Distributed Learning-to-Rank on Streaming Data using Alternating Direction Method of Multipliers

Kevin Duh, Jun Suzuki, and Masaaki Nagata
NTT Communication Science Laboratories, Kyoto, Japan
kevin.duh@lab.ntt.co.jp

Abstract

We show that Alternating Direction Method of Multipliers is an effective method for large-scale learning-to-rank on multi-cores and clusters, especially in scenarios requiring joint distributed and streaming architectures.

1 Introduction

Learning-to-rank algorithms are important tools for building modern search engines. With the growth of the Web, there is an increasing need for learning-to-rank algorithms to scale to ever larger datasets. What are the desiderata for handling scale? We believe that *distributed* and *streaming* are the two most important algorithmic properties. *Distributed* algorithms enable partitioning of training data, so as to speed-up training and enable usage of data that does not fit into a single machine's memory. On the other hand, *streaming* algorithms can continuously learn as new training samples are added. This is very desirable since new data, such as click logs, often becomes available with each additional user interaction with the search engine.

Fig. 1 illustrates this situation. First, large data necessitates that we store data partitions across machines and use a distributed algorithm to get an initial working ranker. Second, as new data streams in, we hope to rapidly update the ranker without incurring the full cost of re-training. Prior learning-to-rank research have mostly focused on the distributed aspect: [17, 18, 20] discuss various ways to parallelize gradient boosted trees for ranking. Generally, the training of weak learners (e.g. finding node splits) is distributed, but the boosting outer-loop is serial. We are not aware of an easy way to extend this kind of distributed algorithm to streaming data, unless we can un-learn weak learners from the ensemble.

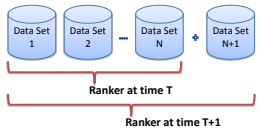


Figure 1: Usage scenario

In this paper, we explore the use of alternating direction method of multipliers (ADMM) [9, 3] for learning-to-rank. ADMM is a distributed optimization method that can also handle streaming data. Our formulation is similar to [8], though theirs is more complex as it deals with dynamic communication schemes. ADMM and its cousin Dual Decomposition [7] are gaining attention recently, with applications in signal processing [6], consensus optimization [13], and graphical models [10]. Here we show it is also effective for scaling up learning-to-rank on distributed, streaming datasets.

2 Proposed Rank-ADMM Algorithm

ADMM is a general optimization method that is naturally suited for partially-decomposable problems like regularized empirical risk: $\min_{\mathbf{w}} \Omega(\mathbf{w}) + \sum_m \ell_m(\mathbf{w})$, where $\Omega(\mathbf{w})$ is a regularizer that controls the complexity of predictor $\mathbf{w}$, and $\sum_m \ell_m(\mathbf{w})$ is a loss function summed across samples. Here we are concerned with a RankSVM solution for learning-to-rank, framed as follows:

$$\arg\min_{\mathbf{w}} \quad \frac{\lambda}{2}||\mathbf{w}||^2 + \sum_{m=1}^{M}(1 - y_m \mathbf{w}^T \mathbf{x}_m)_+ \tag{1}$$

Here, the feature vector $\mathbf{x}_m$ is the difference of two document vectors $d^{(1)} - d^{(2)}$ for a given search query. If document $d^{(1)}$ is more relevant than $d^{(2)}$, we set label $y_m = 1$; otherwise $y_m = -1$. In other words, the ranking problem is reduced to a binary classification problem with a hinge loss $(1 - y\mathbf{w}^T\mathbf{x})_+ = \max(0, 1 - y\mathbf{w}^T\mathbf{x})$ and L_2 regularizer. The number of training samples M in learning-to-rank tasks may easily exceed millions. To distribute training to N workers, we partition the data into $\{\mathcal{M}_n\}_{n=1,..N}$ subsets[1] and frame an analogous objective:

$$\arg\min_{\mathbf{w},\mathbf{v}_n} \quad \frac{N\lambda}{2}||\mathbf{w}||^2 + \sum_{n=1}^{N} \sum_{m \in \mathcal{M}_n} (1 - y_m \mathbf{v}_n^T \mathbf{x}_m)_+ \quad s.t.\ \mathbf{v}_n = \mathbf{w} \ \ \forall n \in (1, \ldots, N) \tag{2}$$

We have introduced additional vectors $\mathbf{v}_n$, $n = 1, 2, \ldots, N$, to de-couple the loss and regularizer terms. Specifically, note that the loss term has been decomposed into N components, where each component computes the loss on subsets of samples with respect to $\mathbf{v}_n$, not $\mathbf{w}$. Intuitively, $\mathbf{v}_n$ are local RankSVM solutions while the constraints $\mathbf{v}_n = \mathbf{w}\ \forall n$ ensure all local solutions are eventually consistent.

