# Low memory profile cross-validation for least squares support vector machines using block recursive inverse

Ormazabal L. do Nascimento*, Samuel Xavier-de-Souza*, Adrião D. D. Neto* and Iria C. S. Cosme*†
*Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil
ormazabalnascimento@gmail.com, samuel@dca.ufrn.br, adriao@dca.ufrn.br
†São Gonçalo do Amarante Campus
Instituto Federal do Rio Grande do Norte,
São Gonçalo do Amarante, RN, Brazil
iria.cosme@ifrn.edu.br

***Abstract*—The Least Squares Support Vector Machine (LS-SVM) requires, as part of its cross-validation algorithm, the calculation of the inverse of the kernel matrix. Although the whole inverse is computed only blocks of elements close to its main diagonal are used in the cross-validation. We present a solution for the computation of the needed inverse blocks as a way to decrease the memory footprint of the algorithm.**

## I. Introduction

There has never been as much information as there is in recent years. Originated from mobile devices, GPS, social networks, among other sources, it is estimated that each day 2.5 quintillion of data bytes are produced [1]. Extracting knowledge from these huge databases is essential for the advancement of science and technology and learning methods and techniques are excellent tools for this purpose.

Support Vector Machine (SVM) [2] is a statistical learning machine that has in its main feature the obtainment of the hyperplane of optimal separation of the analyzed data. Due to its high computational cost and difficult to implement, the Least Square Support Vector Machine (LS-SVM) [3] was developed. In this variation of the SVM, instead of the Quadratic Programming (QP), it's necessary to solve a system of linear equations.

The generalization of the model is an essential step in the learning of the machine. The more precise it is, the better the predictions of the machine will be. In order to evaluate its generalization capacity, the Cross-Validation (CV) is used. In LS-SVM, this technique is directly related to the selection of parameters. For each set of tested parsers, a new kernel matrix is generated and stored. In an input data set with $n$ points, storing this matrix requires $O(n^2)$ memory. For large volumes of data, lack of memory becomes a recurring problem.

In the $l$-fold CV algorithm, proposed by [4], in each of the $l$ iterations, only sub-matrices of the inverse of the kernel matrix are necessary for occlusion, each requesting (approximately) $O(\frac{n^2}{l^2})$ memory. Based on this premise, it is not necessary to store the complete kernel matrix, but only the sub-matrix used in each iteration.

The Block Recursive Inversion (BRI) algorithm [5] will be presented as a solution to the problem. In this algorithm, the input data is used to calculate the inverse of the kernel matrix, but only the necessary sub-matrix will be returned and stored in the current iteration of the $l$-fold.

The remainder of this paper is structured as follows. In Section II, the necessary theory for this paper will be presented. In section III, the tests and results obtained will be presented. Section IV, it concludes this paper with a summary of these improvements.

## II. Preliminaries

### A. Least Squares Support Vector Machine

Given a training set $\{x_i, y_i\}_{i=1}^n$ with binary output $y = \{-1, +1\}$, according to [3], its classifier in the primal space has the following form:

$$y(x) = sign[w^T \phi(x) + b] \tag{1}$$

where $\phi(x)$ represents the kernel function. If the data are not linearly separable in the input space, the kernel function is the one responsible for mapping them to a higher dimension space, called feature space, in which there is a high probability that these data are linearly separable. The LS-SVM has the following formula in the primal problem:

$$\min_{w,b,e} J(w, e) = \frac{1}{2} w^T w + \gamma \frac{1}{2} \sum_{i=1}^{n} e_i^2 \tag{2}$$

limited to $w^T \phi(x_i) + b = y_i - e_i, i = 1, 2, ..., n$

Using Lagrange multipliers, the minimization problem in the primal space can be solved by solving a linear system called the dual problem:

$$\begin{bmatrix} 0 & 1_n^T \\ 1_n & K + \frac{1}{\gamma} I_n \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix} \tag{3}$$

$$K_{ij} = \phi(x_i)^T \phi(x_j) = K(x_i, x_j) \tag{4}$$

K represents the kernel matrix, which depends on a kernel function to be generated. In this work, Radial Base Function (RBF) were used. One advantage of using this function is that it depends on the choice of a parameter. Thus, the parameter selection algorithm needed to deal with only two parameters: The kernel parameter $\sigma$ and the regularization parameter $\gamma$.

$$K(x, x^{'}) = exp\Big( - \frac{(x - x^{'})^2}{2\sigma^2} \Big) \tag{5}$$

The classifier in the dual space has the following form:

$$y(x) = sign \left[ \sum_{i=1}^{n} \alpha_i K(x_i, x) + b \right] \tag{6}$$

Thus, finding $\alpha$ e b solves the problem.

