} \leftarrow \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$$

Pegasos Algorithm

INPUT: S, λ, T, k
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR $t = 1, 2, \ldots, T$
Choose $A_t \subseteq [m]$, where $|A_t| = k$, uniformly at random
Set $A_t^+ = \{i \in A_t : y_i \langle \mathbf{w}_t, \mathbf{x}_i \rangle < 1\}$
Set $\eta_t = \frac{1}{\lambda t}$
Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \frac{\eta_t}{k} \sum_{i \in A_t^+} y_i \mathbf{x}_i$
[Optional: $\mathbf{w}_{t+1} \leftarrow \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$]
OUTPUT: $\mathbf{w}_{T+1}$

Mini-batch version

- Select 'k' examples at each step in a set A_t
- Potentially better approximation of the gradient through averaging over examples in A_t

INPUT: S, λ, T, k
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR $t = 1, 2, \ldots, T$
Choose $A_t \subseteq [m]$, where $|A_t| = k$, uniformly at random
Set $A_t^+ = \{i \in A_t : y_i \langle \mathbf{w}_t, \mathbf{x}_i \rangle < 1\}$
Set $\eta_t = \frac{1}{\lambda t}$
Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \frac{\eta_t}{k} \sum_{i \in A_t^+} y_i \mathbf{x}_i$
[Optional: $\mathbf{w}_{t+1} \leftarrow \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$]
OUTPUT: $\mathbf{w}_{T+1}$

Properties

- **Convergence**
  - To reach an ε-accurate solution in which $f(\boldsymbol{w_T}) - f(\boldsymbol{w}^*) \leq \varepsilon$, the number of iterations is $T = \widetilde{\boldsymbol{O}}\left(\frac{\boldsymbol{d}}{\boldsymbol{\lambda\varepsilon}}\right)$
    - $f(w) = \frac{\lambda}{2}\boldsymbol{w^T w} + \frac{1}{m}\sum_{i=1}^{m} l(\boldsymbol{x_i}, y_i; \boldsymbol{w})$
    - $\boldsymbol{w}^* = argmin_{\boldsymbol{w}}\, f(\boldsymbol{w})$
    - d is the dimensionality of the data
    - $\tilde{O}\big(g(n)\big)$= $O\left(g(n)log^k\big(g(n)\big)\right)$, $\exists k$, Pronounced Soft-O Ignores logarithmic factors because $g(n)$ trumps $log^k\big(g(n)\big)$
    - Number of iterations is Independent of the number of data points
      - Ideally suited for large scale data
    - Inverse dependence on training set size (SVM Optimization: Inverse Dependence on Training Set Size by Shalev-Shwartz and Srebro, 2008) – Training time to reach a certain classification error decreases with increase in training set size!
      - The runtime required to get a generalization error $l(\boldsymbol{w}) \leq \varepsilon + l(\boldsymbol{w}^*)$ using a training set of size m is $\tilde{O}\left(\frac{d}{\left(\frac{\varepsilon}{\|\boldsymbol{w}^*\|} - O\left(\frac{1}{\sqrt{m}}\right)\right)^2}\right)$
- **Easy Implementation & Extensibility to other learning problems**
- **Easy batch processing**
- **Easy Parallelization: Scikit and Spark implementation of SGD**
  - **Simple Averaging the weights from different machines, "Parallel SGD: When does averaging help?", Zhang et al., 2016**
  - **"Slow learners are Fast", Langford et al., 2009**
- **Issues**
  - **Bias (can be added as an additional feature or do a binary search over the bias term), Typically not needed for large number of features**
  - **Sensitive to feature scaling (normalize and standardize your features via preprocessing)**
  - **Averaging: A simpler approach to obtaining an O(1/t) convergence rate for the projected stochastic subgradient method", and "Stochastic Gradient Descent for Non-smooth Optimization: Convergence Results and Optimal Averaging Schemes"**
    - **Minimal impact for convex learning problems, more useful for non-smooth optimization**

For Kernels

- The weight update equation can be written as

$$\mathbf{w}_{t+1} \leftarrow (1 - \frac{1}{t})\mathbf{w}_t + \eta_t \mathbb{1}\left[y_{i_t} \left\langle \mathbf{w}_t, \mathbf{x}_{i_t} \right\rangle < 1\right] y_{i_t} \mathbf{x}_{i_t}$$

- Let's take: $\mathbf{v}_t = \frac{1}{|A_t|} \sum_{i \in A_t} \mathbb{1}\left[y_i \left\langle \mathbf{w}_t, \mathbf{x}_t \right\rangle < 1\right] y_i \mathbf{x}_i$

- Thus: $\mathbf{w}_{t+1} = \left(1 - \frac{1}{t}\right) \mathbf{w}_t + \frac{1}{t\lambda} \mathbf{v}_t$

This equation (12) in the paper has a mistake with a '-' when it should have been '+'

- Assume that at iteration 'i' we choose example 'i' and then try to see what its impact on the weight vector is uptil iteration 't' independent of other examples (assume only one example!)
- For this purpose we take
  - $w_i = 0$
  - The initial multiplier of v_t is thus $\frac{1}{\lambda i}$

Using Mercer's Conditions

- We can now solve the recurrence with
  - $\boldsymbol{w_i = 0}$
  - $\boldsymbol{w_{i+1}} = \left(1 - \frac{1}{i}\right)\boldsymbol{w_i} + \frac{1}{\lambda i}\boldsymbol{v_i} = \frac{1}{\lambda i}\boldsymbol{v_i}$
  - $\boldsymbol{w_{i+2}} = \left(1 - \frac{1}{i+1}\right)\boldsymbol{w_{i+1}} + \frac{1}{\lambda(i+1)}\boldsymbol{v}_{i+1} = \left(\frac{i}{i+1}\right)\frac{1}{\lambda i}\boldsymbol{v_i} + \frac{1}{\lambda(i+1)}\boldsymbol{v}_{i+1} = \frac{1}{\lambda(i+1)}(\boldsymbol{v_i} + \boldsymbol{v}_{i+1})$

- In general, $\boldsymbol{w_{t+1}} = \frac{1}{\lambda t}\sum_{i=1}^{t}\boldsymbol{v_i}$

- Substituting v_i $$\mathbf{w}_{t+1} = \frac{1}{\lambda t}\sum_{i=1}^{t}\mathbb{1}\left[y_{i_t}\left\langle \mathbf{w}_t, \phi(\mathbf{x}_{i_t})\right\rangle < 1\right] y_{i_t}\phi(\mathbf{x}_{i_t})$$

- Let, denote the number of times, uptil the current iteration, the example i_t has been chosen and it has violated the margin

$$\alpha_{t+1}[j] = \left|\left\{t' \leq t : i_{t'} = j \wedge y_j \left\langle \mathbf{w}_{t'}, \phi(\mathbf{x}_j)\right\rangle < 1\right\}\right|$$

Using Mercer's Conditions

- With this definition and $\mathbf{w}_{t+1} = \frac{1}{\lambda t} \sum_{i=1}^{t} \mathbb{1}[y_{i_t} \langle \mathbf{w}_t, \phi(\mathbf{x}_{i_t}) \rangle < 1] \, y_{i_t} \phi(\mathbf{x}_{i_t})$
- We get: $\boldsymbol{w_{t+1}} = \frac{1}{\lambda t} \alpha_{t+1}[j] y_{i_t} \boldsymbol{\phi}(x_{i_t})$
- Since we have a total of m' examples we get

$$\mathbf{w}_{t+1} = \frac{1}{\lambda t} \sum_{j=1}^{m} \alpha_{t+1}[j] \, y_j \phi(\mathbf{x}_j)$$

- The final weight vector will thus be: $\boldsymbol{w} = \frac{1}{\lambda T} \sum_{j=1}^{m} \alpha_{T+1}[j] y_j \boldsymbol{\phi}(x_j)$
- This is inline with the Representer Theorem
  - The weight vector is being expressed as a linear combination of the given examples

Using Mercer's Conditions

- With this, we can now evaluate the output score of an example as:
  - $\boldsymbol{w^T \phi(x)} = \frac{1}{\lambda T}\sum_{j=1}^{m} \alpha_{T+1}[j] y_j \boldsymbol{\phi}^T(x_j)\boldsymbol{\phi(x)} = \frac{1}{\lambda T}\sum_{j=1}^{m} \alpha_{T+1}[j] y_j K(x_j, x)$
- Thus for learning, all we need are the α for each example
- Note that the calculation of α requires us to see if the chosen example violates the margin or not which in turn requires us to compute the loss function which is dependent on the output score for that example at that iteration
- Thus, each iteration now becomes O(m)
  - Overall, runtime with kernels thus becomes $\boldsymbol{O\left(\frac{m}{\lambda \varepsilon}\right)}$
  - As a consequence of this, the runtime is dependent upon the number of examples even though the number of iterations is not
  - Also note that the problem is still being solved in the primal as the optimization is still with respect to **w** and not the lagrange variables in the dual representation