The formulation of Eq. 2 leads to a distributed algorithm where we iterate between parallel optimization of $\mathbf{v}_n$ and successive enforcement of constraints. ADMM achieves this by first formulating an Augmented Lagrangian:

$$\mathcal{L}_\rho(\mathbf{w}, \mathbf{v}_n, \boldsymbol{\mu}_n) : \frac{N\lambda}{2}||\mathbf{w}||^2 + \sum_{n=1}^{N} \left[\sum_{m \in \mathcal{M}_n} (1 - y_m \mathbf{v}_n^T \mathbf{x}_m)_+ + \boldsymbol{\mu}_n^T(\mathbf{v}_n - \mathbf{w}) + \frac{\rho}{2}||\mathbf{v}_n - \mathbf{w}||^2 \right] \tag{3}$$

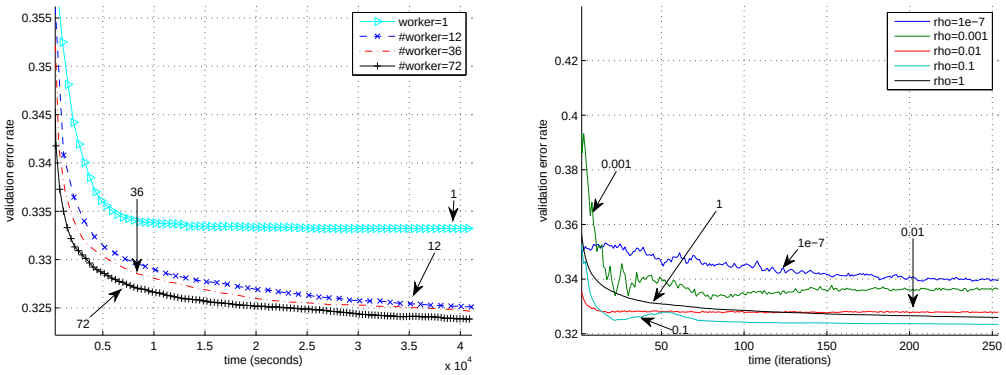
Here, $\boldsymbol{\mu}_n$ are dual variables for the constraints. The final term $\frac{\rho}{2}||\mathbf{v}_n - \mathbf{w}||^2$ with $\rho > 0$ ensures strict convexity and increases robustness (Dual Decomposition can be viewed as having $\rho = 0$). Following [3], we scale the dual variables ($\mathbf{u}_n = \boldsymbol{\mu}_n/\rho$) and algebraically manipulate Eq. 3 to get the following equivalent Augmented Lagrangian, which is simpler to work with:
$\mathcal{L}_\rho(\mathbf{w}, \mathbf{v}_n, \mathbf{u}_n) : \frac{N\lambda}{2}||\mathbf{w}||^2 + \sum_{n=1}^{N} \left[\sum_{m \in \mathcal{M}_n} (1 - y_m \mathbf{v}_n^T \mathbf{x}_m)_+ + \frac{\rho}{2}||\mathbf{v}_n - \mathbf{w} + \mathbf{u}_n||^2 - \frac{\rho}{2}||\mathbf{u}_n||^2 \right]$.

Now ADMM works by iteratively computing one of $\mathbf{v}_n$, $\mathbf{w}$, $\mathbf{u}_n$ while holding the other variables fixed in the following iterations $k = 1, 2, \ldots$ until convergence:

1. $\mathbf{v}_n$-update: Solve $\mathbf{v}_n^{k+1} = \arg\min_{\mathbf{v}_n} \mathcal{L}_\rho(\mathbf{w}^k, \mathbf{v}_n, \mathbf{u}_n^k)$ on N parallel workers.
   For our case, this corresponds to solving standard RankSVM, with the addition of a bias term $(-\mathbf{w} + \mathbf{u}_n)$ in the regularizer:
   $$\mathbf{v}_n^{k+1} := \arg\min_{\mathbf{v}_n} \sum_{m \in \mathcal{M}_n} (1 - y_m \mathbf{v}_n^T \mathbf{x}_m)_+ + \frac{\rho}{2}||\mathbf{v}_n - \mathbf{w}^k + \mathbf{u}_n^k||^2 \tag{4}$$
2. $\mathbf{w}$-update: Solve $\mathbf{w}^{k+1} = \arg\min_{\mathbf{w}} \mathcal{L}_\rho(\mathbf{w}, \mathbf{v}_n^{k+1}, \mathbf{u}_n^k)$
   For our case, this can be computed in closed-form:
   $$\mathbf{w}^{k+1} := \arg\min_{\mathbf{w}} \ \frac{N\lambda}{2}||\mathbf{w}||^2 + \frac{\rho}{2}||\mathbf{v}_n^{k+1} - \mathbf{w} + \mathbf{u}_n^k||^2 = \frac{\rho}{(\lambda + \rho)}(\bar{\mathbf{v}}_n + \bar{\mathbf{u}}_n) \tag{5}$$
   where $\bar{\mathbf{v}}_n = \sum_{n=1}^{N} \mathbf{v}_n/N$ and $\bar{\mathbf{u}}_n = \sum_{n=1}^{N} \mathbf{u}_n/N$. Intuitively, we average local RankSVM and dual vectors from multiple machines and apply a proximity operator. For L_2 this scales the result toward origin, but other regularizers like L_1 are also applicable [5].
3. $\mathbf{u}_n$-update: Perform gradient ascent on dual variables to tighten the constraints.
   $$\mathbf{u}_n^{k+1} := \mathbf{u}_n^k + (\mathbf{v}_n^{k+1} - \mathbf{w}^{k+1}) \tag{6}$$

It can be proved that the above iterations will attain the optimal objective value of Eq. 2 at convergence, where convergence is defined as $\sum_n ||\mathbf{v}_n^k - \mathbf{w}^k|| < \epsilon_1$ and $||\mathbf{w}^{k+1} - \mathbf{w}^k|| < \epsilon_2$ for suitably small ϵ_1, ϵ_2 [3]. With regards to streaming, it can be shown that ADMM converges to the objective given by the available subset of data [8]. Thus it handles both distributed and streaming conditions.

[1] $\{\mathcal{M}_n\}$ specifies a disjoint partitioning of sample indexes, $\bigcup_n \mathcal{M}_n = (1, .., M), \bigcap_n \mathcal{M}_n = \emptyset$. In contrast to some distributed methods, we do not require the partitions to overlap or satisfy certain statistical properties.



(a) Comparing #workers. ($\rho = 1, \lambda = 10^{-7}$)

(b) Comparing ρ. (72-workers, $\lambda = 10^{-7}$)

Figure 2: Learning curves: training time with respect to validation set error.

Table 1: Comparison of speed improvements. Note the skewed distribution of #samples/worker since data is partitioned by queries (not documents). For 72-workers, the largest partition has 327k samples, a 15.6x factor reduction compared to 5.1M. Speed-up is time with respect to 1-worker case. Total time is defined as wall-clock time to reach the same error rate as the converged 1-worker solution (error=33.33%). Last column indicates the final error at time=40000 seconds.

#Worker	**#Samples per worker** $\lvert\mathcal{M}_n\rvert$			**Time per iteration**		**Total time**			**Error**
(N)	median	max	reduction	seconds	speedup	iteration	seconds	speedup	(%)
1	5.1M	5.1M	-	533	-	42	19970	-	33.33
12	335k	1.3M	3.9x	142	3.8x	25	3613	5.5x	32.52
36	87k	522k	9.7x	85	6.3x	28	2422	8.2x	32.48
72	38k	327k	15.6x	51	10.5x	29	1519	13.1x	32.39

3 Practical Implementation

We implemented the ADMM algorithm using MPI. While Eq. 4 and Eq. 6 can be distributed among workers, Eq. 5 requires communicating $\mathbf{v}_n$ and $\mathbf{u}_n$ from all workers. One implementation is to have a master process collect $\mathbf{v}_n$ and $\mathbf{u}_n$ from N slaves and broadcast the updated $\mathbf{w}$. But this creates a possible single-point of failure. We opt for a de-centralized architecture (using MPI's AllReduce function) [3], where individual workers compute all of Eq. 4, 5, 6. This replicates the same $\mathbf{w}$-update computation, but in general proximity operators are not expensive.

