Implementing the Linear and Kernel PEGASOS SVM Algorithms for Binary Classification

Daniel N. Baker
Department of Computer Science
Johns Hopkins University
Baltimore, MD
dnb@cs.jhu.edu

Abstract

The PEGASOS (Primal Estimated subGrAdient SOlver for SVM)[1] algorithm is a highly-efficient stochastic gradient descent (SGD) algorithm for determining an optimal linear classifier for binary classification using ℓ_2 regularization.
We have implemented the PEGASOS classifier in C++14 and benchmarked its performance on standard datasets with varying parameters.
The linear classifier is extremely fast, while the kernel classifier is much less performant.
We supply linear, Radial Basis Function, Laplacian, Hyperbolic Tangent, Circular, Spherical, and Bessel kernel implementations. The source code and benchmarking data is available under the permissive Apache 2.0 license at `https:/github.com/dnbh/denseSVM/`.

1 Introduction

Support Vector Machines (**SVM**s) [6] are a powerful class of classification tools which characterize a decision boundary using a hyperplane with the largest margin between classes, with a convex but nondifferentiable penalty paid for points which are misclassified or classified with insufficient distance between them and the boundary. They have been for many years been state of the art for a variety of problems, although deep neural networks have, in certain applications, have started replacing them. Still, with well-understood theory and interpretability, they represent a scalable, performant solution with high accuracy.

With excellent classification capability and attractive asymptotic training and runtime memory and processing requirements,

Optimization across a training set can be an expensive process involving large amounts of linear algebra and potentially significant memory requirements. We have implemented a fast, memory-efficient trainer for support vector machines for both the linear and kernel cases. For particular applications, their memory requirements and performance could be further tuned. With excellect classification capability and attractive asymptotic training and runtime memory and processing requirements, PEGASOS provides practical stochastic optimization for large-scale classification problems.

2 Algorithm and Theory

The SVM problem involves minizing a loss function $\underset{\mathbf{w}}{\operatorname{argmin}} \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \frac{1}{m}\sum_{\mathbf{x},y\in\mathcal{S}} \ell(\mathbf{w};\mathbf{x},y)$, where $\ell(\mathbf{x};\mathbf{x},y) = \max\{0, 1 - y\langle\mathbf{w},\mathbf{x}\rangle\}$, or, in the case of a kernel, $\max\{0, 1 - yK(\mathbf{w},\mathbf{x})\}$. The subgradient of this problem $\nabla_t = \lambda\mathbf{w}_t - BOOL(y_{i_t}\langle\mathbf{w}_t,\mathbf{x}_{i_t}\rangle < 1)y_{i_t}x_{i_t}$, where $BOOL()$ is 1 if true and 0 if false. (I.E., a coercion from a boolean to an integer.) By randomly sampling from our data, we approximate the subgradient of the whole dataset with a fraction

of the cost. Our stochastic setting requires an expected $\mathcal{O}(1/\lambda\epsilon)$ number of iterations for convergence to within ϵ of the optimum value.

The PEGASOS algorithm additionally adds a rescaling at each iteration to a standard stochastic subgradient descent, as well as an optional projection to a smaller ball around 0. Earlier proofs required the projection step, but later proofs were able to demonstrate that this was unnecessary.

With sufficient samples, our training time to within ϵ of the approximation error is inversely dependent on sample size [2]. This makes the PEGASOS algorithm highly attractive for large-scale machine learning problems, at least within the linear setting. When kernels are applied, the number of updates at each iteration increases by a factor of m, which largely negates its superior performance. By enforcing sparsity, however, it may be possible to make this relatively simple solution within the primal performance and feasible.

3 Design and Implementation

The primary computational cost in SVM optimization is linear algebra and matrix operations. For optimal performance, we chose to take advantage of recent advances in both hardware and software. OpenBLAS [7] is an exceptionally efficient library for linear algebra which generically generates code using a combination of C and x86 assembly for a wide variety of architectures, taking advantage of widening SIMD vectors on the x86 platform and new Intel instructions, such as the instructions marketed for "deep learning" which are actually simply ideal for computing dot products. Benchmark suites (except for those performed by Intel, unsurprisingly) routinely demonstrate that MKL and OpenBLAS vastly outperform competitors and are nearly equivalent, with each library being slightly superior on various operations, depending on the size of the problems. These implementations are optimized for cache efficiency in addition to making full use of hardware-specific acceleration and handwritten assembly in its inner loops.
For the purpose of developing purely open-source software, we chose to use OpenBLAS for our BLAS implementation.

