CG4001 - B.Eng. Dissertation

CA Report

Scaling up Machine Learning techniques via parallelization for large data

By

Akshay Viswanathan

A0074611M

Project ID: H148130

Project Supervisor: Dr. Low, Bryan Kian Hsiang

Main Evaluator: Dr. Leong, Tze Yun

Contents

1 Introduction

My final year research project focuses on developing a parallel implementation of the Support Vector Regression algorithm which makes it suitable for analysing large data by exploiting multi-core processors and supercomputing clusters.

Support Vector Regression (SVR) is a variation of Support Vector Machines which produces a regression model using a subset of the input training data. Unlike traditional regression models that derive a function with the least deviation between the predicted and experimentally observed responses for all training examples, SVR attempts to minimize the generalized error bound so as to achieve generalized performance.

Unfortunately, usage of SVR for processing of large data is limited by the large computation cost in both memory use and computational time. It incurs cubic time and quadratic memory in the size of the data.

To improve scalability, a Parallel Support Vector Regression(PSVR) algorithm has been proposed which uses low-rank matrix approximation to distribute the computational load amongst parallel machines to achieve time efficiency and scalability by improving the computational time to $O(np^2/m)$ and decreasing the memory requirement to $O(np/m)$ where n is the number of training examples, p is the rank of the reduced matrix after approximation and m is the number of processing cores available.

As mentioned above, usage of traditional SVR implementations fails with increase in data which would mean the inability to use these implementations in most real-world applications such as traffic monitoring and prediction. The implementation of the PSVR algorithm would mean:

- Increased efficiency in training the SVR model by reducing the memory as well as time complexity achieved by using computing clusters.

- Potentially allowing real-time predictions by distributedly storing data and using computing clusters.

Thereby alleviating the main bottlenecks of SVRs by using increased hardware.

The specific contributions of my work include:

- Experimentally guaranteeing the predictive performance of PSVR to be approximately equivalent to some of the other existing SVR implementations. This would mean that even with low-rank matrix approximation, PSVR could replace existing SVR implementations while still benefiting by having increased scalability and efficiency. The rank of the approximated matrix is controlled by a parameter and the value of this parameter can varied to achieve the required trade off between accuracy and scalability.
- Analysing the time and space complexity of the PSVR model
- Implementing PSVR using the Message Passing Interface(MPI) and experimentally verifying the predictive performance, scalability and speedup.

2 Related Work

The regression version of SVMs was proposed by in 1996 by Smola et al. and utilized a different loss function from the traditional SVMs(Section 3). Since then, although there have been some work regarding variations and experimentations of the SVR algorithm, there has been no significant development of the algorithm. There have been some attempts to parallelize SVRs by the PiSVM and MRPsvm projects, both of which utilize matrix decomposition to solve the QP problem. The key differences have been highlighted in the next section.

The idea of scaling up traditionally unscalable machine learning techniques however, has received a lot of interest in the past few years (Bekkerman et al.). Parallel Gaussian Process

Regression (Chet et al.) utilizes supercomputing clusters to distribute the computational load among various processing cores during the Gaussian Process Regression. Parallel Support Vector Machines(PSVM) (Chang et al.) also utilizes parallel computing to ensure scalability when using Support Vector Machines for classification. Since SVR is a variation of SVM, several ideas and concepts such as row-based Incomplete Cholesky Factorization for utilizing a low-rank matrix approximation as well as relying on the Sherman-Morrison-Woodbury formula to calculate the inverse of a high-dimension matrix have been reused from PSVM for this project.

There have been related works in speeding up SVMs including Cascade SVMs(Vapnik et al.) which relies on splitting up the input data into several smaller data sets, computing the local Support Vectors and combining the results from these machines and SVMLight, Reduced SVMs, proximal SVMs and Core Vector Machines which propose alternate algorithms to speed up solving the Quadratic Programming problem.

For this project, a variation of the PSVM will be developed which tries to achieve both a speedup in the process of solving the Quadratic Programming process as well as splitting the input data into smaller sets.

