Parallelization of the Nearest-Neighbour Search and the Cross-Validation Error Evaluation for the Kernel Weighted k-nn Algorithm Applied to Large Data Sets in Matlab

Gines Rubio, Alberto Guillen, Hector Pomares, Ignacio Rojas
Dept of Computer Architecture and Technology, University of Granada
C/ Periodista Daniel Saucedo sn, 18071 Granada, Spain
grubio@atc.ugr.es aguillen@atc.ugr.es hector@atc.ugr.es ignacio@atc.ugr.es

Ben Paechter
Centre for Emergent Computing
School of Computing
Napier University
b.paechter@napier.ac.uk

Peter Glösekötter
C. I. Torres-Ceballos
Dept. of Electrical Engineering and CS
University of Applied Sciences Muenster
peter.glosekotter@fh-muenster.de citorres@fh-muenster.de

ABSTRACT
The kernel weighted k-nearest neighbours (KWKNN) algorithm is an efficient kernel regression method that achieves competitive results with lower computational complexity than Least-Squares Support Vector Machines and Gaussian Processes. This paper presents the parallel implementation on a cluster platform of the sequential KWKNN implemented in Matlab. This implies both the parallelization of the k nearest-neighbour search and the evaluation of the cross-validation error on a large distributed data set. The results demonstrate the good performances of the implementation.


1. INTRODUCTION
Kernel methods such as Gaussian Process Regression [1], [2] and Least Square Support Vector Machines (LS-SVMs) [3] have been successfully applied to regression and function approximation problems. Although they show good performances and obtain very accurate results, they suffer from inherent drawbacks such as the difficulty in the selection of the kernel function, the fact that the optimization of the kernel parameters is computationally intensive (they require the evaluation of some cross validation procedure or some Bayesian criteria with a complexity of $O(N^3)$, where $N$ is the number of training points), and the size of the generated models, which could be huge because they include all training data.

Instead of using Gaussian Processes or LS-SVMs, the kernelized version of the weighted k-nearest neighbour (KWKNN) can be used given its good performances and lower computational costs. Nonetheless, when there is a large number of samples (several tens of thousands), a common personal computer is not practical anymore in terms of computational costs and memory requirements. Thus, the parallelization of the algorithm is desirable in a way that data and computation are distributed over several nodes, e.g. on a cluster platform. There are examples in literature of parallel implementations of k-nearest neighbours retrieval such as [4] and [5].

The KWKNN algorithm is implemented in Matlab, using several libraries (Matlab toolboxes, for kernels management for instance) and integrated into a particular framework for experimentation on functional approximation. The re-implementation of all the software in C/C++ (or another language) for its parallelization, would have been an appreciable effort, and also would have led to lose several benefits of the use of Matlab and the particular framework. So, the parallel version of the algorithm is implemented directly in Matlab, using a particular toolbox for using MPI [6].

This paper presents the parallel implementation of the kernel weighted k-nn for function approximation (PKWKNN). The rest of the paper is organized as...
follows: Section 2 will briefly introduce the weighted k-nearest neighbours algorithm, as well as the involved modifications for using kernel based distance instead of Euclidean distance and practical issues associated to the setting of its parameters. Section 3 will present the parallel implementation developed for the algorithm, Section 4 will demonstrate its good performance, and, finally, in Section 5, some conclusions will be drawn.

2. KERNEL WEIGHTED K NEAREST NEIGHBOURS (KWKNN)

2.1. Weighted k Nearest Neighbours Algorithm for Regression

The k nearest neighbours (k-nn) is a simple algorithm that is usually applied to classification tasks although some adaptations of this algorithm have been applied also to regression problems [7], [8].

Given a set of function samples \((\vec{x}_i, y_i), i = 1, \ldots, N\), where \(\vec{x}_i \in \mathbb{R}^d\) and \(y_i \in \mathbb{R}\), let \(X = \{\vec{x}_i\}, i = 1, \ldots, N\) be a set where the distance function \(D\) has been defined over it. The weighted \(K\) nearest neighbor algorithm for regression is able to compute the output for a given new input \(\vec{x}\) as:

\[
f(\vec{x}) = \hat{y} = \frac{\sum_{i=0}^{k} y_i^\prime w^{(k)} (\vec{x}, x_i^\prime)}{\sum_{i=0}^{k} w^{(k)} (\vec{x}, x_i^\prime)}
\]

where \(x_i^\prime\) is the \(i\)-th nearest training point to a sample \(\vec{x}\) according to the distance function \(D\), \(y_i^\prime\) its corresponding output and \(w^{(k)} (\vec{x}, x_i^\prime)\) is the weight assigned to it. As the formulation shows, the method is based on the values of a predefined distance function and it is independent of any possible input structure. Therefore, the weighted k-nn can be interpreted as a kernel method where the distance function represents the kernel. The most commonly used distance measure is the Euclidean.

