Ingeniería y Ciencia
ISSN:1794-9165 | ISSN-e: 2256-4314
ing. cienc., vol. 14, no. 27, pp. 75–99, enero-junio. 2018.
http://www.eafit.edu.co/ingciencia

Leave-one-out Evaluation of the Nearest Feature Line and the Rectified Nearest Feature Line Segment Classifiers Using Multi-core Architectures

Ana-Lorena Uribe-Hurtado[1], Eduardo-José Villegas-Jaramillo[2] and Mauricio Orozco-Alzate[3]

Received: 03-10-2017 | Accepted: 10-04-2018 | Online: 15-06-2018

MSC: 68U01, 11C20, 65Y05 | PACS: 89.20.Ff, 89.70.Hj, 02.70.-c

doi:10.17230/ingciencia.14.27.4

Abstract

In this paper we present the parallelization of the leave-one-out test: a reproducible test that is, in general, computationally expensive. Parallelization was implemented on multi-core multi-threaded architectures, using the Flynn Single Instruction Multiple Data taxonomy. This technique was used for the preprocessing and processing stages of two classification algorithms that are oriented to enrich the representation in small sample cases: the nearest feature line (NFL) algorithm and the rectified nearest feature line segment (RNFLS) algorithm. Results show an acceleration of up to 18.17 times with the smallest dataset and 29.91 times with the largest one, using the most costly algorithm (RNFLS) whose complexity is $O(n^4)$. The paper also shows the pseudo-codes of the serial and parallel

[1] Universidad Nacional de Colombia, Sede Manizales, alhurtadou@unal.edu.co, https://orcid.org/0000-0002-1424-2372, Manizales, Colombia.
[2] Universidad Nacional de Colombia, Sede Manizales, ejvillegasj@unal.edu.co, https://orcid.org/0000-0002-7563-2913, Manizales, Colombia.
[3] Universidad Nacional de Colombia, Sede Manizales, morozcoa@unal.edu.co, https://orcid.org/0000-0002-5937-6382, Manizales, Colombia.

algorithms using, in the latter case, a notation that describes the way the parallelization was carried out as a function of the threads.

Keywords: Multi-core computing; classification algorithms; leave-one-out test.

Evaluación leave-one-out de los clasificadores de la línea de características más cercana y del segmento de línea rectificado más cercano usando arquitecturas multi-núcleo

Resumen

Presentamos en este artículo la paralelización de la prueba leave-one-out, la cual es una prueba repetible pero que, en general, resulta costosa computacionalmente. La paralelización se implementó sobre arquitecturas multi-núcleo con múltiples hilos, usando la taxonomía Flynn Single Instruction Multiple Data. Esta técnica se empleó para las etapas de preproceso y proceso de dos algoritmos de clasificación que están orientados a enriquecer la representación en casos de muestra pequeña: el algoritmo de la línea de características más cercana (NFL) y el algoritmo del segmento de línea rectificado más cercano (RNFLS). Los resultados obtenidos muestran una aceleración de hasta 18.17 veces con el conjunto de datos mas pequeño y de 29.91 veces con el conjunto de datos más grande, empleando el algoritmo más costoso —RNFLS— cuya complejidad es $O(n^4)$. El artículo muestra también los pseudocódigos de los algoritmos seriales y paralelos empleando, en este último caso, una notación que describe la manera como se realizó la paralelización en función de los hilos.

Palabras clave: Computación con múltiples núcleos; algoritmos de clasificación; prueba leave-one-out.

1 Introduction

Classification algorithms can be roughly categorized into dissimilarity-based classifiers, probabilistic classifiers and geometric classifiers [1]. The first ones assign an unlabeled object —represented as a feature vector $\boldsymbol{x}$— to the class of the most similar examples within a set of either labeled feature vectors (also known as training objects) or within models previously built from them; the second ones estimate class-conditional probability densities by using the training objects and, afterwards, assign class labels to the unlabeled ones according to the maximum posterior probabilities; the third

category of classifiers directly construct boundaries between class regions in the feature space by optimizing criteria such as classification error and maximum margin of separation between classes. The Nearest Neighbor (1-NN) rule is the paradigmatic example of the dissimilarity-based classifiers; it is very natural, intuitive for non-experts [2] and exhibits a competitive classification performance provided that a large enough training set is available. Several variants have been proposed to improve 1-NN, among them the so-called Nearest Feature Line (NFL) classifier [3] that enlarges the representational power of a training set of limited cardinality by building a linear model (a feature line) between each pair of training feature vectors of the same class.

Even though NFL effectively counteracts the weakness of 1-NN under representational limitations, it is much more costly than 1-NN and also introduces two drawbacks: *interpolation* and *extrapolation* inaccuracies (see Section 2.1.1) that are reflected into high classification errors in problems having particular data distributions and dimensionalities. Both drawbacks of NFL were diagnosed in [4], whose authors also proposed the so-called Rectified Nearest Feature Line Segment (RNFLS) classifier as a solution to the inaccuracies of NFL via two processes called *segmentation* and *rectification* of the feature lines. Unfortunately, such an improvement of NFL is paid with a significantly increased computational cost for RNFLS, turning its application too time demanding. In spite that most of the computations in RNFLS are associated to the rectification process —which is only performed during the training stage and, therefore, off-line—, current application scenarios often imply the classification of evolving streaming data [5] that, after detecting changes in their distributions, demand an automatic retraining of the classifiers in order to quickly adapt them to new data behaviors.

An additional and often ignored motivation for speeding up classification algorithms is enabling the practitioner to perform fast simulations, in such a way that varying parameters and exploring different configurations over several runs of the classifier become feasible in reasonable amounts of time. Such simulations typically require the repeated estimation of the classification performance. Among the classical performance estimation methods [6], the so-called *leave-one-out* method is very often preferred because it does not imply random splits of the data into training set and

test set and, therefore, reported performances can be exactly confirmed by other researches; in addition, leave-one-out is approximately unbiased for the true classification performance [7, p. 590]. However, these desirable properties are obtained at the expense of a significant computational effort since the classifier must be trained as many times as the number of objects in the data set.

