Applying a Task-Based Approach to Distributed Machine Learning Workflows

Fernando Vázquez-Novoa, Daniele Lezzi, Francesc Lordan
Department of Computer Sciences
Barcelona Supercomputing Center
Barcelona, Spain
{ fernando.vazquez, daniele.lezzi, francesc.lordan }@bsc.es

Fatemeh Baghdadi, Davide Cirillo
Department of Life Sciences
Barcelona Supercomputing Center
Barcelona, Spain
{ fatemeh.baghdadi, davide.cirillo }@bsc.es

***Abstract*—The growing demands across various scientific fields have led to a significant shift in applications that consume data at the edge of the computing continuum. These applications require unified programming models for the composition of components and coordinating the execution of computational workloads, including training machine learning (ML) models on distributed resources. Personalized healthcare often leverages data generated from wearable devices used to train ML models, can be benefited from distributed computing approaches. Specifically, stroke care can be greatly benefited from distributed ML with modifiable risk factors that can be monitored using wearable devices. In this work, we present an implementation that leverages distributed techniques for large-scale ML workflows using electrocardiogram (ECG) recordings for atrial fibrillation (AF) classification. The application was evaluated using the PhysioNet database, showcasing the potential of distributed, ML in stroke care, opening the way for future creation of more advanced models embedded in edge devices.**

I. Introduction

In recent years the requirements of many scientific fields have brought a trasformative shift on how applications are developed, deployed, and operated; the availability of data continuously generated by sensors, instruments and other devices as in the case of farming 4.0 applications, predictive maintenance, machine vision, sismology, personalize healthcare, etc., has fostered the evolution of computing paradigms from centralized data centers to the edge of the network where it has to be processed. One of the most common requirements is the application of sophisticated machine learning (ML) models to the data at the edge, allowing to send only essential data to the HPC data centers, reducing bandwidth usage and associated costs. The development and execution of such distributed applications, involve the orchestration of complex workflows that manage data and computation across the so called computing continuum, requiring that a programmer has to be an expert in both the application domain and the low-level details of the platforms on which that application will be deployed. On the other side, the workload burdens associated with larger data sizes pose several challenges to scaling model training and improving performances efficiently, in particular in health applications where massive amounts of patients' data are key to assisting high-stakes clinical decisions. In this regard, distributed ML is emerging as a prominent approach to address the challenges posed by extensive data volumes in model development and to bring about novel technological advancements in healthcare. Task-based workflows are a good approach to developing compute-intensive applications while exploiting distributed infrastructures as they allow the automatic identification of the parallelism inherent to the application and the task executions are distributed in parallel across the underlying infrastructure.

A medical area that can greatly benefit from distributed ML is stroke care. Stroke is a neurovascular condition due to an acute focal injury in the central nervous system by a vascular cause [1]. It is the second leading cause of death and third leading cause of disability in adults worldwide [2]. 90% of strokes are attributable to modifiable risk factors [3], such as high blood pressure, smoking, diabetes, physical inactivity, and, specifically for ischaemic stroke, atrial fibrillation (AF) [4], which is an arrhythmia of the atrial chambers of the heart. In recent years, new strategies for preventing and monitoring stroke and its recurrence, in a continuous and non-invasive way, have been developed, namely the use of portable and wearable devices for cardiovascular monitoring [5].

In this work, we present the implementation of an application that leverages distributed techniques to build large-scale ML workflows in a completely sequential and effortless manner. The application has been evaluated using different implementations to train and test ML models using electrocardiogram (ECG) recordings, collected with a portable device, obtained from the PhysioNet database [6], specifically the dataset of the Computing in Cardiology (CinC) Challenge 2017 on AF classification [7].

The main contributions of this article are:

- Usage of data augmentation techniques to increase the quantity of available data and preprocessing techniques to improve the quality of the data.
- Comparison between three different machine learning algorithms. The comparison was based on training time, scalability of the execution time and performance of the model.
- Training of a neural network following a distributed approach and evaluating the performance of this solution.

This article is structured in five different sections. This first section contains the introduction. Section II contains some background and state of the art about the problem. In Section

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00166

III we present our application. We describe the dataset we are going to use, the transformations applied to the data to improve its quality, the classical machine learning algorithms we are going to use and their distributed version and the neural network approach used. After describing our proposal we make some tests, analyze the results obtained, evaluating the solution and the algorithms used, this evaluation is contained on Section IV. Finally, we draw some conclusions that are contained on the last Section (V).

II. Distributed ML for healthcare applications

Cardiology is recognized as one of the early adopters of ML within the medical sector. Several studies have revealed that ML applications in this domain outperform traditional risk assessment using well-established cardiovascular disease risk factors [8], [9], which are key for the prevention of cerebrovascular events such as stroke. ML and neural networks, specifically, have been used to identify novel clinical phenotype of AF, which is consistently associated with stroke based on clinical data. With the rapid growth of digital clinical data, ML, coupled with portable and wearable technologies, has proven effective in predicting groups of patients that were missed by conventional methods [10]. Scaling up such capacity with distributed ML approaches would help immensely in processing large volumes of electrophysiological signals as well as analyzing and extracting insights to better identify stroke-associated patterns and perform more accurate tasks, such as AF detection.

The electrocardiogram (ECG) has become the most widely used biomarker for the early diagnosis of AF. ECG is a graphical representation of heart electrical activity that is used to diagnose cardiovascular diseases and irregularities. It consists of five major components: P wave, Q wave, R wave, S wave, and T wave. During AF episodes, the heart's atria are quicker than normal beating resulting in the blood not being ejected completely out of the atria and the formation of blood clots. AF can be detected by observing three main features including the P wave absence, the presence of fluctuating waveform (f-wave) Instead of P wave, and heart rate irregularity. There are several methods to detect these features of ECG during AF episodes. RR interval-based methods are limited when the ECG changes quickly between rhythms or when AF takes place with regular ventricular rates. Moreover, the P wave absence detection is difficult due to its small amplitude [11]. Time-frequency domain techniques have been proposed in this paper to overcome these limitations and provide more accurate detection.

