Solving Large Scale Linear SVM with Distributed Block Minimization

Dmitry Pechyony, Libin Shen and Rosie Jones
Akamai Technologies
{dpechyon,lishen,rejones}@akamai.com

Abstract

Over recent years we have seen the appearance of huge datasets that do not fit into memory and do not even fit on the hard disk of a single computer. Moreover, even when processed on a cluster of machines, data are usually stored in a distributed way. The transfer of significant subsets of such datasets from one node to another is very slow. We present a new algorithm for training linear Support Vector Machines over such large datasets. Our algorithm assumes that the dataset is partitioned over several nodes on a cluster and performs a distributed block minimization along with the subsequent line search. The communication complexity of our algorithm is *independent* of the number of training examples. With our Map-Reduce/Hadoop implementation of this algorithm the accurate training of SVM over the datasets of tens of millions of examples takes less than 11 minutes.

1 Introduction

The Support Vector Machine (SVM) [4] is one of the most robust machine learning algorithms developed to date, and one of the most commonly used ones. Over the last 15 years algorithms for training SVMs have been evolving from being scalable to thousands of examples to being scalable to millions. Nevertheless, current state-of-the-art sequential linear SVM solvers are relatively slow ([3, 19]) if they are trained over tens of millions of high-dimensional examples. The scalability of the state-of-the-art sequential solvers of nonlinear SVM is even worse ([15]).

The scalability of sequential solvers can be improved by parallelization. In this paper we present a new distributed linear SVM solver, referred to as DBM (distributed block minimization). DBM is a distributed version of LIBLINEAR solver of linear SVM [7]. DBM is also influenced by the ideas from LIBLINEAR-CDBLOCK [19] extension of LIBLINEAR. DBM operates on a dataset that is distributed over k nodes on a cluster. During the optimization process DBM maintains the global primal solution $\mathbf{w}$ at the *master node* and k *slave nodes*. Also each slave node maintains a subset of the global dual solution $\boldsymbol{\alpha}$ that corresponds to the examples that are stored in that node. At each iteration the current value of $\mathbf{w}$ is used to compute a new $\boldsymbol{\alpha}$ in a distributed way. The new $\boldsymbol{\alpha}$ is then used to compute an updated $\mathbf{w}$. The former step is done by a simplified version of LIBLINEAR-CDBLOCK, the latter one is done by using a line search.

DBM is designed to minimize the latency of training and communication. We do not require that the data in each slave node to fit in memory so that the number k of nodes used can be independent of the number of examples. The communication complexity of DBM is $O(k \times \text{number of dimensions})$ and with the fixed k is *independent* of the number of examples. We implemented DBM as an extension of LIBLINEAR-CDBLOCK on Hadoop. Our experiments with two large datasets of tens of millions of examples show a significant speedup over the sequential LIBLINEAR-CDBLOCK solver. In particular on the dataset of 79M examples it took LIBLINEAR-CDBLOCK 3 hours to achieve the same test accuracy as the one achieved by DBM within 11 minutes.

Related Work. The idea of training SVM on the subsets of training set goes back to the chunking algorithm of [14]. Unlike DBM, in chunking algorithm the blocks are overlapping. Many previous attempts in distributed SVM training were focused on parallelizing particular steps of sequential solvers [6, 18, 20, 2, 17, 16]. With the exception of [17, 16], those papers focused on parallelizing the solvers of nonlinear SVMs. The approach of [17] needs to keep the entire dataset in the memory of the master node. This limits the size of the training set to one which will fit into memory. Our

approach does not have this limitation and in fact does not require the master node to keep any examples in memory. The algorithm of [16] has the same property. But, as shown by [16], the sequential version of the algorithm of [16] is slower than LIBLINEAR, that is a sequential version of DBM. We plan to do a detailed comparison of DBM with the algorithm of [16] in the future.

Previously published fully distributed algorithms [8, 9] for training SVM are significantly different from ours. Unfortunately both [8] and [9] do not provide experiments with large datasets.

Recently the field of online algorithms has become very active. [5, 11, 13, 21] perform parallelization over examples in the primal space, while DBM does this in the dual one. DBM is conceptually similar to SHOTGUN algorithm of [1] that performs parallelization over the features in the primal space. Since there is one-to-one correspondence between training examples and the coordinates of the dual solution, DBM can be viewed as doing parallelization over the features in the dual space.

