DSL for parallelizing Machine Learning algorithms on multicore architecture

Nel Gerbault Nanvou Tsopgny, Thomas Messi Nguélé, Etienne Kouakam

DSL for parallelizing Machine Learning algorithms on multicore architecture

Nel Gerbault NANVOU TSOPGNY[1], Thomas MESSI NGUELÉ[1], Etienne KOUAKAM[1]

[1]University of yaounde 1, Department of Computer Science, PO:812, yaounde, Cameroon

*E-mail : [gerbault.nanvou, thomas.messi, etienne.kouokam]@facsciences-uy1.cm

Abstract

Machine Learning algorithms must run on large amounts of data in order to produce powerful classification, regression, and clustering models. The larger the size of the data required to run these algorithms, the higher the execution time of these algorithms. Programmers of machine learning-related applications can take advantage of the rise of multi/many core architectures to reduce this long runtime. However, these programmers may find it difficult to write efficient parallel programs that run on these architectures because they used to implemente these algorithms sequentially. It is therefore difficult to write low-level parallel code specific to the platform. Several DSLs have already been proposed in the context of parallelizing Machine Learning algorithms. But most of them are embedded in high level languages such as Python (case of Qjam) or Scala (case of OptiML). In order for such an (embedded) DSL to produce code with good performances (execution time and speedup), the host language must have intrinsic characteristics allowing to have them. In this paper, we propose FastML, a Domain Specific Language embedded in the C language. The idea of FastML is to offer to the programmer learning primitives (such as gradient descent) already parallelized according to the Map-Reduce model, that he will just have to call by specifying the parameters, depending on the Machine Learning algorithm he wants to implement. The first experiments carried out on a machine with 8 cores and 8GB of RAM show that FastML gives promising results in terms of speedup compared to the OptiML DSL and the Scikit-learn platform. For example, with kmeans, FastML produces 4x as speedup (with 7 cores) compared to 1x for Scikit-learn and 0.70x for OptiML.

Keywords

Machine Learning (ML); Domain Specific Language (DSL); Parallelization.

I INTRODUCTION

Machine Learning a sub-branch of Artificial Intelligence that consists in making algorithms automatically learn to perform a task from data. Once trained, the models can be used to solve problems such as regression, classification or clustering on new data with minimal human intervention. Applications include: drug detection, cyber security, automatic driving, genetics, speech recognition. For better inference on data in these applications, the algorithms need to be run on large amounts of data; but the larger the amount of data needed for this execution, the higher the execution time of these algorithms.

For two decades, the solution of increasing the frequency of single-core processors has allowed

to increase the speed of applications. However, since 2003, this solution is no longer feasible due to the problems of power consumption and heat dissipation that have limited the increase in the clock frequency of processors. Thus, several suppliers of processors have moved to models where several processing units, called *core* of processor are used: We speak of multi/many-core processor.

Another solution to reduce this execution time is the use of these multi/many-core processors which are more and more preponderant and which have the advantage of having several low-frequency computing units. However, the programmer encounters difficulties in writing efficient parallel programs that run on these architectures. One way to simplify the job is to choose a language that provides implicit parallelization. This way, the programmer will not have to worry about how to parallelize his algorithm. For that, we will use a DSL approach (*Domain Specific Language*). DSLs are languages designed to solve problems in a particular domain. An example of a DSL is Structured Query Language (SQL), for database queries.

Several DSLs have already been proposed in the context of parallelization of Machine Learning algorithms. But most of them are embedded in high-level languages such as Python (in the case of Qjam) or Scala (in the case of OptiML). Thus, their performance in terms of execution time or speedup is dependent on that of the host language. For such an (embedded) DSL to produce code with good performance (execution time and speedup), the host language must have intrinsic characteristics that allow it to do so. These are characteristics that allow the programmer to optimize his code. Some languages like C have these characteristics.

In this article, we propose FastML, a DSL embedded in the C language for the parallelization of algorithms of the *Machine Learning*. The algorithms concerned by this parallelization are those corresponding to the statistical query model [5] and the paradigm used for the parallelization is the Map-Reduce paradigm [11].

This paper is structured as follows: In section II, we present the background on parallel programming and on DSLs and the related work, in section III, a detailed presentation of FastML; in section IV, we present our experiments, the results obtained and their interpretations and finally, we conclude this work and present some perspectives in section V.