A. *Workflow manager*

In the recent years the data available has increased at a high rate. This increment surpasses the increment in the computational resources and their performance. Training machine learning and neural network models using a single CPU or GPU using the data available would take very long times. For this reason the need to distribute the computation across various nodes and computational devices is significant. This distribution reduces the computation times required at the same time that it makes a more efficient usage of the available resources.

PyCOMPSs is a programming framework whose main aim is to facilitate the development of parallel applications that use distributed computing. PyCOMPSs' interface allows for an easy development, at the same time its runtime system efficiently leverages parallelism during the execution of the applications.

PyCOMPSs is the Python binding of COMPSs [12]. By using PyCOMPSs a regular Python script can be easily transformed into a distributed application just by adding a task decorator to the functions that will run in parallel. Then, the runtime system is able to detect the dependencies between tasks dependencies and exploitting their parallelism. The dependencies are detected by the runtime based on their input and output arguments. A task that has at least one input argument that is the output of another task has a dependency with that previous task.

PyCOMPSs builds a graph which contains the tasks present on the application, and the dependencies between the different tasks. This graph is built in execution time. One example of graph generated by PyCOMPSs is shown in Figure 4. The circles in this graph represent the different tasks (each type of task has a different color in the image). The lines between the different tasks represent the dependencies between the tasks. The tasks placed in the same horizontal line in the image can be executed concurrently.

B. *dislib*

Inspired by scikit-learn, a popular ML library for the Python programming language, dislib [13] provides an estimator-based interface that leverages the distributed data structure (ds-array) that can be operated as a regular Python object. The combination of this data structure and the estimator-based interface makes dislib a distributed version of scikit-learn, where communications, data transfers, and parallelism are automatically handled behind the scenes by the PyCOMPSs runtime [14].

All ML methods in dislib are provided as scikit-learn estimator objects. An estimator is a function that is used to infer the value of an unknown parameter in a statistical model. dislib estimators implement the same API as scikit-learn, which is mainly based on the fit and predict operators. The typical workflow in dislib consists of the following steps:

- Reading input data into a ds-array
- Creating an estimator object
- Fitting the estimator with the input data
- Getting information from the model's estimator or applying the model to new data

III. Design of a distributed ML approach for stroke care

The healthcare application presented in this work concerns with the development of distributed ML models for cardiovascular monitoring using portable or wearable devices. Specifi-

cally, it focuses on the detection of AF from ECG recordings, which is a pivotal task for the future creation of risk stratification models embedded in such devices operating in the edge-cloud continuum in real-time. Continuous and non-invasive ML-driven monitoring through portable or wearable devices represents a promising strategy for preventing stroke and its recurrence. Our work is a first step towards the realization of a framework for the operation of distributed ML applications, including AF detection, in computing continua. This work is framed in the context of a use case on personalized healthcare of the EU H2020 project AI-SPRINT (GA 101016577).

Figure 1 represents an overview of the AI-SPRINT use case, with the ECG data from portable or wearable devices. By harnessing HPC resources, the data is used to build a classification model that is then used to detect AF at the edge or close to where the data is generated (e.g., smartwatches). In this paper, we focus on the first part of this pipeline (Training in Figure 1), which consists in the training of the AF detection model using distributed ML techniques, while the inference analysis on the edge is part of future work.

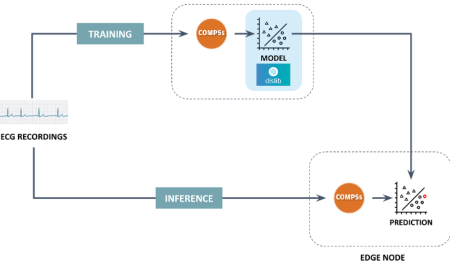


Figure 1: A ML application for stroke care in the edge-cloud continuum. Electrocardiogram (ECG) recordings from portable or wearable devices are used to train an atrial fibrillation (AF) detection model in the cloud or HPC premises using the dislib library, which is then deployed and used for inference at the edge.

A. Dataset description

PhysioNet [6] is a repository of freely-available medical research data, managed by the MIT Laboratory for Computational Physiology. Among many resources, the PhysioNet database of the Computing in Cardiology (CinC) Challenge 2017 on AF classification (https://physionet.org/content/challenge-2017/1.0.0/) contains a total of 12186 single-lead ECG recordings, sampled as 300 Hz, donated by AliveCor, manufacturer of portable ECG hardware. The data, collected from individuals at rest, is available as Matlab V4 files (each including a .mat file containing the ECG and a .hea file containing the waveform information). It consists of 8528 recordings lasting from 9 to 61 seconds. The classes represented in the dataset are (1) Normal (5154 recordings), (2) AF (771 recordings), (3) Other rhythms (2557 recordings), and (4) Noisy recordings (46 recordings). By achieving an F1-score of 0.79 in 5-fold cross-validation on the AF class [15], a classifier based on the Cascade Support Vector Machines (CSVM) algorithm was among the winners of the CinC Challenge 2017. As other classes are out of the scope of this work and its future derivations, we only focused on the classification of AF and Normal classes.

B. Data preparation

1) Data augmentation: Given the imbalance in AF (771 recordings) and Normal classes (5154 recordings) in the PhysioNet dataset, we sought to synthetically augment the minority class using a procedure specifically designed to maintain key properties of ECG signals unaltered. The synthetic augmentation consists in randomly segmenting the signal into stretches of 6 contiguous R peaks (patches), which is considered the minimum ECG length needed to detect irregular rhythms [16], separated by in-between regions (spacers), and then shuffling their order to generate a new synthetic signal (Figure 2). This procedure is performed on all AF signals at random until their total amount is balanced with that of the Normal class. The identification of the R peaks is performed using the Gamboa segmenter of the Python library BioSSPy [17].