2 Review of Sequential Block Minimization for solving SVM [19]

Let $\mathcal{T} = \{(\mathbf{x}'_i, y_i)\}_{i=1}^n, \mathbf{x}'_i \in \mathbb{R}^d, y_i \in \{+1, -1\}$ be a training set. A linear SVM generates a classifier $h(\mathbf{x}') = \mathbf{w}'^T\mathbf{x}' + b$, where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$. Let $\mathbf{w} = [\mathbf{w}'; b]$ and $\mathbf{x} = [\mathbf{x}'; 1]$. Then $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$. The vector $\mathbf{w}$ is obtained by solving $\min_{\mathbf{w}\in\mathbb{R}^d} f_P(\mathbf{w}) = \|\mathbf{w}\|_2^2/2 + C\sum_{i=1}^n \max(1 - y_i\mathbf{w}^T\mathbf{x}_i, 0)$. The corresponding dual optimization problem is

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^n} \; f_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^T Q\boldsymbol{\alpha}/2 - \mathbf{e}^T\boldsymbol{\alpha} \;\; , \text{ s.t. } \forall\, 1 \le i \le n,\; 0 \le \alpha_i \le C \;\; , \tag{1}$$

where $Q_{ij} = y_i y_j \mathbf{x}_i\mathbf{x}_j$ and $\mathbf{e} = [1, \ldots, 1]^T$. Given the dual solution $\boldsymbol{\alpha}$, the primal one is

$$\mathbf{w} = \sum_{i=1}^n y_i\alpha_i\mathbf{x}_i \;\; . \tag{2}$$

Let $\{B_i\}_{i=1}^k$ be a *fixed* partition of $\mathcal{T}$ and the corresponding dual variables $\boldsymbol{\alpha}$ into k blocks. These blocks are disjoint and $\bigcup_{i=1}^n B_i = \mathcal{T}$. We overload the notation and refer to B_i both as a set of indices and a set of corresponding training examples. Yu et al. [19] showed that (1) can be solved using *sequential block minimization* (SBM): at each iteration we consider a block B_i and solve (1) only for the variables in B_i. The remarkable fact, shown in [19], is that when solving (1) for the variables in B_i we do not need to keep in memory the variables from other blocks. We now describe this observation in more details, since our forthcoming distributed algorithm is heavily based on it.

Let $\boldsymbol{\alpha}^t \in \mathbb{R}^d$ be a solution of (1) after t iterations. Suppose that at the $(t+1)$-th iteration we are optimizing (1) for the variables in B_i, and $\mathbf{d}^i = \boldsymbol{\alpha}^{t+1} - \boldsymbol{\alpha}^t$. The direction $\mathbf{d}^i$ has nonzero values only in the coordinates from B_i and is found by solving

$$\min_{\mathbf{d}^i\in\mathbb{R}^d} \; (\boldsymbol{\alpha}^t + \mathbf{d}^i)^T Q(\boldsymbol{\alpha}^t + \mathbf{d}^i)/2 - \mathbf{e}^T(\boldsymbol{\alpha}^t + \mathbf{d}^i) \tag{3}$$

$$\text{subject to } \forall\, j \in B_i,\; 0 \le \alpha_j^t + d_j^i \le C \;\; , \forall\, j \notin B_i, d_j^i = 0 \;\; .$$

Let $\mathbf{d}_{B_i}$ be a vector of $|B_i|$ nonzero coordinates of $\mathbf{d}^i$ that correspond to the indices in B_i. The objective (3) is equivalent to

$$\mathbf{d}_{B_i}^T Q_{B_i,B_i}\mathbf{d}_{B_i}/2 + (\boldsymbol{\alpha}^t)^T Q_{:,B_i}\mathbf{d}_{B_i} - \mathbf{e}^T\mathbf{d}_{B_i} \;\; , \tag{4}$$