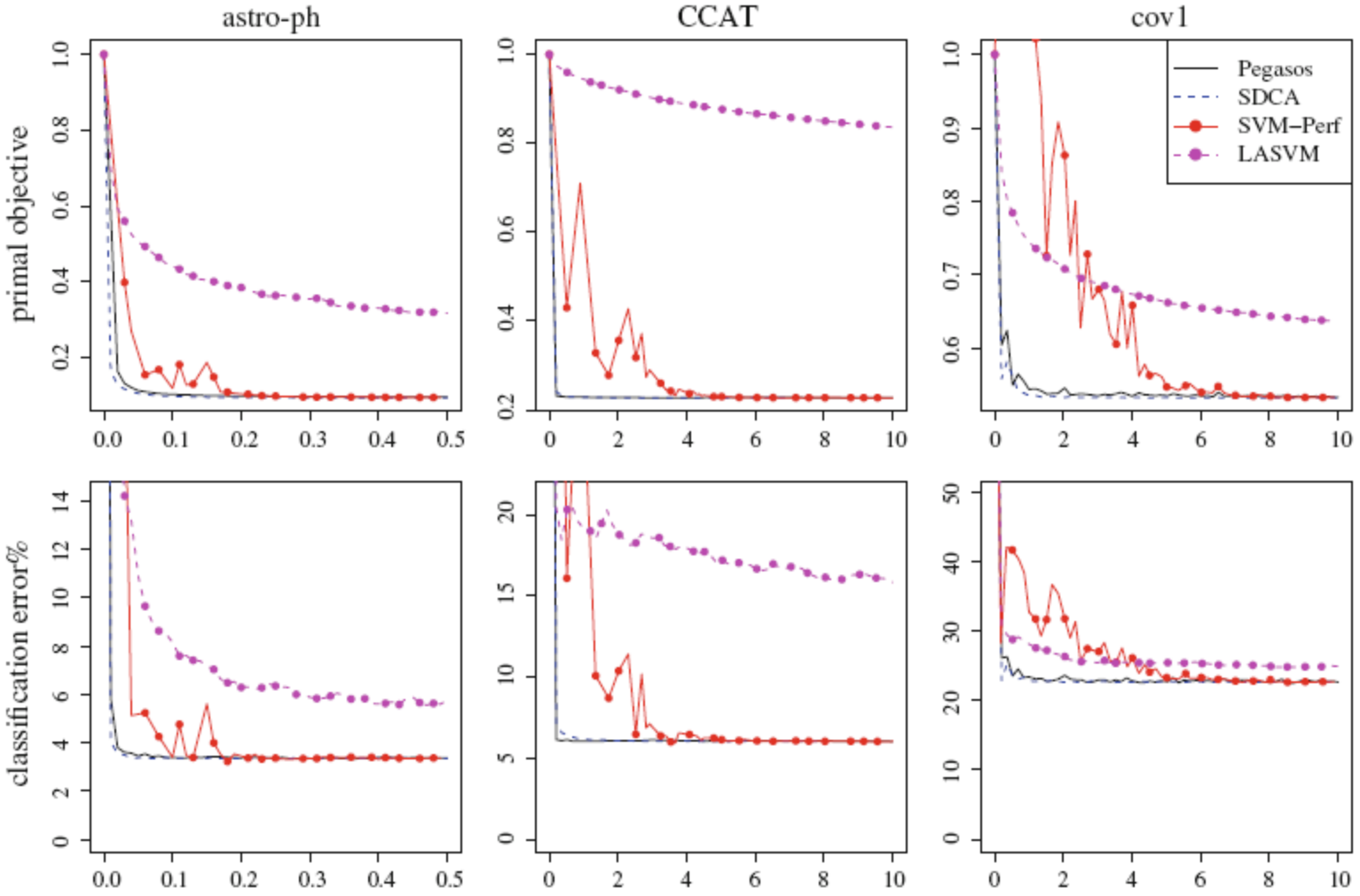


Fig. 4 Comparison of linear SVM optimizers. Primal suboptimality (*top row*) and testing classification error (*bottom row*), for one run each of Pegasos, stochastic DCA, SVM-Perf, and LASVM, on the astro-ph (*left*), CCAT (*center*) and cov1 (*right*) datasets. In all plots the horizontal axis measures runtime in seconds

Timing results (linear)

- Superfast!

Table 1 Training runtime and test error achieved (in parentheses) using various optimization methods on linear SVM problems

Dataset	Pegasos	SDCA	SVM-Perf	LASVM
astro-ph	0.04*s* (3.56%)	0.03*s* (3.49%)	0.1*s* (3.39%)	54*s* (3.65%)
CCAT	0.16*s* (6.16%)	0.36*s* (6.57%)	3.6*s* (5.93%)	$> 18000s$
cov1	0.32*s* (23.2%)	0.20*s* (22.9%)	4.2*s* (23.9%)	210*s* (23.8%)

Kernelized

- Slower!

Dataset	Pegasos	SDCA	SVM-Light	LASVM
Reuters	15*s* (2.91%)	13*s* (3.15%)	4.1*s* (2.82%)	4.7*s* (3.03%)
Adult	30*s* (15.5%)	4.8*s* (15.5%)	59*s* (15.1%)	1.5*s* (15.6%)
USPS	120*s* (0.457%)	21*s* (0.508%)	3.3*s* (0.457%)	1.8*s* (0.457%)
MNIST	4200*s* (0.6%)	1800*s* (0.56%)	290*s* (0.58%)	280*s* (0.56%)

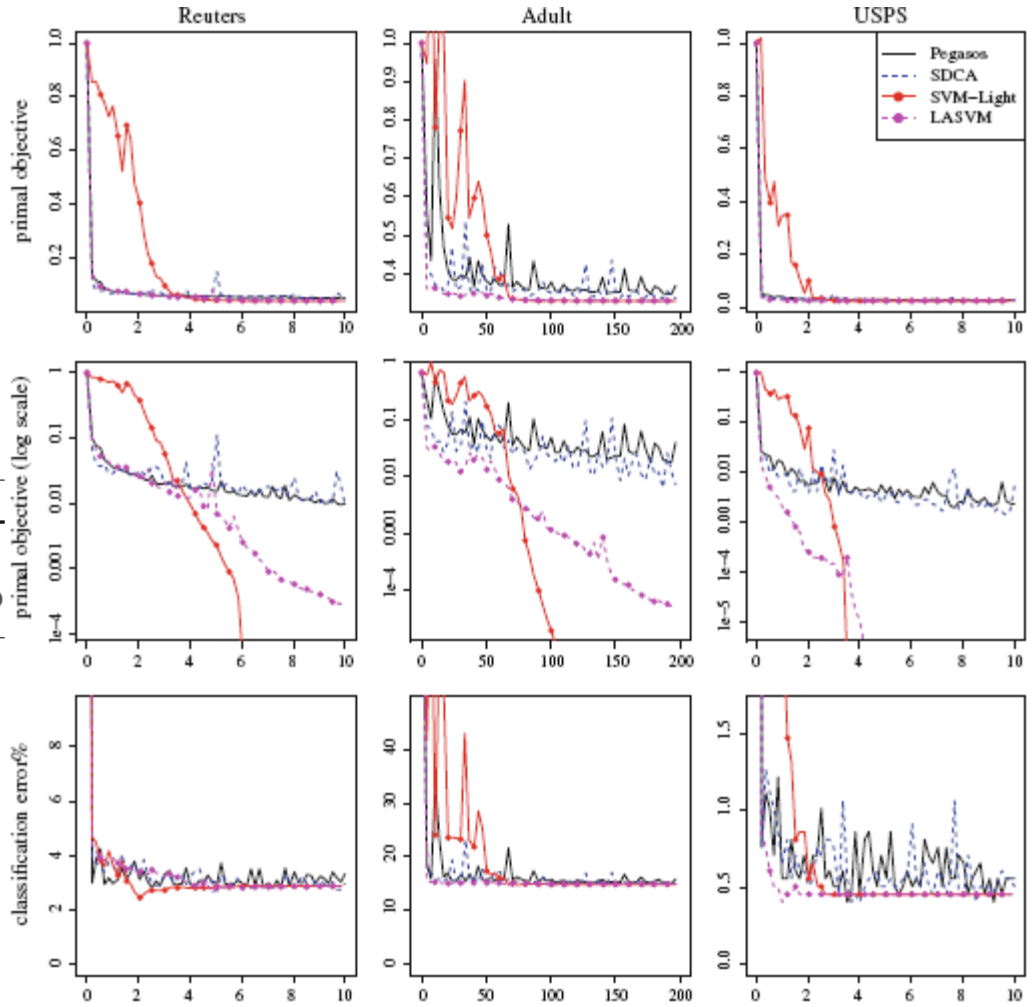


Fig. 5 Comparison of kernel SVM optimizers. Primal suboptimality (*top row*), primal suboptimality in log scale (*middle row*) and testing classification error (*bottom row*), for one run each of Pegasos, stochastic DCA, SVM-Light, and LASVM, on the Reuters (*left column*), Adult (*center column*) and USPS (*right column*) datasets. Plots of traces generated on the MNIST dataset (not shown) appear broadly similar to those for the USPS dataset. The horizontal axis is runtime in seconds