In addition to taking advantage of hardware advances, we gain additional performance by the use of expression templates through the Blaze library [5]. Expression templates use abstract C++ template metaprogramming to develop a form of lazy evaluation, in which an operation is performed only as needed. This allows chained matrix operations to be reduced into fewer matrix operations. Blaze optionally wraps BLAS for operations above a given size and provides its own implementations for cases where the overhead for these methods would exceed the additional efficiency. In addition, Blaze provides efficient, SIMD-accelerated operations which are not ideally-suited for BLAS, such as smaller vectors or setting each element in a vector or matrix to a scalar. As Blaze strongly outperforms its competitors (`https://bitbucket.org/blaze-lib/blaze/wiki/Benchmarks`), such as Eigen [8] and Armadillo [9], we have chosen to use Blaze for our linear algebra library.

For very large-scale problems, the reduced precision due to using 32-bit floats over the standard 64-bit IEEE 784 (1985) double is well worth the reduced memory traffic and the ability to perform twice as many foating-point operations per cycle with SIMD. For this reason, our code has been developed to run with either 32-bit or 64-bit floating-point numbers.

The original PEGASOS paper[1] discusses the incorporation of a bias term. The algorithm does not have one in its primary instantiation. They offer several solutions for adding a bias term. First, one can simply increase the dimensionality of the dataset by 1 and assign all points an arbitrary value (1 is suggested). This has the advantage of not requiring any modification of the algorithm, though it solves a different optimization problem, as it regularizes the bias term with the weights. Their last suggestion is to vary the bias term and optimize PEGASOS in an inner loop, selecting the best-performing bias term. This method still provides the same asymptotic complexity at linear increase in runtime cost.

We have chosen primarily to use OpenMP for parallelization due to its natural appropriateness for the problem.

We currently offer a variety of kernels, including binaries for Radial Basis Function, Circular, Spherical, Laplacian, Hyperbolic Tangent, and Bessel functions.

Table 1: Hyperparameter Optimization for Linear and RBF Kernels

Dataset	Kernel	Test Error	Train Error	Optimal Parameters	Runtime	Dimensions	Training Samples	Test Samples
Breast Cancer	Linear Kernel	1.056%	3.75%	$\lambda = 0.05, b = 32$	0.02s	10	400	285
Breast Cancer	RBF Kernel	1.056%	3.75%	$\lambda = 0.0001, b = 32, \gamma = 0.5$	4.07s	10	400	285
a8a	Linear Kernel	14.93%	15.36%	$\lambda = 0.0001, b = 32$	0.57s	123	22696	9856
a8a	RBF Kernel	14.79%	14.78%	$\lambda = 0.025, b = 128, \gamma = 0.1$	148.58s	123	22696	9856

4 Performance

We used standard benchmark datasets from LibSVM [3] to assess the performance of our software. We chose the breast cancer dataset and the a8a text classification problem. In tuning hyperparameters, we use a python script to explore the parameter space, found in `scripts/hyperpar.py`. For the kernel implementation of PEGASOS, this can become quite slow, as many (and up to all) samples become support vectors. To minimize the time for these experiments, we wrapped our function calls in multiprocessing. For the breast cancer dataset, which did not have separate training and test datasets, we split the data into 400-sample training and 284-sample sets. The a8a text classification dataset came with both training and test.

These algorithms don't parallelize well for low-dimensional datasets due to the dependence , but for dense, large matrices there can be an improvement, as the matrix operations are more parallelizable.

The linear problem fails to parallelize well when datasets are small because potentially frequent updates to the weight matrix prevent parallel updating and the vectors between which inner products are calculated are not sufficiently large for parallel BLAS operations.
The kernel problem, due to its increased computational expense at each iteration, does improve with parallelization.

Significant advantages of the PEGASOS kernel method over static SVM optimization is that memory requirements are $\mathcal{O}(dn)$ rather than $\mathcal{O}(n(n+d))$. In the setting where we have many more training points than dimensions (which would certainly be ideal for avoiding overfitting), this asymptotically linear memory is vastly superior.