where Q_{B_i,B_i} is a submatrix of Q with all the indices in B_i and $Q_{:,B_i}$ is a submatrix of Q with the column indices being in B_i. It follows from (2) that for any $1 \le j \le n$, $\boldsymbol{\alpha}^T Q_{:,B_i} = y_j(\mathbf{w}^t)^T\mathbf{x}_j$, where $\mathbf{w}^t$ is a primal solution after t iterations. Let X_{B_i} be a $d \times |B_i|$ matrix. The j-th column of X_{B_i} is the j-th example in B_i, multiplied by its label. Then the second term of (4) is $\mathbf{w}^T X_{B_i}\mathbf{d}_{B_i}$ and we obtain that in order to solve (3) for the variables in B_i we need to solve

$$\min_{\mathbf{d}_{B_i}} \; \mathbf{d}_{B_i}^T Q_{B_i,B_i}\mathbf{d}_{B_i}/2 + (\mathbf{w}^t)^T X_{B_i}\mathbf{d}_{B_i} - \mathbf{e}^T\mathbf{d}_{B_i} \tag{5}$$

$$\text{subject to } \forall\, j \in B_i,\; 0 \le \alpha_j^t + d_j^i \le C \;\; .$$

To solve (5) we only need to keep in memory the examples from the block B_i and the d-dimensional vector $\mathbf{w}^t$. After solving (5) the vector $\mathbf{w}^t$ is updated as $\mathbf{w}^{t+1} = \mathbf{w}^t + \sum_{j\in B_i} d_j^i y_j\mathbf{x}_j$. In summary, at the $(t+1)$-th iteration SBM solves (5) for a *single* B_i and then updates $\mathbf{w}^t$ using the last formula.

3 Distributed Block Minimization

In distributed block minimization (DBM), at each iteration, we process *all* the blocks B_i, $1 \le i \le k$ simultaneously. This is done by solving (5) in parallel on k *slave nodes*. We denote by $\boldsymbol{\alpha}_i^t$ the local

Algorithm 1 DSVM-AVE - distributed block minimization with averaging

Input: The ith slave node has a block $\{B_i\}_{i=1}^k$; the master node has a feasible solution $\mathbf{w}^1$ of (1).

1: **for** $t = 1, 2, \ldots$ **do**
2: Send $\mathbf{w}^t$ from the master node to k slave nodes.
3: **for** $i = 1$ **to** k, in parallel **do**
4: At the ith slave node: solve (5) for B_i and obtain $\mathbf{d}^i$;
send $\Delta\mathbf{w}_i^t = \sum_{r\in B_i} d_r^i y_r \mathbf{x}_r$ to the master node;
set $\boldsymbol{\alpha}_i^{t+1} = \boldsymbol{\alpha}_i^t + 1/k \cdot \mathbf{d}^i$
5: **end for**
6: Set $\mathbf{w}^{t+1} = \mathbf{w}^t + \frac{1}{k}\sum_{i=1}^k \Delta\mathbf{w}_i^t$.
7: **end for**

Algorithm 2 DSVM-LS - distributed block minimization with the line search

The same as Algorithm 1 with Step 6 replaced by
At the master node: find $\lambda^* = \arg\min_{0\le\lambda\le 1} f_P(\mathbf{w}^t + \lambda\sum_{i=1}^k \Delta\mathbf{w}_i^t)$;
set $\mathbf{w}^{t+1} = \mathbf{w}^t + \lambda^* \sum_{i=1}^k \Delta\mathbf{w}_i^t$.

version of $\boldsymbol{\alpha}^t$ at the ith slave node. $\boldsymbol{\alpha}_i^t$ has nonzero values only in the indices from B_i. The ith node computes $\mathbf{d}^i$ and the difference $\Delta\mathbf{w}_i^t = \sum_{r\in B_i} d_r^i y_r \mathbf{x}_r$ between the current primal solution $\mathbf{w}^t$ and the new one. Then $\Delta\mathbf{w}_i^t$ is sent from the ith slave node to the *master node*. Upon receiving $\{\Delta\mathbf{w}_i^t\}_{i=1}^k$ the master node computes the new primal solution $\mathbf{w}^{t+1} = \mathbf{w}^t + \sum_{i=1}^k \Delta\mathbf{w}_i^t/k$ and sends it back to the slave nodes. In the appendix we provide a justification for $1/k$ multiplier. We denote this straightforward algorithm as DSVM-AVE and describe it formally at Algorithm 1.