2) Zero-padding: The ECGs of the used dataset have different lengths, ranging from 9 seconds to 61 seconds. The inconsistency in signal lengths could introduce problems when developing the ML models. To overcome this issue and make all the ECGs even in terms of length, a zero-padding method has been implemented. In this method, the length of the signal is extended by adding zeros to the series to achieve the desired length. Applying the zero-padding method to ECGs resulted in each signal having a maximum length of 18300, which corresponds to the maximum number of data points of longest signal in the dataset. The corresponding duration in seconds can be obtained by multiplying the length of the signal by the 300 Hz sampling frequency.

3) Short Time Fourier Transform: The ECG signal is non-stationary data whose instantaneous frequency oscillates with time. Thus, the properties of the changes in a signal cannot be fully described by solely utilizing information derived from the frequency components. The Short Time Fourier Transform (STFT) is a non-stationary signal analysis approach that maps signal information from the time domain to the time-frequency domain and has been applied to signals as a feature extraction method. The STFT is an optimized mathematical tool evolved from the discrete Fourier transform (DFT) for discovering the instantaneous frequency as well as the instantaneous amplitude of localized waves with time-varying characteristics within a window function [18]. Using the signal package of the Python library SciPy [19], the spectrogram function computes and returns the STFT of the input signal. Each column in the output of the spectrogram function contains an estimate of the short-term, time-localized frequency components of the input signal. In this work, STFT has been applied to signals as a feature extraction method to compute their frequency, time, and amplitude components in each window segment. The resulting array is a multi-dimensional array, and for

ease of processing and dimensionality reduction, the array elements are concatenated to produce a 1-dimensional array of length 18810 using the flatten function. These arrays of values are used as input for the classifiers after an additional dimensionality reduction by Principal Component Analysis (PCA).

4) Principal Component Analysis: In order to reduce the dimensionality of the data, a Principal Component Analysis (PCA) decomposition is performed before the fitting of the models. In the covariance method, features are centered (the mean is subtracted for each feature) but not standardized (not divided by the standard deviation, which would be the correlation method). Then, the covariance matrix is estimated as x.T@x / (n_samples - 1). Finally, the eigendecomposition of this matrix is computed, yielding the principal components (eigenvectors) and the explained variance (eigenvalues). In the dislib implementation of the covariance method, centering the features and estimating the covariance matrix are computed in two successive map-reduce phases, partitioning the samples only by row blocks. Hence, an unpartitioned covariance matrix of shape (n_features, n_features) is obtained. This matrix is processed by a single task which computes the eigendecomposition using the numpy.linalg.eigh method. The PCA resulted in only a loss of the 5% of the total information, preserving the 95% of the information contained in the features of the original dataset. Despite preserving the majority of the information we reduce the number of features drastically from 18810 features to 3269. The top part of each PyCOMPSs execution graphs, described in the next section, represents the PCA portion of the execution.

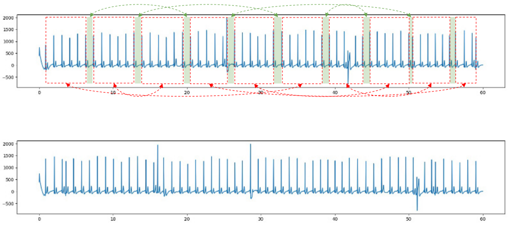


Figure 2: Shuffling-based data augmentation procedure designed to overcome class imbalance.

C. Traditional ML classification models

In this section we describe the three implementations of distributed ML algorithms that we adopted for the training of the models. For each model, a graph of the PyCOMPSs execution is provided (Figures 4, 6, 8, 9). It is worth noticing that these figures are shown to explain the complexity of the workflows and the level of parallelism that can be achieved with our proposal. However, these graphs represent only a part of the actual tests described in the evaluation section. The complete graphs would be indeed too complex to be displayed fully.

1) CSVM: The code performs a training of the model using the dislib implementation of the CascadeSVM (CSVM) algorithm and then calculates the score returning the mean accuracy on a given test data and labels. The CSVM estimator implements a version of support vector machines that parallelises training by using a cascade structure. The algorithm (Figure 3) splits the input data into N subsets, trains each subset independently, merges the computed support vectors of each subset two by two, and trains again each merged group of support vectors. One iteration of the algorithm finishes when a single group of support vectors remains. The final support vectors are then merged with the original subsets, and the process is repeated for a fixed number of iterations or until a convergence criterion is met. The fitting process of the CSVM estimator creates the first layer of the cascade with the different row blocks of the input ds-array. This means that the estimator creates one task per row block at the first layer, and then creates the rest of the tasks in the cascade. Each of these tasks use scikit-learn's SVC (C-Support Vector Classification) internally for training and load a row block in memory. The maximum amount of parallelism of the fitting process is thus limited by the number of row blocks in the input ds-array. In addition to this, the scalability of the estimator is limited by the reduction phase of the cascade.

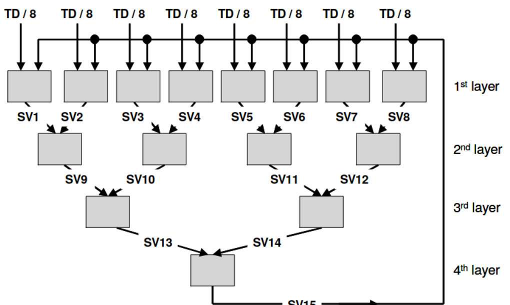


Figure 3: CSVM algorithm illustrative representation

The input dataset is loaded from the PhysioNet repository files into ds-array objects as training set and labels after preprocessing. The data is split by dislib in blocks of 500x500 thus generating 631 tasks managed by PyCOMPSs. Figure 4 depicts a reduced version of the PyCOMPSs execution graph.

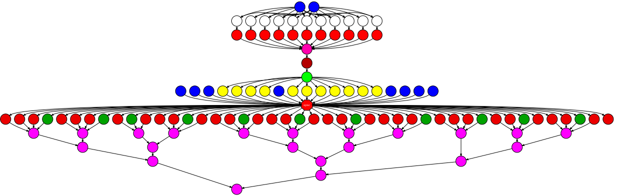


Figure 4: Execution graph of the CSVM algorithm. This is a simplified version of the graph with less tasks than the real executions.