### *B. Inverse Matrix in Blocks*

Calculating the inverse of a matrix is a computationally intense task, as well as the storage of very large matrices may exceed the computer memory limit. The computation of the inverse matrix in blocks arises as a solution that enables a high memory economy and increases the computer's ability to solve larger problems, previously impossible, without the expansion of memory [5].

In the algorithm proposed in [5], the inverse of an M square matrix is obtained by calculating the Schur complement of each submatrix belonging to it. Thus, four similar formulas have been defined for calculating it, one for each submatrix.

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{7}$$

and your respectives Schur complements

$$M/A = D - CA^{-1}B \tag{8}$$

$$M/B = C - DB^{-1}A \tag{9}$$

$$M/C = B - AC^{-1}D \tag{10}$$

$$M/D = A - BD^{-1}C \tag{11}$$

The algorithm works recursively. As long as there are submatrices of dimension $k > 2$, they will be partitioned into four submatrices of the same order $(k-1)$, called frames. This step is called Forward Recursive Procedure. This partition follows some rules described above.

Starting from matrix M, its partition would result in four frames. The frame $M\rangle_A$ results from the removal of the lowermost block row and rightmost block column; $M\rangle_B$ results from the removal of the lowermost block row and the leftmost block column; $M\rangle_C$ results form the removal of the uppermost block row and rightmost block column and $M\rangle_D$ results from the removal of the uppermost block row and leftmost block column of M.

$$M\rangle_A = \begin{bmatrix} M_{11} & \cdots & M_{1(k-1)} \\ \vdots & \ddots & \vdots \\ M_{(k-1)1} & \cdots & M_{(k-1)(k-1)} \end{bmatrix} \tag{12}$$

$$M\rangle_B = \begin{bmatrix} M_{12} & \cdots & M_{1k} \\ \vdots & \ddots & \vdots \\ M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix} \tag{13}$$

$$M\rangle_C = \begin{bmatrix} M_{21} & \cdots & M_{2(k-1)} \\ \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{k(k-1)} \end{bmatrix} \tag{14}$$

$$M\rangle_D = \begin{bmatrix} M_{22} & \cdots & M_{2k} \\ \vdots & \ddots & \vdots \\ M_{k2} & \cdots & M_{kk} \end{bmatrix} \tag{15}$$

Then rows and columns of each block will be exchanged until the $M_{22}$ block is moved to its original place. In this case, in frame $M\rangle_A$ there are no modifications; In $M\rangle_B$ it is necessary to exchange the two leftmost columns; In $M\rangle_C$ it is necessary to exchange the two uppermost rows; In $M\rangle_D$ the two leftmost columns and the two uppermost rows are exchanged.

$$M\rangle_B = \begin{bmatrix} M_{13} & M_{12} & \cdots & M_{1k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{33} & M_{32} & \cdots & M_{3k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(k-1)3} & M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix} \tag{16}$$

$$M\rangle_C = \begin{bmatrix} M_{31} & M_{32} & \cdots & M_{3(k-1)} \\ M_{21} & M_{22} & \cdots & M_{2(k-1)} \\ M_{41} & M_{42} & \cdots & M_{4(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{k(k-1)} & \end{bmatrix} \tag{17}$$

$$M\rangle_D = \begin{bmatrix} M_{33} & M_{32} & \cdots & M_{3k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{43} & M_{42} & \cdots & M_{4k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k3} & M_{k2} & \cdots & M_{kk} \end{bmatrix} \tag{18}$$

These two steps continue to occur in the resulting frames, until all frames have dimension $k = 2$. After this, the Backward Recursive Procedure is started. In this step, the Schur complement is calculated on each frame, applying the formula appropriate to the position of the frame in question.

Each frame of two dimensions, after calculating its Schur complement, will result in frames of one dimension. Then four frames are joined together, forming a new frame of two

dimension, and this step is reapplied. In the last step, only the Schur complement of the upper left frame is calculated and returned to $l$-fold.

It is important to note that through swaps in rows and columns, any block can be positioned in the upper left corner of M. Thus, it will be used of such artifice to move the block sought for that corner and, finally, stores it for use in $l$-fold.