Let $\mathbf{d} = \sum_{i=1}^k \mathbf{d}^i$. Another way of updating $\mathbf{w}^t$ is to do a line search along the direction

$$\bar{\mathbf{w}} = \sum_{i=1}^k \Delta\mathbf{w}_i^t = \sum_{i=1}^n d_i y_i \mathbf{x}_i \ . \tag{6}$$

The resulting algorithm, denoted as DSVM-LS, is formalized at Algorithm 2. Note that the direction $\bar{\mathbf{w}}$ might not be a descent direction in the primal space. We now explain this effect in more details.

Since each slave node solves (5), for any $1 \le i \le k$, $f_D(\boldsymbol{\alpha}^t + \mathbf{d}^i) < f_D(\boldsymbol{\alpha}^t)$. Using the convexity of $f_D(\boldsymbol{\alpha})$ we have that $\mathbf{d}^i \cdot \nabla f_D(\boldsymbol{\alpha}^t) < 0$. Therefore $\sum_{i=1}^k \mathbf{d} \cdot \nabla f_D(\boldsymbol{\alpha}^t) = \mathbf{d}^T Q\boldsymbol{\alpha}^t - \mathbf{d}^T\mathbf{e} < 0$. But it follows from (2) and (6) that

$$\bar{\mathbf{w}}^T \nabla f_P(\mathbf{w}^t) = \bar{\mathbf{w}}^T\mathbf{w}^t - C\bar{\mathbf{w}}^T \sum_{i\ :\ 1-y_i\mathbf{w}^t\cdot\mathbf{x}_i>0} y_i\mathbf{x}_i = \mathbf{d}^T Q\boldsymbol{\alpha}^t - C\bar{\mathbf{w}}^T \sum_{i\ :\ 1-y_i\mathbf{w}^t\cdot\mathbf{x}_i>0} y_i\mathbf{x}_i. \tag{7}$$

In general it is possible that (7) is not negative and thus $\bar{\mathbf{w}}$ is not a descent direction. For example, if the last sum is empty and $\mathbf{d} = \boldsymbol{\alpha}^t$ then (7) reduces to $(\boldsymbol{\alpha}^t)^T Q\boldsymbol{\alpha}^t$. Since Q is positive-semidefinite, the last expression is not negative. *A more general conclusion from this derivation is that if* $\mathbf{d}$ *is a descent direction in the dual space then the corresponding primal direction* $\bar{\mathbf{w}} = \sum_{i=1}^n d_i y_i \mathbf{x}_i$ *is not necessary a descent direction in the primal space.* Nevertheless, we observe empirically in Section 4 that there are iterations when $\bar{\mathbf{w}}$ is a descent direction and this allows DSVM-LS to optimize $f_P(\mathbf{w})$.

It can be verified that at each outer iteration both DSVM-AVE sends $O(kd)$ bytes. Also, if the complexity of the line search is $O(kd)$ (see Section 4 for such implementation of line search) then each iteration of DSVM-LS has the same complexity. Thus if the number of the outer iterations is constant then the communication complexity of these algorithms is independent of the training set size. In our experiments (see Section 4) we observed that the number of outer iterations needed to achieve empirically good results is less than 20.

4 Experiments

We implemented Algorithms 1 and 2 using the Map-Reduce framework on a Hadoop cluster. Each block optimization task is a mapper job. Line search is implemented as multi-round grid search with Map-Reduce on blocks. We used a fixed grid $\{\lambda_r\}_{r=1}^{20}$ of step sizes. At each iteration the master node receives from the slave ones the value of $\Delta\mathbf{w}_i$, computes $\sum_{i=1}^k \Delta\mathbf{w}_i$ and sends it to the slaves. Then each slave node i computes 20 values $\text{er}_{ir} = \sum_{j\in B_i} \max(0, 1 - y_i(\mathbf{w} + \lambda_r\Delta\mathbf{w})^T\mathbf{x}_i)$ and sends them to the master. Finally the master uses er_{ir}'s to choose the best step size.

set	original source	training size	test size	#features	#nonzeros
ADS	Akamai proprietary	79,620,765	81,883,670	5,384	2,603,374,016
MSR	Microsoft Learning to rank	37,010,170	10,659,213	136	3,838,258,824