In recent years, the above-mentioned computational challenges have motivated an increasing interest in developing efficient implementations for classification algorithms, particularly those that are aimed to exploit technologies currently available in personal modern computers such as multi-core processors (multi-core CPU) and general-purpose graphics processing units (GP-GPU). Parallel implementations on the first ones are of special interest because, nowadays, they are omnipresent in domestic machines. Even though the second ones have become very popular, they are still not considered part of the default specifications for commercial computer machines. A throughout study on GPU-based parallel versions of several machine learning algorithms is carried out in [8], including implementations for well-known classifiers such as neural networks and support vector machines. In contrast, classifier implementations for multi-core CPU are scattered in the literature; for example the following, just to cite recent ones per each classifier category: *A dissimilarity-based classifier* – parallel implementation of the k nearest neighbor rule [9] tested on machines having from 2 up to 60 processing cores; *A probabilistic classifier* – the implementation in [10] of the naïve Bayes classifier, whose authors evaluated their algorithm on the publicly available KDD CUP 99 dataset; *A geometric classifier* – the so-called scaling version developed in [11] for support vector machines, that was exhaustively studied for several threads/cores ratios. Nonetheless, to the best of our knowledge, neither NFL nor RNFLS have been studied for parallel implementations except for our own preliminary attempts: [12]. Therefore, in this paper, we propose parallel CPU-based versions for the leave-one-out test of both NFL and RNFLS by accordingly reformulating their tests in terms of a number of available computing cores and giving a thorough experimental evaluation of the proposed parallel evaluations.

The remaining part of this paper is organized as follows. The serial algorithms of NFL and RNFLS are presented in Sec 2.1. The proposed parallel implementations of the leave-one-out test for NFL and RNFLS are

explained in Sec. 2.2. In Sec. 3, we present experimental results of the execution of the multi-core versions on several benchmarking data sets and compare them to the execution of their serial counterparts. Finally, our concluding remarks are given in Sec. 4.

2 Methods

2.1 Serial Algorithms

Let $\mathcal{T} = \{(\boldsymbol{t}_1, \theta_1), \ldots, (\boldsymbol{t}_N, \theta_N)\}$ be the training set, with training feature vectors $\boldsymbol{t}_i \in \mathbb{R}^K$ and class labels $\theta_i \in \{\omega_1, \ldots, \omega_C\}$. Since feature values may have different dynamic ranges, it is customary to normalize each $\boldsymbol{t}_i$ by the mean feature vector ($\boldsymbol{\mu}_\mathcal{T}$) and the vector of standard deviations per feature ($\boldsymbol{\sigma}_\mathcal{T}$), as follows:

$$\boldsymbol{x}_i = (\boldsymbol{t}_i - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}, \quad 1 \leq i \leq N,$$

where $\oslash$ denotes the Hadamard (entrywise) division; notice that, in case that any entry of $\boldsymbol{\sigma}_\mathcal{T}$ is 0, it must be replaced by 1 in order to avoid a division by zero. The normalized training set is thereby $\mathcal{X} = \{(\boldsymbol{x}_1, \theta_1), \ldots, (\boldsymbol{x}_N, \theta_N)\}$. Similarly, given a test feature vector $\boldsymbol{t} \in \mathbb{R}^K$, its normalized version $\boldsymbol{x}$ is obtained by

$$\boldsymbol{x} = (\boldsymbol{t} - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}$$

1-NN classifies $\boldsymbol{x}$ according to the class label of its nearest neighbor in $\mathcal{X}$. In more formal terms, the class label $\hat{\theta}$ that 1-NN estimates for $\boldsymbol{x}$ is given by

$$\hat{\theta} = \theta_j, \quad j = \underset{1 \leq i \leq N}{\operatorname{argmin}}\, d(\boldsymbol{x}, \boldsymbol{x}_i).$$

A detailed pseudocode for 1-NN classification is shown in Algorithm 1.

2.1.1 Nearest feature line classifier (NFL) One of the disadvantages of 1-NN, pointed in [13], is that 1-NN losses accuracy when $|\mathcal{T}|$ is small. In order to enrich the representational power of $\mathcal{T}$, Li and Lu [3] proposed NFL that builds a set —denoted hereafter by $\mathcal{L}$— of feature lines, each one

Algorithm 1 Nearest neighbor classifier (1-NN)

Require: $\boldsymbol{t}$: test feature vector; $\mathcal{T}$: training set
1: *// Normalization phase:*
2: $\mathcal{X} = \emptyset$
3: **for each** $\boldsymbol{t}_i \in \mathcal{T}$ **do**
4: $\quad \boldsymbol{x}_i = (\boldsymbol{t}_i - \boldsymbol{\mu}_{\mathcal{T}}) \oslash \boldsymbol{\sigma}_{\mathcal{T}}$
5: $\quad \mathcal{X} = \mathcal{X} \cup \{(\boldsymbol{x}_i, \theta_i)\}$ ▷ Normalized training set
6: **end for**
7: $\boldsymbol{x} = (\boldsymbol{t} - \boldsymbol{\mu}_{\mathcal{T}}) \oslash \boldsymbol{\sigma}_{\mathcal{T}}$ ▷ Normalized test feature vector
8: *// Test phase:*
9: $\mathcal{D} = \emptyset$
10: **for each** $\boldsymbol{x}_i \in \mathcal{X}$ **do**
11: $\quad d_i = ||\boldsymbol{x} - \boldsymbol{x}_i||$ ▷ Euclidean distances to...
12: $\quad \mathcal{D} = \mathcal{D} \cup \{d_i\}$ ▷ ...training feature vectors
13: **end for**
14: $j = \operatorname{argmin}_i d_i \quad \forall d_i \in \mathcal{D}$ ▷ Index of the smallest distance
15: $\hat{\theta} = \theta_j$
16: **return** $\hat{\theta}$
Ensure: $\hat{\theta}$: estimated class label for $\boldsymbol{t}$

connecting a pair of training feature vectors that belong to the same class. Since a line is geometrically specified by the pair of points it connects, each labeled line L_i can be represented as the triplet $(\boldsymbol{x}_j, \boldsymbol{x}_k, \rho_i)$, where ρ_i is the class label of the feature line. Clearly, ρ_i is equal to both θ_j and θ_k since feature lines are restricted to connect class-mates.

NFL classifies $\boldsymbol{x}$ according to the class label of its nearest feature line in $\mathcal{L}$; that is,

$$\hat{\theta} = \rho_m, \quad m = \underset{1 \leq i \leq N_L}{\operatorname{argmin}} d(\boldsymbol{x}, L_i),$$

where N_L is the number of feature lines. A detailed pseudocode for NFL classification is shown in Algorithm 2.