### C. Cross-Validation

The cross-validation is responsible for the generalization of the machine, and it is connected to the number of parameters. For each set of pairs tested, this technique is applied, so that at the end of the selection, the parameters that took the machine have the best generalization ability are selected.

**Algorithm 1** Efficient Cross-Validation

**Input:** $K$ (Kernel matrix), $l$ (number folds), $y$ (response)

$K_\gamma^{-1} \leftarrow inv(K + \frac{1}{\gamma}I), \quad d \leftarrow 1_n^T K_\gamma^{-1} 1_n$

$C = K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1}$

$\alpha = K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1} y$

$n_k \leftarrow size(y)/l, \quad y^{(k)} \leftarrow zeros(l, n_k)$

**for** $k \leftarrow 1, k \leq l$ **do**

  $C_{kk}\beta_{(k)} = \alpha_{(k)}$

  $y^{(k)} \leftarrow sign[y_{(k)} - \beta_{(k)}]$

  $k \leftarrow k + 1$

$error \leftarrow \frac{1}{2} \sum_{k=1}^{l} \sum_{i=1}^{n_k} |y_i - y^{(k,i)}|$

**Output:** $error$

In the $l$-fold algorithm [6], the matrix C is calculated, and at each iteration of the algorithm, a block $C_{kk}$ belonging to the principal diagonal of C is used as validation. As already seen, the matrix C consumes $O(n^2)$ of memory, but each block $C_{kk}$ consumes only (approximately) $O(n^2/l^2)$ of memory.

$$\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{11}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1l}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} = K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1} \quad (19)$$

If the inverse of the kernel matrix were calculated, there would be a concern to perform intermediate computations to obtain the matrix C. However, it was observed that by calculating the inverse of matrix A, we also obtain the matrix C, coming in the form of a sub-matrix $A^{-1}$. Thus, it is only necessary to exclude the first row and column of $A^{-1}$.

$$A = \begin{bmatrix} 0 & 1_n^T \\ 1_n & K_\gamma \end{bmatrix} A^{-1} = \begin{bmatrix} C & C_1^T & C_2^T & \cdots & C_l^T \\ C_1 & C_{11} & C_{12} & \cdots & C_{1l} \\ C_2 & C_{11}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_l & C_{1l}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} \quad (20)$$

The BRI algorithm [5] will be used to obtain only the block $C_{kk}$ used in the $l$-fold iteration. Using the data matrix X, the matrix A will be calculated at run time, as well as its inverse C, and only returning to the machine the block $C_{kk}$ in question. It will be seen that in this way it will only be necessary to store the matrix X, which in a space of low dimensionality is much smaller than the matrix K, and the matrix $C_{kk}$ generated from X.

## III. Experimental Results

In this section the experimental results will be presented. All tests were run on a laptop with 1.7GHz Intel Core i3 CPU and 4G of RAM with Matlab R2106b. The tests were applied to the $l$-fold algorithm with 3, 4, 5 and 6 folds, using the RBF kernel. Among the kernel functions, RBF requires the least number of parameters, only two, and so it was chosen. Tests of memory consumption and time spent were performed. Leave-one-out tests were not performed. They are very costly, especially when involving lots of data. However, the conclusions obtained in this paper also apply to this special case of $l$-fold.

TABLE I
Time Comparison

| Dataset size | Standard | BRI l=3 | l=4 | l=5 | l=6 |
|---|---|---|---|---|---|
| 1000 | 0.94s | 2.06s | 6.11s | 16.2s | 54.33s |
| 2000 | 7.88s | 8.55s | 21.43s | 63.35s | 205.17s |
| 3000 | 24.04s | 26.5s | 62.26s | 157.46s | 495.33s |
| 4000 | 58.83s | 58.95s | 136.25s | 344.33s | 1017.84s |
| 5000 | 111.18s | 110.92s | 243.82s | 652.65s | 1866.21s |
| 6000 | 202.17s | 193.25s | 411.64s | 1089.6s | 3112.96s |
| 7000 | 340.56s | 275.87s | 638.36s | 1999.1s | 4787.66s |
| 8000 | 501.95s | 425.91s | 961.04s | 2641.1s | 6959.88s |
| 9000 | 899.15s | 570.17s | 1216.54s | 3715.7s | 9747.36s |