II BACKGROUND

2.1 parallel programming

Traditionally, software is based on sequential calculations: A problem is broken down into instructions; these instructions are executed sequentially one after the other by a single processor. Parallelization refers to being able to perform multiple tasks simultaneously in order to reduce application execution time. There are three (03) major parallelization techniques [12] : instruction level parallelization, task level parallelization and data level parallelization. The last one considered in this paper, consists of dividing the initial dataset into several blocks, and simultaneously executing a task on these different data blocks.

To make it possible to run parallel programs, specific machine architectures have been developed. The different types of architecture are characterized by the different models of interconnection between processors and memory. Among these architectures, we have SMPs (*Symmetric Multiprocessors*), cc-NUMA (*non-uniform memory access coherent cache*), clusters or clusters, grids, graphics processors GPU (*Graphic Processing Unit*), FPGA (*Field Programmable*

Gate Array) and multi/many core systems. A multi/many core processor is made up of at least two (02) computing units (cores) etched on a single chip. A many-core processor is often considered as a multi-core processor, with more than 30 cores. Each core can execute a thread at a given time.

Parallel programming models are programming techniques that make it possible to exploit the architectures mentioned above. More precisely, a parallel programming model expresses the way in which a parallel application will be programmed. Models based on distributed memory and shared memory architectures [12] are the most widely used for parallel programming. In shared memory systems, all of the parallel machine's processors share the same memory space through which they can exchange information through memory reads and writes. On these machines, mechanisms are provided to create processes. The programmer then manages the synchronization between the processes using low-level primitives such as locks, semaphores, etc. On multi/many core architectures, several libraries have been developed to provide these mechanisms among which, Cilk, Intel TBB, OpenMp, and POSIX Thread.

The efficiency due to parallelization is often measured by:

- **Speedup**: this is the ratio between the execution time of the sequential program and the execution time of the parallel program.
- **Efficiency**: this is the ratio between speedup and the number of computing units.
- **Scalability**: This expresses the increase in efficiency with the number of compute units.

2.2 DSLs

To write computer applications, we need a language that will mediate between humans and machines and vice versa. In computer science, programming languages are divided into two: GPLs (General Purpose Language) designed to solve general problems (Java, C, Python ...) and DSLs (Domain Specific Language) designed to solve problems in a specific domain; (OptiML[15], QJAM[16] Green-marl, Galois ...).

There are two categories of DSLs:

- external or stand-alone DSLs: They have their compilers or interpreters independent of other languages.
- Embedded DSLs: they are usually implemented in a host language as a library. FastML proposed in this paper is embedded in C.

2.3 Related work

Since the advent of multi-core systems, several works have been done in the field of machine learning parallelization algorithms:

Cheng-Tao Chu et al.[10] adapt the MapReduce paradigm to demonstrate how 10 machine learning algorithms can be parallelized on multi-core systems. Among others, locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). FastML was inspired by this adaptation.

Juan Batiz-Benet et al. [16] present Qjam, a python framework for the rapid prototyping of parallelization of machine learning algorithms. That is, Qjam doesn't care about the final execution time. The idea behind FastML was to do a DSL that does more than the prototyping (produce also programs that use the full potential of the machine).

Tiark Rompf et al. [15] present Optiml, DSL for the parallelization of machine learning algorithms which aims to bridge the gap between algorithms and heterogeneous hardware in order to provide a productive and efficient programming environment. Although it seems efficient, OptiML is embedded in scala, which is a high level language compared to C. We expect that a parallel DSL embedded in C like FastML may produce more efficient programs.

	Chu et al (2007)	OptiML (2011)	Qjam (2012)
Implemented	no	yes	yes
Code avialable	-	yes	no
Prototyping	-	no	yes
Hight level implemented language	-	yes (scala)	yes (python)
Usability	-	difficult to use	-
number of code lines	-	few	-

As shown in the table above, for the DSLs proposed in the literature, the Qjam code is not available, which makes it difficult to compare. As for OptiML, although its number of lines of code is small, it remains quite difficult to use, which may explain the fact that since its release in 2011, it is very little used, and even worse, very little known.

We hope that the DSL we propose in this article will be easy to use, popular and embedded in a low-level language.

