PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

MATHEUS LYRA DA SILVA

EVALUATING THE PERFORMANCE AND IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS

Porto Alegre

2020

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

EVALUATING THE PERFORMANCE AND IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS

MATHEUS LYRA DA SILVA

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. César A. F. De Rose

Porto Alegre
2020

Ficha Catalográfica

D111e Da Silva, Matheus Lyra

Evaluating the performance and improving the usability of parallel and distributed Word Embedding tools / Matheus Lyra Da Silva . – 2020.

66 p.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Fonticielha De Rose.

1. Word2vec. 2. HPC. 3. Shared Memory. 4. Multicomputers. 5. MPI/OpenMP. I. De Rose, César Augusto Fonticielha. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

MATHEUS LYRA DA SILVA

EVALUATING THE PERFORMANCE AND IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on Mar 30, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

Prof. Dr. Leandro Krug Wives (PPGC/UFRGS)

Prof. Dr. César A. F. De Rose (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

This study was financed by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001 and by the Forense Science National Institute of Science and Technology (INCT) from CNPq Brazil. This study has also been supported by the project “GREEN-CLOUD: Computação em Cloud com Computação Sustentável” (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil.

AVALIANDO A PERFORMANCE E MELHORANDO A USABILIDADE DE FERRAMENTAS DE WORD EMBEDDING PARALELAS E DISTRIBUÍDAS

RESUMO

A representação de palavras por meio de vetores chamada de *Word Embeddings* (WE) vem recebendo grande atenção do campo de Processamento de Linguagem natural (NLP). Modelos WE são capazes de expressar similaridades sintáticas e semânticas, bem como relacionamentos e contextos de palavras em um determinado corpus. Apesar de as implementações mais populares de algoritmos de WE apresentarem baixa escalabilidade, existem novas abordagens que aplicam técnicas de *High-Performance Computing* (HPC). Nesta dissertação é apresentado um estudo interdisciplinar direcionado a utilização de recursos e aspectos de desempenho dos algoritmos de WE encontrados na literatura. Para melhorar a escalabilidade e usabilidade, o presente trabalho propõe uma integração para ambientes de execução locais e remotos, que contém um conjunto das versões mais otimizadas. Usando estas otimizações é possível alcançar um ganho de desempenho médio de 15x para *multicores* e 105x para *multinodes* comparado à versão original. Há também uma grande redução no consumo de memória comparado às versões mais populares em Python. Uma vez que o uso apropriado de ambientes de alta performance pode requerer conhecimento especializado de seus usuários, neste trabalho também é proposto um modelo de otimização de parâmetros que utiliza uma rede neural *Multilayer Perceptron* (MLP) e o algoritmo *Simulated Annealing* (SA) para sugerir conjuntos de parâmetros que considerem os recursos computacionais, o que pode ser um auxílio para usuários não especialistas no uso de ambientes computacionais de alto desempenho.

Palavras-Chave: Word2vec, HPC, Memória Distribuída, Multicomputadores, MPI, OpenMP.

EVALUATING THE PERFORMANCE AND IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS

ABSTRACT

The representation of words by means of vectors, also called Word Embeddings (WE), has been receiving great attention from the Natural Language Processing (NLP) field. WE models are able to express syntactic and semantic similarities, as well as relationships and contexts of words within a given corpus. Although the most popular implementations of WE algorithms present low scalability, there are new approaches that apply High-Performance Computing (HPC) techniques. This is an opportunity for an analysis of the main differences among the existing implementations, based on performance and scalability metrics. In this Dissertation, we present an interdisciplinary study that addresses resource utilization and performance aspects of known WE algorithms found in the literature. To improve scalability and usability we propose an integration for local and remote execution environments that contains a set of the most optimized versions. Utilizing these optimizations it is possible to achieve an average performance gain of 15x for multicores and 105x for multinodes compared to the original version. There is also a big reduction in the memory footprint compared to the most popular Python versions. Since an appropriated use of HPC environments may require expert knowledge, we also propose a parameter tuning model utilizing an Multilayer Perceptron (MLP) neural network and Simulated Annealing (SA) algorithm to suggest the best parameter setup considering the computational resources, that may be an aid for non-expert users in the usage of HPC environments.

Keywords: Word2vec, HPC, Shared Memory, Multicomputers, MPI, OpenMP.

LIST OF FIGURES

Figure 2.1 – CBOW e Skip-Gram model architectures. 24
Figure 2.2 – A typical Multilayer Perceptron neural network architecture. 26
Figure 2.3 – Hill Climb problem. 28
Figure 3.1 – Speed-up of Word Embedding algorithms generators. *Estimated values based on experiments. 32
Figure 3.2 – Memory consumption of the analyzed implementations over time. . . 33
Figure 3.3 – Memory consumption of the analyzed algorithms. 34
Figure 4.1 – Wrapper library modules scheme. 40
Figure 4.2 – Wrapper library for the optimized Word2vec execution environments. 41
Figure 4.3 – Sequence diagram of the wrapper library Job Execution. 42
Figure 4.4 – Speed-up comparison among wrapper library and original Word2vec executions. *Estimated values based on experiments. 44
Figure 4.5 – Parameter tuning model utilizing Multi-layer Perceptron and Simulated Annealing. 45
Figure 4.6 – Learning process example. 47
Figure 5.1 – Accepted solutions vs Execution time. 55

LIST OF TABLES

Table 3.1 – Fastest processing time of the analyzed implementations. 31
Table 3.2 – Extrinsic evaluation of trained models. 35
Table 3.3 – Models evaluation. 36
Table 3.4 – HPC and WE algorithms from state-of-the-art. 37
Table 4.1 – Range of values from dataset parameters. 46
Table 4.2 – Multilayer Perceptron predictions. 47
Table 4.3 – Simulated Annealing predictions. 49
Table 4.4 – Simulated Annealing predictions for locked variables. 49
Table 5.1 – Comparison between default parameters and suggested parameters. . 54

LIST OF ACRONYMS

API – Application Program Interface

ASSIN – *Avaliação de Similaridade Semântica e Inferência Textual*

CBOW – Continuous Bag-of-Words

HPC – High-Performance Computing

LM – Language Models

ML – Machine Learning

MSE – Mean Squared Error

MLP – Multi-layer Perceptron

MPI – Message Passing Interface

NLP – Natural Language Processing

PBS – Portable Batch System

PROPOR – International Conference on the Computational Processing of Portuguese

SA – Simulated Annealing

WE – Word Embedding

CONTENTS

1 INTRODUCTION 21

2 BACKGROUND AND STATE-OF-THE-ART 23

2.1 WORD EMBEDDING ALGORITHMS 23

2.1.1 WORD2VEC TRAINING PROCESS 23

2.1.2 WANG2VEC TRAINING PROCESS 24

2.1.3 FASTTEXT TRAINING PROCESS 24

2.2 PARALLELISM IN SHARED AND DISTRIBUTED MEMORY 25

2.3 NEURAL NETWORKS 26

2.3.1 MULTILAYER PERCEPTRON 27

2.4 SIMULATED ANNEALING ALGORITHM 27

3 EVALUATION OF THE AVAILABLE WORD EMBEDDING IMPLEMENTATIONS 29

3.1 PERFORMANCE EVALUATION 30

3.2 MEMORY CONSUMPTION EVALUATION 32

3.3 MODELS' QUALITY EVALUATION 35

3.4 USABILITY OF WORD EMBEDDING ALGORITHMS IN THE NATURAL LANGUAGE PROCESSING CONTEXT 36

4 IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS 39

4.1 INTEGRATION FOR OPTIMIZED WORD EMBEDDINGS 39

4.2 PARAMETER TUNING TECHNIQUE AND AUTOMATICALLY SUGGESTIONS 44

4.2.1 PARAMETER TUNING MODEL 44

5 WRAPPER LIBRARY AND PARAMETER TUNING EVALUATION 51

5.1 WRAPPER LIBRARY EVALUATION 51

5.2 PARAMETER TUNING EVALUATION 54

6 RELATED WORK 57

6.1 HIGH-PERFORMANCE COMPUTING AND NATURAL LANGUAGE PROCESSING FOCUSED FRAMEWORKS 57

6.2 SIMULATED ANNEALING ALGORITHM APPLIED TO PARAMETER TUNING PROBLEMS 58

7 CONCLUSION **61**

REFERENCES **63**

1. INTRODUCTION

Natural Language Processing (NLP) is a theory-motivated range of computational techniques for automatic analysis and representation of human language [6]. This area has grown into both scientific research and practical technology that is increasingly being incorporated into consumer products, employing computational techniques for the purpose of learning, understanding, and producing human language content [18]. For several decades the NLP research has been focusing on tasks such as machine translation, information retrieval, text summarization, question answering, information extraction, topic modeling, and opinion mining [6]. More recently, the language representation model Word Embedding (WE) has gained visibility in the NLP field. This kind of model represents words by means of vectors and enables findings such as syntactic and semantic similarities, as well as relations and word contexts within a given corpus (input text). The algorithms identify linguistic patterns and allow algebraic operations over the represented words. For example:

$$[Madrid] - [Spain] + [France] \simeq [Paris] \tag{1.1}$$

Where [*Paris*] would have the vector that most approximates the operation in Equation 1.1[33].

The example above shows an operation over the respective vectors of the words Madrid, Spain, and France. The result of this operation will be closer to the equivalent vector of the word Paris than any other. In this case, the model was able to capture the relationship between a country and its capital. This example shows that the WE output model learns what each word represents in a given context. These WE models can be applied in different fields, such as forensic science, where the models could be used to extract text from documents related to criminal investigations. In this case, the models could serve as an aid for investigators to identify patterns and associated words. Also, they could be applied to associate people with places, performing researches at suspicious activities.