Effect of batches

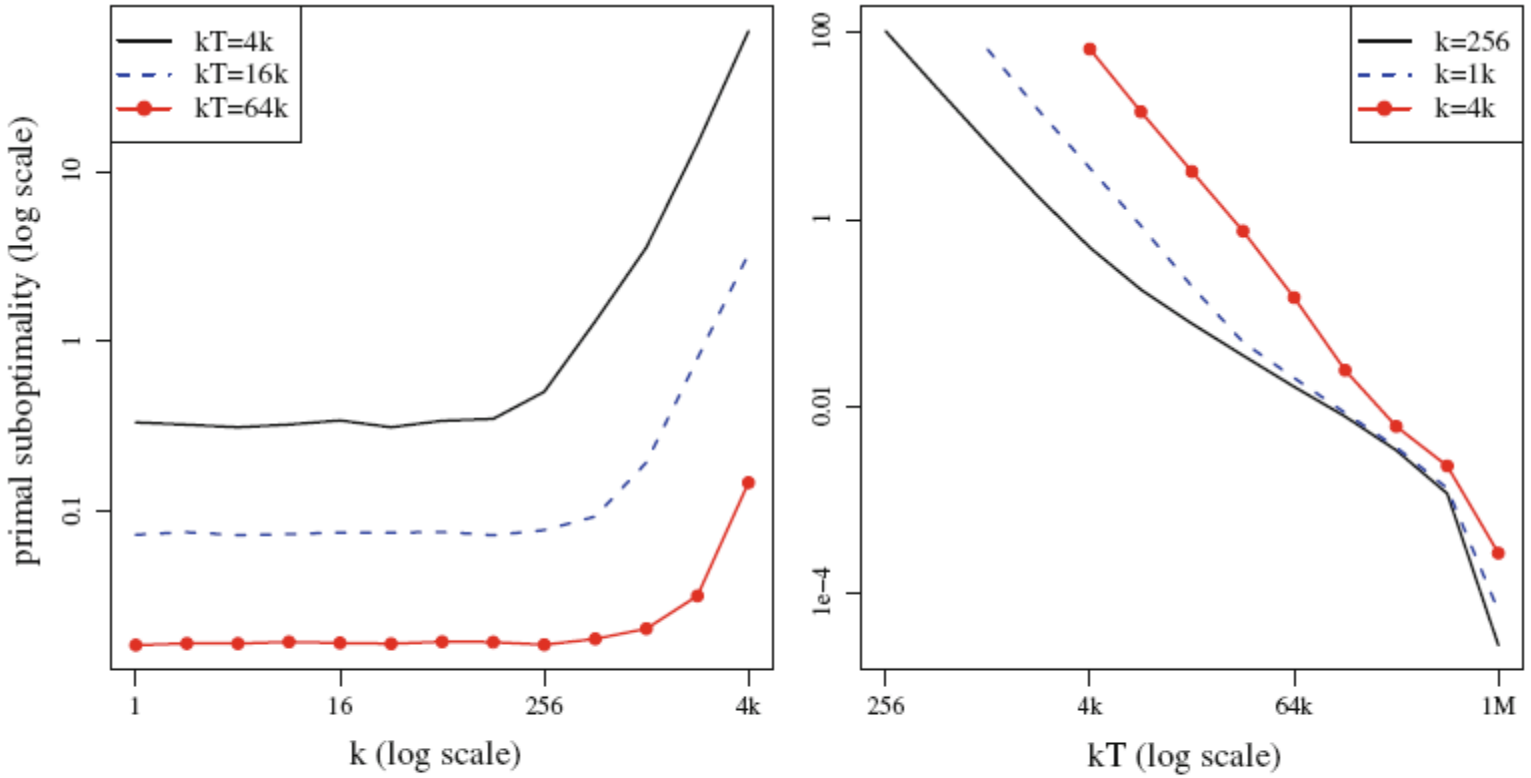
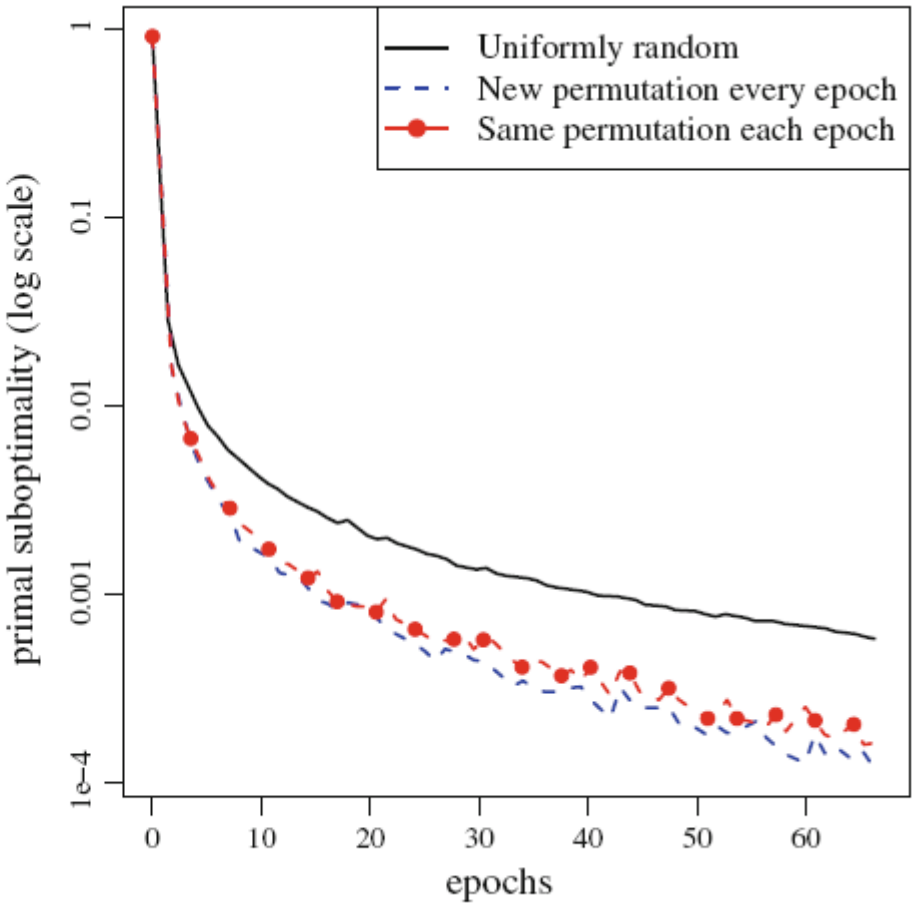


Fig. 7 The effect of the mini-batch size on the runtime of Pegasos for the astro-ph dataset. The first plot shows the primal suboptimality achieved for certain fixed values of overall runtime kT, for various values of the mini-batch size k. The second plot shows the primal suboptimality achieved for certain fixed values of k, for various values of kT. Very similar results were achieved for the CCAT dataset

Effect of Sampling

Fig. 8 The effect of different sampling methods on the performance of Pegasos for the astro-ph dataset. The curves show the primal suboptimality achieved by uniform i.i.d. sampling, sampling from a fixed permutation, and sampling from a different permutation for every epoch



- One epoch is when the classifier has used all input examples once

Solving other problems

- A number of other problems can be solved using SGD just by choosing the appropriate loss function

Loss function	Subgradient
$\ell(z, y_i) = \max\{0, 1 - y_i z\}$ **Binary classification**	$\mathbf{v}_t = \begin{cases} -y_i \mathbf{x}_i & \text{if } y_i z < 1 \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \log(1 + e^{-y_i z})$ **Binary classification with log-loss**	$\mathbf{v}_t = -\frac{y_i}{1+e^{y_i z}} \mathbf{x}_i$
$\ell(z, y_i) = \max\{0, \lvert y_i - z \rvert - \epsilon\}$ **Regression**	$\mathbf{v}_t = \begin{cases} \mathbf{x}_i & \text{if } z - y_i > \epsilon \\ -\mathbf{x}_i & \text{if } y_i - z > \epsilon \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \max_{y \in \mathcal{Y}} \delta(y, y_i) - z_{y_i} + z_y$ **Multi-class classification**	$\mathbf{v}_t = \phi(\mathbf{x}_i, \hat{y}) - \phi(\mathbf{x}_i, y_i)$ where $\hat{y} = \arg\max_y \delta(y, y_i) - z_{y_i} + z_y$
$\ell(z, y_i) = \log\left(1 + \sum_{r \neq y_i} e^{z_r - z_{y_i}}\right)$ **Multi-class classification**	$\mathbf{v}_t = \sum_r p_r \phi(\mathbf{x}_i, r) - \phi(\mathbf{x}_i, y_i)$ where $p_r = e^{z_r} / \sum_j e^{z_j}$

Required Reading

- Shalev-Shwartz, Shai, Yoram Singer, Nathan Srebro, and Andrew Cotter. "Pegasos: Primal Estimated Sub-Gradient Solver for SVM." *Mathematical Programming* 127, no. 1 (March 1, 2011): 3–30. doi:10.1007/s10107-010-0420-4.