TABLE II
Memory Comparison

| Dataset size | Standard | BRI l=3 | l=4 | l=5 | l=6 |
|---|---|---|---|---|---|
| 1000 | 23003.34Kb | 872.39Kb | 489.66Kb | 312.64Kb | 286.94Kb |
| 2000 | 91523.9Kb | 4048.69Kb | 1954.77Kb | 1251.15Kb | 396.77Kb |
| 3000 | 209443.21Kb | 6205.81Kb | 3813.66Kb | 2819Kb | 1761.59Kb |
| 4000 | 374902.87Kb | 13882.06Kb | 6877.48Kb | 4979.64Kb | 2538.52Kb |
| 5000 | 585905.65Kb | 20282.06Kb | 11254.72Kb | 7831.75Kb | 4972.66Kb |
| 6000 | 926485.3Kb | 28526.56Kb | 16682.95Kb | 11244.79Kb | 7806.03Kb |
| 7000 | 1148460.61Kb | 42523.7Kb | 23929.53Kb | 15347.76Kb | 10631.22Kb |
| 8000 | 1837536.78Kb | 55530.8Kb | 31273.47Kb | 20063.33Kb | 13632.33Kb |
| 9000 | 3307564.8Kb | 70355.75Kb | 39574.3Kb | 25383.62Kb | 17690.23Kb |

Table 1 shows the time comparison between the standard algorithm and the modification with BRI algorithm. As can be seen, the standard algorithm can be faster during most tests. Something that must be noticed in the standard algorithm, the inverse matrix does not need to be calculated more than once. During the iterations of the $l$-fold, only read operations are done on the matrix C. The block recursive inverse needs to calculate $l$ blocks $C_{kk}$, so it executes $l$ times throughout the algorithm. The table shows the total time spent, ie. time of computation C in the standard and time of computation $C_{kk}$ $l$ times for BRI algorithm.

Table 2 shows the memory consumption comparison between this algorithms. Note that, in the BRI algorithm, the memory spends on all calls is constant, equivalent to the space

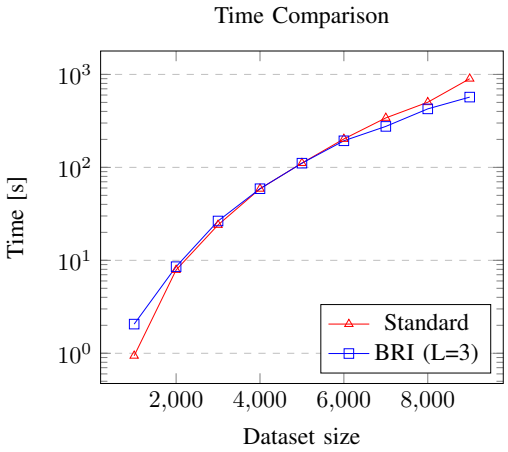


Fig. 1. Total time comparison between standard inverse and block recursive inverse with 3 folds

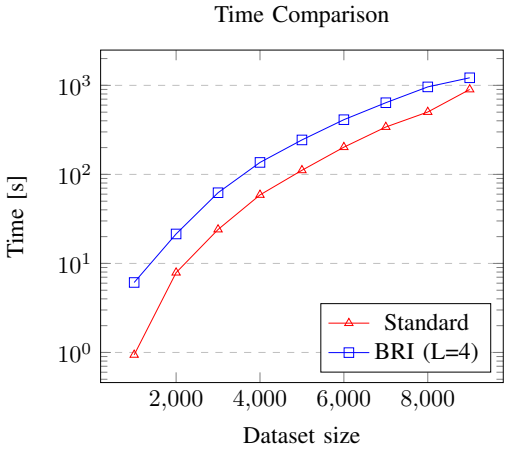


Fig. 2. Total time comparison between standard inverse and block recursive inverse with 4 folds

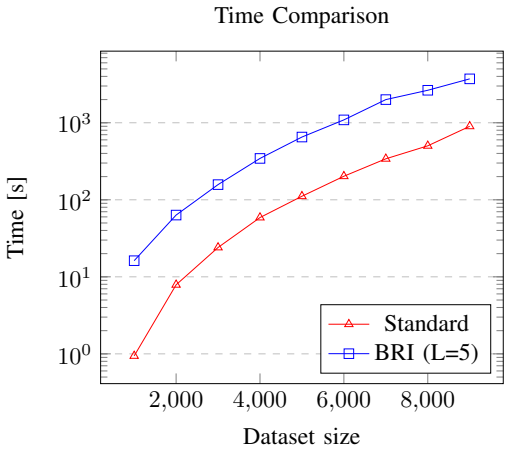


Fig. 3. Total time comparison between standard inverse and block recursive inverse with 5 folds