Word2vec, introduced by Mikolov et al. [33], is a WE algorithm widely used that brought several improvements for the NLP area. However, its implementation has scalability problems, not increasing performance based on the amount of resources available. Thus, it requires several hours to complete its executions on a large corpus, which can be seen in Chapter 3. The scalability problem remains in recent approaches based on Mikolov's Word2vec. However, there are opportunities for improvements by applying High-Performance Computing (HPC) solutions, which are based on parallel computing and provide performance and scalability by splitting the problem in multiple statements concurrently processed. Besides Word2vec, there are others WE algorithms, e.g., FastText [20] and Wang2vec (an extension of the original Word2vec). These are capable of generating vec-

tor representations from larger corpora using different training strategies and consequently producing language models with particular characteristics.

Besides the versions mentioned above, there are several optimizations, also able to achieve better results in terms of performance and resource utilization. However, they suffer from limited usability since they are not integrated with the most popular tools for the usage of these algorithms, e.g., Gensim [45], NLPNET [11], and spaCy [1]. The other dilemma to be faced is beyond the choice of the most appropriate algorithm or environment, which is the parameterization problem. Since each input parameter influences resource consumption and the program behavior at runtime, the usage of appropriate sets of parameters does not seem to be a task that could be easily played by any user without specialized knowledge.

This master thesis presents an analysis of the main differences between the above-mentioned implementations, considering points directly related to performance and scalability. We investigate the established WE NLP implementations and the optimizations proposed by the HPC field aiming to provide insights about the resource utilization and performance issues of the different analyzed versions. Furthermore, we investigate the usage of a neural network and a metaheuristic algorithm for a parameter tuning strategy, aiming to find the best parameter combination in a specific scenario, where we consider the computational resource.

The main contributions of the current work include:

- An analysis of the most known WE algorithms and implementations concerning performance, memory consumption, model quality, and usability;
- A way to integrate parallel and distributed versions of WE to the most popular environments used by the NLP community, improving their workflow regarding performance and usability.
- An evaluation of the proposed wrapper library to implement the above integration;
- An adaptive parameter tuning of the pWord2vec algorithm for executions in regular and High-Performance computational environments, utilizing Machine Learning (ML) techniques.

The remainder of this master thesis is organized as follows. Chapter 2 summarizes the concepts used in this study. Chapter 3 presents performance, memory consumption, output quality, and usability evaluations of the WE algorithms considered in this work. Chapter 4 describes in detail how our proposed solution works and its functionalities. Chapter 5 presents an evaluation of the integration and parameter tuning proposed in this work. Chapter 6 lists related work. Finally, Chapter 7 depicts our conclusions.

2. BACKGROUND AND STATE-OF-THE-ART

In this Chapter, we detail the most important areas to understand the remainder of this study. Firstly, we characterize WE algorithms and their usability context. Secondly, we introduce HPC concepts. Lastly, we explain the adopted ML techniques.

2.1 Word Embedding Algorithms

In recent years, the extensive usage of Language Models (LMs) led to significant results in the NLP field [9]. A standard approach for generating LMs is the usage of algorithms such as Word2vec, FastText, and Wang2vec [26]. These algorithms receive large volumes of text in a given language. From a training strategy, they construct a vocabulary and learn vector representations (in a given vector space $\mathbb{R}^n$ of dimension n) for these words, based on the context in which they are inserted.

2.1.1 Word2vec training process

Word2vec is the algorithm for generation of WE models proposed by Mikolov et al. [33]. This method attempts to predict the neighborhood of a target word within a context window, producing vectors representations for the found vocabulary. Two architectures are used by this algorithm: the Continuous Bag-of-Words (CBOW) and the Skip-Gram. The former predicts a target word from a given context. The latter predicts the context words for a target word, as presented in Figure 2.1. Both architectures require significant processing time, which can get worse according to the data volume and parameters used. However, it is noteworthy that, due to the increase in the range of words considered for training, the Skip-Gram model has greater complexity [32], therefore requires greater computational power.

WE models are precise forms of word representation, and their use is common in NLP systems that use words as basic input units [17]. These models are capable of representing document vocabularies, capturing the word context, its syntactic and semantic meaning, as well as, words relations in a given corpus [17]. Different methods for embedding generation have been developed since the publication of the most popular among them, Word2vec [33]. Most of these WE learning processes require high-computational power, as we detail in Section 3.1. Parameters such as vector dimension, negative sample, architecture, and window, also the usage of large datasets, directly influence resource consumption.

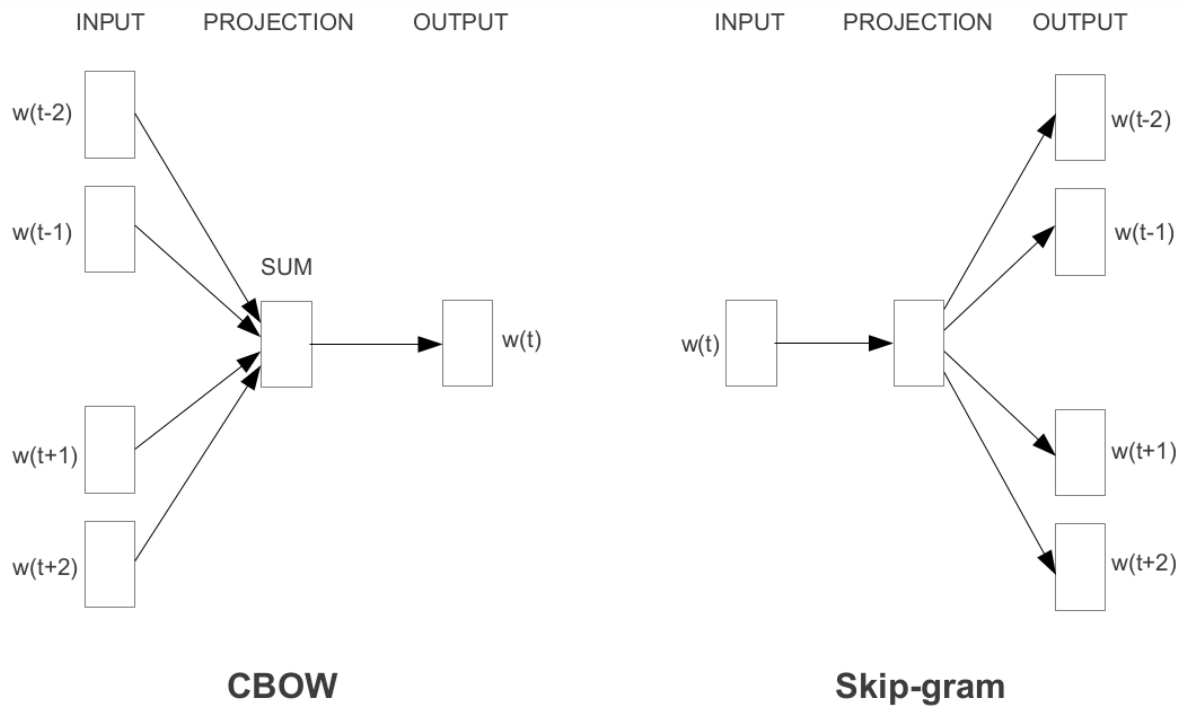

Figure 2.1 – CBOW e Skip-Gram model architectures.

2.1.2 Wang2vec training process

Wang2vec, as mentioned in previous Sections, is also an algorithm for embedding generation, and it is fundamentally based on Word2vec. Two modifications of the Word2vec algorithm give rise to Wang2vec. These modifications allow the model to capture greater detail of the syntactic features of a language. In the CBOW architecture, the input tokens are the concatenation of the one-hot vectors of the context words that appear. In the Structured Skip-Gram architecture, the prediction parameters change to predict each context word, depending on the position concerning the target word [27].

2.1.3 FastText training process

FastText is a WE algorithm that is also divided into two architectures: CBOW and Skip-Gram. This type of embedding is used with success in many NLP tasks such as Text Classification and Named Entity Recognition. One of the main differences between Word2vec and FastText is that FastText can estimate values for words that are not part of its pre-trained model. It happens because the training of the model uses N-grams instead of whole words. For example, given the token “matter" and n = 3, we will have the *3-grams:* $<$*ma, mat, att, tte, ter, er*$>$ [20].

The FastText algorithm uses a method where each representation is induced by the sum of the N-grams vectors with the surrounding word vectors. The N-grams is a sequence of N words used to generate estimates in the probabilities of words attribution [21]. With its method, the FastText aims to capture morphological information to induce the process of generating WE [17, 4].

2.2 Parallelism in shared and distributed memory

Parallel programming is an alternative to achieve high-performance and scalability in software development [31]. Using languages that allow to explicitly indicate parts of code that must be executed at the same time on different processors, algorithms can be modeled with parallel patterns. Regarding parallelism support, computers fall into two categories: multicores and multinodes. The former has great potential for performance gain, despite the limited processors number. The latter allows the usage of a higher number of physical processor cores. However, new issues must be considered, such as the need to exchange messages due to the absence of shared memory. Currently, the most widely used commercial multinodes consists of networked multicores called clusters [39].

Environments based on on-premise clusters commonly use batch-jobs schemes. Thus, a point to be observed during the development of applications focused on remote environments is the integration with resource management systems, which orchestrate and manipulate tasks on remote computers [22, 24]. An example is the TORQUE resource manager, which is based on a job scheduler called Portable Batch System (PBS). TORQUE schedules a conference room of requested jobs submissions, providing primitive job operations such as starting and stopping jobs on compute nodes [8].