2.2. Distance in Feature Space Defined by a Kernel Function

A kernel function is defined as a scalar product of the projected input \((\Phi)\) from the input space to a feature space [9]. Thus a kernel can be defined as:

\[
k(\vec{x}, x') = \langle \Phi(\vec{x}), \Phi(x') \rangle
\]

The quadratic norm of two points in feature space can be written in terms of the values of the kernel function:

\[
\|\Phi(\vec{x}) - \Phi(x')\|^2 = \langle \Phi(\vec{x}), \Phi(\vec{x}) \rangle + \langle \Phi(x'), \Phi(x') \rangle - 2 \langle \Phi(\vec{x}), \Phi(x') \rangle
\]

\[
= k(\vec{x}, \vec{x}) + k(x', x') - 2k(\vec{x}, x')
\]

\[
= \sqrt{k(\vec{x}, \vec{x}) + k(x', x') - 2k(\vec{x}, x')}
\]

So we only need to substitute in the Equation (2) the quadratic terms of the distance by the expression of Equation (4) to get the kernel-distance-based version of the weighted k-nn algorithm for regression (KWKNN).

2.3. Cross-Validation Error Evaluation for KWKNN

Cross-validation is a robust method to avoid over-fitting the training samples in a modelling problem. To compute the \(l\)-fold cross validation error of a model we must

1) divide the training data set into \(l\) subsets
2) for each of the \(l\) subsets
   a) create a model that uses as training set the remaining training data
   b) use the created model to predict the selected subset and compute the approximation error

The mean or average of the \(l\) (for each subset) errors is the \(l\)-fold cross-validation error. An extreme case is the Leave-One-Out error, when the number of subsets is equal to the size of the training data set, so each has 1 element. The implementation of the \(l\)-fold cross validation of KWKNN for \(N\) data samples implies \(l\) evaluations of the distances between sample pairs, whose complexity is

\[
O \left( \left( N - \frac{N}{l} \right) \cdot \frac{N}{l} \right)
\]

and \(\frac{N}{l}\) (for each sample) re-orderings of the distances to find the \(k + 1\) nearest neighbours, being the most efficient algorithm of complexity:

\[
O \left( \left( N - \frac{N}{l} \right) \cdot \log(N - \frac{N}{l}) \right)
\]
So, the evaluation of the $l$-fold cross validation of KWKNN has a total complexity:

$$O \left( l \cdot \left( N - \frac{N}{p} \right) \cdot \frac{N}{p} \cdot (1 + \log(N - \frac{N}{p})) \right) = O \left( N^2 - \frac{N^2}{p} + N \cdot \left( N - \frac{N}{p} \right) \log(N - \frac{N}{p}) \right)$$

(8)

For the particular case of the Leave-One-Out error, we have:

$$O \left( N \cdot ((N - 1) + (N - 1) \cdot \log(N - 1)) \right) = O \left( N^2 - N + N \cdot (N - 1) \cdot \log(N - 1) \right) \simeq$$

$$O \left( N^2 - N + N^2 \cdot \log(N) \right)$$

2.4. Optimization of the Parameters of the Kernel Function

For each kernel, its parameters must be set for each particular data. In order to obtain the parameter values for any arbitrary kernel, the parameters can be optimized by means of the cross-validation error of the KWKNN. The $l$-fold cross-validation ($l$-CV), and particularly the Leave-One-Out (LOO), error of KWKNN can be derived very efficiently from the distances between training samples. In our case, the kernel parameters are optimized with respect to the cross-validation error using a version of the Variable Neighbourhood Search (VNS) [10]. Since it is not possible to differentiate the $l$-CV error function w.r.t the kernel parameters, VNS is used instead of the classic conjugate gradient.

3. PARALLELIZATION OF KWKNN FOR LARGE DATA SETS

The main issue when working with large data sets is the computational costs and the memory requirements that come along with the application of the KWKNN and the calculation of the $l$-CV error. With a parallelized methodology however (using PKWKNN), the samples of the training data are distributed among the different processing nodes, see Figure 1. For $N$ samples and $p$ processes, each process has $\frac{N}{p}$ samples (in case of not being divisible $N$ between $p$, processes $0, \cdots, p-2$ hold $\frac{N}{p}$ samples, $\frac{N}{p}$ rounded to the nearest upper natural, and process $p-1, N - (p-1) \cdot \left( \frac{N}{p} \right)$ samples).