III FASTML PRESENTATION

We proposed FastML [1], a DSL embedded in C for the parallelization of Machine Learning algorithms. In this section, we present in detail FastML starting with the idea of parallelization behind it, its architecture, then its programming interface and finally an example of two algorithms implemented with FastML (Linear regression and Logistic regression).

3.1 Parallelization idea behind FastML

The idea of FastML is to offer programmers learning primitives (such as gradient descent) already parallelized according to the Map-Reduce model. All they have to do is call them by specifying the parameters, depending on the Machine Learning algorithm they want to implement. In this paper, we parallelize two categories of learning algorithms; those based on gradient descent and those based on distance calculation. The principle consists in separating the dataset, and assigning different fragments to different threads, while making sure during the aggregation that the result (in term of accuracy) of the algorithm is maintained, compared to the sequential version of this algorithm.

[1] **FastML** is avialable at ***https://gitlab.com/gnelnanvou/fastml.git***

3.2 FastML Architecture

We modeled FastML parallelisation on Google's Map-Reduce architecture [11]. The Map-Reduce is a programming model that provides a framework to automate parallel computing on big data. This is the "divide, share and rule" model. That means, the resolution of a problem is done by dividing the data set into several blocks or fragments, and assigning these fragments to the available computing nodes which will perform the same operation on it in parallel. It is based on two main operations, which are MAP which specifies the operation that will be performed on different data fragments and REDUCE which is the operation of aggregating the results of the different mappers (nodes performing the MAP operation) to produce the final result.

In order to allow Machine Learning application developers to take advantage of multicore systems, FastML provides for them a set of parallelized learning functions commonly used in Machine Learning. For a Machine Learning algorithm to be implemented, the programmer just needs to identify the ideal learning function and call it by filling in the necessary parameters.

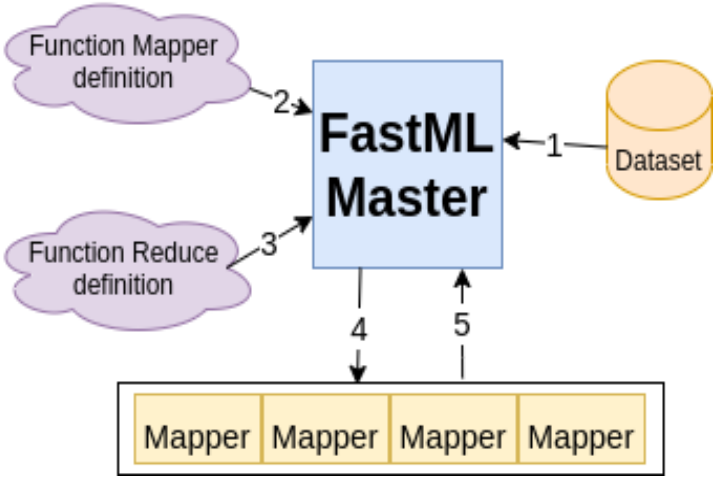

Figure 1: FastML parallelisation architecture

Figure 1 shows the FastML parallelisation architecture. In order to make parallelisation using FastML, the following steps shown in figure 1 are respected:

1. The user defines his dataset.
2. The user defines the task that will run on different compute nodes (*Mapper function*).
3. The user defines how the intermediate results will be aggregated to produce the final result (Reduce function).
4. The FastML engine divides the dataset into fragments and creates threads that will operate on these fragments.
5. The FastML engine aggregates the intermediate results and makes the final results available to the programmer.

3.3 FastML programming interface

Since FastML is embedded in C, its syntax is therefore similar to that of the C language. However, the FastML code must be written according to the formalism described below:

- **Base type**: in addition to the base types of the C language (int, float, double, char ...), FastML defines two new types *matrice* and *matriceChar*. These are the *double* and *string* matrices, respectively. They support some operations such as *copy()*, *ones()*, *sum()*, *product()*, *freeMatrice()* ... Each *matrice* or *matricechar* has three (03) entities: n the number of rows, m the number of columns and *contenu* which the matrix itself.
- **Dataset Definition**: The dataset is defined in matrix form. They are four (04) primitives used to read a file dataset and to load it in a matrices: *readMatrice (matrice * data, char * filename)*, *readMatrice_char (matricechar * data, char * filename)*, *readDatas(matrice * data, matrice * label, char * filename ,int nberOfColumn)* and *readDatas_char(matricechar * data, matricechar * label, char * filename ,int nberOfColumn)*.
- **FastML project content**: The FastML project contains a set of files. The matrix.h file contains the signatures of the functions and operations supported by the types *matrix* and *matrixchar*. The file dsl.h contains operations specific to our DSL. These are some functions (already parallelized) used for Machine Learning which the programmer just has to call by filling in the parameters. We will find for example the stochastic gradient descent, used for the learning of many Machine Learning algorithms. In the files *fonctions.h* and *fonctions.c* the programmer has to place respectively the signatures of his own functions and the code of these functions to finally make the call in the file *main.c*. Once completed, compilation and execution is done like any C project.