3 Comparison with PiSVM and MRPsvm

As mentioned above, the key difference between the proposed PSVR and both PiSVM and MRPsvm is that PSVR solves the QP problem by approximating the kernel matrix using a low rank matrix compared to PiSVM and MRPsvm which decompose the original kernel matrix into several smaller matrices and utilize several processing cores to independently work on the decomposed matrices before reducing the results; MRPsvm uses the Map-Reduce framework to perform this task and PiSVM uses MPI when suitable.

Several other minor differences and similarities exist between the the frameworks. MRPsvm maintains a shared cache of the entire kernel matrix which all the processes access while PiSVM, similar to PSVR, maintains a distributed copy of the kernel matrix on each of the parallel processes. MRPsvm maintains a shared copy of all the training instance while PiSVM maintains a copy of the training instances on each process, thereby requiring more resources; PSVR distributedly maintains a single copy of the training instances on the parallel processes.

4 Background

In this section, the details of Support Vector Regression will briefly be explained. Most of the content has been extracted from Elements of Statistical Learning and A Tutorial on Support Vector Regression.

Given training data: $\{(x_1, y_i) \ldots (x_l, y_l)\} \subset X\text{x}R$, where X denotes the space of input patterns, in ϵ-SV regression our goal is to find a function f(x) that has at most ϵ deviation from the actually obtained targets y_i for all training data while at the same time is as flat as possible. We describe this function as:

$$f(x) = \langle w, x \rangle + b \; with \; w \; \epsilon \; X, \;\; b \; \epsilon \; R$$

where $\langle \; . \; , \; . \; \rangle$ denotes the dot product in X. We can ensure flatness of this system by minimizing $||w||^2$. Similar to the case of soft-margin SVMs, we may want to allow some degree of errors and this is represented by the slack variables ζ, ζ^*(Slack variables for input data above and below the regression hyperplane respectively). From this, we arrive at the formulation:

$$\begin{aligned} minimize \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{l}(\zeta_i + \zeta_i^*) \\ subject\ to\ y_i - \langle w, x_i\rangle - b \ & \le \ \epsilon + \zeta_i \\ \langle w, x_i\rangle + b \ - y_i \le \ & \epsilon + \zeta_i^* \\ & \zeta_i, \zeta_i^* \ge 0 \end{aligned}$$

where C > 0 determines the trade-off between flatness of f and the extent upto which deviations from ϵ are tolerated. In most cases, the above formulation can be solved more easily in its dual, hence Lagrange multipliers are used, leading to the Lagrangian Dual:

$$\begin{aligned} L \ := \ & \frac{1}{2}||w||^2 + C\sum_{i=1}^{l}(\zeta_i + \zeta_i^*) - \sum_{i=1}^{l}(\eta_i\zeta_i + \eta_i^*\zeta_i^*) \\ & -\sum_{i=1}^{l}\alpha_i(\epsilon + \zeta_i - y_i + \langle w, x_i\rangle + b) \\ & -\sum_{i=1}^{l}\alpha_i^*(\epsilon + \zeta_i^* + y_i - \langle w, x_i\rangle - b) \end{aligned}$$

Here $\eta_i, \eta_i^*, \alpha_i, \alpha_i^*$ are Lagrange multipliers and satisfy the constraints

$$\eta_i^{(*)}, \alpha_i^{(*)} \ge 0$$

After some tedious partial differentials and substitutions, we arrive at the dual optimization problem:

$$max\ \frac{-1}{2}\sum_{i,j=0}^{n}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)K(x_i, x_j) - \epsilon\sum_{i=0}^{n}(\alpha_i + \alpha_i^*) + \sum_{i=0}^{n}y_i(\alpha_i - \alpha_i^*)$$

$$Subject\ to : \sum_{i=0}^{n}(\alpha_i - \alpha_i^*) = 0,\ 0 \le \alpha, \alpha^* \le C,\ \alpha_i\alpha_i^* = 0$$

and

$$w = \sum_{i=1}^{l} (\alpha_i - \alpha_i^*) x_i, and\ f(x) = \sum_{i=1}^{l} (\alpha_i - \alpha_i^*) \langle x_i, x \rangle + b$$

Similar to SVMs, only a subset of $(\alpha_i - \alpha_i^*)$ are non-zero and these constitute the Support Vectors for the Regression model.The parameter b can be computed as a by-product of the optimization process.