Algorithm 2 Nearest feature line classifier (NFL)

Require: $\boldsymbol{t}$: test feature vector; $\mathcal{T}$: training set

// Normalization phase:
$\mathcal{X} = \emptyset$
for each $\boldsymbol{t}_i \in \mathcal{T}$ **do**
    $\boldsymbol{x}_i = (\boldsymbol{t}_i - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}$
    $\mathcal{X} = \mathcal{X} \cup \{(\boldsymbol{x}_i, \theta_i)\}$ ▷ Normalized training set
end for
$\boldsymbol{x} = (\boldsymbol{t} - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}$ ▷ Normalized test feature vector
// Test phase:
$\mathcal{L} = \emptyset$ ▷ Optional to store triplets that define feature lines
$\mathcal{D} = \emptyset$

Algorithm 2 Nearest feature line classifier (NFL). Cont...

for $j = 1$ to $|\mathcal{X}|$-1 **do**
    for $k = j + 1$ to $|\mathcal{X}|$ **do**
        if $\theta_j == \theta_k$ **then**
            $\rho_i = \theta_j$
            $\mathcal{L} = \mathcal{L} \cup \{(\boldsymbol{x}_j, \boldsymbol{x}_k, \rho_i)\}$ ▷ Optional to store triplets
            $\tau = (\boldsymbol{x} - \boldsymbol{x}_j) \cdot (\boldsymbol{x}_k - \boldsymbol{x}_j)/(||(\boldsymbol{x}_k - \boldsymbol{x}_j)||)^2$
            $\boldsymbol{p} = \boldsymbol{x}_j + \tau(\boldsymbol{x}_k - \boldsymbol{x}_j)$
            $d_i = ||\boldsymbol{x} - \boldsymbol{p}||$ ▷ Distances to...
            $\mathcal{D} = \mathcal{D} \cup \{d_i\}$ ▷ ...feature lines
        end if
    end for
end for
$m = \text{argmin}_i \, d_i \quad \forall d_i \in \mathcal{D}$ ▷ Index of the smallest distance
$\hat{\theta} = \rho_m$
return $\hat{\theta}$

Ensure: $\hat{\theta}$: estimated class label for $\boldsymbol{t}$

2.1.2 Rectified nearest feature line segment (RNFLS) classifier

NFL suffers from two drawbacks: interpolation and extrapolation inaccuracies. Du and Chen [4] diagnosed both problems and proposed a modified version of the classifier, called rectified nearest feature line segment (RNFLS), that includes two additional processes to overcome the inaccuracies: segmentation and rectification. The first one consists in computing the distance from $\boldsymbol{x}$ to the feature line exactly as done in the NFL algorithm only if the orthogonal projection of $\boldsymbol{x}$ onto the feature line lies on the interpolating part of the line; otherwise, the distance is replaced by the distance from $\boldsymbol{x}$ to either $\boldsymbol{x}_j$ or $\boldsymbol{x}_k$ according to the side of the extrapolating parts of the line where the orthogonal projection of $\boldsymbol{x}$ would appears.

The second process —rectification— consists in checking whether the feature lines violates the territory of other classes or not; in case of violation, that line is excluded from $\mathcal{L}$ and, thereby, it is not taken into account to compute the distances. A feature line violates the territory of other classes if there exists at least one training object for which the radius of its territory is less than its distance to the feature line. For each training object, the radius of its territory is defined as the distance to the closest training object belonging to a different class. A detailed pseudocode for RNFLS classification is shown in Algorithm 3.

Algorithm 3 Rectified nearest feature line segment classifier

Require: $\boldsymbol{t}$: test feature vector; $\mathcal{T}$: training set

// Normalization phase:
$\mathcal{X} = \emptyset$
for each $\boldsymbol{t}_i \in \mathcal{T}$ **do**
    $\boldsymbol{x}_i = (\boldsymbol{t}_i - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}$
    $\mathcal{X} = \mathcal{X} \cup \{(\boldsymbol{x}_i, \theta_i)\}$ ▷ Normalized training set
end for
$\boldsymbol{x} = (\boldsymbol{t} - \boldsymbol{\mu}_\mathcal{T}) \oslash \boldsymbol{\sigma}_\mathcal{T}$ ▷ Normalized test feature vector
// Radii of the territories:
$\mathcal{R} = \emptyset$
for each $\boldsymbol{x}_i \in \mathcal{X}$ **do**
    $r = \text{Inf}$
    for all $\boldsymbol{x}_j \in \mathcal{X}$ **do**
        if $\theta_i \neq \theta_j$ **then**
            $d = ||\boldsymbol{x}_i - \boldsymbol{x}_j||$
            if $d < r$ **then**
                $r = d$
            end if
        end if
    end for
    $\mathcal{R} \cup \{r\}$
end for
// Test phase:
$\mathcal{L} = \emptyset$ ▷ Optional to store triplets that define feature lines
$\mathcal{D} = \emptyset$
for $j = 1$ to $|\mathcal{X}|$-1 **do**
    for $k = j$ to $|\mathcal{X}|$ **do**
        if $\theta_j == \theta_k$ **then**
            $\rho_i = \theta_j$
            // Verification of invasion to class territories
            $check = True$

Algorithm 3 Rectified nearest feature line segment classifier, Cont...

for $j = m$ to $|\mathcal{X}|$ **do** ▷ Check lines
  if $\theta_m \neq \rho_i$ **then**
    $\tau = (\boldsymbol{x} - \boldsymbol{x}_j) \cdot (\boldsymbol{x}_k - \boldsymbol{x}_j)/(||(\boldsymbol{x}_k - \boldsymbol{x}_j)||)^2$
    if $\tau < 0$ **then**
      $\boldsymbol{p} = \boldsymbol{x}_j$
    else if $\tau > 0$ **then**
      $\boldsymbol{p} = \boldsymbol{x}_k$
    else
      $\boldsymbol{p} = \boldsymbol{x}_j + \tau(\boldsymbol{x}_k - \boldsymbol{x}_j)$
    end if
    $d2line = ||\boldsymbol{x}_m - \boldsymbol{p}||$
  end if
  if $d2line < r_m$ **then**
    $check = False$
  end if
end for
if $check = True$ **then**
  $\mathcal{L} = \mathcal{L} \cup \{(\boldsymbol{x}_j, \boldsymbol{x}_k, \rho_i)\}$ ▷ Optional to store triplets
  $\tau = (\boldsymbol{x} - \boldsymbol{x}_j) \cdot (\boldsymbol{x}_k - \boldsymbol{x}_j)/(||(\boldsymbol{x}_k - \boldsymbol{x}_j)||)^2$
  if $\tau < 0$ **then**
    $\boldsymbol{p} = \boldsymbol{x}_j$
  else if $\tau > 0$ **then**
    $\boldsymbol{p} = \boldsymbol{x}_k$
  else
    $\boldsymbol{p} = \boldsymbol{x}_j + \tau(\boldsymbol{x}_k - \boldsymbol{x}_j)$
  end if
  $d_i = ||\boldsymbol{x} - \boldsymbol{p}||$ ▷ Distances to...
  $\mathcal{D} = \mathcal{D} \cup \{d_i\}$ ▷ ...feature lines
end if
end if
end for
end for
$m = \text{argmin}_i\, d_i \quad \forall d_i \in \mathcal{D}$ ▷ Index of the smallest distance
$\hat{\theta} = \rho_m$
return $\hat{\theta}$

Ensure: $\hat{\theta}$: estimated class label for $\boldsymbol{t}$

Notice that, in Algorithm 3, the second inner loop starts with $k = j$ instead of $k = j + 1$ as in Algorithm 2. This difference is because RNFLS is defined to also include degenerated feature lines; that is, those lines connecting points with themselves.