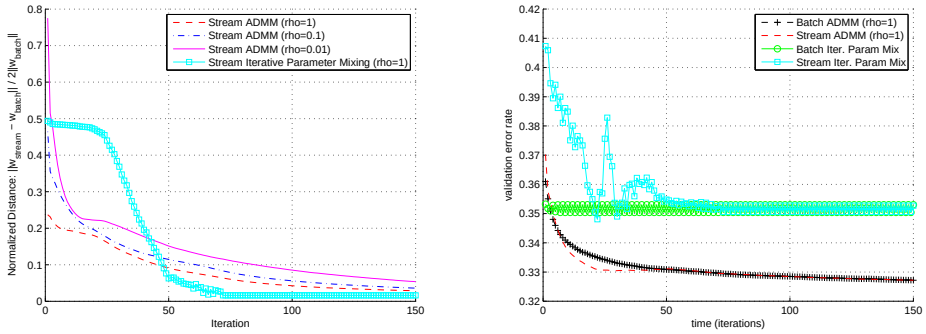
Eq. 4 is solved using stochastic gradient descent (SGD) [2]. For each sample, we perform the update $\mathbf{v}_n - \eta\delta_{\mathbf{v}_n}$ with subgradient $\delta_{\mathbf{v}_n} = \rho(\mathbf{v}_n - \mathbf{w} + \mathbf{u}_n) - y_m\mathbf{x}_m$ for misclassified samples and step size $\eta = \frac{1}{\rho m}$ inspired by [15, 14].[2] We make only one-pass (1 epoch) through the data in each iteration, since ADMM does not require exact minimization of Eq. 4.

It's worth mentioning the ADMM parameter ρ interacts with $\mathbf{v}_n$-update and $\mathbf{w}$-update in an interesting way. A small ρ makes Eq. 4 aggressively reduce individual loss (leading to vectors with large norms), which is later scaled back aggressively by Eq. 5. Conversely, a large ρ encourages the individual RankSVMs to stay close to the global $\mathbf{w}$. As a result the $\mathbf{w}$-update is conservative.

4 Results and Discussion

We experiment with the Yahoo Learning to Rank Challenge Dataset [4]. The training data consists of ≈20k queries and 473k documents, for a total of 5.1 million pairwise samples for RankSVM. Our computing environment consists of 6 blade servers, each with 2 units of 6-core 3.07GHz Xeon and 192GB memory (for a total of 72 cores) and 1Gb Ethernet connection between servers.

[2] We have also experimented with projection onto $\frac{1}{\sqrt{\lambda}}$ ball like [15], but it appears to give unstable results due to the difficulty of balancing the norms of various $\mathbf{v}_n$, $\mathbf{u}_n$, and $\mathbf{w}$ after projection.



(a) Difference with batch solution

(b) Learning curve

Figure 3: Streaming with ADMM.

How does ADMM scale with more workers? We partition the data by queries into 1, 12, 36, and 72 subsets and evaluate the training speedup due to distributed optimization. Fig. 2(a) shows the learning curve on the validation set; Table 1 gives detailed timing breakdown. The 1-worker case (SGD on full data) begins to converge at around error = 33.33%; ADMM with 12, 36, and 72 workers achieves the same error respectively with 5.5x, 8.2x, and 13.1x factor less time. The speedup is substantial but does not seem linear in the number of workers. This is influenced by 3 reasons, in order of importance: (1) size of the largest partition, which limits the end-time of $\mathbf{v}_n$-update, (2) communication costs in $\mathbf{w}$-update, and (3) the number of iterations to reach desired accuracy. For example, consider the 72-worker case: largest partition is only a factor of 15.6 smaller than full data, so we cannot expect time/iteration to be more than 15.6 times faster than 1-worker; it turns out to be 10.5 times faster, due to communication overhead across servers. In contrast, 12 workers fit on the same server so the max sample size reduction (3.9x) and time/iteration speedup (3.8x) are similar; further, the final speedup (total time) increases to 5.5x since fewer iterations are required than the 1-worker case. Finally, distributed solutions not only converge faster but are converging at a *better* result (e.g. $\sim$1% better for 72-worker). This may be due to the ensemble generalization effect [19].