The $K(x_i, x_j)$ component of the above formulation will further be represented as Q. This is an inner product matrix between each of the training instance and is one of the key bottlenecks of the SVR process since the size of the matrix is quadratic in the number of training instances. Similar to SVMs, the Kernel Trick can be used to map the input training instances to higher dimensional spaces if required.

5 Technical Approach

This section will describe the implementation details for PSVR, specifically the steps taken to improve efficiency and scalability.

5.1 Distributed Data Loading and Storage

Each of the n training instances are distributedly loaded onto the m machines in a round robin manner so that instance i is loaded onto machine $i\%m$. All variables computed in subsequent steps are local to the instances on that machine. Global variables are replicated on the machines as required. The support vectors computed as a result of the model training are retained locally and used in the prediction process. The matrix as computed by PICF is also distributedly stored. This is done to distribute the overall memory requirement.

5.2 Parallel Incomplete Cholesky factorization

A key step in PSVR is parallel ICF (PICF). Traditional column-based ICF reduces computational cost, but the initial memory requirement is $O(np)$, and hence not very practical for a large data set. PSVR uses a row-based ICF which loads training instances onto parallel machines and performs the factorization on these machines. At the end of ICF, we get a low-rank approximation of the matrix Q. The details of PICF have been described in Appendix A.

5.3 Interior Point Method

The bottlenecks of SVRs exist primarily in solving the Quadratic Programming problem. We utilize Interior Point Methods to solve the Quadratic Programming problem with linear constraints. Currently, the most effective IPM algorithm is the Primal Dual Interior point method which utilizes barrier functions to eliminate the inequalities following which uses the iterative Newton's method to reach the optimal solution. This is adapted for PSVR in the following equations.

The dual to be maximized is of the form:

$$max\ \frac{-1}{2}\sum_{i,j=0}^{n}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)K(x_i,x_j)-\epsilon\sum_{i=0}^{n}(\alpha_i+\alpha_i^*)+\sum_{i=0}^{n}y_i(\alpha_i-\alpha_i^*)$$

$$Subject\ to:\sum_{i=0}^{n}(\alpha_i-\alpha_i^*)=0,\ 0\leq\alpha,\alpha^*\leq C$$

To optimize this dual equation, we consider its Lagrangian dual:

$$min\frac{1}{2}\sum_{i,j=0}^{n}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)Q + \epsilon\sum_{i=0}^{n}(\alpha_i + \alpha_i^*) - \sum_{i=0}^{n} y_i(\alpha_i - \alpha_i^*) - \sum_{i=0}^{n}\lambda_i(C - \alpha_i)$$
$$-\sum_{i=0}^{n}\xi_i(C - \alpha_i^*) - \sum_{i=0}^{n}\theta_i\alpha_i - \sum_{i=0}^{n}\phi_i\alpha_i^* + \nu\sum_{i=0}^{n}(\alpha_i - \alpha_i^*)$$
$$Subject\ to:\ \lambda_i,\ \xi_i,\ \theta_i,\ \phi_i \geq 0$$

From the dual and modified KKT conditions we get:

$$\frac{dL}{d\alpha} = Q(\alpha - \alpha^*) + 1_n.\epsilon - y + \lambda - \theta + 1_n.\nu = 0$$
$$\frac{dL}{d\alpha^*} = -Q(\alpha - \alpha^*) + 1_n.\epsilon + y + \xi - \phi - 1_n.\nu = 0$$
$$\lambda_i(C - \alpha_i) = \frac{1}{t} \qquad \theta_i\alpha_i = \frac{1}{t}$$
$$\xi_i(C - \alpha_i^*) = \frac{1}{t} \qquad \phi_i\alpha_i^* = \frac{1}{t}$$
$$\sum_{i=0}^{n}\alpha_i - \alpha_i^* = 0$$
$$where\ Q = K(x_i, x_j)$$