2.2 Proposed parallel algorithms

The parallel programs for the leave-one-out evaluation were implemented in two phases: Preprocessing (P) and Classification (C). Figure 1 shows the preprocessing and classification steps, the sequence of functions, their associated complexity and which ones are running in parallel or in sequential mode. Each step is described in Sections 2.2.1 and 2.2.2.

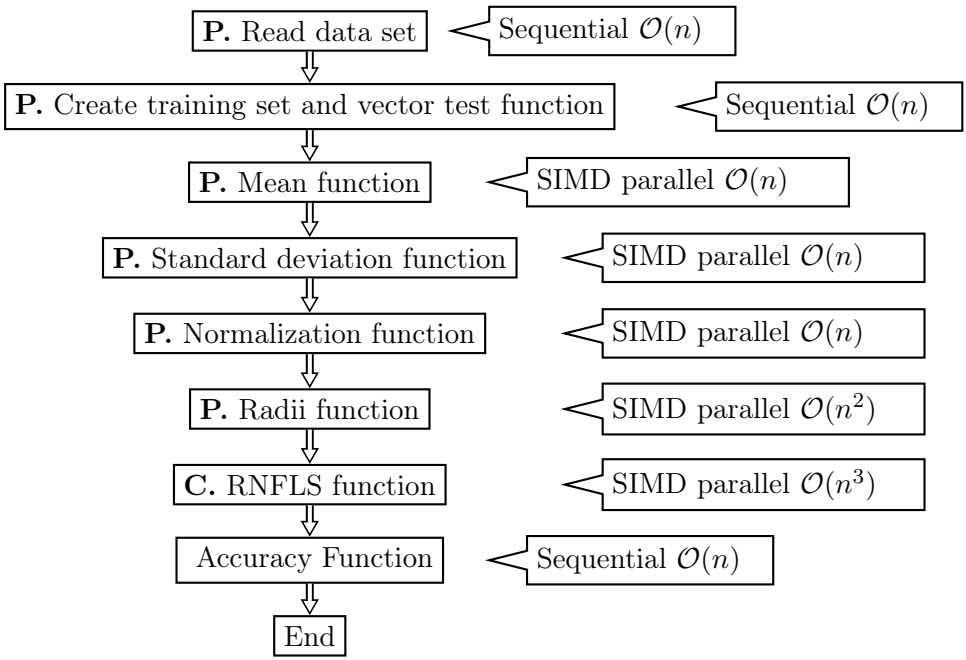

Figure 1: Block diagram of RNFLS algorithm. Mean, Standard Deviation and Radii functions are written in parallel using Single Instruction Multiple Data (SIMD), in front of each one their complexities are shown.

The first step is loading the data set, then a storage structure (shown

in Figure 2) is created by the Create training set and vector test function, this structure contains: (i) the training set $\mathcal{T}^{(i)}$ and the test vector $\boldsymbol{t}^{(i)}$ where $0 < i < N$ and N stands for the number of partitions; (ii) the mean feature vector $\boldsymbol{\mu}_{\mathcal{T}} = \mathbf{0}$; (iii) the standard deviations vector per feature $\boldsymbol{\sigma}_{\mathcal{T}} = \mathbf{0}$ for both NFL and RNFLS are initialized in zero and also the algorithm calculates the radii vector $\mathcal{R}^{(i)} = \mathbf{0}$ initialized in zero too, only for the RNFLS algorithm. After that, the mean parallel and the standard deviation functions are calculated for each $\mathcal{T}^{(i)}$ (see Figure 2). These two vectors are used in order to normalize each $\mathcal{T}^{(i)}$ and each $\boldsymbol{t}^{(i)}$, resulting $\mathcal{X}^{(i)}$ and $\boldsymbol{x}^{(i)}$, see Figure 3. So, each thread executes the NFL and RNFLS algorithms over each position of the structure in an independent way as shown in Figure 2. The size of the structure depends on the number of rows containing the initial data file.

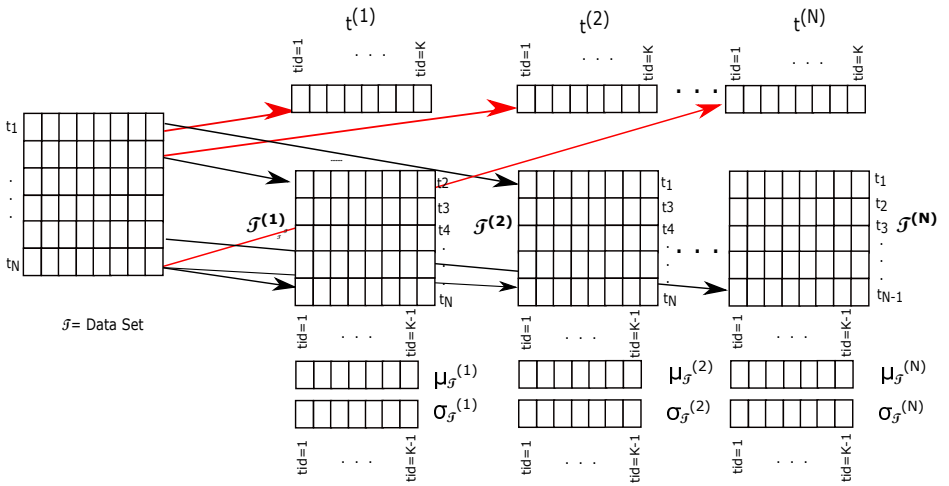

Figure 2: Leave-one-out parallel representation for RNFLS, mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ parallel representation.

2.2.1 Preprocessing phase In this phase, we use the SIMD Flynn taxonomy to implement parallel functions for computation of standard deviation, mean, radii, training matrix and test vector normalization (see Figure 1). This section describes the algorithms involved in the preprocessing phase.

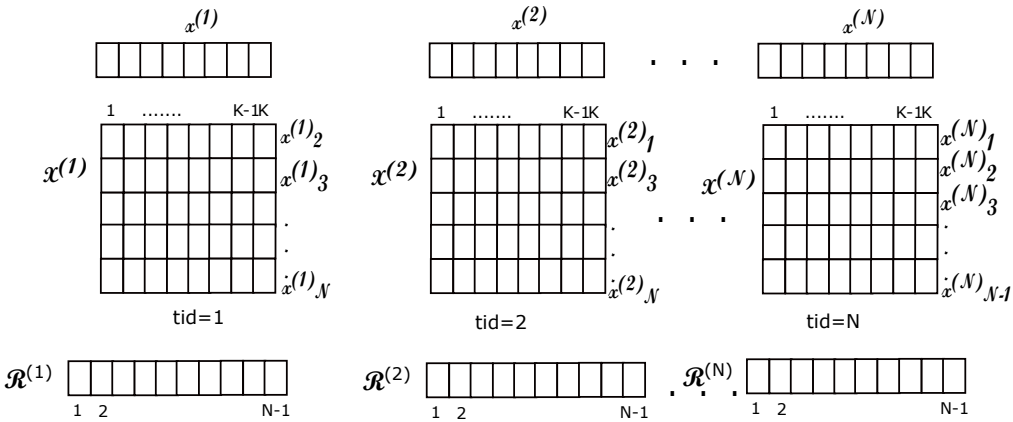

Figure 3: Structure after normalization for threads executions.