2) KNN: The method implements a k-nearest neighbors (KNN) algorithm that classifies the data based on the proximity to a given point (Figure 5). The k value in the KNN

algorithm defines how many neighbors will be checked to determine the classification of a specific query point. The parameters of the method are the following: (1) the number of neighbors to use by default for kneighbors() queries; (2) an optional Weight function used in prediction whose possible values are: 'uniform' to have uniform weights meaning that all points in each neighborhood are weighted equally, or 'distance' to weight points by the inverse of their distance; in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away; (3) a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

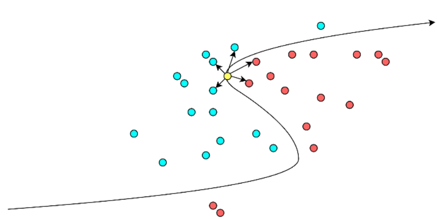


Figure 5: KNN algorithm illustrative representation

The fit function uses the NearestNeighbors algorithm in dislib that has parallelism based on the number of row blocks the dataset is divided into. It launches a fit from the scikit-learn NN into each row block. The predict also makes a task per block in the row axis of the dataset. Figure 6 depicts the execution graph of the workflow with K=5.

Figure 11b shows the times of scaling the data with the StandardScaler and fitting the KNN classifier. The measure of time was done with a block size of 250x250.

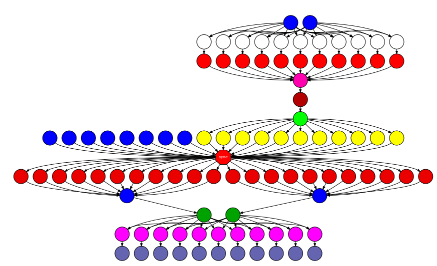


Figure 6: Execution graph of the KNN algorithm

3) Random Forest: RandomForest (RF) is a classification algorithm that constructs a set of individual decision trees, also known as estimators. Each estimator classifies a given input into classes based on decisions taken in random order. The final classification of the model is the aggregate of the result of all the estimators; thus, the accuracy of the model depends on the number of estimators composing it.

Figure 7 illustrates how RF works depicting a 2-estimator model. In this case, decision trees are limited to a depth of 3. The leaves of the decision trees are the probability

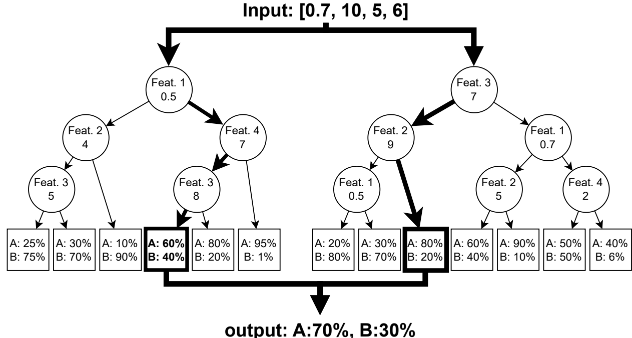


Figure 7: Random Forest algorithm graphical representation

distribution of those samples that fulfill the conditions required by all the nodes in the path. Every input sample undergoes a classification process on both estimators; each estimator returns a predicted class according to the input values of each feature. The first estimator returns a probability of 60% that the sample belongs to class A while the second one estimates it on an 80%. To compute the final prediction of the overall model, the predictions of the composing estimators are averaged.

These tests use the implementation provided in dislib. This is the only algorithm in dislib in which the number of blocks and their size does not have a direct impact on the computational time and number of tasks created during its training; its parallelism is based on the number of estimators and the parameter $distr_depth$ (limit of the depth of the tree where the decisions are no longer computed in parallel). The graph in Figure 8 depicts the workflow resulting from the execution of this algorithm to train a model with 40 estimators.

The time results of the executions of RF are shown on Figure 11c. The results show a very bad scalability, this can have two causes. The first one is the small number of tasks that this algorithm generates (it does not depend on the block size). If there are a small number of tasks the use of more nodes do not improve the execution times. The second cause can be the unbalanced load. The division of the data on the different decision trees can cause some tasks handle considerably more data than other, increasing the execution time of the whole algorithm. In addition, the difference between the execution times with 2 and 3 nodes can be caused due to the increment in the transference of data with 3 nodes. If all the tasks do not fit concurrently in execution, neither with 2 nodes nor with 3, it can happen that the execution with 3 nodes takes a little bit longer than the task that generates the data it depends on due to the transference of data to tasks that are executed in a different node.

D. Neural Network classification model

Neural Networks can solve a wide range of problems and can be used for classification. In order to use Neural Networks for this task, our initial step involved searching for an appropriate architecture. Ultimately, we discovered an

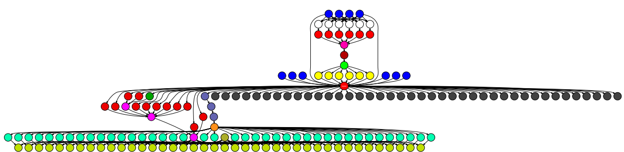


Figure 8: Execution graph of the RF algorithm

architecture that demonstrates high accuracy by using two 1-dimensional convolutional layers with 32 filters and a final dense layer with 32 neurons. Our architecture search involved assessing numerous alternatives with varying numbers of layers, filters, and other modifications. Additionally, we examined architectures solely comprising dense layers; however, their accuracy fell short of our expectations.

The training was parallelized using PyCOMPSs and in addition we used EDDL [20], a deep learning library that enables the parallelization of data between the resources of the same node. As with the previous models, we performed a cross validation (K-fold) with 5 folds.

In order to parallelize the training, we used different approaches. In the first approach, we used PyCOMPSs to distribute the data between different nodes, with EDDL running on 4 GPUs inside each node, being EDDL in charge of distributing the data between the different GPUs.

A different approach was to use the PyCOMPSs workers to distribute the data between the different GPUs, and use EDDL to train the model using only one GPU.