According to Convex Optimization(Boyd, 2004), the increments in the Newton's iterations can be described as:

$$\begin{pmatrix}
Q_{nn} & -Q_{nn} & I_{nn} & 0_{nn} & -I_{nn} & 0_{nn} & 1_n \\
-Q_{nn} & Q_{nn} & 0_{nn} & I_{nn} & 0_{nn} & -I_{nn} & -1_n \\
-diag(\lambda)_{nn} & 0_{nn} & diag(C-\alpha)_{nn} & 0_{nn} & 0_{nn} & 0_{nn} & 0_n \\
0_{nn} & -diag(\xi)_{nn} & 0_{nn} & diag(C-\alpha^*)_{nn} & 0_{nn} & 0_{nn} & 0_n \\
diag(\theta)_{nn} & 0_{nn} & 0_{nn} & 0_{nn} & diag(\alpha)_{nn} & 0_{nn} & 0_n \\
0_{nn} & diag(\phi)_{nn} & 0_{nn} & 0_{nn} & 0_{nn} & diag(\alpha^*)_{nn} & 0_{nn} \\
1_n^T & -1_n^T & 0_n^T & 0_n^T & 0_n^T & 0_n^T & 0
\end{pmatrix}$$

$$\begin{pmatrix} \Delta\alpha \\ \Delta\alpha^* \\ \Delta\lambda \\ \Delta\xi \\ \Delta\theta \\ \Delta\phi \\ \Delta\nu \end{pmatrix} = - \begin{pmatrix} Q(\alpha-\alpha^*) + 1_n.\epsilon - y + \lambda - \theta + 1_n.\nu \\ -Q(\alpha-\alpha^*) + 1_n.\epsilon + y + \xi - \phi - 1_n.\nu \\ vec(\lambda(C-\alpha) - \frac{1}{t}) \\ vec(\xi(C-\alpha^*) - \frac{1}{t}) \\ vec(\theta\alpha - \frac{1}{t}) \\ vec(\phi\alpha^* - \frac{1}{t}) \\ \sum_{i=0}^{n} (\alpha_i - \alpha_i^*) \end{pmatrix}$$

From this matrix we get:

$$\Delta\lambda_i = -\lambda_i + diag(\frac{\lambda_i}{(C-\alpha_i)})\Delta\alpha_i + vec(\frac{1}{t(C-\alpha_i)})$$
$$\Delta\theta_i = -\theta_i - diag(\frac{\theta_i}{\alpha_i})\Delta\alpha_i + vec(\frac{1}{t\alpha_i})$$
$$\Delta\xi_i = -\xi_i + diag(\frac{\xi}{(C-\alpha_i^*)})\Delta\alpha_i^* + vec(\frac{1}{t(C-\alpha_i^*)})$$
$$\Delta\phi_i = -\phi_i - diag(\frac{\phi_i}{\alpha_i^*})\Delta\alpha_i^* + vec(\frac{1}{t\alpha_i^*})$$
$$Q\Delta\alpha - Q\Delta\alpha^* + \Delta\lambda - \Delta\theta + 1_n\Delta\nu = -Q(\alpha-\alpha^*) - 1_n.\epsilon + y - \lambda + \theta - 1_n.\nu$$
$$-Q\Delta\alpha + Q\Delta\alpha^* + \Delta\xi - \Delta\phi - 1_n\Delta\nu = Q(\alpha-\alpha^*) - 1_n.\epsilon - y - \xi + \phi + 1_n.\nu$$