- **Mean parallel function** (Algorithm 4) To parallelize the average of each training set $\mathcal{X}^{(i)}$, as many threads are created as columns in the training set, each thread is identified with its *tid* where $0 \leq tid < K-1$ (K is the number of columns), so *tid* is used as a column index for the training set $\mathcal{T}^{(i)}(j, tid)$. Each thread stores the results of the executions of the mean function in a temporal vector called $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}$, where each vector position is calculated for a thread through their respective *tid*, leaving the result of executing the mean function in the same *tid* vector position, each thread obtains the average of each $\mathcal{T}^{(i)}(j, tid)$ column, in a independent way, see Figure 2. The mean parallel function is shown in Equation (1). The mean function complexity for each thread is $\mathcal{O}(n)$.

$$\boldsymbol{\mu}_{\mathcal{T}^{(i)}} = \sum_{j=0}^{N-1} \mathcal{T}^{(i)}(j, tid) \qquad i = \{0, \ldots, N\} \tag{1}$$

Algorithm 4 Mean parallel function

Require: $\mathcal{T}^{(i)}$ $i = \{0, \ldots, N\}$: training set

1:
2: **function** MEANPARALLEL(tid)
3: $\boldsymbol{\mu}_{\mathcal{T}^{(i)}} = \mathbf{0}$ ▷
4: **for each** $j \in \{0, \ldots, N-1\}$ **do**
5: $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid) = \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid) + \mathcal{T}^{(i)}(j, tid)$ ▷ $\Rightarrow 0 \leq tid < K-1$ and tid is a thread identifier
6: **end for**
7: $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)/N-1$
8: **end function**

- **Standard deviation parallel function** (Algorithm 5) The parallelization of the standard deviation $\boldsymbol{\sigma}_{\mathcal{T}}$ is similar to the media parallel function method, as many threads are created as columns in the training set, each thread is identified with its own tid where $0 \leq tid < K-1$, the tid is used as index for the $\mathcal{T}^{(i)}$ training set columns, each thread stores the standard deviation function results in a temporal vector called $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}$, each position of $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}$ is accessed by a thread through their respective tid, leaving the standard deviation function results in the same tid vector position, each thread executes the standard deviation for each $\mathcal{T}^{(i)}(j, tid)$ columns in an independent way (see Figure 2). This function has the same complexity that Algorithm 4. The standard deviation parallel function is shown in Equation (2).

$$\boldsymbol{\sigma}_{\mathcal{T}^{(i)}} = \sqrt{\frac{\sum_{j=0}^{N-1} [\mathcal{T}^{(i)}(j, tid) - \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)]^2}{N-1}} \qquad i = \{0, \ldots, N\} \qquad (2)$$

- **Training matrix and test vector normalization parallel function** (Algorithm 6) Similarly, we apply $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}$ and $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}$ to its corresponding $\mathcal{T}^{(i)}$ in order to normalize the training set and obtain $\mathcal{X}^{(i)}$ as a result (see Figure 3). The process is represented in Equation

Algorithm 5 Standard deviation parallel function

Require: $\mathcal{T}^{(i)}$: training set; $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}$: media vector $i = \{0, \ldots, N\}$

2: **function** STDPARALLEL(tid)
  $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}} = \mathbf{0}$
4: **for each** $j \in \{0, \ldots, N-1\}$ **do**
    $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid) = \boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid) + [\mathcal{T}^{(i)}(j, tid) - \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)]^2$
6: $\triangleright \Rightarrow 0 <= tid < K-1$ and tid is a thread identifier
  end for
8: $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid) = \sqrt{\frac{\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid)}{N-1}}$
end function

(3). It is important to note that we use the same structure $\mathcal{T}^{(i)}$ in the program to store the normalized results. Each thread obtains the value of $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)$ and $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid)$ in order to normalize each column element in $\mathcal{T}^{(i)}(j, tid)$.

$$\mathcal{X}^{(i)}(j, tid) = (\mathcal{T}^{(i)}(j, tid) - \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)) \oslash \boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid) \tag{3}$$

Algorithm 6 Parallel normalization of training matrix and test vector

Require: $\mathcal{T}^{(i)}$: training set, $\boldsymbol{t}^{(i)}$: test vector $\boldsymbol{\mu}_{\mathcal{T}^{(i)}}$: media vector, $\boldsymbol{\sigma}_{\mathcal{T}^{(i)}}$: standard deviation vector

function PARALLELNORMALIZATION(tid)
3: **for each** $j \in \{0, \ldots, N-1\}$ **do**
    $\mathcal{X}^{(i)}(j, tid) = (\mathcal{T}^{(i)}(j, tid) - \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)) \oslash \boldsymbol{\sigma}_{\mathcal{T}^{(i)}}(tid)$
    $\triangleright\ 0 \leq tid < K$ and tid is a thread identifier
6: **end for**
  $\boldsymbol{x}^{(i)}(tid) = (\boldsymbol{t}^{(i)}(tid) - \boldsymbol{\mu}_{\mathcal{T}^{(i)}}(tid)) \oslash \boldsymbol{\sigma}_{\mathcal{T}^{(i)}}$ $\triangleright$ test vector normalization
end function

- **Radii parallel function** (Algorithm 7) The radii function computes the territories of the each object in the data set. This function creates

the radii vector for each $\mathcal{X}^{(i)}$ implementing the segmentation into blocks of the "*Radii of the territories*" part in the RNFLS algorithm described in Algorithm 3. The parallel process redistributes on the cores of the machine the structure created in Figure 2; each core calculates the complete radii function code. Algorithm 7 shows how the structure distribution is performed.