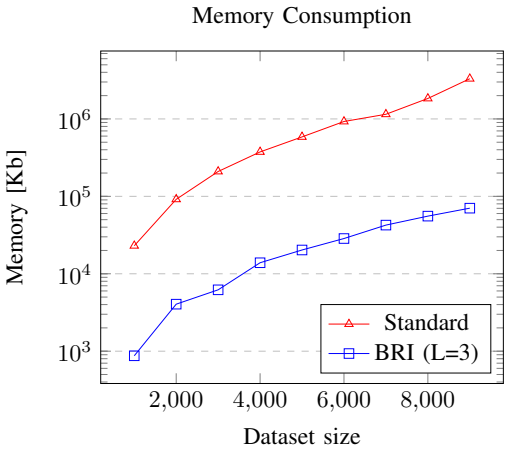


Fig. 4. Memory consumption comparison between standard inverse and block recursive inverse with 3 folds

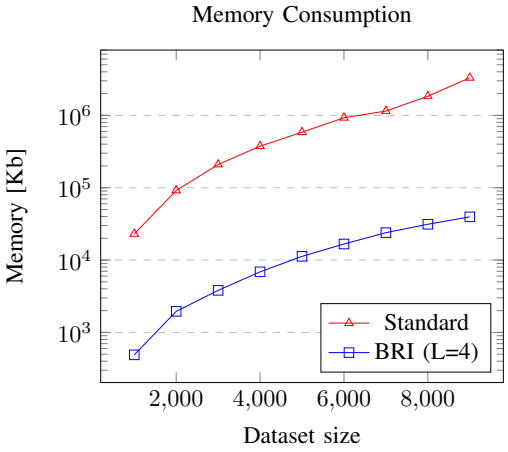


Fig. 5. Memory consumption comparison between standard inverse and block recursive inverse with 4 folds

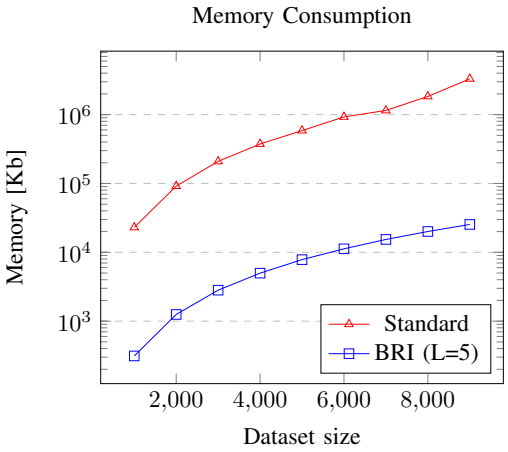


Fig. 6. Memory consumption comparison between standard inverse and block recursive inverse with 5 folds

allocated to the block $C_{kk}$. In the standard algorithm, only one call is necessary, to allocate the full matrix C.

In the 3-folds test it is noticed that the times between the two algorithms are quite identical, with the standard algorithm obtaining the best time in the tests in which $n < 5000$. It is noticed that as the size of the problem increases, the BRI algorithm improves its time, reducing the difference between it and the standard, until from $n = 5000$ it can guarantee the best times. These results demonstrate a tendency that, for high values of $n$, the BRI algorithm can perform the work in a more agile way than the standard algorithm. In comparison in memory the block recursive inverse shows its full potential. There is an average saving of 95 % of memory compared to the standard.

Analyzing all the results of the time comparison, it is verified that the increase of $l$ causes an increase of time in the BRI algorithm. This is already expected, because in this way more calls are made to the algorithm, accentuating the total time difference compared to the standard, which makes only one call. However, as already mentioned, as the size $n$ of the problem grows, this time difference falls. In problems with large volumes of data, this difference will be subtle even with the increase of $l$, being able to reach the case already mentioned with 3-folds in which the BRI algorithm managed to be faster than the standard.

In the comparison of memory, it will be seen that as $l$ increases, the smaller the block size $Ckk$, and therefore less memory is spent by the BRI algorithm. As already seen, the block $Ckk$ consumes $O(n^2/l^2)$ of memory. With $n$ constant and $l$ growing, this figure falls more and more. With 3-folds, this saving was 95%; Increasing to 6-folds resulted in savings of 99%. These results show that not so powerful machines, once considered insufficient to compute large volumes of data, will now have their capacities expanded.