Given a new input point $x$, PKWKNN computes its output $\hat{y}$ as shown in Algorithm 1. First, each node computes the $k + 1$ nearest point to $x$ of their local data of $x$ in terms of a kernel based distance. Then, all nodes send to node 0 (that participates in the operation) their local $k + 1$ nearest-neighbours distances to $x$ and associated outputs. Finally, node 0 computes the $k + 1$ nearest point to $x$ of the global data using the information gathered in the previous step, and applies the Equation (1) to compute $\hat{y}$, the output of $x$. Without taking into account communications, the complexity is given by finding the $k + 1$ nearest neighbours to the local samples of each node, and, finally, the global $k + 1$ nearest neighbours is retrieved from the $p$ local ones. That gives, see Algorithm 1:

$$O \left( \left[ \frac{N}{p} \right] \cdot \log(\left[ \frac{N}{p} \right]) + p(k + 1) \cdot \log(p(k + 1)) \right)$$

(10)

and if we consider $p(k + 1) < \left[ \frac{N}{p} \right]$ the term corresponding to the computation of global $k + 1$ nearest neighbours can be neglected:

$$O \left( \left[ \frac{N}{p} \right] \cdot \log(\left[ \frac{N}{p} \right]) \right)$$

(11)

In most applications, the $k$ value is lower than 10. If we consider $N$ about 30000, it implies that the influence of the second term will be appreciable with $k = 10$ for $p \approx 55$ processors. It is worth to recall that communications are not considered in these complexity expressions.

Given the samples distributed among $p$ processes, the arbitrary $l$-fold CV error evaluation can be quite complex in terms of communications, but there are two very efficient and simple cases: $p$-fold CV and LOO error. The Algorithm 2 illustrates both cases in pseudo-code.

4. EXPERIMENTS

The third time series, Figure 2, proposed in the ESTSP 20082 competition (31614 values) is the large data set that was used in our experiments, based on the work [11]. This series was specially chosen by the organizers to penalize the use of LS-SVM or Gaussian Processes. We used as inputs the sample indices, $X = 1, \cdots, 31614$, and as output the time series values $Y = y_1, \cdots, y_{31614}$. We used the kernel of Equation (14). It is worth to recall the quite high number of parameters of the resulting kernel (i.e. distance) function, 7. For the required value $k$ of the PKWKNN, the usual value of $k = 10$ was taken.

1. each process is mapped to a processor, or node

The PKWKNN algorithm was implemented directly in Matlab. A wrapper to the Message Passing Interface (MPI) library MPICH2 [12] [6] was used, and the result compiled as a stand-alone application in order to use it in a cluster (each node with an AMD Opteron of 64 bits of 2.6 GHz processor with 2 GB of RAM). The algorithm was run on 2, 4, 8 and 16 processors (see Table 1).

It is worth to recall that the Matlab language has characteristics that does not make suitable some implementation approaches which are common in other languages, such as C. For instance, it is not recommended to use iterative structures and it is very recommended the use of matrix oriented instructions to get efficient codes. These features were taken into account in our implementation, for instance, computing in an matrix oriented mode the distances between pairs of points at

\[ k(x_i, x_j; \Theta) = k_{ratquad}(x_i, x_j; \{\theta_1, \alpha, \beta\}) + \\
 k_{7-periodic}(x_i, x_j; \{\theta_2, \theta_3\}) + \\
 k_{8736-periodic}(x_i, x_j; \{\theta_4, \theta_5\}) \] (14)

\[ \Theta = \{\theta_1, \alpha, \beta, \theta_2, \theta_3, \theta_4, \theta_5\} \]

The PKWKNN algorithm was implemented directly in Matlab. A wrapper to the Message Passing Interface (MPI) library MPICH2 [12] [6] was used, and the result compiled as a stand-alone application in order to use it in a cluster (each node with an AMD Opteron of 64 bits of 2.6 GHz processor with 2 GB of RAM). The algorithm was run on 2, 4, 8 and 16 processors (see Table 1).