In both approaches, when an epoch is ended, the weights of the neural network in each worker are retrieved and they are merged and used in the next epoch.

We observed that these approaches raised an issue. As depicted in 9, after each epoch a synchronization is required in order to retrieve the updated weights of the neural network in each worker. Each fold runs seven epochs corresponding to a group of four training tasks each one running on a GPU in the node and represented by a green circle. Each synchronization stops the generation of tasks and prevents the possibility of executing the training of the 5 folds in parallel. In order to solve this issue, we decided to use a new PyCOMPSs feature called *nesting*. This new paradigm enables the generation of tasks inside other tasks and encapsulates the synchronizations within a task.

In order to use nesting, for each of the folds we declared an individual task that runs in parallel with respect to the other folds. In this way, each synchronization is local to the fold and does not block the other 4 folds. In Figure 10 the workflow of this new approach is depicted; the training tasks of each fold are now grouped and can be executed in parallel on five nodes.

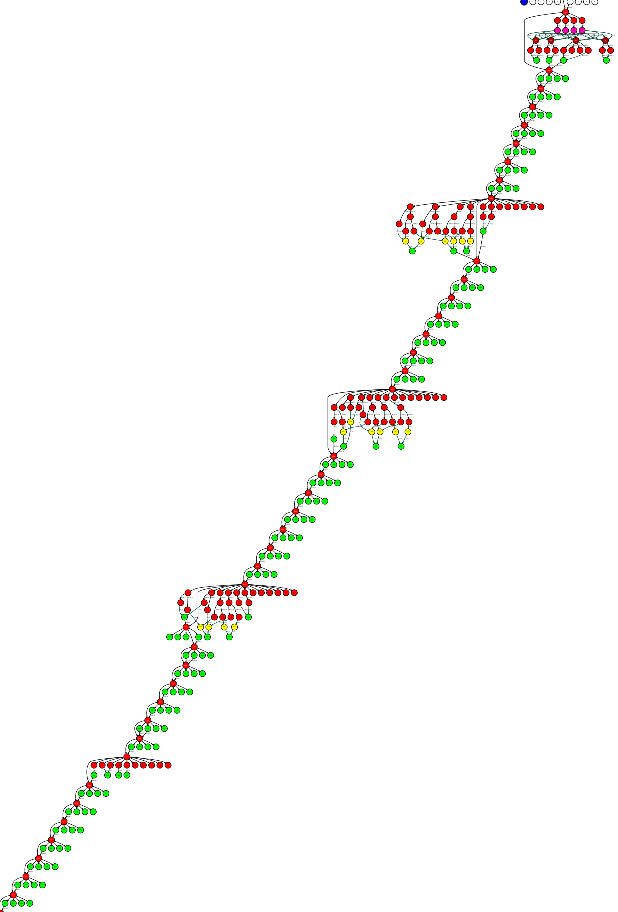


Figure 9: Execution graph of the CNN algorithm

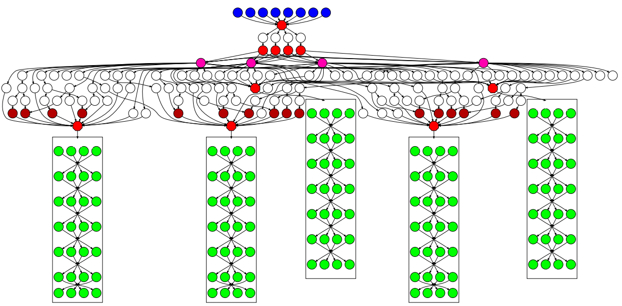


Figure 10: Execution graph of the CNN algorithm with nesting enabled

IV. Evaluation

A. Testbed

In order to evaluate the performance of the implemented algorithms, we executed a set of runs in the MareNostrum IV supercomputer. In particular, we tested the ML algorithms' implementations using the PhysioNet database on the general purpose part of the cluster that contains 3456 nodes (48 servers of 72 nodes) with two 24-core Intel Xeon Platinum 8160 and 98 GB of main memory each.

For the evaluation of the CNN we executed a set of runs in the CTE-Power cluster, a Power 9 GPU partition

of the MareNostrum supercomputer. This cluster contains 52 compute nodes, each of them with 2 IBM Power9 8335-GTH @ 2.4GHz CPUs, 512GB of main memory and 4 GPU NVIDIA V100 (Volta) with 16GB HBM2.

B. Test results

Each algorithm is implemented with an ensemble of runs, trained with K-fold (K=5). In the results, we did not consider the time of executing the PCA, that is the same for each algorithm and takes about 850 seconds.

The experiments conducted were slightly different depending on the algorithm used. The pre-processing of the PhysioNet data was the same for all the experiments but one of them included an extra step, namely the KNN algorithm, which included the application of a StandardScaler to the data. This scaler removes the mean value of the features and divides the data by its standard deviation in order to reduce the variance to a unit. The StandardScaler is part of the dislib library, the parallelism being based on the number of row blocks. This additional step is necessary for the KNN to adjust the values of all the features used to the same range (if the values of the features are in different ranges some features will have a higher impact on the distance of the neighbours than others).

Figure 11 compares the evolution of the training time using the traditional ML algorithms according to the number of cores in the infrastructure. Figure 11a represents the results of the tests on CSVM implementation where each node of the cluster hosted the execution of 6 tasks, each using 8 cores. The results highlight that, for this specific configuration, we can achieve performance improvements thanks to the PyCOMPSs parallelisation, up to 192 cores. Figure 11b depicts the execution of the training of the model using the dislib KNN implementation. The test was conducted using, on each node, up 12 PyCOMPSs tasks (each using 4 cores). Figure 11c depicts the performance of the execution of the RF algorithm. In this case, we experienced some issues related with the scalability, due to a possible mismatch between the number of the blocks, their size and the number of nodes.