In the hardware described, the standard algorithm had problems compiling datasets above 10000 points. It resulted in out of memory. With the block recursive inverse, larger problems could be computed. Thus, it has been shown that increasing the number of folds $l$ reduces memory expenditure, but produces longer times for all computing, however, increasing problems produce better and better times compared to the standard algorithm.

It is up to the researcher to make a good analysis of the size of the problem he is dealing with and the hardware of the computer on which he will be executed. This study will lead you to select a good value for the number of folds. However, the contribution of this technique is undeniable, especially for problems involving large volumes of data, in which time and memory are saved when compared to the standard approach.

## IV. Conclusion

In this paper, we have presented a more efficient way, in terms of memory usage, for $l$-fold CV of LS-SVM. In order to calculate the inverse of matrix $A$ and to store it in memory for later reading of the blocks of interest, we used the BRI algorithm, proposed in [5]. This algorithm calculates

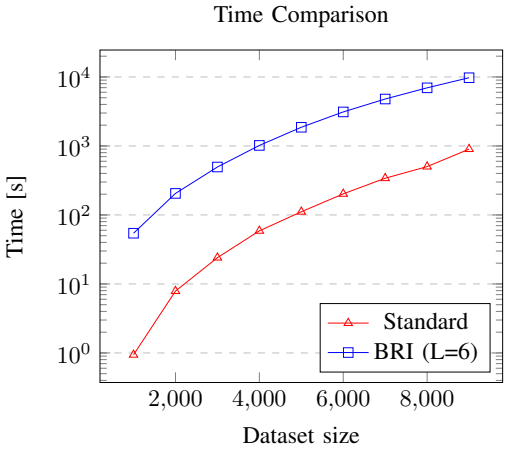


Fig. 7. Total time comparison between standard inverse and block recursive inverse with 6 folds

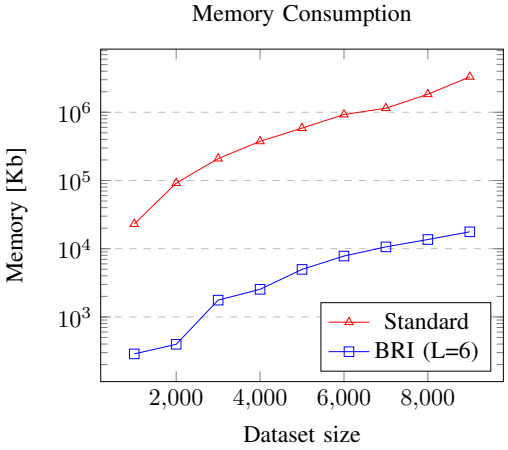


Fig. 8. Memory consumption comparison between standard inverse and block recursive inverse with 6 folds

the matrix $A^{-1}$ as from the matrix $X$, at run time, and stores the block $C_{kk}$ of interest at that time.

By means of tests, it was seen that the memory savings obtained using this method was over 90%, compared to the standard method. Linked to this, although the execution time required get larger, it was observed that this difference of times, in comparison to the standard algorithm, tends to decrease as the size of the input data increases.

## References


[1] R. Devakunchari, *Analysis on big data over the years*, International Journal of Scientific and Research Publications, vol. 4, no. 1, pp. 383, 2014.

[2] V. Vapnik, *The Nature of Statistical Learning Theory*, Springer, New York, 1995.

[3] J. Suykens and J. Vandewalle, *Least squares support vector machine classifiers*, Neural Process, Lett. 9, pp. 293-300, 1999.

[4] S. An, W. Liu, S. Venkatesh, *Fast cross-validation algorithms for least squares support vector machine and kernel ridge regression*, Pattern Recognition. Lett. 40, pp. 2154-2162, 2007.

[5] I. C. S. Cosme, I. F. Fernandes, J. L. de Carvalho, S. Xavier-de-Souza, *Block Recursive Matrix Inverse*, [Online] Available: https://arxiv.org/abs/1612.00001 , 2016.

[6] R. Edwards, H. Zhang, L. Parker, J. New, *Approximate l-fold Cross-Validation with Least Squares SVM and Kernel Ridge Regression*, Machine Learning and Applications (ICMLA), 2013 12th International Conference on, Vol. 1. IEEE, pp. 58–64, 2013.