In distributed-memory contexts, the Message Passing Model is a widespread approach. In this case, processes have only local memory and communicate with each other by sending and receiving messages through the network [15]. This model allows the usage of a higher number of physical cores for processing tasks, but with an additional cost of communication between processes. The Message Passing Interface (MPI) is a library specification based on the Message Passing Model. It allows data to be moved from one process address space to another by using specific functions [15].

The shared-memory model is related to the concept of multicores, where the same memory address on two different CPUs refers to the same global location, as presented by Quinn [39]. Thus, there is no need to exchange messages between the running processes, which allows the exploration of the parallelism model with multiple threads. Some libraries and applications provide support to explore parallel programming in shared-memory contexts. It allows programmers to explicitly specify the actions to be taken by the compiler to execute the program in parallel [38]. OpenMP is an Application Program Interface (API) that

explores parallelism on shared-memory systems, assisting programmers in parallel implementations through the fork-join template at threads level [39].

2.3 Neural Networks

A neural network is a processing structure formed by units called artificial neurons. These neurons can have connections in analogy to neurobiology synapses, where each connection between neurons is associated with weights. A neural network utilizes layers to organize the multiple neurons and, the network topology defines the arrangement of layers.

The principle of a neural network is that, based on the input layer data, the network runs calculations in its consecutive hidden layers until obtaining an output value at each neuron of the output layer. [12], producing a model as the final output. When applied to classification problems, these models indicate the appropriate class for the input data, and, when applied to regression problems, the model should be able to predict output values for any given input. In early iterations, neuron connections receive random weights. As the training proceeds, the network updates the weights considering the mean square error between real and predicted outputs. A popular learning method capable of finding the correct combination of weights during the training process is the backpropagation algorithm.

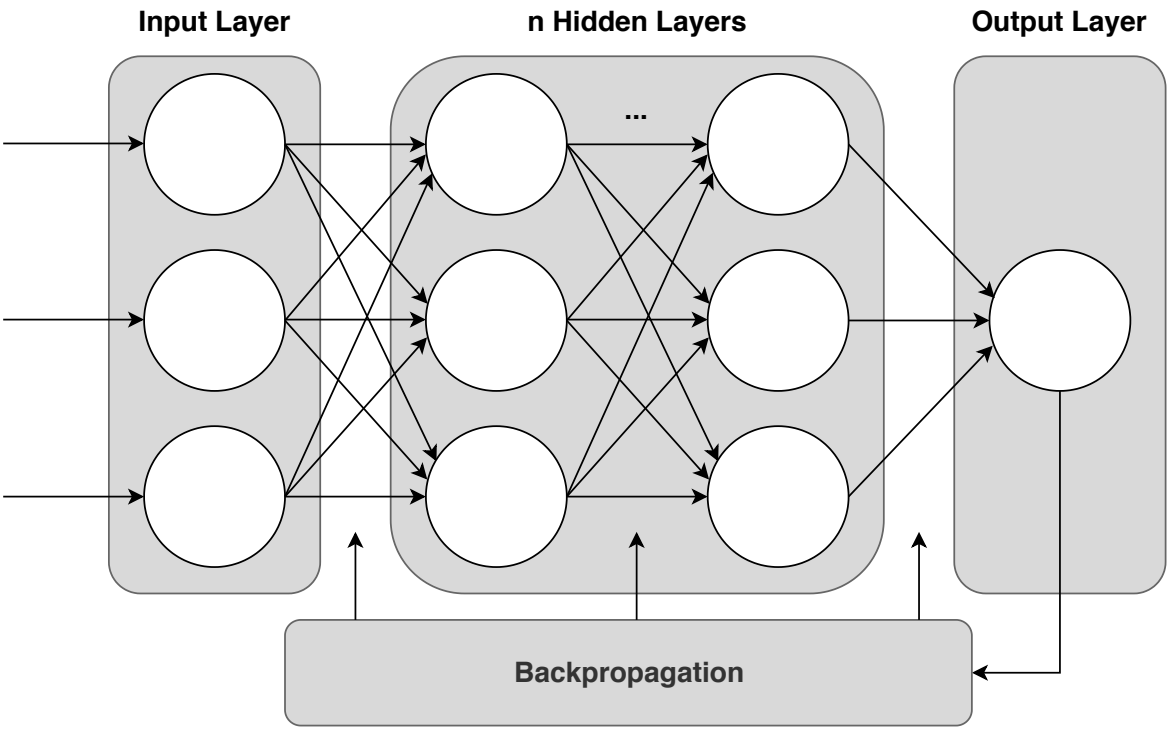

Figure 2.2 – A typical Multilayer Perceptron neural network architecture.

2.3.1 Multilayer Perceptron

Multilayer Perceptron (MLP) is one of the most popular and simplest feed-forward neural network models. An MLP network contains one or more layers of nodes between its input and output layers, and due to the nonlinear activation function with each node, an MLP is capable of forming complex decisions in the pattern space [16]. Figure 2.2 presents a typical MLP architecture. In general, the first layer is the input layer, followed by N hidden layers and then the output layer. The increase in the number of hidden layers is generally related to the accuracy expected and the complexity of the task.

2.4 Simulated Annealing Algorithm

A metaheuristic is a method that performs a robust search of a solution space, orchestrating interactions between local improvement procedures and higher-level strategies, being able to escape from local optima [13, 5]. These methods can be applied to many types of complex scenarios, and are an extensively used approach for optimization problems, particularly those of a combinatorial nature. The correct sets of parameters are indispensable in the search for better results, either for faster executions or better output quality. Since algorithms can be highly-sensitive for its parameters setup, metaheuristics such as Random Walk, Genetic Algorithm, Simulated Annealing, etc. are possible solutions to parameter optimization tasks [28].

The Simulated Annealing (SA), first introduced by Kirkpatrick et al. [23], is a method based on the cooling processes technique to get the ground state of metals. One of the main characteristics of this method is the acceptance of worse solutions during iterations as a strategy to escape local optima. Being $\Delta_{\omega,\omega'}$ the difference between the current solution and its neighbor, and t_k the temperature for k iteration. SA decreases the acceptance of worse solutions as the number of iterations increases and the temperature cools down. The probability of worse solutions acceptance is controlled by the following equation:

$$p(\omega) = exp^{(\frac{-\Delta_{\omega,\omega'}}{t_k})} \tag{2.1}$$

In case of optimization problems, the standard optimization procedure of SA is given by the following steps [42]:

1. Generate initial solution vector;
2. Initialize the temperature;

3. Select a new solution in the neighborhood of the current solution;
4. Evaluate a new solution;
5. Decrease the temperature periodically;
6. Repeat step 2 – 6 until a stopping criterion is met.

We can take the hill-climbing problem as an example of the simulated annealing optimization scope. This problem consists of finding the highest peak in a mountain range. Starting from an initial state, the algorithm accepts as next, the closest neighbor that leads to a higher peak than the current state. To solve the local optima problem, early iterations take worse solutions, over the probability presented in Equation 2.1. Figure 2.3 exemplifies how the algorithm may find local optimal values during the searching process.

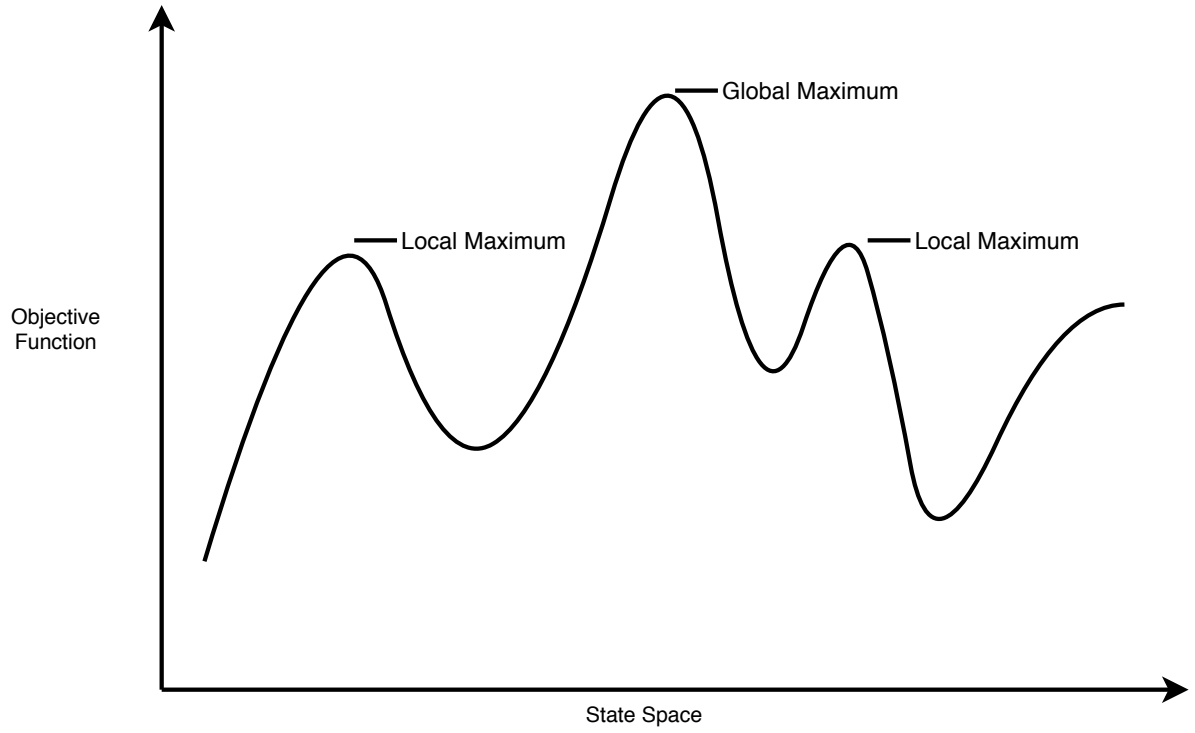

Figure 2.3 – Hill Climb problem.