3.4 Implementation Examples

Here we present the outline of the Linear and Logistic Regression and Kmeans implementation using FastML. We use stochastic gradient descent to train both algorithms. In red, the keywords and types of the C language, in brow, the types specific to FastML, in green, the operations specific to the latter and in black, the functions and operations specific to the programmer.

3.4.1 *Linear Regression*

```
1  matrice linearRegressionAlgorithm (double alpha, double seuil,
                    int n_epoch, char * filename){
2       matrice  X,Y,aux;
3       readDatas(&aux,&Y,filename,12);
4       X = ones(aux.n,1);
5       concatColoumn(&X,aux);
6       freeMatrice(&aux);
7      top_(&w_time_par1,&tz);
8      matrice w = SGD( X, Y,means_square_error,
                means_square_error_gradient,
                alpha, seuil, n_epoch);
9      top_(&w_time_par2,&tz);
10      work_time_par = cpu_time_(w_time_par1, w_time_par2);
11      printf(" ^temps d'execution Regression Logistique avec %d
                threads=%lld.%03lldms",N_THREADS,
                work_time_par/1000, work_time_par%1000);
12      freeMatrice(&X);
13      freeMatrice(&Y);
14      return w;
15  }
```

3.4.2 *Logistic Regression*

```
1  matrice logisticRegressionAlgorithm (double alpha, double seuil,
                    int n_epoch, char * filename){
2       matrice  X,Y,aux;
3       readDatas(&aux,&Y,filename,12);
4       X = ones(aux.n,1);
5       concatColoumn(&X,aux);
6       freeMatrice(&aux);
7       for (int i = 0; i < Y.n; i++){
8           Y.contenu[i][0] = (Y.contenu[i][0]<=5)0:1;
9       }
10      top_(&w_time_par1,&tz);
11      matrice w = SGD( X, Y,cross_entropy_binaryclass,
                cross_entropy_binaryclass_gradien,
                alpha, seuil, n_epoch);
12      top_(&w_time_par2,&tz);
13      work_time_par = cpu_time_(w_time_par1, w_time_par2);
14      printf(" temps d'execution Regression Logistique avec %d
                threads=%lld.%03lldms",N_THREADS,
                work_time_par/1000, work_time_par%1000);
15      freeMatrice(&X);
16      freeMatrice(&Y);
17      return w;
18  }
```

3.4.3 *Kmeans*

```
1  matrice * kmeans(int k,int n_epoch,double seuil,char * filename){
2       matrice  X;
3       readMatrice(&X,filename);
4       matrice * clusters = (matrice *)malloc(k * sizeof(matrice));
5       matrice * old_clusters;
6       initialize_centroid(clusters, X, k);
7       int iter = 1;
8       double error = seuil;
9       top_(&w_time_par1,&tz);
10      while(n_epoch >= iter && error >= seuil){
11          matrice index = determineMeilleurCluster(X, clusters, k);
            old_clusters = clusters;
12          clusters = update_centroid(X, index, k);
13          freeMatrice(&index);
14          error = calcul_erreur(old_clusters,clusters, k);
15          printf("Epoch %d : error  = %4.10f",iter,error);
16          iter++;
17     }
18      top_(&w_time_par2,&tz);
19      work_time_par = cpu_time_(w_time_par1, w_time_par2);
20      printf("temps d'execution Kmeans avec %d threads
                = %lld.%03lldms",N_THREADS,
                work_time_par/1000, work_time_par%1000);

21      return clusters;
22 }
```

The linear and logistic regression are almost similar. The gradient descent called in lines 8 and 11 respectively in the linear and logistic regression code marks the small difference. Indeed, in this function the third and fourth parameters represent respectively the loss function used and the gradient of this one. For example, for linear regression, it is the mean square error. For the Kmeans algorithm, a particular function "*determineBestCluster*" allows to compute the distances in parallel, and determine for each point the cluster to which it is closer.