Table 1: Data sets

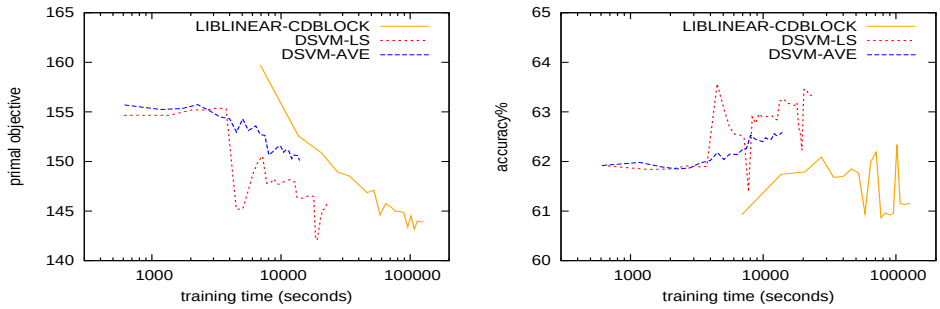

Figure 1: Experimental results on ADS.

We tested DBM on two large data sets, ADS and MSR. Their statistics are listed in Table 4. The ADS set is a proprietary dataset of Akamai. It has contains data for advertising modeling. The MSR dataset is transformed from Microsoft learning to rank dataset [10] by pairwise sampling. We only use pairs whose ranks are consecutive. We use 20 blocks for ADS experiments, and 50 blocks for MSR. Such number of blocks guarantee that each block fits into memory.

Figures (1) and (2) show the primal objective function and the accuracy on the test set. The X axis represents the wall-clock time of training. Each box is the result of the iteration of LIBLINEAR-CDBLOCK, that passes over all blocks sequentially. Each circle/staris is result of the iteration of DSVM-LS/DSVM-AVE that processes all blocks in parallel. Since LIBLINEAR-CDBLOCK training is slow on these large datasets, we use logarithmic scale for the X axis. As shown in the graphs DSVM-LS converges more quickly than DSVM-AVE with the help of line search, and significantly faster than LIBLINEAR-CDBLOCK.

DSVM-AVE completed the first iteration in 613 seconds on ADS and in 337 seconds on MSR, and the first round results provided reasonably good performance. It took LIBLINEAR-CDBLOCK 6,910 and 8,090 seconds respectively to complete the first iteration on each set, and its results are not as good as DSVM-AVE's. As to optimal values, DSVM-LS obtained better results in much less time as compared to LIBLINEAR-CDBLOCK, on both the objective function and test accuracy.

These graphs show also that all three algorithms do not necessary decrease value of the primal objective function at each iteration. This is an empirical confirmation of the effect discussed in Section 4. But despite this we observe the global convergence. For LIBLINEAR-CDBLOCK this was expected, since its global convergence was already proven in [19]. But for two other algorithms the global convergence has yet to be proven.

5 Conclusions and Future Work

We have presented a new algorithm for distributed training of linear SVM. Our Hadoop implementation allows us to obtain within 11 minutes accurate results on the datasets of tens of millions of examples. The future work includes the proof of the global convergence of DBM algorithms, combination of DBM with selective block minimization [3] and feature hashing [12].

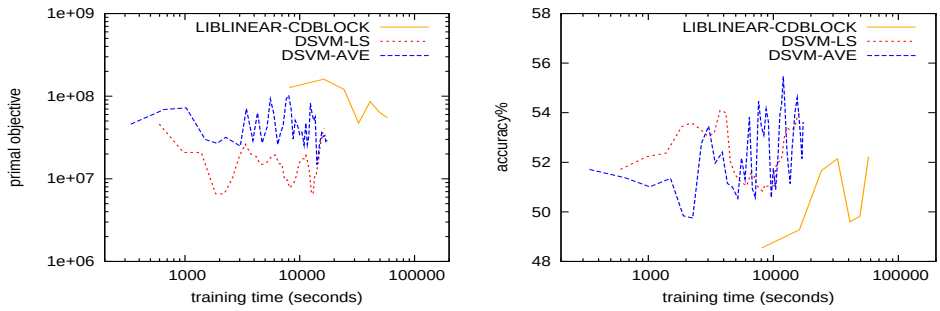

Figure 2: Experimental results on MSR.

References

[1] J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for L_1-regularized loss minimization. In *ICML*, pages 321–328, 2011.

[2] E.Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. Psvm: Parallelising support vector machines on distributed computers. In *NIPS*, 2008.