Algorithm 7 Radii Parallel Function

Require: $\mathcal{X}^{(i)}$: training set normalized
  function RadiiTrainingVectors(tid)
    $\mathcal{R}^{(i)} = \emptyset$
    for each ${\boldsymbol{x}^{(tid)}}_{(i)} \in \mathcal{X}^{(tid)}$ **do**
      $r = \text{Inf}$
      for each ${\boldsymbol{x}^{(tid)}}_{(j)} \in \mathcal{X}^{(tid)}$ **do**
        if $\theta_{(i)}^{(tid)} \neq \theta_{(j)}^{(tid)}$ **then**
          $d = ||\boldsymbol{x}_{(i)}^{(tid)} - \boldsymbol{x}_{(j)}^{(tid)}||$
          if $d < r$ **then**
            $r = d$
          end if
        end if
      end for
      $\mathcal{R}^{(tid)} \cup r$
    end for
  end function

2.2.2 Classification phase The classification phase implements the NFL and the RNFLS algorithms, redistributing in each case the function over the machine cores and processing data that are on the positions of the structure according to the identifier of each thread. Algorithm 9 shows how the function is loaded using initial and final range partitions, according to the number of cores. For example, if the machine has 48 cores, each time the machine sends 48 processes to process the data in the structure shown in Figure 3. Each thread calls Algorithm 8, which verifies the invasion to class territories and calculates the best distance to each test vector $\boldsymbol{x}^{(tid)}$ with respect to $\mathcal{X}^{(tid)}$ and leaves the result in a vector Θ_{tid} of estimated labels in

the corresponding *tid* position, making in parallel each leave-one-out test.

Algorithm 8 Rectified nearest feature line parallel function

Require: $\mathcal{X}^{(i)}$: training set standardized, $\boldsymbol{x}^{(i)}$: test vector standardized
function RectifiedNearestFeatureLineParallel(tid)
// Test phase:
$\mathcal{L}^{(tid)} = \emptyset$ ▷ Optional to store triplets that define feature lines
$\mathcal{D}^{(tid)} = \emptyset$
for $j = 1$ to $|\mathcal{X}^{(tid)}|$-1 **do**
for $k = j$ to $|\mathcal{X}^{(tid)}|$ **do**
if $\theta_j == \theta_k$ **then**
$\rho_i = \theta_j$
// Verification of invasion to class territories
$check = True$
for $m = 1$ to $|\mathcal{X}^{(tid)}|$ **do** ▷ Check lines
if $\theta_m \neq \rho_i$ **then**
$\tau = (\boldsymbol{x}^{(tid)} - \boldsymbol{x}_j^{(tid)}) \cdot (\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})/(||(\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})||)^2$
if $\tau < 0$ **then**
$\boldsymbol{p} = \boldsymbol{x}_j^{(tid)}$
else if $\tau > 0$ **then**
$\boldsymbol{p} = \boldsymbol{x}_k^{(tid)}$
else
$\boldsymbol{p} = \boldsymbol{x}_j^{(tid)} + \tau(\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})$
end if
$d2line = ||\boldsymbol{x}_m^{(tid)} - \boldsymbol{p}||$
end if
if $d2line < r_m^{(tid)}$ **then**
$check = False$
end if
end for
if $check = True$ **then**
$\mathcal{L}^{(tid)} = \mathcal{L}^{(tid)} \cup \{(\boldsymbol{x}_j^{(tid)}, \boldsymbol{x}_k^{(tid)}, \rho_i)\}$
$\tau = (\boldsymbol{x}^{(tid)} - \boldsymbol{x}_j^{(tid)}) \cdot (\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})/(||(\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})||)^2$
if $\tau < 0$ **then**
$\boldsymbol{p} = \boldsymbol{x}_j^{(tid)}$
else if $\tau > 0$ **then**
$\boldsymbol{p} = \boldsymbol{x}_k^{(tid)}$
else
$\boldsymbol{p} = \boldsymbol{x}_j^{(tid)} + \tau(\boldsymbol{x}_k^{(tid)} - \boldsymbol{x}_j^{(tid)})$
end if

Algorithm 8 Rectified nearest feature line parallel function, Cont...

```
                    d_i^(tid) = ||x^(tid) − p||                ▷ Distances to...
                    D^(tid) = D^(tid) ∪ {d_i}^(tid)             ▷ ...feature lines
                end if
            end if
        end for
    end for
    m = argmin_i d_i^(tid)   ∀d_i^(tid) ∈ D^(tid)              ▷ Index of the smallest distance
    θ̂ = ρ_m

    Θ_tid = θ̂                                                  ▷ Θ: Vector of estimated labels
end function
```

Algorithm 9 Rectified nearest feature line parallel threads Invocation

```
Require: cores: number of cores
    procedure RNFL(cores)                                      ▷ number of machine's cores
        labelsestimated = ∅
        partition = numfil/cores;
        for each i ∈ {1, ..., partition} do
            beginP = i * cores;
            if rest == 0&&i == partition then
                break;
            else if i == partition then
                finalP = NumFil;
            else
                finalP = beginP + cores;
            end if
            let tid ∈ beginP < tid < finalP
            pthread_t threads[cores];                          ▷ define threads array
            for each tid do
                pthread_create(&threads[tid], NULL, RectifiedNearestFeatureLineParallel,
    .                                                          ▷ Run NFL in a i − th  core
                (void *)tid);
            end for

        end for
    end procedure
```

3 Results and discussion

The experiments were run on a Dell server with two (2) Intel(R) Xeon(R) CPU E7−4860 v2 @ 2.60GHz, each one with 12 real cores (24 cores in total and 48 Hyper−Threading), operative system GNU/Linux, Scientific Linux 3.10 for x86_64, using postfix threads, C library and Hyper−Threading

(HT). The following data sets, taken from the UCI Machine Learning Repository (`https://archive.ics.uci.edu/ml/datasets.html`), were employed for the experiments: Iris, Wine, Glass, Ionosphere, Bupa, WDBC and Pima.

Algorithms 2 and 3 were run 20 times each one. Results shown in Table 1 are the mean of the executions for the sequential and the most parallelized versions; that is, using one core and 48 cores, respectively. These results illustrate how efficient an algorithm turns out if it can be adapted to a multi-core architecture.

Figure 4 shows that an acceptable performance of the algorithm is reached at around 24 cores, presenting a similar behavior when we increase the number of cores to $24, \ldots, 48$. Such a behavior is due to the Amdahl's law limitation and other aspects such as low-level kernel primitives, shared memory and how the machine scheduler attends the threads[1].

Figure 4 (left) shows the elapsed times achieved for the smaller data set called, iris with 150 samples and 4 attributes and Figure 4 shows the elapsed times achieved for the largest (right) data set called WDBC with 569 samples and 30 attributes running the leave-one-out test with the NFL algorithm. For sequential versions of iris, the time was 0.148 *sec* and the parallel time with 48 cores was the best time 0.038 *sec*, with an speed up of 3.79 times best than the sequential version. For the sequential version of WDBC the time was 46.35 *sec* and the parallel time with 48 cores was the best time (4.23 *sec*), with an speed up of 10.96 times better than the sequential version. A similar behavior happens with the RNFLS version, see Table 1.

Table 1: Sequential and parallel Elapsed time (in seconds) of the small and bigger data.