After substitutions, we get

$$Q\Delta\alpha - Q\Delta\alpha^* + \frac{1}{t(C-\alpha)} - \lambda + diag(\frac{\lambda}{(C-\alpha)})\Delta\alpha - \frac{1}{t\alpha} + \theta + diag(\frac{\theta}{\alpha})\Delta\alpha + \Delta\nu$$
$$= -Q(\alpha-\alpha^*) - 1_n.\epsilon + y - \lambda + \theta - 1_n.\nu$$
$$-Q\Delta\alpha + Q\Delta\alpha^* + \frac{1}{t(C-\alpha^*)} - \xi + diag(\frac{\xi}{(C-\alpha^*)})\Delta\alpha^* - \frac{1}{t\alpha^*} + \phi + diag(\frac{\phi}{\alpha^*})\Delta\alpha^* - \Delta\nu$$
$$= Q(\alpha-\alpha^*) - 1_n.\epsilon - y - \xi + \phi + 1_n.\nu$$

Simplifying the above, we get

$$(Q + diag(\frac{\lambda}{C-\alpha} + \frac{\theta}{\alpha}))\Delta\alpha - Q\Delta\alpha^* + \Delta\nu = -Q(\alpha-\alpha^*) - 1_n.\epsilon + y + \frac{1}{t}(\frac{1}{\alpha} - \frac{1}{(C-\alpha)}) - 1_n.\nu$$
$$-Q\Delta\alpha + (Q + diag(\frac{\xi}{C-\alpha^*} + \frac{\phi}{\alpha^*}))\Delta\alpha^* - \Delta\nu = Q(\alpha-\alpha^*) - 1_n.\epsilon - y + \frac{1}{t}(\frac{1}{\alpha^*} - \frac{1}{(C-\alpha^*)}) + 1_n.\nu$$

$$Setting:\ \delta = diag(\frac{\lambda}{C-\alpha} + \frac{\theta}{\alpha});\ \delta^* = diag(\frac{\xi}{C-\alpha^*} + \frac{\phi}{\alpha^*})$$
$$\rho = -Q(\alpha-\alpha^*) - 1_n.\epsilon + y + \frac{1}{t}(\frac{1}{\alpha} - \frac{1}{(C-\alpha)}) - 1_n.\nu\ and$$
$$\rho^* = Q(\alpha-\alpha^*) - 1_n.\epsilon - y + \frac{1}{t}(\frac{1}{\alpha^*} - \frac{1}{(C-\alpha^*)}) + 1_n.\nu$$

Giving us:

$$\begin{pmatrix} Q+\delta & -Q_{nn} & 1_n \\ -Q_{nn} & Q_{nn}+\delta^* & -1_n \\ 1_n & -1_n & 0 \end{pmatrix} \begin{pmatrix} \Delta\alpha \\ \Delta\alpha^* \\ \Delta\nu \end{pmatrix} = \begin{pmatrix} \rho \\ \rho^* \\ -\sum_{i=0}^{n}(\alpha_i - \alpha_i^*) \end{pmatrix}$$

$$From\ this\ matrix,\ we\ get:$$
$$\Delta\alpha^* = (\delta^*)^{-1}(\rho+\rho^*) - (\delta^*)^{-1}\delta\Delta\alpha$$
$$\Delta\alpha = \Sigma^{-1}(z - \Delta\nu)$$
$$\Delta\nu = \frac{\sum_{i=0}^{n}(I + (\delta^*)^{-1}\delta)\Sigma^{-1}z + \sum_{i=0}^{n}(\alpha_i - \alpha_i^*) - \sum_{i=0}^{n}(\delta^*)^{-1}(\rho+\rho^*)}{\sum_{i=0}^{n}(I + (\delta^*)^{-1}\delta)\Sigma^{-1}1_n}$$
$$where\ z = \rho + Q((\delta^*)^{-1}(\rho+\rho^*))$$
$$and\ \Sigma = Q(I + (\delta^*)^{-1}\delta) + \delta$$

After each iteration, t is updated as $t = \frac{4n}{\lambda(C-\alpha) + \xi(C-\alpha^*) + \theta\alpha + \phi\alpha^*}$

We perform the iterations until we satisfy certain conditions as described by the IPM. The computational bottleneck is on the matrix inverse, which takes place in solving both $\Delta\nu$ and Δz. Both these inverses depend on Q which we have approximated though PICF as HH^T. So the bottleneck of the Newton Step can be sped up from $O(n^3)$ to $O(p^2n)$, and can be parallelized to $O(p^2n/m)$ using the Sherman-Morrison-Woodbury formula (Appendix B).