How does ρ impact convergence? Fig. 2(b) shows the learning curve for 72-workers with varying ρ. Tuning ρ can matter much in practice. $\rho = 0.1$ gives the fastest convergence, achieving the same 33.33% error in only 8 iterations (total time of 171 seconds). A large $\rho = 1$ leads to smoother but slower convergence, while extremely small ρ, e.g. $\rho = \lambda = 10^{-7}$ leads to zig-zag behavior. In our implementation, ρ serves double duty as Lagrangian penalty and SGD step size due to our desire to limit the number of tunable parameters. We recommend selecting ρ by tuning SGD once on a single partition; good settings there seem to carry over to ADMM.

How does ADMM behave with streaming data? To evaluate this, we start ADMM with 1 of the 72 data partitions, then add a new subset with each additional iteration. As a figure of merit, we compare the distance of $\mathbf{w}$ obtained under streaming vs. $\mathbf{w}$ obtained by training with all 72 partitions in batch from the start. Fig. 3(a) shows that distance rapidly decreases to 0.1, but takes long to reach 0. However, the streaming version tracks the validation error of the batch-data version without lag, as seen in Fig. 3(b). Interestingly, it can decrease error *faster* initially (iterations 10-30) because less data implies less stalling in $\mathbf{v}_n$-updates and not all samples are needed this large-scale data. Recent work on parallel online learning [21, 11, 12] also shows promise in joint streaming/distributed learning. We implemented a streaming version of Iterative Parameter Mixing [12] as comparison in Fig. 3. In Fig. 3(b), this streamed version tracks the learning curve of its batch version very well after iteration 50, but differs wildly beforehand. So ADMM appears more robust in this regard.

In conclusion, we advocate ADMM in large-scale applications, especially since it handles both distributed and streaming conditions. Future work includes: (1) asynchronous update scheme [1], since unequal data partition poses the most serious challenge to speed-up here, and (2) extension of ADMM to non-linear functions, possibly by modifying the constraints in Eq. 2 with e.g. co-regularization norms [16].

References

[1] Dimitri Bertsekas and John Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[2] Leon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *NIPS*, 2008.

[3] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Foundations and Trends in Machine Learning, 2011.

[4] Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. *JMLR: Workshop and Conference Proceedings*, 14, 2011.

[5] John Duchi and Yoram Singer. Efficient online and batch learning using forward backward splitting. *JMLR*, 10, 2009.

[6] Ernie Esser. Applications of lagrangian-based alternating direction methods and connections to split bregman. Technical report, UCLA CAM report, 2009.

[7] Hugh Everett. Generalized lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11(3):399–417, 1963.

[8] Pedro Forero, Alfonso Cano, and Georgios Giannakis. Consensus-based distributed support vector machines. *JMLR*, 2010.

[9] Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers and Mathematics with Applications*, 2(1):17 – 40, 1976.

[10] Nikos Komodakis, Nikos Paragios, and Georgios Tziritas. MRF optimization via dual decomposition: Message-passing revisited. In *ICCV*, 2007.

[11] John Langford, Alex Smola, and Martin Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[12] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *NAACL HLT*, 2010.

[13] Angelia Nedić and Asuman Ozdaglar. Cooperative distributed multi-agent optimization. In Yonina Eldar and Daniel Palomar, editors, *Convex Optimization in Signal Processing and Communications*. Cambridge Univ. Press, 2008.

[14] D. Sculley. Large scale learning to rank. In *NIPS 2009 Workshop on Advances in Ranking*, 2009.

[15] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-GrAdient SOlver for SVM. In *ICML*, 2007.

[16] Vikas Sindhwani and David Rosenberg. An RKHS for multi-view learning and manifold co-regularization. In *ICML*, 2008.

[17] Krysta M. Svore and Christopher J.C. Burges. Large-scale learning to rank using boosted decision trees. In Ron Bekkerman, Misha Bilenko, and John Langford, editors, *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge Univ. Press, May 2011.

[18] Stephen Tyree, Kilian Wienberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In *WWW*, 2011.