In the following section, we present our experiments by comparing them with the results obtained with the famous DSL OptiML and Scikit-learn tool.

IV PRELIMINARY RESULTS WITH FASTML

In this section, we present the experimentation environment and for linear and logistic regression and Kmeans algorithm, the speedup gotten with FastML in comparison with the speedup gotten with OptiML and Scikit-learn.

4.1 Experimentation environment

For this preliminary results, the experiments were done on a user multi-core machine with the following characteristics: 8 cores at 2.20 GHz, 8 GB Ram.

We implemented and executed the linear regression, the logistic regression and Kmeans algorithm with FastML, OptiML and Scikit-learn. We used the wineQuality dataset from the UCI Machine Learning repository.

4.2 Improving Speedup with FastML

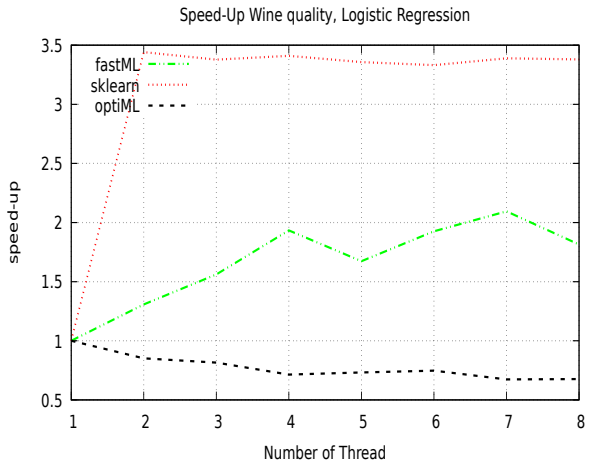

Figure 2: Logistic regression Speedup

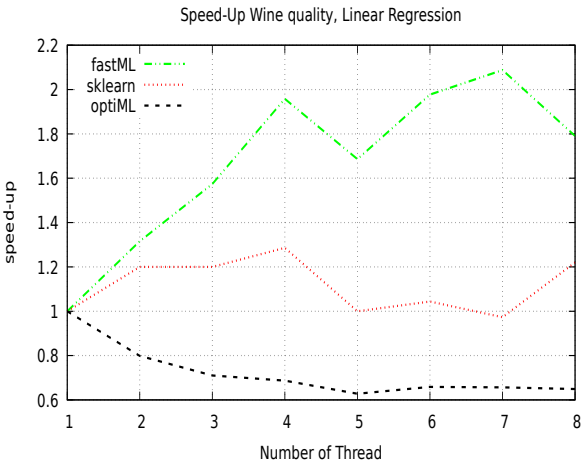

Figure 3: Linear regression Speedup

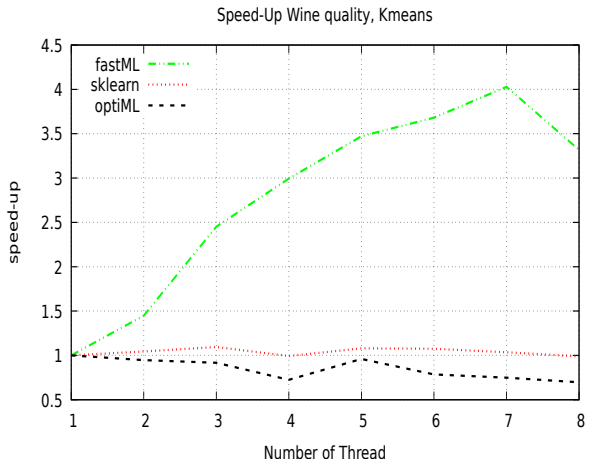

Figure 4: Kmeans Speedup

Figure 2, 3 and 4 show the speedup of logistic and linear regression and Kmeans algorithms implemented with FastML, OptiML and Scikit-learn respectively. The executions were made with different numbers of cores, from one (sequential execution) to height cores.

We note that the speedup gotten with FastML (4x for Kmeans and 2.1x for linear regression) is greater than the speedup gotten both with OptiML (1x for Kmeans and 0.70x for linear regression) and Scikit-learn (0.73x for Kmeans and 1.00x for linear regression). These results means that, for these algorithms, the parallelization with FastML is better than the parallelization with OptiML.