It is worth to recall that the Matlab language has characteristics that does not make suitable some implementation approaches which are common in other languages, such as C. For instance, it is not recommended to use iterative structures and it is very recommended the use of matrix oriented instructions to get efficient codes. These features were taken into account in our implementation, for instance, computing in an matrix oriented mode the distances between pairs of points at

\[ k_{\lambda-periodic}(x_i, x_j; \{\theta_1, \theta_2\}) = \theta_1 \exp \left( -2 \frac{\sin(\frac{\pi}{\lambda}(x_i-x_j))^2}{\theta_2} \right) \] (12)

\[ k_{ratquad}(x_i, x_j; \{\theta, \alpha, \beta\}) = \theta \left( 1 + \frac{||x_i-x_j||^2}{2\alpha\beta^2} \right)^{-\alpha} \] (13)
Algorithm 1 PKWKNN

INPUT: $X$, $m$ input points (in process 0)
INPUT: $P$, number of processes
INPUT: $r$, the rank of the local process
INPUT: $S^{(r)} = [X^{(r)}, Y^{(r)}]$, local to process $r$ samples (points and outputs)
INPUT: $N$, global number of samples
INPUT: $n = |X^{(r)}|$, local number of samples
INPUT: $k$, $k$-nearest-neighbours parameter
INPUT: $D^2(x, x'; \Theta)$, kernel based quadratic norm between points $x, x'$, where $\Theta$ represents the fixed values of the kernel parameters

OUTPUT: $\hat{Y}$, the outputs computed for each point in $X$ (in process 0)

(1) Broadcast of $X$ with root process 0
(2) compute $D^2(X_j, X^{(r)}_i; \Theta)$ where $j = 1, \ldots, m, i = 1, \ldots, n$
(3) compute the $k + 1$ nearest neighbours to each $X_j$ ($|\text{nn}[X_j, X^{(r)}]|, i = 1, \ldots, k + 1$ is the index of the $i$-th nearest-neighbour of $X^{(r)}$ to $X_j$)
(4) set $K^{(r)} = \{ (D^2(X_j, X^{(r)}_i; \Theta), Y^{(r)}_i) \}$, $j = 1, \ldots, m, i = 1, \ldots, k + 1$
(5) Gather in process 0 all $K^{(r)}$ in $K^{(X)}$
if $r = 0$ then
(7) compute the global $k + 1$ nearest neighbours to samples for each $X_j$ using the distance information in $K^{(X)}$
(8) compute the output $\hat{Y}_j$ for each $X_j$ using Equation (1) with the distance and output information in $K^{(X)}$
end if

Algorithm 2 $p$-fold CV/LOO Error Evaluation of PKWKNN

INPUT: $P$, number of processes
INPUT: $r$, the rank of the local process
INPUT: $S^{(r)} = [X^{(r)}, Y^{(r)}]$, local to process $r$ samples (points and outputs)
INPUT: $N$, global number of samples
INPUT: $n = |X^{(r)}|$, local number of samples
INPUT: $k$, $k$-nearest-neighbours parameter
INPUT: $D^2(x, x'; \Theta)$, kernel based quadratic norm between points $x, x'$, where $\Theta$ are the fixed values of the kernel parameters

OUTPUT: $p$-fold CV/LOO error for global sample data with PKWKNN
for $p = 1, \ldots, P$ do
(1) Broadcast of $S^{(p)}$ from process $p$
if $r \neq p$ then
(2) compute $D^2(X^{(p)}_j, X^{(r)}_i; \Theta)$
(3) compute the $k + 1$ nearest neighbours to each $X^{(p)}_j$ ($|\text{nn}[X^{(p)}_j, X^{(r)}]|, i = 1, \ldots, k + 1$ is the index of the $i$-th nearest-neighbour of $X^{(r)}$ to $X^{(p)}_j$)
(4) set $K^{(p,r)} = \{ (D^2(X^{(p)}_j, X^{(r)}_{nn[X^{(p)}_j, X^{(r)}]}; \Theta), Y^{(r)}_{nn[X^{(p)}_j, X^{(r)}]}(p) \}$
(5) Gather of $K^{(p,r)}$ with root process $p$
else
if LOO error then
(6) compute $D^2(X^{(p)}_j, X^{(p)}_i; \Theta)$
(7) compute the $k + 1$ nearest neighbours to each $X^{(p)}_j$ ($|\text{nn}[X^{(p)}_j, X^{(p)}]|, i = 1, \ldots, k + 1$ is the index of the $i$-th nearest-neighbour of $X^{(p)}$ to $X^{(p)}_j$)
(8) set $K^{(p,p)} = \{ (D^2(X^{(p)}_j, X^{(p)}_{nn[X^{(p)}_j, X^{(p)}]}; \Theta), Y^{(p)}_{nn[X^{(p)}_j, X^{(p)}]}(p) \}$
(9) Gather of $K^{(p,p)}$ at process $p$ in $K^{(p)}$
(10) compute the global $k + 1$ nearest neighbours for each sample $X^{(p)}_j$ using the distance information in $K^{(p)}$
(11) compute the output $\hat{Y}_j$ for each $X^{(p)}_j$ using Equation (1) with the distance and output information in $K^{(p)}$
compute the the sum of the squared error between $\hat{Y}_j$ and $Y^{(p)}$ as $SSE^{(p)}$
end if
end for
(12) Gather of $SSE^{(p)}$ at process 0 in $SSE$
(13) compute in process 0 the $p$-fold CV/LOO error as $\sum SSE/N$