3. EVALUATION OF THE AVAILABLE WORD EMBEDDING IMPLEMENTATIONS

The execution of WE algorithms is a task that requires high-computational power and may also face performance problems. The existence of different WE algorithms was our initial motivation to give a closer look at this research field and led us to our first goal, which is to distinguish state-of-the-art algorithms and analyze their behaviors. In the literature, we found both NLP and optimizations proposed from the HPC field. The most widespread WE algorithms in the NLP community uses programming with multiple processing threads on computers with shared memory. However, from an HPC perspective, the implementation proposed by Mikolov [33], as well as the other versions based on it, have low scalability and low performance. Consequently, it can take several hours to finish its executions.

The first WE algorithm implemented was the Word2vec proposed by Mikolov et al. [33]. The original version already performed the model training in parallel for shared memory machines, since a sequential execution would be possibly unfeasible for real applications. In this case, the parallelism level increment occurs in a scenario where the main memory is global for all processor cores. The input text is divided by the number of threads in order to execute the training process and update the final output file, thus, ignoring the race conditions [41]. From this proposal, other implementations based on the Mikolov's et al. algorithm emerged, and libraries such as Gensim provides its own versions of those algorithms, some examples are the FastText and the Word2vec itself.

Recent works show concern over the Word2vec algorithm scalability and performance improvements, such as the proposal of the called pWord2vec [19], an optimization for shared and distributed memory contexts. Its implementation consists of a Word2vec extension with a negative sample optimization focused on the Skip-Gram architecture. In highlighted points, there is a mini-batching based scheme (a division of the training data in smaller loads) and shared negative samples to convert Basic Linear Algebra Subprograms (BLAS) operations of level 1 vectors to multiplication operations of level 3 matrices.

Another similar study was proposed by Rengasamy et al. [41]. The authors aimed to increase throughput by sharing positive/negative samples in several context windows using Skip-Gram architecture. Similarly, Ji et al. [19] proposed an optimization also based on the Word2vec algorithm, which addresses the distribution and parallelization in environments with distributed memory using MPI and OpenMP technologies. The gains, in this case, are significant, and show scalability for up to 32 nodes with 76 cores each.

The behavior of an algorithm at runtime is crucial to define how it must be handled in a real-world scenario [30]. If too much processing power or too much memory is required for simple inputs, it may imply that even a modern desktop machine will not support a large workload. In a scenario where the user must previously pay for infrastructure, under-

standing algorithms behavior may lead to the best affordable choice. Furthermore, due to its specific requirements and approaches, each programming language influences resource consumption. Thus, comparisons over implementations based on the same algorithm, but using different programming languages may also lead to a better understanding of the WE state-of-the-art.

The remainder of this chapter investigates and compares performance and efficiency levels of established WE algorithms and high-performance optimizations, aiming to answer specific questions about performance and memory consumption. Furthermore, we present an evaluation of the output models and investigate the usability of the considered WE algorithms in the NLP context.

3.1 Performance evaluation

WE algorithms have become a study object for the HPC area [19, 41] due to the computational intensity even for small datasets. Implementations focused on performance and scalability were developed, but the versions most popular and present in NLP academic research, do not surpass state-of-the-art results, such as implementations of [19, 41]. Table 3.1 presents the best results regarding the processing time of the versions considered in this work.

For our baseline experiments, we utilized the parameter set with vectors size equal to 200, window 8, negative 25, sample 1e-4, iter 15, min-count 5 and a PT-BR 4 GB file [25] as input. The computational testbed consists of a Dell EMC PowerEdge R740 server with two sockets, each one containing a 5118 Xeon Gold processor of 2.30 GHz, 12 Cores/24 Threads 12 MB L2 cache. The total number of cores is 24/48, with 16.5 MB of L3 cache shared by the two sockets and 322 GB of main memory.

Based on our experiments, considering a PT-BR 4 GB input file, Mikolov's Word2vec presents the highest execution time overall in the Skip-Gram architecture. Its fastest result is about 15 to 19 hours longer than Python implementations. Wang2vec presents a similar performance to Mikolov's version. The Gensim's Word2vec, on the other hand, presents a faster execution time in comparison to the original version, finishing its execution about 15 hours earlier. The Gensim's FastText finishes its execution 4.2 hours faster. In general, both algorithms show similar results.

Regarding the analyzed versions which focus on CPU-parallelism, the pWord2vec is the algorithm that brings better results in terms of execution time, as presented in Table 3.1. This approach presents an execution time of 1.48 hours, for the same workload used in all experiments. However, the pWord2vec contains only Skip-Gram architecture. The pWord2vec_MPI, in this case, consider a cluster of two identical servers from the above testbed.

Table 3.1 – Fastest processing time of the analyzed implementations.

Algorithm	Processing Time	
	CBOW	Skip-Gram
Word2vec	1.3 h	22.1 h
Gensim.Word2vec	2.0 h	7.2 h
Gensim.FastText	1.5 h	3.0 h
Wang2vec	1.3 h	7.7 h
pWord2vec	-	1.4 h
pWord2vec_MPI	-	30 min

The possibility to process a corpus of nearly unlimited size through Gensim is an interesting feature mentioned by Radim [40]. The library allows reading the input text through multiple files. Thus, it is not required a total corpus size, smaller than the main memory. Such a possibility can be useful when executions are running on machines with limited storage capacity. However, within an HPC scenario, the main memory is usually equipped with a few hundred gigabytes and is not usually a limiting factor as in conventional personal computers. Furthermore, dividing the input data to be processed in different files, the time dedicated to inbound and outbound operations during execution increases. This effect potentially damages the efficiency of applications developed in this format, which can also change the output quality.

Figure 3.1 presents the speed-up means computed over the slowest execution time of the original Word2vec. The considered algorithms are pWord2vec, pWord2vec_MPI, Wang2vec, Gensim.Word2vec and Gensim.FastText over the variation on the number of threads/workers. All tests were repeated five times for each sample to compute the throughput average and standard deviation. From the speed-up levels shown in Figure 3.1, it is possible to identify how each algorithm deals with computational resources usage. A bad efficiency score generally implies in processor cores getting idle during execution, where efficiency is the achieved speed-up per processor cores utilized. A low speed-up gain or a non-gain, in the worst case, may indicate that perhaps the addition of more computational resources may not be an interesting attempt.

The highlight goes to the pWord2vec, which presents a speed-up of 15x, being the fastest among the CPU-parallelism versions. For the pWord2vec's MPI version we generated projections for 4 and 8 nodes, based on the performance experienced on our testbed and the values demonstrated in S. Ji's paper [19]. Gensim's Word2vec CBOW architecture maintains acceptable speed-up and efficiency levels until it reaches 12 cores. However, from this point on, the speed-up level stagnates and does not increase significantly when the number of processing cores is incremented from 12 to 24 and 48. The non-acceleration, even with additional resources, characterizes the low scalability of these versions.

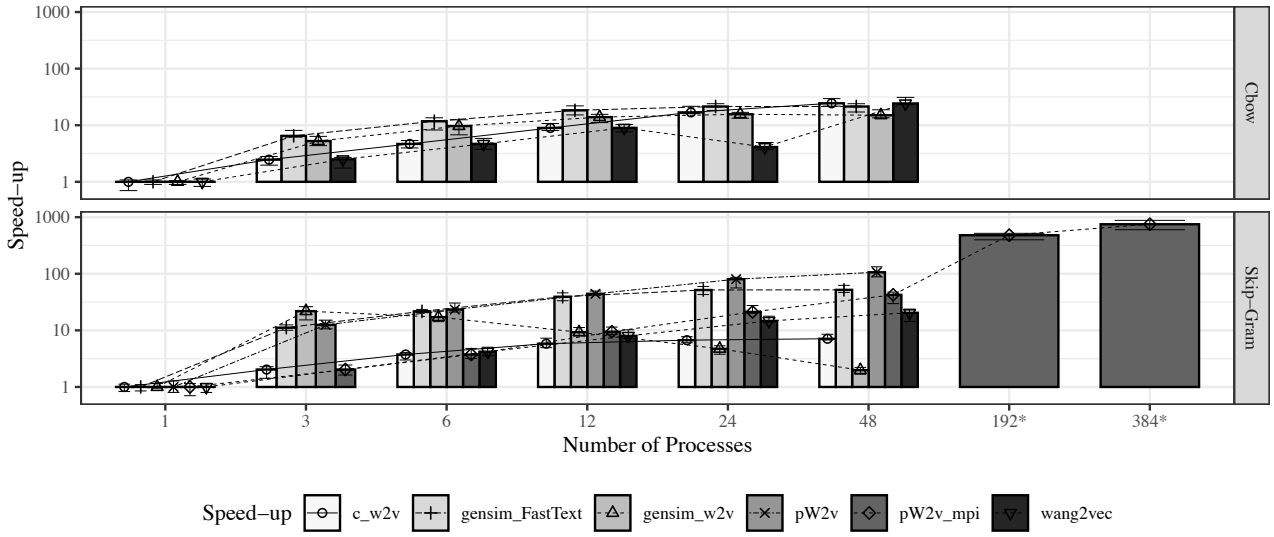

Figure 3.1 – Speed-up of Word Embedding algorithms generators. *Estimated values based on experiments.