- http://ciml.info/ "Efficient Learning" and "Linear Models"

Algorithm	Citation	SVM type	Optimization type	Style	Runtime
SMO	[Platt, 1999]	Kernel	Dual QP	Batch	$\Omega(n^2 d)$
SVM$^{\text{light}}$	[Joachims, 1999]	Kernel	Dual QP	Batch	$\Omega(n^2 d)$
Core Vector Machine	[Tsang et al., 2005, 2007]	SL Kernel	Dual geometry	Batch	$O(s/\rho^4)$
SVM$^{\text{perf}}$	[Joachims, 2006]	Linear	Dual QP	Batch	$O(ns/\lambda\rho^2)$
NORMA	[Kivinen et al., 2004]	Kernel	Primal SGD	Online(-style)	$\tilde{O}(s/\rho^2)$
SVM-SGD	[Bottou, 2007]	Linear	Primal SGD	Online-style	Unknown
Pegasos	[Shalev-Shwartz et al., 2007]	Kernel	Primal SGD/SGP	Online-style	$\tilde{O}(s/\lambda\rho)$
LibLinear	[Hsieh et al., 2008]	Linear	Dual coordinate descent	Batch	$O(nd \cdot \log(1/\rho))$
SGD-QN	[Bordes and Bottou, 2008]	Linear	Primal 2SGD	Online-style	Unknown
FOLOS	[Duchi and Singer, 2008]	Linear	Primal SGP	Online-style	$\tilde{O}(s/\lambda\rho)$
BMRM	[Smola et al., 2007]	Linear	Dual QP	Batch	$O(d/\lambda\rho)$
OCAS	[Franc and Sonnenburg, 2008]	Linear	Primal QP	Batch	$O(nd)$

Table 1: A comparison of various SVM solvers discussed in this document. "QP" refers to a quadratic programming technique, "SGD" to stochastic (sub)gradient descent, and "SGP" to stochastic (sub)gradient projection. "SL" means the method only works with square-loss. The runtime is for a problem with n training examples and d features, with an average of s non-zero features per example. λ is the SVM regularization parameter, and ρ the optimization tolerance. "Unknown" means there is no known formal bound on the runtime.

Excellent read: "Large Scale Support Vector Machines: Algorithms and Theory", Research Exam by Aditya Krishna Menon

Further Developments

- Excellent read: “Large Scale Support Vector Machines: Algorithms and Theory”, Research Exam by Aditya Krishna Menon
- SARAH: A Novel Method for Machine Learning Problems Using Stochastic Recursive Gradient by Nguyen et al, March 2017
- Large Scale Kernel Learning using Block Coordinate Descent by Tu et al., 2016
- Diving into the shallows: a computational perspective on large-scale shallow learning, Mar 2017
- Efficient handling for d>>n: S. Shalev-Shwartz, A. Tewari Stochastic Methods for L1-regularized Loss Minimization [Jul.] J. Mach. Learn. Res., 12 (2011), pp. 1865–1892
- Faster Kernelization
  - Lee, Sangkyun, and Stephen J. Wright. “Stochastic Subgradient Estimation Training for Support Vector Machines.” In *Mathematical Methodologies in Pattern Recognition and Machine Learning*, edited by Pedro Latorre Carmona, J. Salvador Sánchez, and Ana L. N. Fred, 67–82. Springer Proceedings in Mathematics & Statistics 30. Springer New York, 2013. http://link.springer.com/chapter/10.1007/978-1-4614-5076-4_5.

Further Developments: Nonlinear SVM Approximations

- Budgeted SVM: A Toolbox for Scalable SVM Approximations by Djuric et al. 2013
  - Trading Representability for Scalability: Adaptive Multi-Hyperplane Machine for Nonlinear Classification
    - Super Simple!
    - Cannot handle implicit feature representations
    - BudgetedSVM: AMM
      - http://www.dabi.temple.edu/budgetedsvm/index.html

Algorithm	Training time	Prediction time	Model size
Pegasos	$O(N \cdot C \cdot S)$	$O(C \cdot S)$	$O(C \cdot D)$
AMM	$O(N \cdot S \cdot B)$	$O(S \cdot B)$	$O(D \cdot B)$
LLSVM	$O(N \cdot S \cdot B^2 + N \cdot S \cdot B)$	$O(S \cdot B^2 + S \cdot B)$	$O(D \cdot B + B^2)$
BSGD	$O(N \cdot (C + S) \cdot B)$	$O((C + S) \cdot B)$	$O((C + D) \cdot B)$
RBF-SVM	$O(I \cdot N \cdot C \cdot S)$	$O(N \cdot C \cdot S)$	$O(N \cdot C \cdot S)$

Datasets	Error rate (%)					Training time (seconds)[1]				
	AMM *batch*	AMM *online*	Linear (Pegasos)	Poly2 SVM	RBF SVM	AMM *batch*	AMM *online*	Linear (Pegasos)	Poly2 SVM	RBF SVM
a9a	15.03±0.11	16.44±0.23	15.04±0.07	14.94	14.97	2	0.2	1	2	99
ijcnn	2.40±0.11	3.02±0.14	7.76±0.19	2.16	1.31	2	0.1	1	11	27
webspam	4.50±0.24	6.14±1.08	7.28±0.09	1.56	0.80	80	4	12	3,228	15,571
mnist_bin	0.53±0.05	0.54±0.03	2.03±0.04	NA	0.43[2]	3084	300	277	NA	2 days[2]
mnist_mc	3.20±0.16	3.36±0.20	8.41±0.11	NA	0.67[3]	13864	1200	1180	NA	8 days[3]
rcv1_bin	2.20±0.01	2.21±0.02	2.29±0.01	NA	NA	1100	80	25	NA	NA
url	1.34±0.21	2.87±1.49	1.50±0.39	NA	NA	400	24	100	NA	NA

Where have we used it?

- PAIRPred
  - Significant reduction in training time
- PyLemmings
  - Pegasos inspired algorithm for multiple instance learning
- pRANK
  - Prion classification using PyLemmings style MIL
- CAMELS
  - PyLemmings style MIL for CAM Binding site prediction
- MILIAMP
  - PyLemmings for Amyloid Prediction
- Implementation of Learning with Rejection
- Feature selection and Mapping
  - CAFÉ-Map: Context Aware Feature Mapping for mining high dimensional biomedical data by Minhas, Asif and Arif (uses coordinate descent)

What's Next?

- How can we approximate kernels to do nonlinear classification?

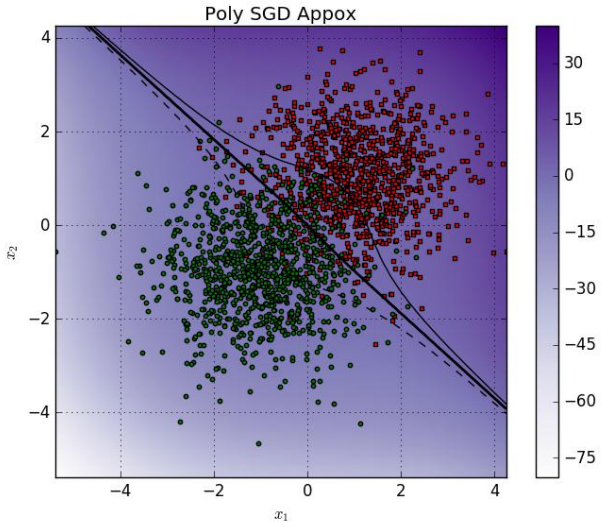
Moore-Aronszajn Theorem

- Suppose K is a symmetric, positive definite kernel on a set X. Then there is a unique Hilbert space of functions on X for which K is a reproducing kernel, i.e., $K(\boldsymbol{a}, \boldsymbol{b}) = \langle \varphi(a), \varphi(b) \rangle$
  - Can be used to do "kernel untricking", i.e., get the feature representation that reconstructs the kernel
- Example:
  - Polynomical Kernel Features based on multinomial expansion (sklearn.preprocessing.PolynomialFeatures)
    - The number of features in the polynomial expansion of degree p for an input feature dimension d is: $D = \binom{p+d}{p}$

$$\varphi(x) = \langle x_n^2, \ldots, x_1^2, \sqrt{2}x_n x_{n-1}, \ldots, \sqrt{2}x_n x_1, \sqrt{2}x_{n-1}x_{n-2}, \ldots, \sqrt{2}x_{n-1}x_1, \ldots, \sqrt{2}x_2 x_1, \sqrt{2c}x_n, \ldots, \sqrt{2c}x_1, c \rangle$$

Polynomial Features

- Can construct nonlinear boundaries by transforming the input features to a polynomial feature space and then using fast optimization algorithms
- Thought question
  - Using SGD, we can solve a linear SVM very fast
  - When will it be useful to first transform the data into polynomial features and then use SGD instead of using a nonlinear SVM (say, one based on SMO style methods)?
    - Useful for when number of training examples is large and the number of dimensions is small or a regularized solution is needed (small “C”)



Randomized Kernel Approximations

- Random Features for Large Scale Kernel Machines by Rahimi and Recht
  - Obtains a D-Dimensional approximation φ of a shift-invariant kernel $k(\boldsymbol{a}, \boldsymbol{b}) = k(\boldsymbol{a} - \boldsymbol{b}) \approx \boldsymbol{\varphi}(\boldsymbol{a})^T \boldsymbol{\varphi}(\boldsymbol{b})$
    - RBF Kernel $k(\boldsymbol{a}, \boldsymbol{b}) = k(\boldsymbol{a} - \boldsymbol{b}) = \exp(-\gamma \|\boldsymbol{a} - \boldsymbol{b}\|^2)$
  - Based on the Bochner's theorem: The Fourier transform $p(\boldsymbol{\omega})$ of a kernel $k(\boldsymbol{a} - \boldsymbol{b})$ can be written as follows if ω is drawn from p with $\psi_{\boldsymbol{\omega}}(\boldsymbol{x}) = e^{j\boldsymbol{\omega}^T \boldsymbol{x}}$

$$k(\boldsymbol{a} - \boldsymbol{b}) = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^T(\boldsymbol{a}-\boldsymbol{b})} d\boldsymbol{\omega} = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^T \boldsymbol{a}} e^{j\boldsymbol{\omega}^T \boldsymbol{b}} d\boldsymbol{\omega} = E[\psi_{\boldsymbol{\omega}}(\boldsymbol{a}) \psi_{\boldsymbol{\omega}}(\boldsymbol{b})^*]$$