Figure 12 depicts the comparison of the execution time required to train the model with the different implementations of neural networks using EDDL. In the version without nesting, each epoch can run up to four tasks in parallel; for this reason, we considered two options: i) assign four GPUs in one node to each task (therefore we needed four nodes to host all the epoch training in parallel); ii) assign one GPU to each task, with a single node we can run the epoch in parallel. Since the version with nesting allows more parallelism (5 folds with 4 tasks in parallel), we could use five nodes assigning one GPU to each task.

Using one single GPU per task achieves better performance than using 4 GPUs since it reduces the communication between the different GPUs as each task will use only one. The dataset is not big enough to fill the 4 GPUs in each node, and the communication between the GPUs is causing unnecessary overhead. Hence, removing the communication will be translated into a reduction of the training time (1.2x faster). Since the nested version allows a higher degree of parallelism; it requires a larger infrastructure to cope with the workload. By parallelizing the training of each fold in a different node, the overall execution time is reduced to 340 seconds (2.24x faster). Although the 5 folds can be trained in parallel, the solution does not achieve a 5x scalability due to the part of the workflow previous to the training of the folds which includes the partitioning and distribution of the dataset.

The scalability of the three machine learning algorithms is limited, no one of them has a good scalability. The solution using Neural Networks scales better than the other three algorithms. However, the solution based on Neural Networks requires specific hardware like GPUs, which is specialized hardware, and their availability may be limited.

Table I compares the accuracy obtained with each algorithm. CSVM obtains an accuracy of 74.9%. Table Ia reports the confusion matrix of one of the 5-folds (a total of 2006 samples) that contains, in the rows, the fraction of instances of true AF and Normal classes, and in the columns the predicted classes. In this run, out of 1013 AF samples, the algorithm correctly predicted 762 samples as AF and 251 were wrongly classified as Normal (false negatives). Out of 993 Normal samples, the algorithm wrongly predicted 251 as AF (false positives) and 742 as Normal (true negatives).

The accuracy obtained with the KNN algorithm is 52%. The confusion matrix of one of the 5-folds (2006 samples) in Table Ib shows that, out of 1003 AF samples, 999 were correctly predicted as AF class (true positives), 4 wrongly classified as Normal class (false negatives); out of 1003 Negative samples, 983 were wrongly classified as AF (false positives) and 20 classified as Normal (true negatives).

The best accuracy was reached using the RF algorithm (86.8%). The confusion matrix of one of the 5 folds (2006 samples) is shown in Table Ic. Where we can see that out of 1012 AF samples it correctly classified 915 as AF (true positives) and 97 as Normal (false negatives). With regard to the Normal class samples, there are a total of 994 and the model incorrectly classifies 143 as AF (false positives) and 851 correctly as Normal (true negatives).

With the CNN implementation, the accuracy obtained is 90%. That increases the performance with respect to the

	Prediction	
	AF	N
AF	0.379	0.125
N	0.125	0.369

(a) CSMV

	Prediction	
	AF	N
AF	0.498	0.001
N	0.490	0.009

(b) KNN

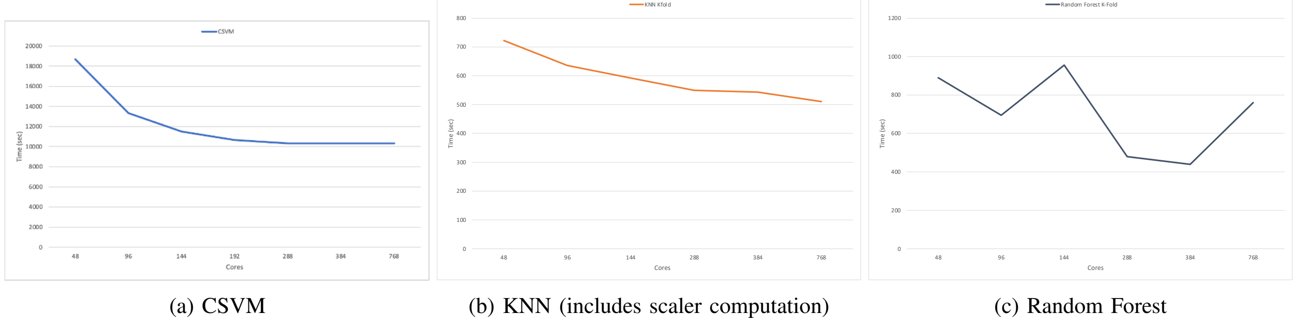
	Prediction	
	AF	N
AF	0.456	0.048
N	0.071	0.424

(c) RF

	Prediction	
	AF	N
AF	0.454	0.066
N	0.009	0.469

(d) CNN

Table I: Average confusion matrices of the 5-folds obtained for the executed algorithms



(a) CSVM (b) KNN (includes scaler computation) (c) Random Forest

Figure 11: Execution time of the algorithms in the Marenostrum cluster

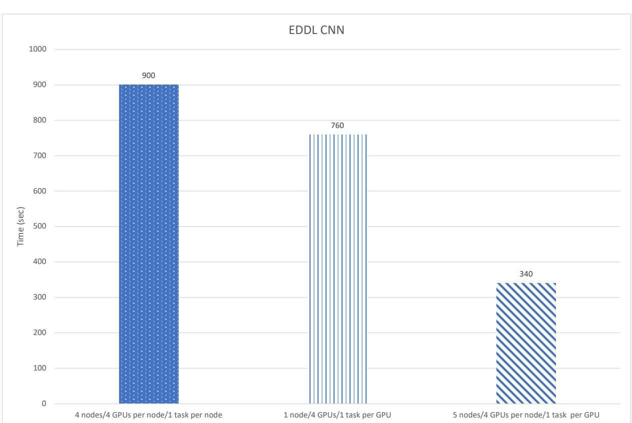


Figure 12: Performance results of the runs of training executions using EDDL on a GPU cluster

previous tests using classic ML algorithms. This accuracy is slightly lower than the accuracy obtained with the most recent models like the one present on [21], which was of 93%. However, the accuracy obtained using both solutions is comparable and the main focus of this paper was not to reach or surpass the actual solutions accuracy but to also tackle the scalability and distribution of the computation. The confusion matrix obtained using our CNN solution is shown in Table Id. Out of 1045 AF samples, it correctly classified 911 as AF (true positives) and 134 as Normal (false negatives). For the 961 Normal class samples, it correctly classified 19 as AF (false positives) and 942 as N (true negatives).