V CONCLUSION AND REFERENCES

This paper proposes a Domain Specific Language embedded in the C language called FastML. The idea of FastML is to offer to the programmer already parallelized learning primitives (such as gradient descent) according to the Map-Reduce model. We test FastML on three machine learning algorithms: linear regression, logistic regression and kmeans. The experiments carried out on a user machine with 8 cores show that FastML gives promising results in terms of speedup compared to the OptiML DSL and the Scikit-learn platform. For example, with kmeans, FastML produces 4x as speedup (with 7 cores) compared to 1x for Scikit-learn and 0.70x for OptiML.

In this paper, we did not measure the ease of programming with FastML compared to other DSLs and platforms. This should be done later in this work. As future work, we plan to include other famous machine learning algorithms in the study.

REFERENCES

Publications

[1] A. L. Samuel. "Some studies in machine learning using the game of checkers. II—Recent progress". In: *IBM Journal of research and development* 11.6 (1967), pages 601–617.

[2] L. G. Valiant. "A theory of the learnable". In: *Communications of the ACM* 27.11 (1984), pages 1134–1142.

[3] R. Duncan. "A survey of parallel computer architectures". In: *Computer* 23.2 (1990), pages 5–16.

[4] P. Hudak. "Building domain-specific embedded languages". In: *Acm computing surveys (csur)* 28.4es (1996), 196–es.

[5] M. Kearns. "Efficient noise-tolerant learning from statistical queries". In: *Journal of the ACM (JACM)* 45.6 (1998), pages 983–1006.

[6] A. Van Deursen, P. Klint, and J. Visser. "Domain-specific languages: An annotated bibliography". In: *ACM Sigplan Notices* 35.6 (2000), pages 26–36.

[7] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4 (2005), pages 316–344.

[8] H. Sutter and J. Larus. "Software and the Concurrency Revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry." In: *Queue* 3.7 (2005), pages 54–62.

[9] B. Chapman. "The multicore programming challenge". In: *International Workshop on Advanced Parallel Processing Technologies*. Springer. 2007, pages 3–3.

[10] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. "Map-reduce for machine learning on multicore". In: *Advances in neural information processing systems* 19 (2007), page 281.

[11] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pages 107–113.

[12] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen. *Introduction to concurrency in programming languages*. CRC Press, 2009.

[13] D. Patterson. "In Praise of Programming Massively Parallel Processors: A Hands-on Approach". In: *Parallel Computing* (2010).

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pages 2825–2830.

[15] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. “OptiML: an implicitly parallel domain-specific language for machine learning”. In: *ICML*. 2011.

[16] J. Batiz-Benet, Q. Slack, M. Sparks, and A. Yahya. “Parallelizing machine learning algorithms”. In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, Pittsburgh, PA, USA*. 2012, pages 25–27.

[17] T. White. *Hadoop: The definitive guide*. " O’Reilly Media, Inc.", 2012.

[18] M. R. Ghazi and D. Gangodkar. “Hadoop, MapReduce and HDFS: a developers perspective”. In: *Procedia Computer Science* 48 (2015), pages 45–50.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. In: *arXiv preprint arXiv:1603.04467* (2016).

[20] I. Portugal, P. Alencar, and D. Cowan. “A survey on domain-specific languages for machine learning in big data”. In: *arXiv preprint arXiv:1602.07637* (2016).

[21] D. Dua and C. Graff. *UCI Machine Learning Repository*. 2017.

[22] N. Ketkar. “Introduction to pytorch”. In: *Deep learning with python*. Springer, 2017, pages 195–208.

[23] O. Theobald. *Machine learning for absolute beginners: a plain English introduction*. Scatterplot press, 2017.

[24] J. Moolayil, J. Moolayil, and S. John. *Learn Keras for Deep Neural Networks*. Springer, 2019.

[25] H. Al-Sahaf, Y. Bi, Q. Chen, A. Lensen, Y. Mei, Y. Sun, B. Tran, B. Xue, and M. Zhang. “A survey on evolutionary machine learning”. In: *Journal of the Royal Society of New Zealand* 49.2 (2019), pages 205–228.

[26] B. Mahesh. “Machine Learning Algorithms-A Review”. In: *International Journal of Science and Research (IJSR).[Internet]* 9 (2020), pages 381–386.