[3] K.-W. Chang and D. Roth. Selective block minimization for faster convergence of limited memory large-scale linear models. In *KDD*, pages 699–707, 2011.

[4] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20(3):273–297, 1995.

[5] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction. In *ICML*, pages 713–720, 2011.

[6] I. Durdanovic, E. Cossatto, and H.-P. Graf. Large-scale parallel SVM implementation. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 105–138. MIT Press, 2007.

[7] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *JMLR*, 9:1871–1874, 2008.

[8] P.A. Forero, A. Cano, and G. Giannakis. Consensus-based distributed support vector machines. *JMLR*, 11:1663–1707, 2010.

[9] T. Hazan, A. Man, and A. Shashua. A parallel decomposition solver for SVM: distributed dual ascend using Fenchel duality. In *CVPR*, 2008.

[10] http://research.microsoft.com/en us/projects/mslr/.

[11] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, pages 2331–2339, 2009.

[12] P. Li, A. Shrivastava, J. Moore, and C. Konig. Hashing algorithms for large-scale learning. In *NIPS*, 2011.

[13] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, pages 1231–1239, 2009.

[14] E. Osuna, R. Freund, and F. Girosi. an improved training algorithm for support vector machines. In *Proceedings of 1997 IEEE Workshop on Neural Networks for Signal Processing*, pages 276–285, 1997.

[15] S. Sonnenburg and V. Franc. Coffin: A computational framework for linear svm. In *ICML*, 2010.

[16] C.H. Teo, S.V.N. Vishwanathan, A. Smola, and Q.V. Le. Bundle methods for regularized risk minimization. *JMLR*, 11:311–365, 2010.

[17] K. Woodsend and J. Godsio. Hybrid mpi/openmp parallel linear support vector machine training. *JMLR*, 10:1937–1953, 2009.

[18] E. Yom-Tov. A distributed sequential solver for large-scale svms. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 139–154. MIT Press, 2007.

[19] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin. Large linear classification when data cannot fit in memory. In *KDD*, pages 833–842, 2010.

[20] L. Zanni, T. Serafini, and G. Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *JMLR*, 7:1467–1492, 2006.

[21] M. Zinkevich, M. Weimer, A.J. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

A Justification of $1/k$ factor in DSVM-AVE

Let $f_D(\boldsymbol{\alpha}^t)+\lambda \mathbf{d}^i \nabla f_D(\boldsymbol{\alpha}^t)$ be a linear approximation of $g_i(\lambda) = f_D(\boldsymbol{\alpha}^t+\lambda \mathbf{d}^i)$. Since $\mathbf{d}^i$ solves (5), $\lambda = 1$ minimizes $g_i(\lambda)$. We would like to find μ that will minimize $g(\mu) = f_D(\boldsymbol{\alpha}^t + \mu \sum_{i=1}^k \mathbf{d}^i)$. We have that

$$
\begin{aligned}
g(\mu) = f_D(\boldsymbol{\alpha}^t + \mu \sum_{i=1}^{k} \mathbf{d}^i) &\approx f_D(\boldsymbol{\alpha}^t) + \sum_{i=1}^{k} \mu \mathbf{d}^i \nabla f_D(\boldsymbol{\alpha}^t) \\
&= \frac{1}{k} \sum_{i=1}^{k} f_D(\boldsymbol{\alpha}^t) + \frac{1}{k} \sum_{i=1}^{k} k\mu \mathbf{d}^i \nabla f_D(\boldsymbol{\alpha}^t) \\
&= \frac{1}{k} \sum_{i=1}^{k} \left(f_D(\boldsymbol{\alpha}^t) + k\mu \mathbf{d}^i \nabla f_D(\boldsymbol{\alpha}^t) \right) \\
&\approx \frac{1}{k} \sum_{i=1}^{k} g_i(k\mu) \ .
\end{aligned}
$$

Since $\mu = 1/k$ minimizes each of $g_i(k\mu)$, we assume that $\mu = 1/k$ approximately minimizes $g(\mu)$. Combining this further with (2) we obtain the update rule $\mathbf{w}^{t+1} = \mathbf{w}^t + \frac{1}{k}\sum_{i=1}^{k}\sum_{j\in B_i} d_j^i y_j \mathbf{x}_i$ of DSVM-AVE.