V. Conclusions

In this work, we presented an application leveraging distributed techniques for large-scale ML workflows, specifically for AF classification using ECG recordings. Our implementation has shown promising results, highlighting the potential of distributed ML in improving prevention and patient management in stroke care. Further research and technical developments in this domain hold great promise for advancing healthcare and improving patient outcomes.

From an application standpoint, it is important to emphasize that different scenarios and requirements can influence the selection of the most suitable model and training approach, especially in healthcare applications. Specifically, our exploration of different models and training solutions for stroke care focuses on three main dimensions: (1) training time; (2) available resources; (3) model performance. Depending on the application, it is desirable to prioritize one or more of these dimensions. Considering training time is crucial when processing; speed and responsiveness are important for instance when retraining a model with real-time data streams. In such cases, models with poor training scalability, like RF in our case, should be avoided. The number of available resources is important when there are limitations related to this aspect. In such a scenario, models trained on GPUs, such as CNN in our case, should be avoided. Considering model performance is crucial for high-stakes applications, such as in stroke care. In such cases, more important than overall accuracy is choosing a model based on clinical priorities, specifically whether it should have a precision focus or a recall focus. A precision focus aims to minimize false positives, while a recall focus aims to minimize false negatives. In the context of real-world stroke intervention, it is preferable for a classifier to predict a normal signal as AF (false positive) rather than predicting AF as a normal signal (false negative).

Future work encompasses the improvement of the scalability of all the presented algorithms, specifically RF as it showed the worst scalability compared with the others. Moreover, our approach could incorporate federated learning in the future to train multiple models, which is particularly relevant for healthcare applications due to privacy constraints on data sharing. In this setup, various devices with local data contribute to training local models, and the resulting outcomes are then combined by a general model.

Acknowledgment

Author Fernando Vázquez is supported by PRE2022-104134 funded by MICIU/AEI /10.13039/501100011033 and by the FSE+. This work has been supported by the Spanish Government (PID2019-107255GB) and by MCIN/AEI /10.13039/501100011033 (CEX2021-001148-S), by Generalitat de Catalunya (contract 2021-SGR-00412), and by the European Commission through the Horizon Europe Research

and Innovation program under Grant Agreement 101016577 (AI-SPRINT project).

REFERENCES

[1] B. Campbell and P. Khatri, "Stroke," *The Lancet*, vol. 396, no. 10244, pp. 129–142, Jul. 2020. [Online]. Available: https://doi.org/10.1016/s0140-6736(20)31179-x

[2] "Global, regional, and country-specific lifetime risks of stroke, 1990 and 2016," *New England Journal of Medicine*, vol. 379, no. 25, pp. 2429–2437, Dec. 2018. [Online]. Available: https://doi.org/10.1056/nejmoa1804492

[3] M.J. O'Donnell et al., "Risk factors for ischaemic and intracerebral haemorrhagic stroke in 22 countries (the INTERSTROKE study): a case-control study," *The Lancet*, vol. 376, no. 9735, pp. 112–123, Jul. 2010. [Online]. Available: https://doi.org/10.1016/s0140-6736(10)60834-3

[4] S.S. Chugh et al., "Worldwide epidemiology of atrial fibrillation," *Circulation*, vol. 129, no. 8, pp. 837–847, Feb. 2014. [Online]. Available: https://doi.org/10.1161/circulationaha.113.005119

[5] K. Bayoumy et al., "Smart wearable devices in cardiovascular care: where we are and how to move forward," *Nature Reviews Cardiology*, vol. 18, no. 8, pp. 581–599, Mar. 2021. [Online]. Available: https://doi.org/10.1038/s41569-021-00522-7

[6] A.L. Goldberger et al., "PhysioBank, PhysioToolkit, and PhysioNet," *Circulation*, vol. 101, no. 23, Jun. 2000. [Online]. Available: https://doi.org/10.1161/01.cir.101.23.e215

[7] G. Clifford et al., "Af classification from a short single lead ecg recording: The physionet/computing in cardiology challenge 2017," Feb 2017. [Online]. Available: https://physionet.org/content/challenge-2017/1.0.0/

[8] I.A. Kakadiaris et al., "Machine learning outperforms ACC/AHA CVD risk calculator in MESA," *Journal of the American Heart Association*, vol. 7, no. 22, Nov. 2018. [Online]. Available: https://doi.org/10.1161/jaha.118.009476

[9] A.M. Alaa et al., "Cardiovascular disease risk prediction using automated machine learning: A prospective study of 423, 604 UK biobank participants," *PLOS ONE*, vol. 14, no. 5, p. e0213653, May 2019. [Online]. Available: https://doi.org/10.1371/journal.pone.0213653

[10] A.S. Tseng and P.A. Noseworthy, "Prediction of atrial fibrillation using machine learning: A review," *Frontiers in Physiology*, vol. 12, Oct. 2021. [Online]. Available: https://doi.org/10.3389/fphys.2021.752317

[11] S. Aziz, S. Ahmed, and M.-S. Alouini, "ECG-based machine-learning algorithms for heartbeat classification," *Scientific Reports*, vol. 11, no. 1, Sep. 2021. [Online]. Available: https://doi.org/10.1038/s41598-021-97118-5

[12] F. Lordan, R. M. Badia *et al.*, "ServiceSs: an interoperable programming framework for the Cloud," *Journal of Grid Computing*, vol. 12, no. 1, pp. 67–91, 3 2014.

[13] J. Álvarez Cid-Fuentes et al., "dislib: Large Scale High Performance Machine Learning in Python," in *Proceedings of the 15th International Conference on eScience*, 2019, pp. 96–105.