  - We can approximate this averaging using a monte-carlo approximation by randomly drawing $\boldsymbol{\omega}$ from $p(\boldsymbol{\omega})$ with $z_{\boldsymbol{\omega}}(\boldsymbol{x}) = e^{j\boldsymbol{\omega}^T \boldsymbol{a}} = \cos(\boldsymbol{\omega}^T \boldsymbol{a}) + j\sin(\boldsymbol{\omega}^T \boldsymbol{a}) \equiv \left[\cos(\boldsymbol{\omega}^T \boldsymbol{a}) \quad \sin(\boldsymbol{\omega}^T \boldsymbol{a})\right]^T$ or $z_{\boldsymbol{\omega}}(\boldsymbol{x}) = \sqrt{2}\cos(\boldsymbol{\omega}^T \boldsymbol{a} + c)$

$$k(\boldsymbol{a} - \boldsymbol{b}) = \frac{1}{D} \sum_{i=1}^{D} z_{\omega_i}(a) z_{\omega_i}(b)$$

Randomized Kernel Approximations

- Thus, the features can be written as:

$$\tilde{z}(x) := \sqrt{\frac{2}{D}} \begin{bmatrix} \sin(\omega_1^\mathsf{T} x) \\ \cos(\omega_1^\mathsf{T} x) \\ \vdots \\ \sin(\omega_{D/2}^\mathsf{T} x) \\ \cos(\omega_{D/2}^\mathsf{T} x) \end{bmatrix}, \ \omega_i \overset{iid}{\sim} P(\omega)$$

and another of the form

$$\check{z}(x) := \sqrt{\frac{2}{D}} \begin{bmatrix} \cos(\omega_1^\mathsf{T} x + b_1) \\ \vdots \\ \cos(\omega_D^\mathsf{T} x + b_D) \end{bmatrix}, \quad \begin{matrix} \omega_i \overset{iid}{\sim} P(\omega) \\ b_i \overset{iid}{\sim} \mathrm{Unif}_{[0,2\pi]} \end{matrix}$$

- We know that, for the Fourier Transform of a Gaussian is a Gaussian
  - Thus the $\boldsymbol{\omega}$ can be drawn from a Normal Distribution
- In general

Kernel Name	$k(\Delta)$	$p(\omega)$
Gaussian	$e^{-\frac{\|\Delta\|_2^2}{2}}$	$(2\pi)^{-\frac{D}{2}} e^{-\frac{\|\omega\|_2^2}{2}}$
Laplacian	$e^{-\|\Delta\|_1}$	$\prod_d \frac{1}{\pi(1+\omega_d^2)}$
Cauchy	$\prod_d \frac{2}{1+\Delta_d^2}$	$e^{-\|\Delta\|_1}$

- The scaling parameter can be viewed as a simple scaling of the input data as

$$\exp(-\gamma\|\boldsymbol{a} - \boldsymbol{b}\|^2) = \exp(-\|\sqrt{\gamma}\boldsymbol{a} - \sqrt{\gamma}\boldsymbol{b}\|^2)$$

Algorithm 1 Random Fourier Features.

Require: A positive definite shift-invariant kernel $k(\mathbf{x}, \mathbf{y}) = k(\mathbf{x} - \mathbf{y})$.
Ensure: A randomized feature map $\mathbf{z}(\mathbf{x}) : \mathcal{R}^d \rightarrow \mathcal{R}^{2D}$ so that $\mathbf{z}(\mathbf{x})'\mathbf{z}(\mathbf{y}) \approx k(\mathbf{x} - \mathbf{y})$.

Compute the Fourier transform p of the kernel k: $p(\omega) = \frac{1}{2\pi} \int e^{-j\omega'\Delta} k(\Delta)\, d\Delta$.

Draw D iid samples $\omega_1, \cdots, \omega_D \in \mathcal{R}^d$ from p.

Let $\mathbf{z}(\mathbf{x}) \equiv \sqrt{\frac{1}{D}} \left[\cos(\omega_1'\mathbf{x}) \ \cdots \ \cos(\omega_D'\mathbf{x}) \ \sin(\omega_1'\mathbf{x}) \ \cdots \ \sin(\omega_D'\mathbf{x}) \right]'$.

Dataset	Fourier+LS	Binning+LS	CVM	Exact SVM
CPU regression 6500 instances 21 dims	3.6% 20 secs $D = 300$	5.3% 3 mins $P = 350$	5.5% 51 secs	11% 31 secs ASVM
Census regression 18,000 instances 119 dims	5% 36 secs $D = 500$	7.5% 19 mins $P = 30$	8.8% 7.5 mins	9% 13 mins SVMTorch
Adult classification 32,000 instances 123 dims	14.9% 9 secs $D = 500$	15.3% 1.5 mins $P = 30$	14.8% 73 mins	15.1% 7 mins $\text{SVM}^{\text{light}}$
Forest Cover classification 522,000 instances 54 dims	11.6% 71 mins $D = 5000$	2.2% 25 mins $P = 50$	2.3% 7.5 hrs	2.2% 44 hrs libSVM
KDDCUP99 (see footnote) classification 4,900,000 instances 127 dims	7.3% 1.5 min $D = 50$	7.3% 35 mins $P = 10$	6.2% (18%) 1.4 secs (20 secs)	8.3% < 1 s SVM+sampling

Approximation Error

- Given a shift-invariant kernel $K(a, b) = K(a - b)$, for $a, b \in \mathbb{R}^d$ the kernel can be approximated to within ϵ with only $D = O\left(\frac{d}{\epsilon^2} log \frac{1}{\epsilon^2}\right)$ and excellent classification and regression performance can be achieved with smaller D
- Downstream Error (Error in classification/regression performance)
  - Scales with $O\left(\frac{1}{\sqrt{D}}\right)$
    - **"On the Error of Random Fourier Features" by Sutherland and Schneider**

Implementation

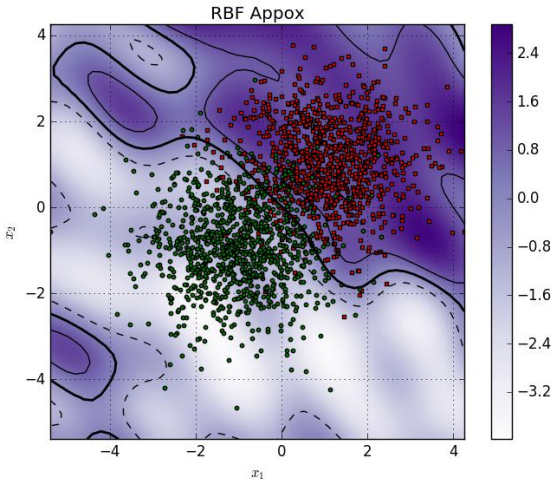
- from sklearn.kernel_approximation import RBFSampler
  - Implements the cosine only
- Code in Notes
  - Implements the sin-cos projection
    - Better than cosine only
  - Also implements the Orthogonal Random Features
    - Better than sin-cos
    - Orthogonal Random Features by Felix Xinnan Yu et al. (2016)
      - https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Orthogonal-Random-Features
      - **Derived from: Orthogonal Random Features https://arxiv.org/abs/1610.09072 , Implementation: https://github.com/NICTA/revrand**
- How can you test?

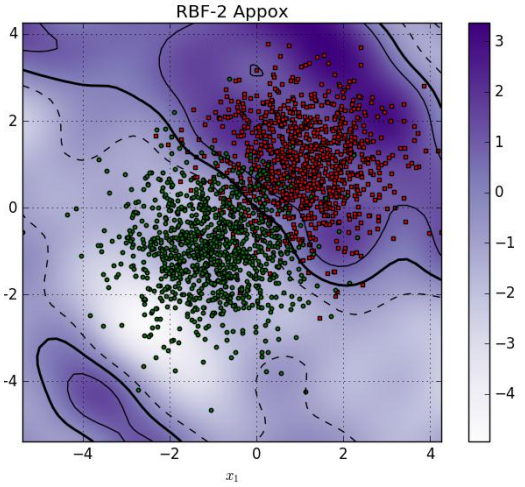
$$\tilde{z}(x) := \sqrt{\frac{2}{D}} \begin{bmatrix} \sin(\omega_1^\mathsf{T} x) \\ \cos(\omega_1^\mathsf{T} x) \\ \vdots \\ \sin(\omega_{D/2}^\mathsf{T} x) \\ \cos(\omega_{D/2}^\mathsf{T} x) \end{bmatrix}, \ \omega_i \overset{iid}{\sim} P(\omega)$$