Depending on the nature of the training instances, only a subset of instances correspond to a non-zero $(\alpha - \alpha^*)$. These are the computed Support Vectors and will be used in the prediction phase.

6 Current Progress

I started working on this project by familiarizing myself with Support Vector Machines and Regression by consulting Elements of Statistical Learning (Trevor Hastie, 2003) as well as several web-lectures.

Following this, I looked into the 'Parallelizing Support Vector Machines on Distributed Computers' (PSVM) paper as well as source code to understand how SVM had been parallelized and identified portions of the paper and the code that could be reused for this project.

The next step I took was to propose a corresponding PSVR formulation, which took a few iterations to complete. After the correct formulation was devised, I proceeded to implement the code to do the same, while reusing parts of the PSVM code. The formulation has currently been implemented and the program converges to a solution for several datasets. This is the current state of the project.

The implementation is currently available at `https://github.com/akshayv/psvr` and is implemented in C++ using MPI.

7 Plan for next semester

I intend to experiment with several regression datasets such as SARCOS and AIMPEAK to determine the speedup achieved by using various number of processor cores while also monitoring the prediction accuracy and scalabilty of the entire process.

Depending on the time taken to complete the above tasks, I hope to also integrate the Bayesian evidence framework with the SVR algorithm which would allow prediction of a function value with an uncertainty value with associated with the prediction.

8 Appendix

8.1 A: Parallel Incomplete Cholesky factorization

ICF can approximate $Q(Q \in R^{n \times n})$ by a smaller matrix $H(H \in R^{n \times p}, p \ll n)$, i.e., $Q \approx HH^T$ Our row-based parallel ICF (PICF) works as follows: Let vector v be the diagonal of Q and suppose the pivots (the largest diagonal values) are $\{i_1, i_2, ..., i_k\}$, the k^{th} iteration of ICF computes three equations:

$$H(i_k, k) = \sqrt{v(i_k)}$$

$$H(J_k, k) = \frac{(Q(J_k, k) - \sum_{j=1}^{k-1} H(J_k, j)(H(i_k, j))}{H(i_k, k)}$$

$$v(J_k) = v(J_k) - H(J_k, k)^2,$$

where J_k denotes the complement of $\{i_1, i_2, ..., i_k\}$. The algorithm iterates until the approximation of Q by $H_k H_k^T$ (measured by $trace(QH_kH_k^T)$) is satisfactory, or the predefined maximum iterations (or say, the desired rank of the ICF matrix) p is reached.

At the end of the algorithm, H is stored distributedly on m machines, ready for parallel IPM. PICF enjoys three advantages: parallel memory use ($O(np/m)$), parallel computation

$(O(p^2n/m))$, and low communication overhead $(O(p^2log(m)))$.

8.2 B: Sherman-Morrison-Woodbury formula for performing Matrix Inversion

Let us set

$$D = diag(\frac{\lambda_i}{(C - z_i)} + \frac{\theta_i}{(C + z_i)}),$$
$$Q + D = \Sigma,\ and$$
$$u_i = -Qz + y_i - \nu + \frac{1}{t}vec(\frac{1}{(C + z_i)} - \frac{1}{(C - z_i)}).$$