For this experiment, we analyze the behavior of each version mentioned in the previous Sections over the "workers" parameter (number of threads) variation. Some implementations do not perform as expected in this scenario. The first point we may consider is that the comparison occurs between implementations from compiled and interpreted programming languages. Compiled language tends to be faster and more efficient to execute than interpreted languages. However, as we observe long-running executions, it may not make much difference for the complete scenario. The optimizations, namely, the pWord2vec and its derivate pWord2vec_MPI, implements optimizations strategies for cache and main memory. Using these resources, the algorithm minimizes the Input/Output waiting at runtime, making usage of the operating system mechanisms for cache gain, and by them reducing processor idleness. The multiprocessor version is an improvement of the former, but by using the MPI resources, it may consider a much larger number of processor cores.

3.2 Memory consumption evaluation

Since each algorithm presented a particular behavior during the performance experiments, we decided to give a closer look over of memory consumption of the state-of-the-art WE implementations. The purpose of this Section is to emphasize the different requirements at runtime. Thus, we analyzed each algorithm specified in Chapter 2, considering the number of threads variation over the executions, intending to comprehend the memory consumption behavior of each observed algorithm.

Figure 3.2 presents the memory profiling of the experiment specified in Figure 3.1. We split the experiment into the two WE architectures and the algorithms behavior over time.

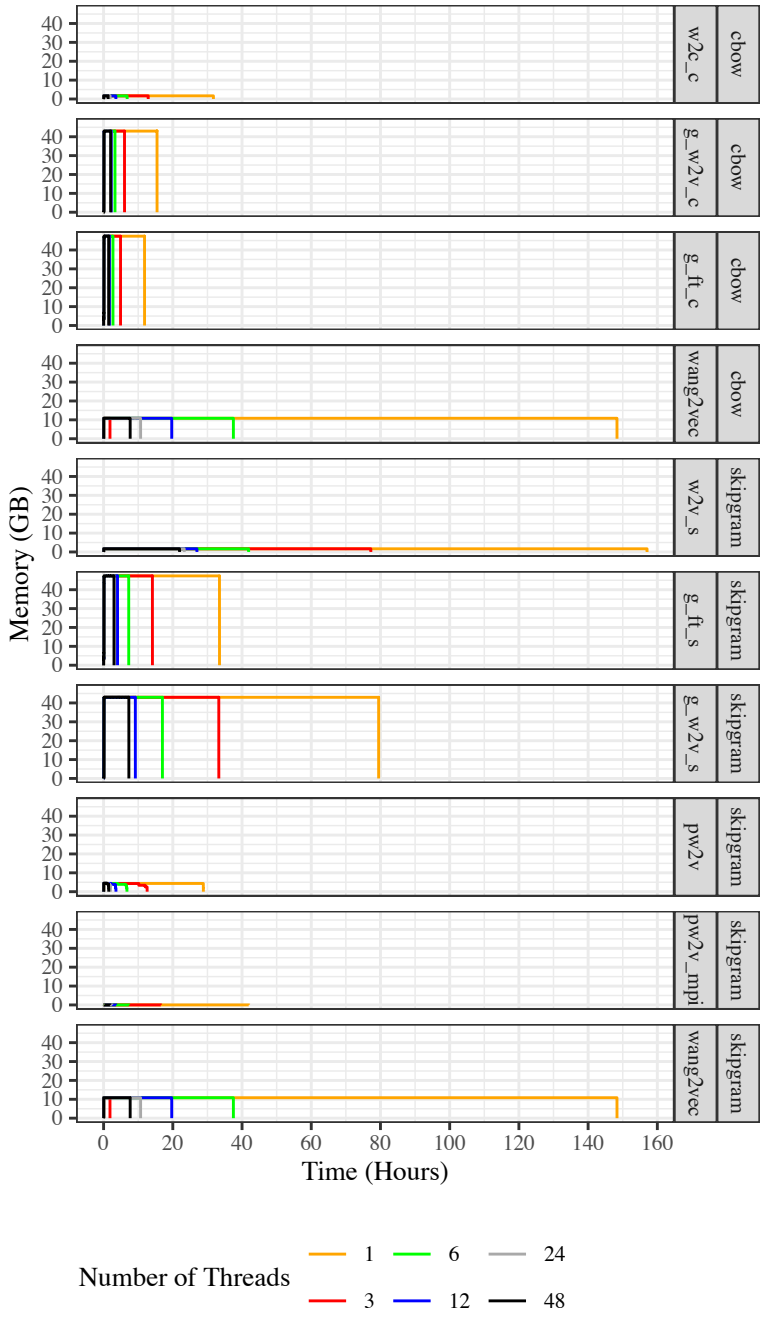

Figure 3.2 – Memory consumption of the analyzed implementations over time.

In general, it is possible to verify that in terms of memory consumption, there are significant differences in the footprints comparing one version to another. Meanwhile, compared to itself, the algorithms do not show any significant variation from a single core to 48, once the memory footprint remains the same over the executions, even with the gradual increment of processor cores. The testbed utilized for this experiment is the same presented in Section 3.1 and, the file size was also PT-BR 4 GB, size 200, window 8, negative 25, sample 1e-4, iter 15, min-count 5.

Regarding the differences over the programming languages, in Figure 3.2 it is possible to verify the most meaningful contrasts, reaching a variance bigger than 39 GB. The Python implementations show higher memory consumption, both in Gensim.Word2vec and Gensim.Fastext. The C implementations, in general, present a much minor memory consumption. Although, it is important to discern that low memory consumption does not imply low execution time. Therefore, it is possible to observe implementations with a lower memory consumption but with higher execution time.

In Skip-Gram architecture, Mikolov's Word2vec presents its memory consumption as one of the lowest, being close to 1.5 GB. The Wang2vec performs the highest memory consumption compared to the other C versions, close to 10 GB. The Gensim's Word2vec, on the other hand, consumes much more resources, being close to 40 GB of memory. The Gensim.FastText consumes 3 GB more memory than the Gensim.Word2vec. Regarding the analyzed versions, the pWord2vec is the algorithm that brings the best results in terms of execution time and memory consumption. Figure 3.3 summarizes the memory consumption of the analyzed versions, concentrating the two architectures and it's memory footprints.

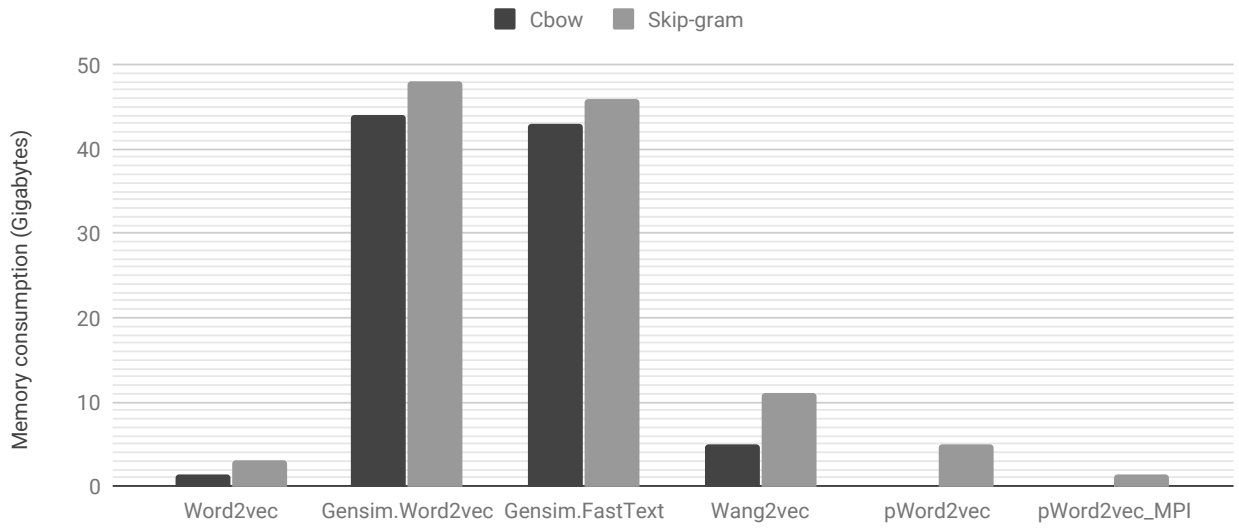

Figure 3.3 – Memory consumption of the analyzed algorithms.

In real scenarios, a high memory consumption may turn the usage of an algorithm unfeasible. Some algorithms in Figure 3.3 presents memory footprints over ten times the input file size. A common machine may not support such a workload. Despite the possibility of running the input files through batch-jobs, it would clearly have negative consequences, which reinforces the idea of considering environment resources in the choice of the appropriate algorithm to utilize.

3.3 Models' quality evaluation

Besides resource consumption, another important concern in the choice of the appropriate algorithm is the quality loss. Thus, we present an analysis of the models quality through an extrinsic evaluation of semantic similarity. The purpose of the semantic similarity task is to predict a degree of similarity (from 1 to 5) between two sentences. The corpus used in this evaluation is the corpus proposed in shared task *Avaliação de Similaridade Semântica e Inferência Textual*[1] (ASSIN) in the International Conference on the Computational Processing of Portuguese (PROPOR) 2016. The evaluation of this task is made using two metrics: Pearson's Correlation (ρ) for semantic similarity and Mean Squared Error (MSE).

Table 3.2 – Extrinsic evaluation of trained models.

Algorithm	**Architecture**	ρ	**MSE**
Word2vec	Cbow	0.48	0.58
	Skip-Gram	0.54	0.54
Gensim.Word2vec	Cbow	0.51	0.56
	Skip-Gram	0.54	0.54
Gensim.FastText	Cbow	0.51	0.56
	Skip-Gram	0.55	0.53
Wang2vec	Cbow	0.48	0.58
	Skip-Gram	0.53	0.55
pWord2vec	Cbow	-	-
	Skip-Gram	0.53	0.55
pWord2vec_MPI	Cbow	-	-
	Skip-Gram	0.53	0.54

Table 3.2 contains the results obtained with this extrinsic evaluation. Although we perceive some variation, the results are close to each other with regard to the evaluated metrics. These evaluations may be used as a supplement when choosing which algorithm should be used in a specific scenario. Another point to be analyzed is the variance and standard deviation of the metrics presented in Table 3.2. We present these results in Table 3.3. With these additional metrics, we may observe that the generated models are quite similar, as we do not have loss in the similarity evaluation.