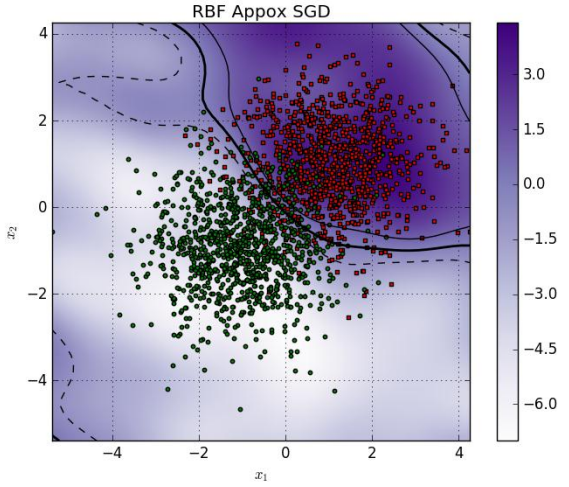
and another of the form

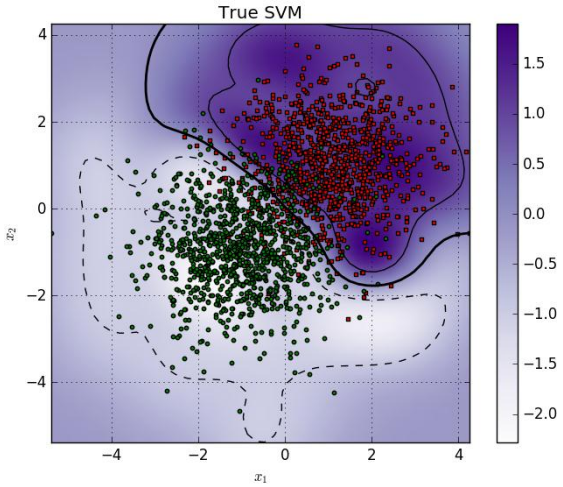
$$\check{z}(x) := \sqrt{\frac{2}{D}} \begin{bmatrix} \cos(\omega_1^\mathsf{T} x + b_1) \\ \vdots \\ \cos(\omega_D^\mathsf{T} x + b_D) \end{bmatrix}, \quad \begin{matrix} \omega_i \overset{iid}{\sim} P(\omega) \\ b_i \overset{iid}{\sim} \mathrm{Unif}_{[0,2\pi]} \end{matrix}$$

Implementation



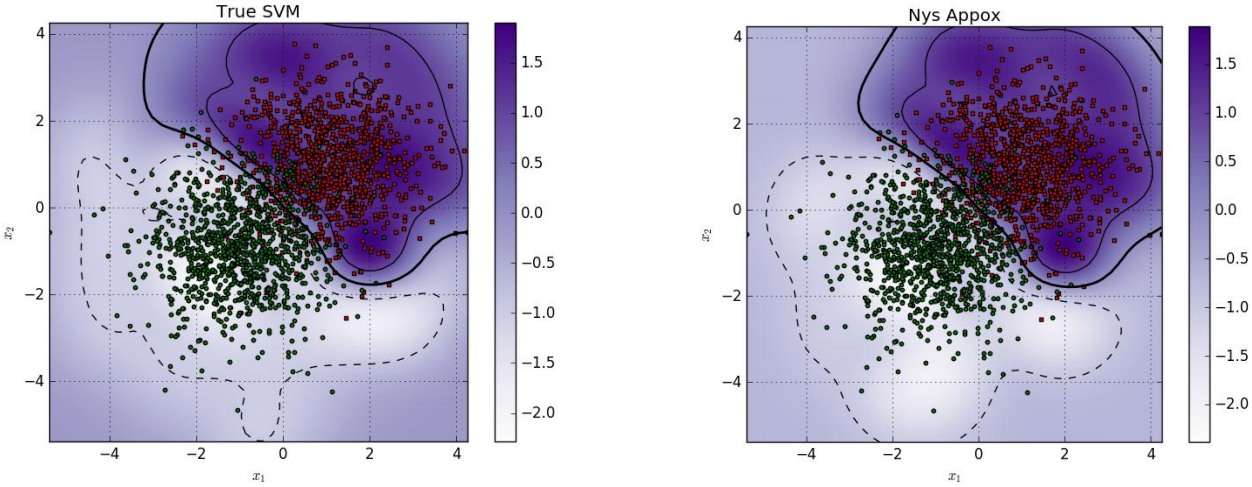






Other Methods

- Nystroem Approximation
  - Better approximation than random features
- FASTFood: Le, Quoc Viet, Tamas Sarlos, and Alexander Johannes Smola. "Fastfood: Approximate Kernel Expansions in Loglinear Time." *arXiv:1408.3060 [cs, Stat]*, August 13, 2014. http://arxiv.org/abs/1408.3060.
- Look at scikit-learn implementations
  - http://scikit-learn.org/stable/modules/classes.html#module-sklearn.kernel_approximation



Extensions based on Random Approximations

- Extreme Learning Machines
  - A least-squares SVM with random features
    - Closed form solution, extremely fast!

$$\min_{\boldsymbol{\beta}\in\mathbf{R}^{L\times m}} \quad \frac{1}{2}\|\boldsymbol{\beta}\|^2 + \frac{C}{2}\sum_{i=1}^{N}\|\mathbf{e}_i\|^2$$

$$\text{s.t.} \quad \mathbf{h}(\mathbf{x}_i)\boldsymbol{\beta} = \mathbf{t}_i^T - \mathbf{e}_i^T, \quad i = 1, \ldots, N.$$

$$f_L(\mathbf{x}) = \sum_{i=1}^{L}\boldsymbol{\beta}_i h_i(\mathbf{x}) = \mathbf{h}(\mathbf{x})\boldsymbol{\beta}$$

$$h_i(\mathbf{x}) = G(\mathbf{a}_i, b_i, \mathbf{x}), \quad \mathbf{a}_i \in \mathbf{R}^d, b_i \in R$$

OR $$\min_{\boldsymbol{\beta}\in\mathbf{R}^{L\times m}} \quad L_{\text{ELM}} = \frac{1}{2}\|\boldsymbol{\beta}\|^2 + \frac{C}{2}\|\mathbf{T} - \mathbf{H}\boldsymbol{\beta}\|^2.$$

Thus, $$\boldsymbol{\beta}^* = \left(\mathbf{H}^T\mathbf{H} + \frac{\mathbf{I}}{C}\right)^{-1}\mathbf{H}^T\mathbf{T},$$

OR (for N<L) $$\boldsymbol{\beta}^* = \mathbf{H}^T\boldsymbol{\alpha}^* = \mathbf{H}^T\left(\mathbf{H}\mathbf{H}^T + \frac{\mathbf{I}}{C}\right)^{-1}\mathbf{T},$$

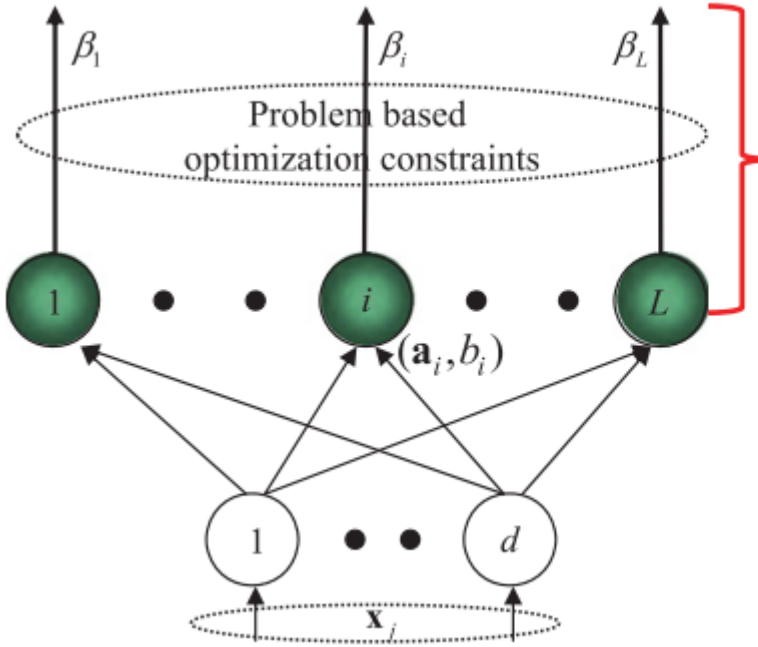
Commonly used mapping functions in ELM.

Sigmoid function	$G(\mathbf{a}, b, \mathbf{x}) = \frac{1}{1+\exp(-(\mathbf{a}\cdot\mathbf{x}+b))}$
Hyperbolic tangent function	$G(\mathbf{a}, b, \mathbf{x}) = \frac{1-\exp(-(\mathbf{a}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{a}\cdot x+b))}$
Gaussian function	$G(\mathbf{a}, b, \mathbf{x}) = \exp(-b\|\mathbf{x}-\mathbf{a}\|)$
Multiquadric function	$G(\mathbf{a}, b, \mathbf{x}) = (\|\mathbf{x}-\mathbf{a}\| + b^2)^{1/2}$
Hard limit function	$G(\mathbf{a}, b, \mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{a}\cdot\mathbf{x}+b \le 0 \\ 0, & \text{otherwise} \end{cases}$
Cosine function/Fourier basis	$G(\mathbf{a}, b, \mathbf{x}) = \cos(\mathbf{a}\cdot\mathbf{x}+b)$

Parameters **a** and b are chosen randomly

ELM as a Single Layer Neural Network

- ELM can be viewed as a single hidden layer neural network such that the weights of the hidden layer are initialized randomly – has universal approximation ability
- However, the idea is not new but is interesting! Many Python implementations available!