Table 1. Number of Samples by Process for the Time Series Data

<table>
<thead>
<tr>
<th>Processors</th>
<th>Number of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0, 1 → 15907</td>
</tr>
<tr>
<td>4</td>
<td>0, 2 → 7904, 3 → 7902</td>
</tr>
<tr>
<td>8</td>
<td>0, 6 → 3952, 7 → 3950</td>
</tr>
<tr>
<td>16</td>
<td>0, 14 → 1976, 15 → 1972</td>
</tr>
</tbody>
</table>

once and sorting them instead of computing them one by one and getting them into a sorted array to get the $k + 1$ nearest neighbours.

Figure 3 shows the mean execution times of 10 runs of PKWKNN (see Algorithm 1) versus the input size for different number of processors. The implementation scales linearly with the number of processors, as can be seen in Figure 4 with the speed-up (time of sequential algorithm / time of parallel) computed on series with
Figure 3. Mean Evaluation Time (10 executions) of PKWKNN vs Number of Inputs data.

Figure 4. Speed-up Achieved by PKWKNN and PKWKNN LOO with Data of Size 10240.

Figure 5. Mean Evaluation Times of CV Error of PKWKNN vs Number of Processors.

Figure 6. Mean Evaluation Times and Standard Deviation of PKWKNN Times vs Number of Training Samples (Execution with 8 Processors)

A normal application of the method implies first the optimization of the CV error w.r.t the kernel parameters by means of the VNS procedure. In Figures 8 and 9 for 16 processors, and in Figures 6 and 7 for 8 processors, the execution times versus the training size are drawn both for 10 iterations of the optimization procedure VNS using PKWKNN LOO error (100 evaluations of the CV error) and the posterior prediction for using the same training data. The results confirm the computational complexities inferred in Section 3.

5. CONCLUSIONS

In this paper, we have shown how the kernel weighted k-nearest neighbours (KWKN) algorithm can be parallelized by distributing the samples among the different computing nodes. The kernel based distance

10240 examples\(^3\).

The CV procedure is critical for applying PKWKNN, as it is used to optimize the parameters of the kernel based distance (i.e., the parameters of a kernel). Figure 5 reveals the mean evaluation times of CV (PKWKNN) as a function of the number of processors involved. Average times of the evaluation of \( p \)-CV and LOO error differ very little (see figure 5), about 4\% in the case of 2 processors, 3\% in the case of 4 and 8 and about 2\% in the case of 16, but it represents about 53, 16, 6 and 1.5 seconds respectively. The difference in the evaluation times of the performed implementation of \( p \)-CV and LOO error with PKWKNN is practically negligible in absolute terms when an higher number of processors is used.

\(^3\) not with the complete data set because the sequential case could not be computed due to its huge memory requirements
used by the KWKNN requires a first step for the optimization of the cross validation (CV) error w.r.t the parameters of distance for the particular training data. An efficient implementation of the CV error for PKWKNN based on the same scheme of data distribution was also realized. This allows us to apply efficiently the KWKNN algorithm to large data sets, and let us chose between the $p$-fold cross-validation and the LOO error for the optimization of the kernel parameters with acceptable costs. Although it is a very classical parallelization scheme, the performances of the implementation are good, moreover if it is considered the fact that it was realized in Matlab. It is worth to recall that maintaining the original language of the sequential implementation, the work of parallelization was made faster and easier, as well as other benefits from maintaining the original framework. The implemented algorithm can be improved in several ways, and in future works some optimizations will be studied. For instance, instead of a global gather operation to get $k + 1$ nearest neighbours of each process and after computing the final results, a scheme based on a reduction in a binary tree could be used, increasing scalability and distributing computations and communications.

ACKNOWLEDGMENT

This work was carried out using the facilities of the Edinburgh Parallel Computing Centre$^4$ (EPCC) and under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme. This work has been also partially supported by the Spanish CICYT Project TIN2007-60587 and Junta Andalucia Project P07-TIC-02768.

REFERENCES