One of the biggest concerns of NLP researchers regarding WE generation is the quality of the output models. As the evaluations presented in Table 3.2 and Table 3.3 demonstrate how the outputs are very similar, and, as previously discussed, each algorithm presents specific resource consumption characteristics. The concern about outputs,

[1] http://propor2016.di.fc.ul.pt/?page_id=381

Table 3.3 – Models evaluation.

Model	s^2	s
ρ	0,000576	0,024008
MSE	0,000262	0,016181

in a certain way, could be replaced by the concern of which algorithm presents better performance or minor resource consumption.

3.4 Usability of Word Embedding algorithms in the Natural Language Processing context

The concept of usability is related to different software quality attributes, such as user performance and satisfaction [3]. In this Master thesis, the term usability is used with the meaning given in ISO 9126-1: understandability, learnability, and operability [43]. Thus, the capability of a software product to be understood, learned, used, and attractive to the user, when utilized under specified conditions. Regarding NLP applications, the Python language offers libraries focused on preprocessing, manipulation, and analysis, providing methods that simplify the processes to implement NLP applications. Examples include the Gensim [45], NLPNET [11] and spaCy [7] libraries. Among the mentioned examples, the Gensim library, firstly defined as a framework, aims to fulfill a gap in NLP applications. This library contains some of the main word processing algorithms, allowing them to be used from a single system. Python was the programming language used for its development due to its easy learning curve, compact syntax, multiplatform nature, and easy deployment [40].

Currently, Gensim is a free Python library, designed for raw and unstructured text processing for semantic data extraction [45]. Even though Gensim is one of the most popular options from the NLP community, there are some limitations in its use, e.g., elevated memory consumption and resource utilization, which may be a barrier to large data processing. Also, Gensim does not support the HPC optimizations proposed for WE models generation, which could lead to significant performance improvements. Similarly to Gensim, also other frameworks as the mentioned spaCy and some recent attempts such as NLPNET, have been trying to offer a series of functions to mitigate steps of preprocessing data and then extract information. However, those libraries still show similar limitations.

Regarding the usability of WE algorithms, besides resource utilization, another subject to be considered is parametrization. Each parameter of an algorithm influences its runtime behavior. Resource consumption is directly affected by the input parameter sets. Consequently, the task of choosing the most appropriate set of parameters does not seem to be a task that a non-expert user would be able to efficiently conduct. We discuss a possible solution in the remainder of this work.

Table 3.4 – HPC and WE algorithms from state-of-the-art.

	Features		Parallelism exploration		Usability	Performance
Implementations	**Cbow**	**Skip-Gram**	**Multicore**	**Multinode**	**Gensim compatible**	**Language**
Gensim.Word2vec	✓	✓	✓	-	✓	Python
Gensim.FastText	✓	✓	✓	-	✓	Python
Word2vec	✓	✓	✓	-	-	C
pWord2vec	-	✓	✓	-	-	C
pWord2vec_MPI	-	✓	✓	✓	-	C
Wang2vec	✓	✓	✓	-	-	C

Table 3.4 shows data regarding state-of-the-art algorithms implementations for the generation of WE Word2vec and FastText available in the Gensim library. It also indicates optimizations found in the literature, relating to features, parallelism exploration, usability, and performance. Notice that those developed in Python are present in the Gensim library and provide the user with the generation of WE templates on the two architectures proposed by Mikolov et al. [33]. However, they are limited to the scalability requirements, since they do not use resources that provide scale-out. The C language optimizations, despite its performance and scalability gains, are impaired as to their usability. As presented in Table 3.4, those versions are not present in the most popular tools, which affects points such as operability.

4. IMPROVING THE USABILITY OF PARALLEL AND DISTRIBUTED WORD EMBEDDING TOOLS

Analyzing the established optimizations proposed by the High-Performance community, we have compared performance, efficiency, and memory consumption aspects, as well as the quality of the output models and usability aspects. In our experiments, the algorithms developed in the C language presented considerable speed-up and lower memory consumption compared to the Python versions. Nevertheless, these optimizations are not commonly adopted by the NLP community. We believe that this situation may be caused due to the lack of integration between these versions and the most popular NLP tools, such as Gensim. Furthermore, as shown in previous Chapters, each algorithm has its features and behavior to solve similar problems, as well as, each parameter can increase processing time. To minimize its scalability and performance problems utilizing the best parameter set, demonstrated to be a relevant subject.

4.1 Integration for optimized Word Embeddings

Aiming improve tasks such as pre-processing and data analysis, several NLP tools have emerged. Through them, users are able to solve complex problems related to the mentioned topics. As early discussed, WE algorithms are one of the target subjects present in established NLP frameworks. Those provide a series of functions allowing users to perform preprocessing and raw text processing. Some of them, as Gensim, also allow entries in batch-files mode, intending to support executions in regular machines. However, despite the improvements to generate output models, data processing is still an issue, in fact, as demonstrated in the former chapter, some of the algorithms to generate WE models, have a very high workload.

As shown in the previous analysis, pWord2vec optimization seems to already solve the problems mentioned above. However, for a large amount of data, machines with high-computational power are still necessary. To run applications in more powerful environments through the network may expose users to certain difficulties such as data transfer, format conversion, remote access, and authentication. Also, to choose the right parameters considering the architectures may be a problem for non-experts users. Therefore, a tool that allows executions to be done remotely and simultaneously, considering different processing environments, would be an improvement for the NLP area. A better resource management for WE algorithms may become an aid for researchers. Such integration, over HPC environments, may allow researchers to solve larger problems and achieve better results since

the corpus size, number of iterations, number of layers, and other parameters that directly influence the output quality could be explored.

Due to the study presented in Section 3.1, we may assume that according to the problem to be solved and the available resources, the answer to which is the most appropriate algorithm may change. To tackle these problems, as well as the mentioned performance and usability issues, we have developed a wrapper [2], which the main functionality modules are presented in Figure 4.1, namely, resource provisioning and WE algorithms. The former addresses environment configuration and the transference of data from the user desktop to the target environment. The latter addresses the execution of specific algorithms considering their requirements. The wrapper library combines these two modules to manage simultaneous executions. The main goal is to integrate the high-performance optimizations early mentioned into a single library, in the most transparent way as possible regarding resource allocation and programming languages related issues. The implementations can be used for different purposes and applied to different computer architectures, e.g. multicore or multinode. In the search of better resource utilization in the usage of the mentioned WE algorithms, the developed wrapper improves the usability and performance of NLP frameworks.

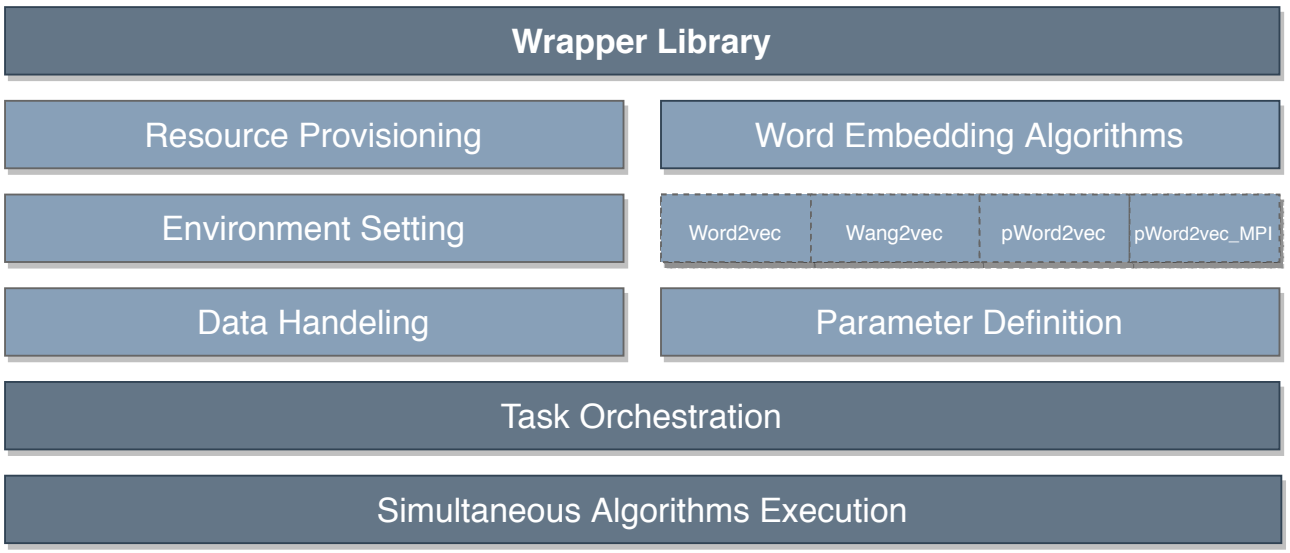

Figure 4.1 – Wrapper library modules scheme.