[14] E. Tejedor et al., "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017. [Online]. Available: https://doi.org/10.1177/1094342015594678

[15] S. Datta et al., "Identifying normal, af and other abnormal ecg rhythms using a cascaded binary classifier," in *2017 Computing in Cardiology (CinC)*, 2017, pp. 1–4.

[16] Hesi, *Comprehensive review for the nclex-pn(r) examination - E-book*, 7th ed., D. M. Korniewicz, Ed. Elsevier, Oct. 2022.

[17] C. Carreiras, A. P. Alves, A. Lourenço, F. Canento, H. Silva, A. Fred *et al.*, "BioSPPy: Biosignal processing in Python," 2015–, [Online; accessed]. [Online]. Available: https://github.com/PIA-Group/BioSPPy/

[18] J. Huang et al., "ECG arrhythmia classification using STFT-based spectrogram and convolutional neural network," *IEEE Access*, vol. 7, pp. 92 871–92 880, 2019. [Online]. Available: https://doi.org/10.1109/access.2019.2928017

[19] P. Virtanen et al., "SciPy 1.0: fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, Feb. 2020. [Online]. Available: https://doi.org/10.1038/s41592-019-0686-2

[20] J. Flich, C. Hernandez, E. Quiñones, and R. Paredes, "Distributed training on a highly heterogeneous hpc system," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2020, pp. 359–370.

[21] X. Zhao, R. Zhou, L. Ning, Q. Guo, Y. Liang, and J. Yang, "Atrial fibrillation detection with single-lead electrocardiogram based on temporal convolutional network–resnet," *Sensors*, vol. 24, no. 2, p. 398, 2024.

Appendix

Artifact Description

In order to facilitate the reproducibility of the experiments contained in this article we uploaded to a public repository the scripts with the code used in the experiments. The public repository is a GitHub project and its link is: https://github.com/lezzidan/aisprint_dislib. The repository contains also the launching scripts used in MareNostrum4 and Power-9. The unique requirement to reproduce correctly the experiments is to adjust the number of nodes desired to use.

In the repository there is a folder which contains the scripts of the Neural Networks, both the nesting and the normal execution together with the corresponding launching scripts.

This repository is still alive, and their scripts may suffer changes from the versions used in this article. For this reason we made a release in GitHub available at https://doi.org/10.5281/zenodo.13836996.

The dataset used in the experiments can be downloaded from B2Drop, using the following URLs: https://b2drop.bsc.es/index.php/s/8Q8MefXX2rrzaWs/download?path=%2Fdata&files=balanced_training2017.tar.gz&downloadStartSecret=kndjafzsfu, for the data used in the training experiments, and https://b2drop.bsc.es/index.php/s/8Q8MefXX2rrzaWs/download?path=%2Fdata&files=balanced_validation2017.tar.gz for the data used to compute the validation accuracy of the models.

Using the data contained on this repository together with the scripts in the GitHub the tests can be easily reproduced on a supercomputer or cluster where dislib, EDDL and PyCOMPSs are installed. A docker container using the Dockerfile contained in the GitHub repository can be generated in order to execute the tests without the need of installing PyCOMPSs neither dislib.

An execution of each of the machine learning algorithms were documented using provenance. The information of the executions obtained using provenance was uploaded to workflowhub in order to ensure the reproducibility and register the details of the executions. The CascadeSVM information execution is on https://workflowhub.eu/workflows/1124, the execution of kNN is on https://workflowhub.eu/workflows/1123 and the Random Forest information execution on https://workflowhub.eu/workflows/1122.

We registered the logs of various executions of the kNN algorithm, which are uploaded to Zenodo: https://zenodo.org/records/7426459, because we included traces of the executions.

This document has been uploaded to Zenodo and a DOI has been created for it. The DOI is: https://zenodo.org/doi/10.5281/zenodo.13691927

Artifact Evaluation

In order to reproduce the experiments it is required to have installed dislib-0.9.0 (https://dislib.readthedocs.io/en/release-0.9/), COMPSs (https://compss-doc.readthedocs.io/en/stable/), the python binding PyCOMPSs, and the Neural Network library EDDL(https://deephealthproject.github.io/eddl/) and its python binding PyEDDL(https://deephealthproject.github.io/pyeddl/).

COMPSs and its binding PyCOMPSs both have an installation manual. This manual is publicly available at: https://compss-doc.readthedocs.io/en/stable/Sections/01_Installation.html. Following the instructions on this manual it is possible to install them in a personal laptop or in a Supercomputer or cluster.

Like COMPSs and PyCOMPSs, both EDDL and PyEDDL have installation manual. The manual to install the EDDL library is available at https://deephealthproject.github.io/eddl/intro/installation.html, and the instructions to install PyEDDL are available at https://deephealthproject.github.io/pyeddl/installation.html.

Dislib can be easily installed through the usage of pip:

```
> python3 -m pip install dislib
```

Once all the requirements are installed, the executions registered in WorkflowHub can be reproduced by executing the instructions to re-execute a COMPSs workflow without data persistence that are in the following URL: https://compss-doc.readthedocs.io/en/stable/Sections/05_Tools/04_Workflow_Provenance.html?highlight=reproducibility#re-execute-a-compss-workflow-published-in-workflowhub.

Once this workflows have been correctly executed, in order to execute the rest of the executions present on the paper it will only be required to change the number of computing nodes in the enqueue_compss command.

Other form to reproduce the machine learning experiments is to download the source code and the launching script from the Zenodo link https://doi.org/10.5281/zenodo.13836996, adapting the paths, changing the python script for the correct one in the bash scripts, and execute them just like (e.g.):

```
> ./launch_train_kfold.sh
```

The README.md contains additional instructions on how to change the configuration for different machines, including in a local machine.

In order to reproduce the Neural Network experiments it will be required to download the source code and the launching scripts from the GitHub repository https://github.com/lezzidan/aisprint_dislib. Upload them to the supercomputer or cluster where the experiments are going to be reproduced and launch each of the experiments by adjusting the required paths in the enqueue and bash scripts. Then this scripts can be easily launched just by doing (e.g.):

```
> ./launch_train_4_gpus_per_work.sh
```