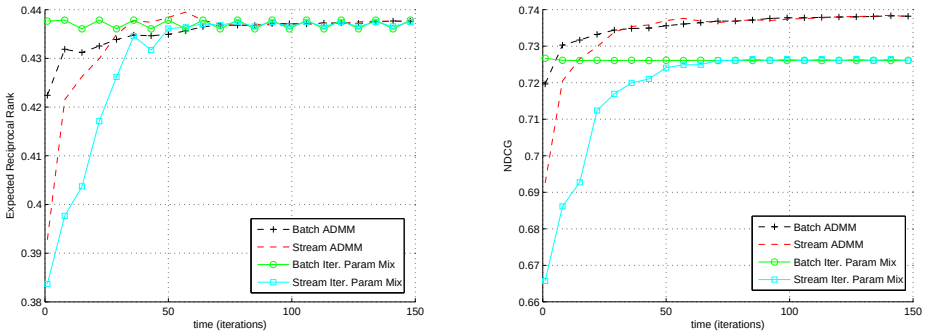
[19] Naonori Ueda and Ryohei Nakano. Generalization error of ensemble estimators. In *IEEE International Conference on Neural Networks*, volume 1, 1996.

[20] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM*, 2009.

[21] Martin Zinkevich, Alexander J. Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.

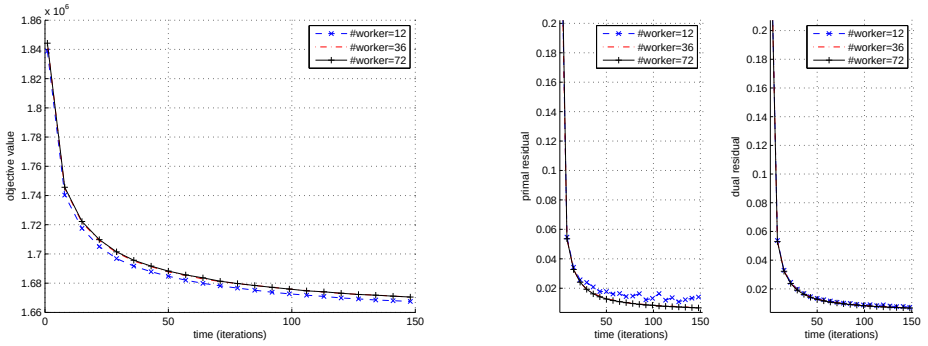
Appendix: Additional Results

Additional numerical results and statistics from the experiments are reported here for reference.



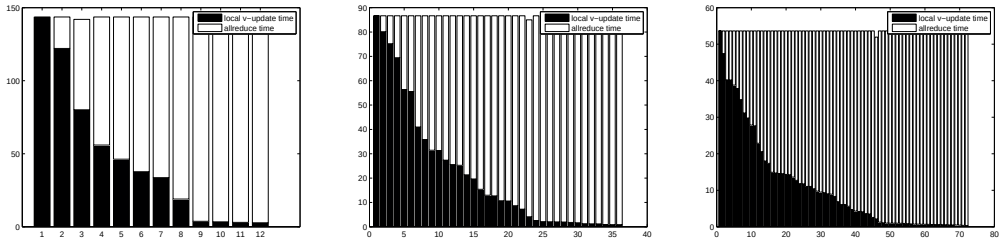
(a) Expected Reciprocal Rank (b) NDCG@10

Figure 4: Evaluation by common Learning-to-Rank metrics. These learning curves are analogous to Fig. 3(b), except that they plot the improvement in application-specific metrics. Results are comparable to baselines (Expected Reciprocal Rank = 0.43, NDCG = 0.75) reported in [4].



(a) Convergence of Objective Value (b) Convergence of Primal/Dual Residuals

Figure 5: ADMM converges to modest objective values very fast but slows down drastically in later iterations. Observing the decrease in primal residuals ($\sum_n ||\mathbf{v}_n^k - \mathbf{w}^k||$) and dual residuals ($||\mathbf{w}^{k+1} - \mathbf{w}^k||$), we would recommend that less than 50 iterations is sufficient here. The convergence rate by iteration is approximately the same for different number of workers (though of course the total wall-clock time differs substantially).



(a) 12 workers (b) 36 workers (c) 72 workers

Figure 6: Breakdown of time usage for each MPI process in one iteration. Black bars indicate time spent in $\mathbf{v}_n$-update. White bars indicate time in $\mathbf{w}$-update, which is dominated by AllReduce passing and summing $\sum_n \mathbf{v}_n$ and $\sum_n \mathbf{u}_n$ in Eq. 5. The time in $\mathbf{u}_n$-update is negligible. Many workers are spending time in AllReduce, likely waiting for the slowest worker to finish. The slowest worker, when finished, already has partial summation results available so its AllReduce time is $\sim$0. Skewed data partition occurs in practice so we believe asynchronous updates are the most promising future direction for practical speed-up improvements.