Considering the demand of different computational contexts, in our wrapper library, we consider three possible environment interactions, as presented in Figure 4.2. We encapsulated bash scripts into Python commands to run each considered WE algorithm respecting its specific requirements and considering different contexts. As we can perceive, each different environment requires specific efforts in order to successfully manipulate data and handle executions. In order to orchestrate simultaneous jobs over the network, the first considered environment, as shown in Figure 4.2, is the on-premise cluster. In this scenario, we uti-

[2] https://github.com/mmatheuslyra/Wrapper

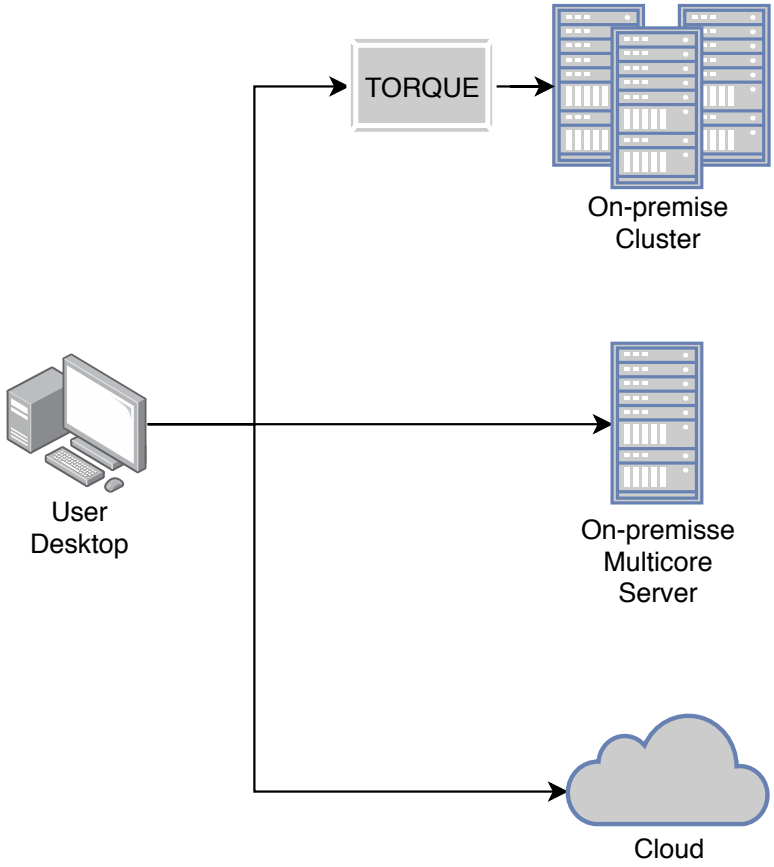

Figure 4.2 – Wrapper library for the optimized Word2vec execution environments.

lize the job queue from the TORQUE resource manager to orchestrate job requests over the cluster infrastructure. The library transfers data input files to the on-premise cluster, makes a batch-job request to the TORQUE manager, and lastly brings the output back. Another possible computational context is the on-premise multicore server. In this case, the interaction with the TORQUE manager is no longer necessary once the machine can be manipulated by establishing a secure connection. The other scenario considered is the cloud. In general, cloud services provide access to its virtual machines through the internet, which make the access steps in the wrapper point o view, similar to the on-premise multicore server scenario. Figure 4.3 presents a sequence diagram of the wrapper main functionality described above. In this example, we consider Gensim to be the tool used for prepossessing data, and the interaction with the On-Premise Cluster environment.

The computational environments to be considered as valid parameters must be previously configured, informing the required access credentials. Regarding security problems, the cluster access occurs through an SSH channel. Using the wrapper library users may run the algorithms integrated with their regular environments. This tool allows simultaneous executions over HPC environments and brings the possibility of solving problems over bigger corpora. Thus, many environments can be active at the same time, executing user operations. Job requests may be allocated and schedule, running simultaneous tasks over the on-premise cluster, according to the Job queue.

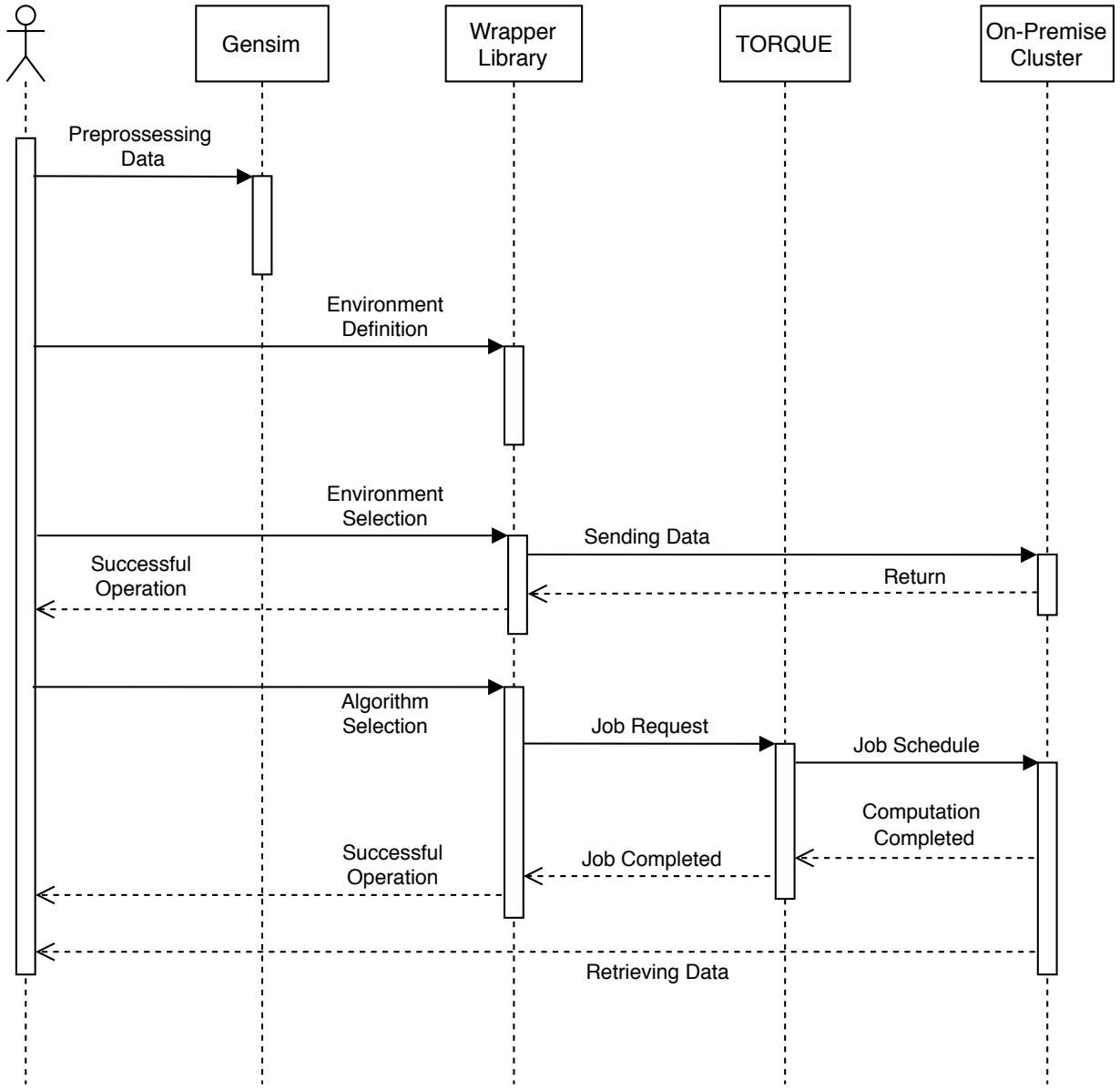

Figure 4.3 – Sequence diagram of the wrapper library Job Execution.

To simplify the process of switching between environments, the library allows the definition of different computational contexts by JSON files. These files must contain the parameter informations and access credentials for the respective environments. The actual switch occurs by a function call. As the user may define several different environments, the library allows the execution of the algorithms defined on the JSON files simultaneously, which may be useful to generate models through each algorithm separately. Each environment may have its specific settings, which is desirable once possibly they will be heterogeneous between them and, a generic configuration, in this case, may lead to resource wastage. Listing 1 presents an example of environment definition and the parameters considered by the library.

The analysis from previous Sections, combined with the developed library, brings a better understanding of the analyzed WE implementations and allows users to choose the

```
{
        "train" : "filename.txt",
        "output" : "vectors.txt",
        "cbow" : 0,
        "size" : 10,
        "window" : 8,
        "negative" : 5,
        "hs" : 0,
        "sample" : 1e-4,
        "threads" : 24,
        "binary" : 0,
        "iter" : 15,
        "mincount" : 5,
        "batch-size": 17,
        "local" : 0,
        "user" : "username",
        "UserServer" : "username@domain",
        "password" : "password",
        "nodes" : "1",
        "cluster" : "clusterName",
        "ppn" : "24",
        "walltime" : "00:10:00",
        "email" : "email",
        "type" : "0"
}
```

Listing 1: Wrapper library environment definition.

one that better suits their objectives. To approximate the HPC optimizations from data extraction libraries, may enable researchers to solve larger problems, since, these algorithms, in general, requires a minor workload. The algorithms composing the library are the original Word2vec, pWord2vec, pWord2vec_MPI, and Wang2vec.

The goal of the wrapper library is to allow researchers who do not have knowledge in HPC ecosystems to use its resources more easily. To remotely execute the mentioned algorithms, making use of the predefined computational settings, the user only needs to instantiate an object from the library, and then specify the environment. As the environment changes, the library takes care of resource management. It is noteworthy that by applying the wrapper, application performance does not get worse, which enforces its usability. Figure 4.4 summarizes the speed-up in comparison to the fastest time of the original Word2vec version, which used 48 threads. Since the testbed has 24 physical cores, it is also possible to identify how each version deals with additional resources.