Feature learning
Clustering
Regression
Classification

L Random Hidden Neurons (which need not be algebraic sum based) or other ELM feature mappings. Different type of output functions could be used in different neurons:

$$h_i(\mathbf{x}) = G_i(\mathbf{a}_i, b_i, \mathbf{x})$$

d Input Nodes

Trends in extreme learning machines: A review by Huang et al. (2015)
Extreme Learning Machine for Multilayer Perceptron: Interesting extension to multilayer networks

Towards Infinitely Wide Neural Networks

- **"Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning"** by Recht and Rahimi, 2009
- Originates from the concept of random projections
  - **"The Unreasonable Effectiveness of Random Orthogonal Embeddings", 2017**

Algorithm 1 The Weighted Sum of Random Kitchen Sinks fitting procedure.

Input: A dataset $\{x_i, y_i\}_{i=1\ldots m}$ of m points, a bounded feature function $|\phi(x; w)| \leq 1$, an integer K, a scalar C, and a probability distribution $p(w)$ on the parameters of ϕ.

Output: A function $\hat{f}(x) = \sum_{k=1}^{K} \phi(x; w_k)\alpha_k$.

Draw $w_1, \ldots, w_K$ iid from p.

Featurize the input: $z_i \leftarrow [\phi(x_i; w_1), \ldots, \phi(x_i; w_K)]^\top$.

With w fixed, solve the empirical risk minimization problem

$$\underset{\alpha \in \mathbb{R}^K}{\text{minimize}} \quad \frac{1}{m} \sum_{i=1}^{m} c\left(\alpha^\top z_i,\ y_i\right) \tag{3}$$

$$\text{s.t.} \quad \|\alpha\|_\infty \leq C/K. \tag{4}$$

Doubly Stochastic Gradient Learning

- "Scalable Kernel Methods via Doubly Stochastic Gradients" by Dai et al. 2014
  - *"The general perception is that kernel methods are not scalable, and neural nets are the methods of choice for nonlinear learning problems. Or have we simply not tried hard enough for kernel methods?"*
- Based on kernel approximation
- The number of approximation elements is not fixed a priori

Doubly Stochastic Gradient Learning

- Random Feature Generation and Stochastic Gradient Combined into one algorithm

$$\underset{f \in \mathcal{H}}{\operatorname{argmin}} \quad R(f) := \mathbb{E}_{(x,y)}[l(f(x), y)] + \frac{\nu}{2} \|f\|_{\mathcal{H}}^2$$

Algorithm 1: $\{\alpha_i\}_{i=1}^t = \textbf{Train}(\mathbb{P}(x, y))$

Require: $\mathbb{P}(\omega)$, $\phi_\omega(x)$, $l(f(x), y)$, ν.

1: **for** $i = 1, \ldots, t$ **do**
2: Sample $(x_i, y_i) \sim \mathbb{P}(x, y)$.
3: Sample $\omega_i \sim \mathbb{P}(\omega)$ with seed i.
4: $f(x_i) = \textbf{Predict}(x_i, \{\alpha_j\}_{j=1}^{i-1})$.
5: $\alpha_i = -\gamma_i l'(f(x_i), y_i)\phi_{\omega_i}(x_i)$.
6: $\alpha_j = (1 - \gamma_i \nu)\alpha_j$ for $j = 1, \ldots, i-1$.
7: **end for**

Algorithm 2: $f(x) = \textbf{Predict}(x, \{\alpha_i\}_{i=1}^t)$

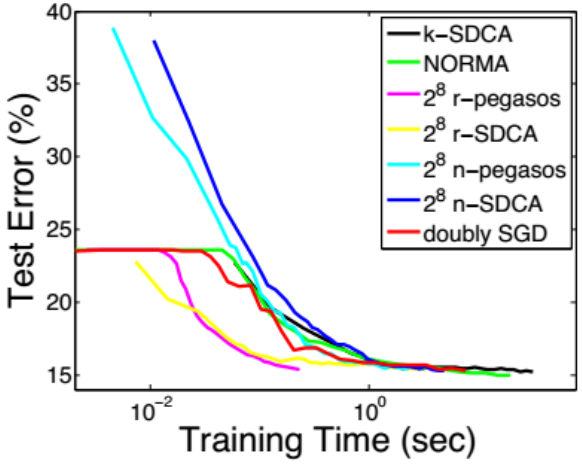
Require: $\mathbb{P}(\omega)$, $\phi_\omega(x)$.

1: Set $f(x) = 0$.
2: **for** $i = 1, \ldots, t$ **do**
3: Sample $\omega_i \sim \mathbb{P}(\omega)$ with seed i.
4: $f(x) = f(x) + \alpha_i \phi_{\omega_i}(x)$.
5: **end for**

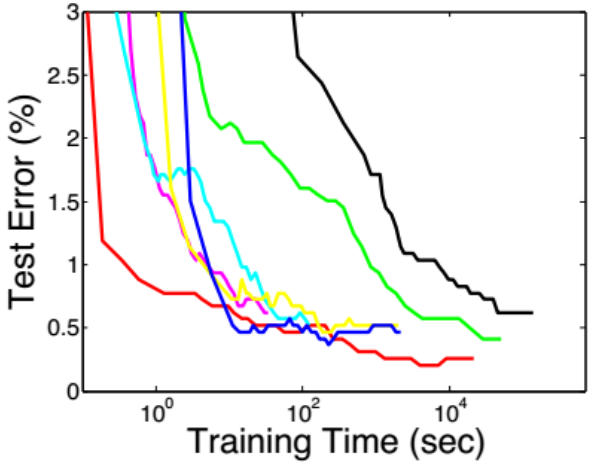
For the Hinge Loss Kernel SVM, Line 5 becomes

$$\alpha_i = \begin{cases} 0 & \text{if } y_i f(x_i) \geqslant 1 \\ \gamma_i y_i \phi_{\omega_i}(x_i) & \text{if } y_i f(x_i) < 1 \end{cases}$$