5 Conclusions & Future Work

The PEGASOS stochastic subgradient descent algorithm is extremely efficient for linear applications but suffers heavily from kernelization. Our classification accuracy on the Breast Cancer dataset is exceptional (1%) for both linear and kernel SVMs, while text classification remains difficult. This cost arises due to our inability to collapse multiple support vectors into a single vector in our feature space. This performance cost can be minimized by using a budget, which limits the number of support vectors [4,11,12]. Under the assumption that there is some dependence or similarity between points in a given class, it seems likely that there could exist classifiers with similar accuracy using fewer support vectors.

Our performance in the sparse matrix setting could be improved by using a specialized sparse matrix (CompressedMatrix) from Blaze for sparse data, such as text classification. While these are no longer able to use BLAS and vectorization, they can outperform dense matrix operations by performing many fewer arithmetic operations.

The algorithm can also be expanded for multiclass classification with relatively few modifications. One takes an argmax over the input weight matrices rather than simply the sign of single weight matrix. We could also adapt the algorithm to use other forms of regularization, such as ℓ_1 or the four additional loss functions detailed, with subgradients, in [1].

Regarding our particular implementation, further care for cache efficiency could improve the performance of the kernel method. The primary cost (85%+ of runtime) is calculating $\sum_{v \in SupportVectors} y_v K(y, x)$ for each x in a mini batch. By switching the outer loop from samples from a minibatch to support vectors, we could potentially improve our cache efficiency by only loading each support vector once per iteration.

For the linear case, one could entirely dispense with bringing the full dataset nto memory and simply process a stream under the assumption that each datapoint is randomly selected from our target distribution. This is particularly well-suited to big-data applications.

Alternatively, we restricted our random datapoints to rows which are close or adjacent to each other, we could dramatically reduce the number of cache misses required for a batch. However, the number of samples would have to be sufficiently large that a randomly selected but colocalized subset would be an appropriate approximation of uniformly random sampling. Additionally, if adjacent samples were used, more heavily optimized linear algebra functions could be applied and SIMD intrinsics for specialized functions (including square root, tanh, and exp) could be applied.

Further extensions could be using the algorithm for regression problems [13], or perhaps an output layer of a neural network.

[1] Shalev-Shwartz, S., Singer, Y. & Srebro, N. (2007) Pegasos: Primal estimated sub-gradient solver for svm. {Proceedings of the 24th International Conference on Machine learning. ACM.}

[2] Shalev-Shwartz, S. & Srebro N. (2008) SVM optimization: inverse dependence on training set size. {Proceedings of the 25th International Conference on Machine learning. ACM.}

[3] Fan, R., & Chih-Jen Lin. (2011) LIBSVM data: Classification, regression and multi-label. {https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/}

[4] Wang, Z., Crammer K., & Vucetic S. (2010) Multi-Class Pegasos on a Budget {Proceedings of the 27th International Conference on Machine Learning. ACM.}

[5] Iglberger, K., Hager G., Treibig J., & Rde U. (2012) Expression Templates Revisited: A Performance Analysis of Current Methodologies. {SIAM Journal on Scientific Computing, 34(2): C42–C69, 2012}

[6] Cortes, C., & Vapnik C. (1995) Support-vector networks {Machine Learning, Vol. 20, Issue 3, pp. 273297, Sep. 1995}

[7] Qian W., Xianyi, Z., Yunquan, Z., & Qing Yi, Q. (2013) AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. {International Conference on High Performance Computing, Networking, Storage and Analysis.}

[8] Gaël Guennebaud and Benoît Jacob and others (2010) Eigen v3. {https://eigen.tuxfamily.org}

[9] Sanderson C., & Ryan Curtin R. (2016) Armadillo: a template-based C++ library for linear algebra. {Journal of Open Source Software, Vol. 1, pp. 26, 2016.}

[10] Iglberger, K., and others. (2015) Blaze Benchmarks {https://bitbucket.org/blaze-lib/blaze/wiki/Benchmarks}

[11] Callavanti, G., Cesa-Bianchi, N., & Gentile, C. (2007) Tracking the best hyperplane with a simple budget Perceptron {Machine Learning. Volume 69, Issue 2, pp 143167}

[12] Dekel, Ofer., Shalev-Schwartz S., Singer, Y. (2008) The Forgetron: A Kernel-Based Perceptron on a Budget. {SIAM Journal of Computing, Vol. 37, No. 5, pp. 1342-1372}

[13] Drucker, H., Burges, C., Kaufman, L., Smola, A., Vapnik, V. (1997) Support Vector Regression Machines {Advances in Neural Information Processing Systems 9, NIPS 1996, 155161, MIT Press.}