Dataset	Sequential Time	48 cores Time	Speedup
Iris	7.92	0.44	18.17
WDBC	8843.55	316.85	29.91

In each run, we were careful in throwing one thread per core without surpassing the total Hyper-Threading that the machine can support in

[1] $https://www.ibm.com/developerworks/library/l-htl/index.html$

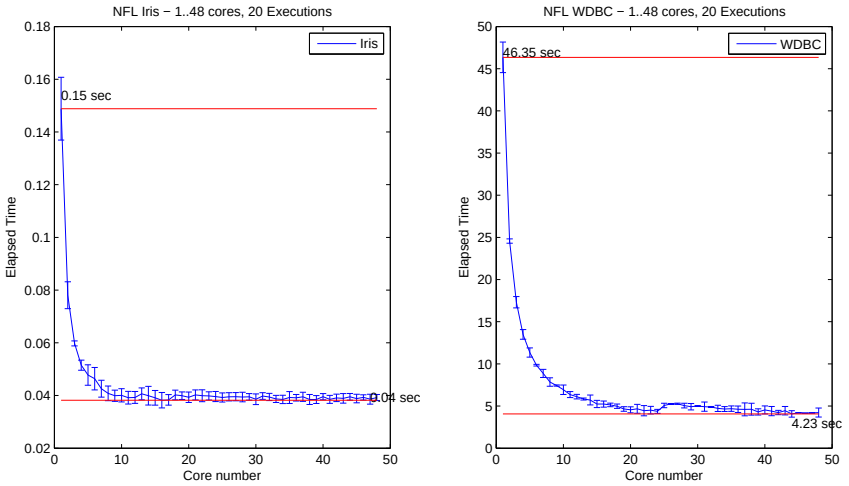

Figure 4: Elapsed time for the smallest data set (on the left) and the largest data set running Leave-one-out with NFL.

order to avoid overload and context change of the threads by the scheduler, in this architecture we can use 2 threads by core. When the number of threads exceeds 24 cores, the behavior of the speed-up changes, presenting ups and downs around 24 cores; see Figure 5. We note that the efficiency is achieve when each algorithm uses only the real base core equivalents (BCE) [14]. In spite of that, when we use the total capacity of multi-threads, the parallel algorithms (NFL and (RNFLS), achieve efficiency but not as significant as with the BCE.

Figure 6 and Figure 7 show the elapsed time and speed up achieved with the leave-one-out test using the NFL and RNFLS algorithms. The speed up was calculated with an approximation of the Amdahl's formula $S_{up} = Et_s/Et_p$ [15], where Et_s is the elapsed time of the sequential version and Et_p is the elapsed time of the parallel version using from 2 to 48 cores. Table 2 shows results for all the data sets used in these tests.

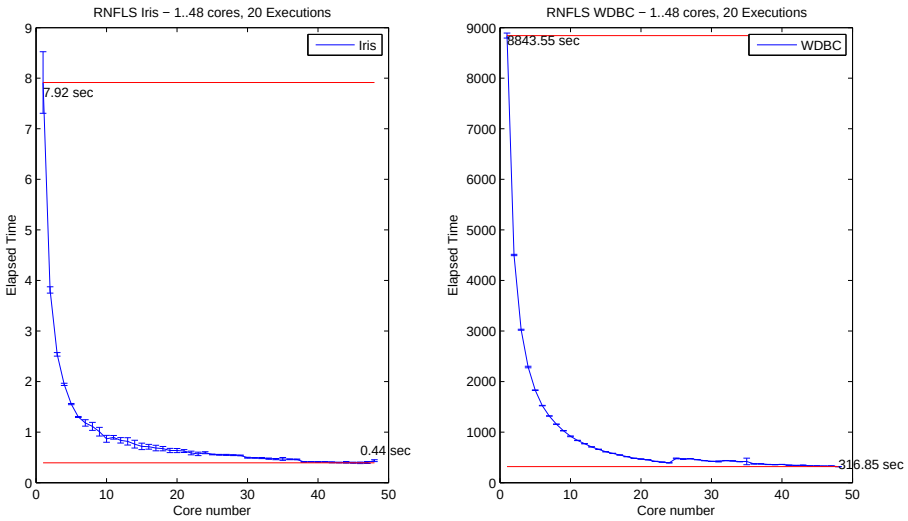

Figure 5: Elapsed time for the smallest data set (on the left) and the largest data set running Leave-one-out with RNFLS.

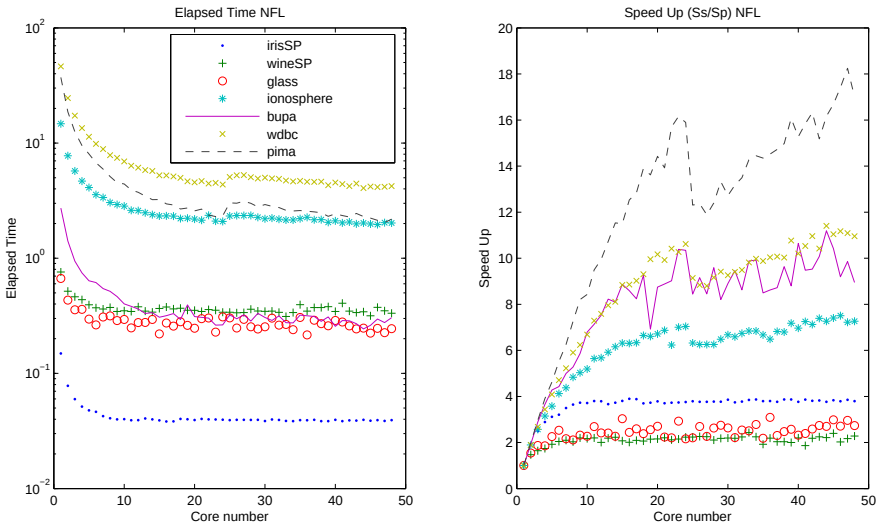

Figure 6: Average speed up for all data sets running the leave-one-out test with the NFL algorithm.

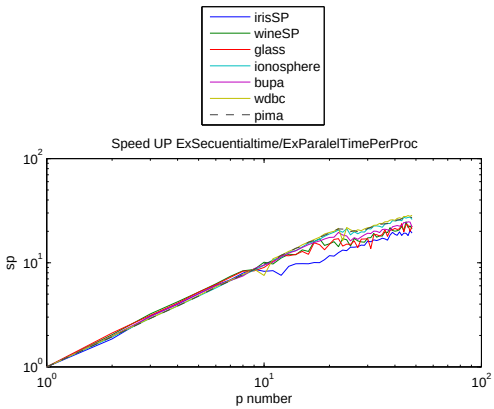

Figure 7: Average speed up for all data sets running the leave-one-out test with the RNFLS algorithm.