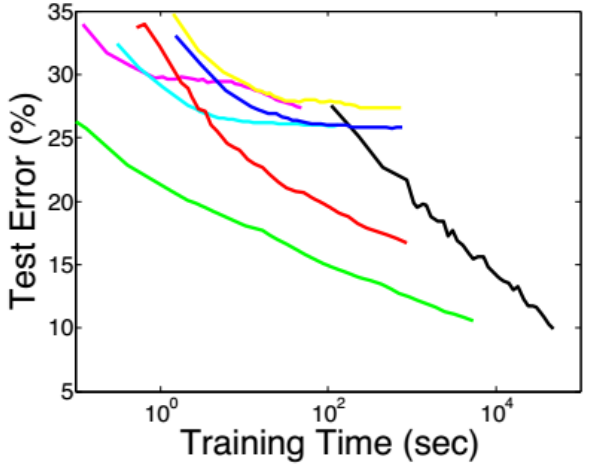
Comparison with other SVMs



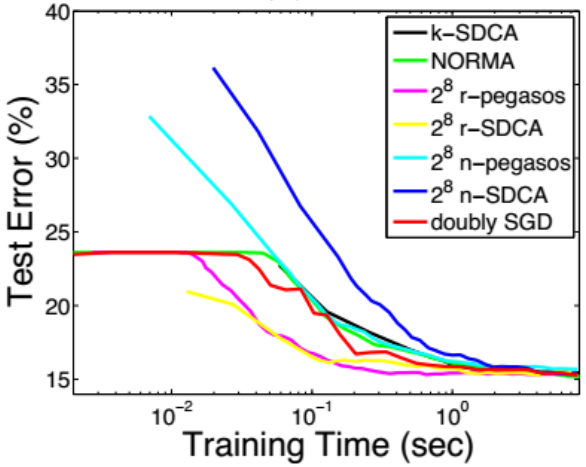
SC1: (1) Adult



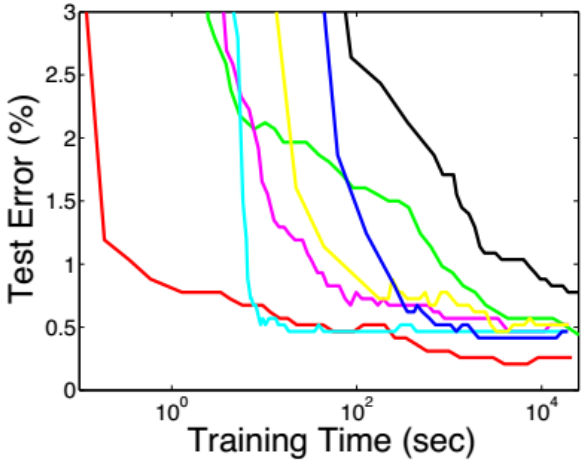
(2) MNIST 8M 8 vs. 6



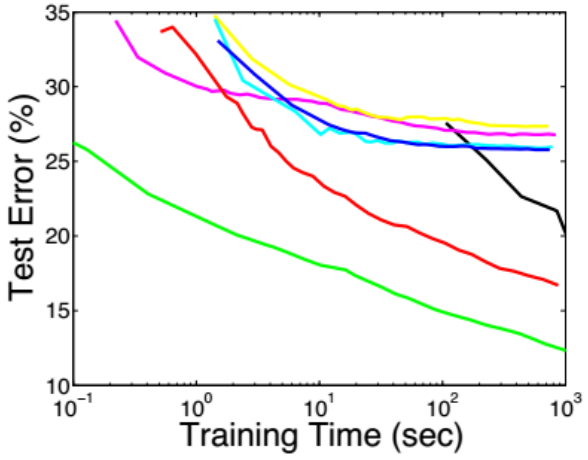
(3) Forest



SC2: (4) Adult



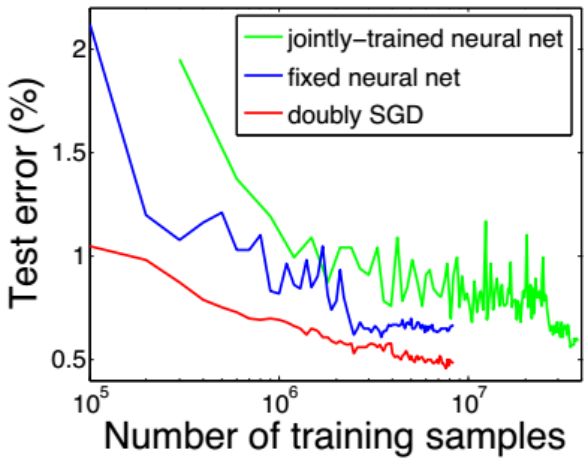
(5)MNIST 8M 8 vs. 6



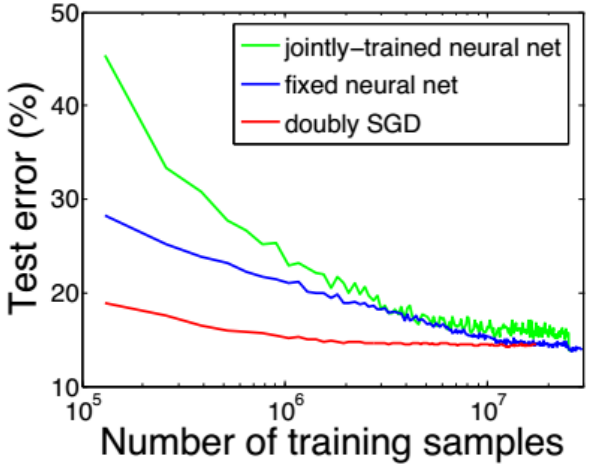
(6) Forest.

Figure 6: Comparison with other kernel SVM solvers on datasets (3) – (5) with two different stopping criteria.

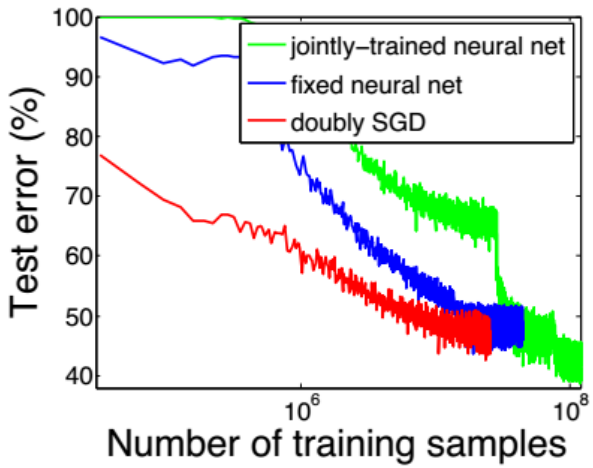
Comparison with Neural Nets



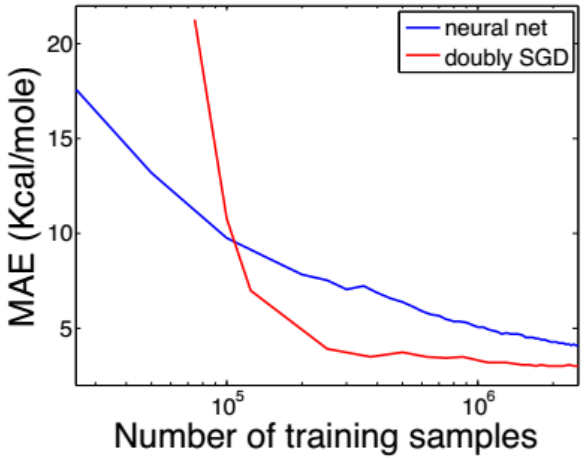
(1) MNIST 8M



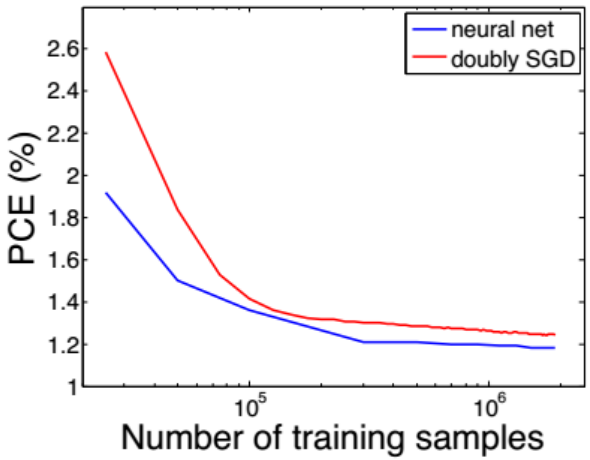
(2) CIFAR 10



(3) ImageNet



(4) QuantumMachine



(5) MolecularSpace.

Implementation

- The paper details implementation for a wide variety of kernels and a number of learning problems including SVM, Logistic Regression, Ridge Regression, Robust Regression, SVR, Quantile Regression, Novelty Detection, Density Estimation, Gaussian Processes, etc.

- Available: https://github.com/zixu1986/Doubly_Stochastic_Gradients
- Julia Implementation by Dr. Fayyaz Minhas

What are the random features doing?

- I think that the random features are making the SVM have data based basis instead of a priori basis (or kernel) which remains fixed!
- Extension
  - Triply Stochastic Sub-Gradient Learning
- Parallelization
  - Block processing: On each machine m=1…M, take D' different dimensions and randomly select from the data, and calculate α_i, then simply average the outputs for each machine

Semi Random Features (skip)

- "Deep Semi-Random Features for Nonlinear Function Approximation" by Kawaguchi et al., 2017
- "Can we have the best of both worlds? Can we develop a framework for big nonlinear problems which has the ability to adapt basis functions, has low computational and storage complexity, while at the same time retaining some of the theoretical properties of random features? Towards this goal, we propose semi-random features to explore the space of trade-off between flexibility, provability and efficiency in nonlinear function approximation. We show that semirandom features have a set of nice theoretical properties, like random features, while possessing a (deep) representation learning ability, like deep learning. More specifically:
  - Despite the nonconvex learning problem, semi-random feature model with one hidden layer has no bad local minimum;
  - Semi-random features can be composed into multi-layer architectures, and going deep in the architecture leads to more expressive model than going wide;
  - Semi-random features also lead to statistical stable function classes, where generalization bounds can be readily provided.
- They propose semi-random features: In experiments, we show that semi-random features can match the performance of neural networks by using slightly more units, and it outperforms random features by using significantly fewer units.
- The single hidden layer network is:
  - R is randomly drawn, $W^{(1)}$ and $W^{(2)}$ are optimized
  - $\sigma_s(z) = \begin{cases} z^s & z > 0 \\ 0 & else \end{cases}$, is a ReLU style function
- Code: http://github.com/zixu1986/semi-random

$$\hat{f}_n^s(x; w) = \left(\sigma_s(\mathbf{x}^\top \mathbf{R}) \odot (\mathbf{x}^\top \mathbf{W}^{(1)})\right) W^{(2)},$$

$$\mathbf{W}^{(1)} = (\mathbf{w}_1^{(1)}, \mathbf{w}_2^{(1)}, \ldots, \mathbf{w}_n^{(1)}) \in \mathbb{R}^{(d+1)\times n},$$

$$\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n) \in \mathbb{R}^{(d+1)\times n}, \text{and}$$

$$\mathbf{W}^{(2)} = (w_1^{(2)}, \ldots, w_n^{(2)})^\top \in \mathbb{R}^{n+1}.$$

Scaling up further (skip)

- "How to Scale Up Kernel Methods to Be As Good As Deep Neural Nets" by Lu et al., 2014
  - Multinomic logistic regression based on random kitchen sinks with block parallelization and stochastic gradient descent
  - Compared to DNN
  - Equivalent performance

Interesting Papers

- Faster Kernel Ridge Regression Using Sketching and Preconditioning
- Fast randomized kernel methods with statistical guarantees
- Deep Semi-Random Features for Nonlinear Function Approximation
- Steps toward deep kernel methods from infinite neural networks
- A comparative study on large scale kernelized support vector machines
- Stochastic Methods for l1-regularized Loss Minimization
- Distributed Coordinate Descent Method for Learning with Big Data
- The Tradeoffs of Large Scale Learning
- Large-Scale Support Vector Machines: Algorithms and Theory

Deep learning papers

- JMLR Dropout
- Failures of deep learning
- Skflow
- keras
- http://nuit-blanche.blogspot.com/2017/01/understanding-deep-learning-requires.html
- Deep Nets Don't Learn via Memorization
- https://medium.com/@phelixlau/deep-nets-dont-learn-via-memorization-6fd692dea63e

End of Lecture

I believe that learning has just started, because whatever we did before, it was some sort of a classical setting known to classical statistics as well. Now we come to the moment where we are trying to develop a new philosophy which goes beyond classical models.

- Vapnik

http://www.learningtheory.org/learning-has-just-started-an-interview-with-prof-vladimir-vapnik/