We notice that in the IPM process, we need to calculate Σ^{-1}. An interesting observation is that parallelizing $\Sigma^{-1}u$ (or $\Sigma^{-1}.1_n$) is simpler than parallelizing Σ^{-1}. Let us explain how parallelizing $\Sigma^{-1}u$ works, and parallelizing $\Sigma^{-1}.1_n$ can follow suit. According to SMW (the Sherman-Morrison-Woodbury formula), we can write $\Sigma^{-1}u$ as

$$\Sigma^{-1}u = (D + Q)^{-1}u(D + HH^T)^{-1}u$$
$$= D^{-1}u - D^{-1}H(I + H^TD^{-1}H)^{-1}H^TD^{-1}u$$
$$= D^{-1}u - D^{-1}H(GG^T)^{-1}H^TD^{-1}u.$$

$\Sigma^{-1}u$ can be computed in four steps:

1. Compute $D^{-1}u$. D can be derived from locally stored vectors, as part of IPM. $D^{-1}u$ is a $n \times 1$ vector, and can be computed locally on each of the m machines.

2. Compute $t_1 = H^TD^{-1}u$. Every machine stores some rows of H and their corresponding part of $D^{-1}u$. This step can be computed locally on each machine. The results are sent to the master (which can be a randomly picked machine for all PIPM iterations) to aggregate into t_1 for the next step.

3. Compute $(GG^T)^{-1}t_1$. This step is completed on the master, since it has all the required data. G can be obtained from H in a straightforward manner as shown in SMW. Computing $t_2 = (GG^T)^{-1}t_1$ is equivalent to solving the linear equation system $t_1 = (GG^T)t_2$. PIPM first solves $t_1 = Gy_0$, then $y_0 = G^T t_2$. Once it has obtained y_0, PIPM can solve $G^T t_2 = y_0$ to obtain t_2. The master then broadcasts t_2 to all machines.

4. Compute $D^{-1}Ht_2$ All machines have a copy of t_2, and can compute $D^{-1}Ht_2$ locally to solve for $\Sigma^{-1}u$.

Similarly, $\Sigma^{-1}.1_n$ can be computed at the same time. Once we have obtained both, we can solve for $\Delta\nu$.

9 References

[1] Smola A., Scholkopf B. *A Tutorial on Support Vector Regression* 2004: Statistics and computing 14.3: 199-222.

[2] Boyd S. *Convex optimization* 2004: Cambridge University Press.

[3] Hastie T. *Elements of Statistical Learning* 2009: Vol. 2. No. 1. New York: Springer.

[4] Wu G., Chang E., Chen Y.K., Hughes C. *Incremental Approximate Matrix Factorization for Speeding up Support Vector Machines* 2006: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM.

[5] Chen J., Cao N., Low K. H., Ouyang R., Tan C. K., Jaillet P. *Parallel Gaussian Process Regression with Low-Rank Covariance Matrix Approximations* 2013: arXiv preprint arXiv:1305.5826

[6] Graf, H. P., Cosatto, E., Bottou, L., Dourdanovic, I., Vapnik, V. *Parallel Support Vector Machines: The Cascade SVM* 2005: In Advances in neural information processing systems 17, 521-528

[7] Chang E., Zhu K., Wang H., Bai H. *PSVM: Parallelizing Support Vector Machines on Distributed Computers* 2007: In Proc. NIPS.

[8] Mehrotra S. *On the implementation of a primal-dual interior point method* 1992: SIAM J. Optimization, 2.

[9] Smola A. *Regression Estimation with Support Vector Learning Machines* 1996: Master's thesis, Technische Universit at Munchen.

[10] Woodsend K. *Using Interior Point Methods for Large-scale Support Vector Machine training* 2010

[11] Bekkerman, R., Bilenko, M., and Langford, J. *Scaling up Machine Learning: Parallel and Distributed Approaches* 2011: Cambridge Univ. Press, NY.

[12] H. X. Zhao, F. Magoules. *New parallel support vector regression for predicting building energy consumption*. IEEE Symposium Series on Computational Intelligence (SSCI 2011), Apr 2011, Paris, France. IEEE Computer Society