Table 2: Elapsed times in seconds (parallel vs. sequential), for the leave-one-out evaluation of NFL and RNFLS using ANSI C. Data sets (DS): (1)Iris, (2)Wine, (3)Glass, (4)Ionosphere, (5)Bupa, (6)WDBC, (7)Pima

			NFL				RNFLS			
DS	**Attr.**	**Inst.**	**Accu**	**Hits**	**Seq.**	**48Cores**	**Accu**	**Hits**	**Seq.**	**48Cores**
1	150	4	0.893	134	0.148	0.039	0.966	145	7.915	0.435
2	178	4	0.960	171	0.771	0.349	0.972	173	35.155	1.829
3	214	4	0.687	147	0.685	0.225	0.729	156	30.128	1.698
4	351	34	0.852	299	14.697	2.023	0.928	326	1318.653	49.928
5	345	4	0.614	212	2.724	0.304	0.660	228	163.474	7.334
6	569	30	0.954	543	46.345	4.229	0.966	550	8843.550	316.847
7	768	8	0.665	511	37.180	2.185	0.700	538	6836.055	250.063

4 Conclusions

The leave-one-out test was parallelized and tested with the NFL and RNFLS algorithms, observing that the best acceleration is achieved under the actual machine base architecture (24 cores) and that multi-threaded activations, although they improve performance, do not achieve accelerations that exceed those achieved when using the base cores of the machine; the latter demonstrates that the acceleration of the algorithms is controlled by the limit imposed by the Amdahl's law. The tests of the algorithms were repeated 20 times for each dataset in order to demonstrate the stability of

the implementation, obtaining results with low standard deviations with respect to the mean. By using more threads than real machine cores, the clock speed of the cores decreases, this is a typical behavior of the Xeon processors to keep the heat levels generated by the processors stable when they are working at 100% of their capacity, which also affects the acceleration of the algorithms. It is important to note that the parallel functions for average, standard deviation and normalization do not behave efficiently for small datasets. The results showed that, using algorithms based on multi-thread multi-core architectures, an improvement of the leave-one-out test can be achieved with the RNFLS algorithm of 29.91 times compared against the sequential algorithm.

Acknowledgements

The authors acknowledge support from Universidad Nacional de Colombia - Sede Manizales, within "Convocatoria interna de investigación la Facultad de Administración 2015, para la formulación y ejecución de proyectos de consolidación y/o fortalecimiento de los grupos de investigación. Modalidad 1 - Formulación y ejecución de proyectos de consolidación", under the project "Consolidación de las líneas de investigación del Grupo de Investigación en Ambientes Inteligentes Adaptativos GAIA" (HERMES code: 32059).

References

[1] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, 2000. [Online]. Available: https://doi.org/10.1109/34.824819 76

[2] R. P. W. Duin, M. Bicego, M. Orozco-Alzate, S.-W. Kim, and M. Loog, "Metric Learning in Dissimilarity Space for Improved Nearest Neighbor Performance," in *Structural, Syntactic and Statistical Pattern Recognition: Proceedings of the Joint IAPR International Workshop, S+SSPR 2014*, ser. Lecture Notes in Computer Science, P. Fränti, G. Brown, M. Loog, F. Escolano, and M. Pelillo, Eds., vol. 8621, IAPR. Berlin Heidelberg: Springer, 2014, pp. 183–192. [Online]. Available: https://doi.org/10.1007/978-3-662-44415-3{_}19 77

[3] S. Z. Li and J. Lu, "Face Recognition Using the Nearest Feature Line Method," *IEEE Transactions on Neural Networks*, vol. 10, no. 2, pp. 439–443, 1999. 77, 79

[4] H. Du and Y. Q. Chen, "Rectified nearest feature line segment for pattern classification," *Pattern Recognition*, vol. 40, no. 5, pp. 1486–1497, 2007. [Online]. Available: https://doi.org/10.1016/j.patcog.2006.10.021 77, 82

[5] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A Survey on Concept Drift Adaptation," *ACM Computing Surveys*, vol. 46, no. 4, pp. 44:1—-44:37, mar 2014. 77

[6] M. Bramer, *Principles of Data Mining*, 2nd ed., ser. Undergraduate Topics in Computer Science. London, UK: Springer, 2013. 77

[7] B. Clarke, E. Fokoué, and H. H. Zhang, *Principles and Theory for Data Mining and Machine Learning*, ser. Springer Series in Statistics. New York, NY: Springer New York, 2009. 78

[8] N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines - A Practical Approach*, ser. Studies in Big Data. Cham, Switzerland: Springer International Publishing, 2015, vol. 7. 78

[9] A. Ahmadzadeh, R. Mirzaei, H. Madani, M. Shobeiri, M. Sadeghi, M. Gavahi, K. Jafari, M. M. Aznaveh, and S. Gorgin, "Cost-efficient implementation of k-NN algorithm on multi-core processors," in *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014*, oct 2014, pp. 205–208. 78

[10] V. D. Katkar and S. V. Kulkarni, "A novel parallel implementation of Naive Bayesian classifier for Big Data," in *International Conference on Green Computing, Communication and Conservation of Energy, ICGCE 2013*. IEEE, 2013, pp. 847–852. [Online]. Available: https://doi.org/10.1109/ICGCE.2013.6823552 78

[11] Y. You, H. Fu, S. L. Song, A. Randles, D. Kerbyson, A. Marquez, G. Yang, and A. Hoisie, "Scaling Support Vector Machines on modern HPC platforms," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 16–31, 2015. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.09.005 78

[12] A.-L. Uribe-Hurtado and M. Orozco-Alzate, "Acceleration of Dissimilarity-Based Classification Algorithms Using Multi-core Computation," in *Trends in Cyber-Physical Multi-Agent Systems. The PAAMS Collection - 15th International Conference, PAAMS 2017*, ser. Advances in Intelligent Systems and Computing, A. T. Campbell, F. de la Prieta, Z. Vale, L. Antunes, M. N.

Moreno, V. Julian, T. Pinto, and A. J. R. Neves, Eds., vol. 619, IEEE Systems Man and Cybernetics Society Spain Section Chapter. Cham, Switzerland: Springer, jun 2017, pp. 231–233. 78

[13] E. Pekalska and R. P. W. Duin, "Dissimilarity representations allow for building good classifiers," *Pattern Recognition Letters*, vol. 23, no. 8, pp. 943–956, 2002. [Online]. Available: https://doi.org/10.1016/S0167-8655(02)00024-7 79

[14] H. Che and M. Nguyen, "Amdahl's law for multithreaded multicore processors," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 3056–3069, 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.06.012 94

[15] J. Nutaro and B. Zeigler, "How to apply Amdahl's law to multithreaded multicore processors," *Journal of Parallel and Distributed Computing*, vol. 107, no. Supplement C, pp. 1–2, 2017. 94