Figure 4.4 complements the analysis, showing the best results in terms of performance and the maximum number of threads that provides speed-up. It is possible to identify that the pWord2vec reaches 15x speed-up over the fastest executions of the original Word2vec, and an estimated 105x in an 8-node cluster for its MPI version. The Gensim.Fastext, Gensim.Word2vec and Wang2vec algorithms achieved speed-up for the Skip-Gram architecture, being 7.3x, 3.1x and 2.8x respectively. For the Cbow architecture despite very close to the original the algorithms do not achieve speed-up.

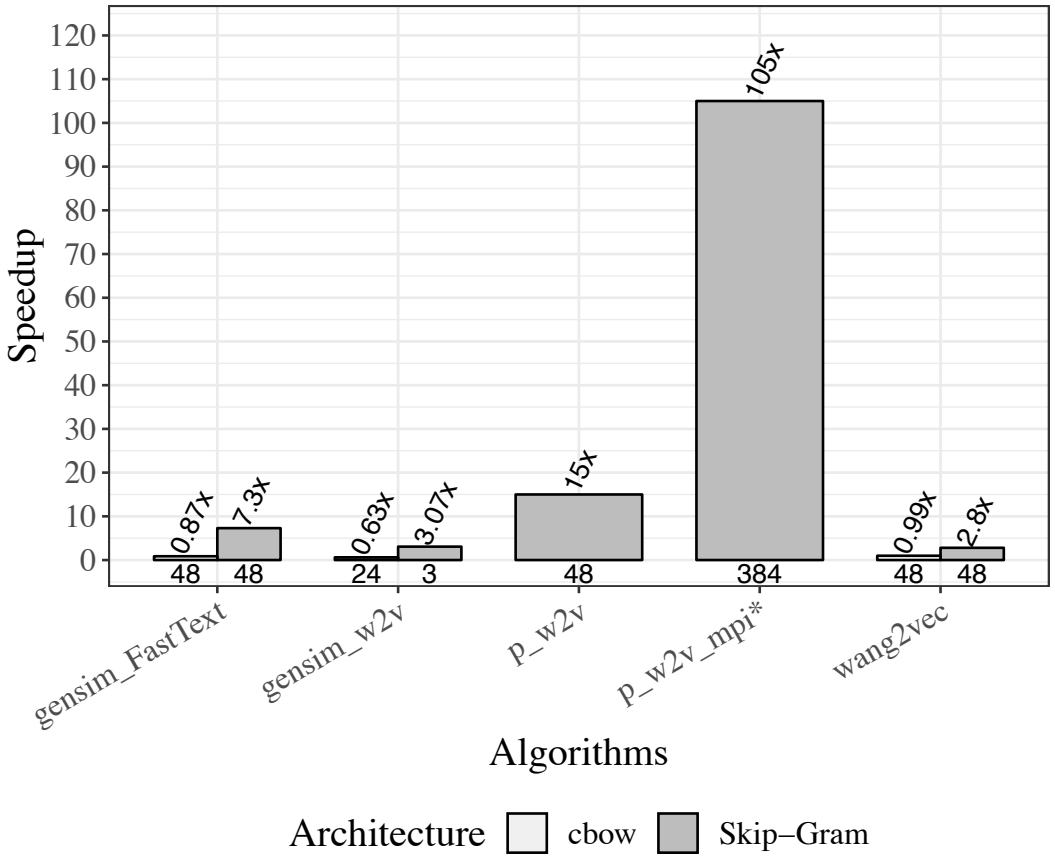

Figure 4.4 – Speed-up comparison among wrapper library and original Word2vec executions. *Estimated values based on experiments.

4.2 Parameter Tuning Technique and Automatically Suggestions

As early mentioned, the developed wrapper library allows non-experts in HPC or resource management to utilize WE tools in different computational environments. Nevertheless, it is not possible to employ efficient parameter setups, in terms of high-performance usage, without having such specific knowledge. In this Section, we investigate techniques to automatically suggest parameter settings aiming to further optimize the usage of WE algorithms and also the parallel and distributed optimizations presented in this work by applying ML techniques.

4.2.1 Parameter tuning model

Currently, given the high-computational power available, several areas of computing have been applying ML techniques to solve combinatorial problems. Learning in the context of ML can be characterized in a simple way, as historical data or observations used

to make predictions or derive tasks [14]. Parameter tuning is a method to select the optimal parameter set for a specific algorithm. Moreover, learning combinations based on previous executions is an alternative for non-expert users to utilize efficient parameter setups.

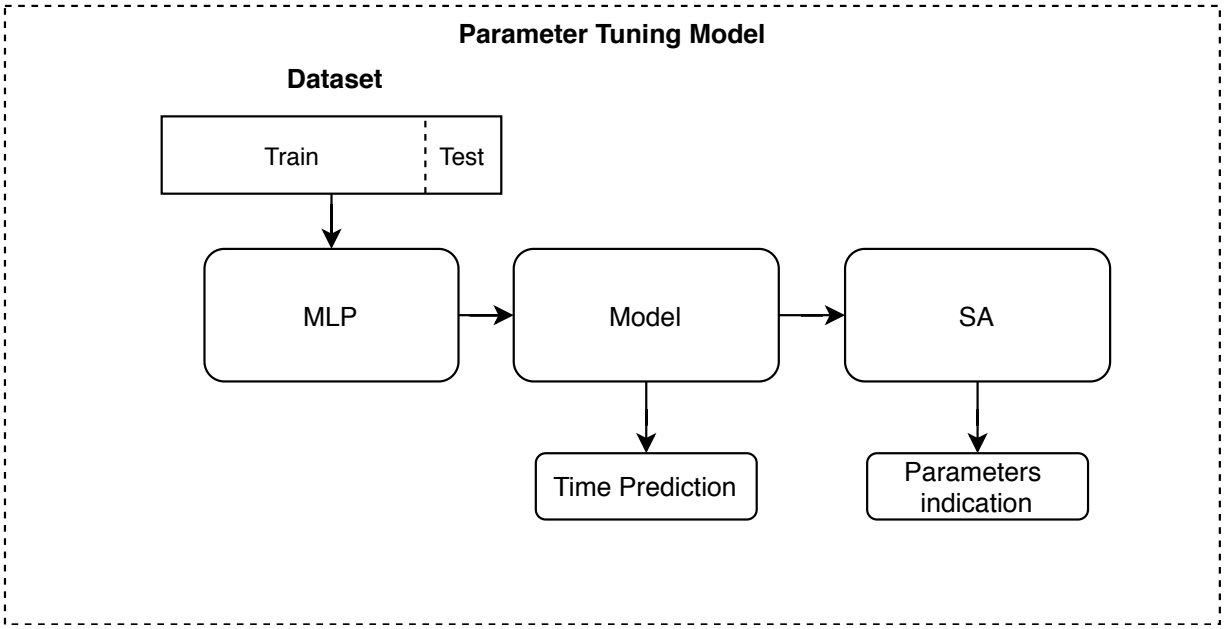

Figure 4.5 – Parameter tuning model utilizing Multi-layer Perceptron and Simulated Annealing.

As presented in Chapter 2, studies discuss the state-of-the-art algorithms to find optimal parameter combinations based on profiling datasets. Figure 4.5 presents our parameter tuning model, integrating a neural network and a metaheuristic. This tuning model is composed of two main parts, the MLP network, and the SA algorithm. The former generates a model that is able to predict the execution time of any given parameter set. The latter utilizes the MLP model to decide which combination leads to the minor execution time.

Since each WE parameter potentially increases computational cost, in our experiment, it was not feasible to utilize all parameter combinations nor all WE algorithms due to the elevated experiment duration. Therefore, to develop our parameter tuning model, we first defined a viable concept proof experiment, generating a dataset that represents the behavior of the algorithm which best performed on the analyses presented in Chapter 3, namely, pWord2vec. Based on the functioning of our wrapper library proposed in Section 4.1, we focus on the scenario of local user desktop to On-premise Clusters. Furthermore, based on the literature, we predefined ranges for the parameters Window, Negative and Batch-size. Moreover, we kept the parameters Size, Iter, and Sample with static values, as presented in Table 4.1.

Table 4.1 – Range of values from dataset parameters.

Parameter	Values
Architecture	Skip-Gram
Window	5 - 9
Negative	5 - 25
Batch-size	10 - 20
Threads	6 - 48
Size	200
Iter	10
Sample	1e-4
File size	100 MB

Execution time prediction

After data acquisition and preparation, we delimited the dataset into two parts, training data, and test data. Therefore we divided and applied the dataset about 90% for the training phase and the other 10% for the testing phase. The training dataset is used to build the predictor. The test dataset refers to examples of data that are chosen randomly and checked against the built-in predictor, which can help to tune the accuracy and validate the output. In accord with Gollapudi, S. [14], regarding time prediction, our experiment has three phases:

- **Training phase:** Training data is used by a training model to match the input provided with the expected output. The result of this phase is the learning model itself.
- **Test phase:** This phase measures how the correctness of the trained model, and estimates the properties of the model, such as error, accuracy, and other measures. This phase uses the validation dataset and the output is a sophisticated learning model.
- **Application phase:** The model is subject to real-world data for which the results are to be derived.

Linear regressions attempt to make predictions based on previous executions by learning connections between input and output values of a particular function. As presented in Chapter 2, an MLP is a widely utilized Neural Network. Figure 4.6 represents the process of how learning can be applied to predict behavior based on the topics early mentioned by using a MLP [14]. In our experiment, the neural network intents to infer a hypothesis of the real-valued output for any given set of parameters, minimizing the mean square error between predicted and real values calculated by the cost function presented in Equation 4.1.

$$J(\theta) = \frac{1}{2m} \sum_{n=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{4.1}$$

Where $h_\theta(x)$ is the hypothesis function and m the dataset size .

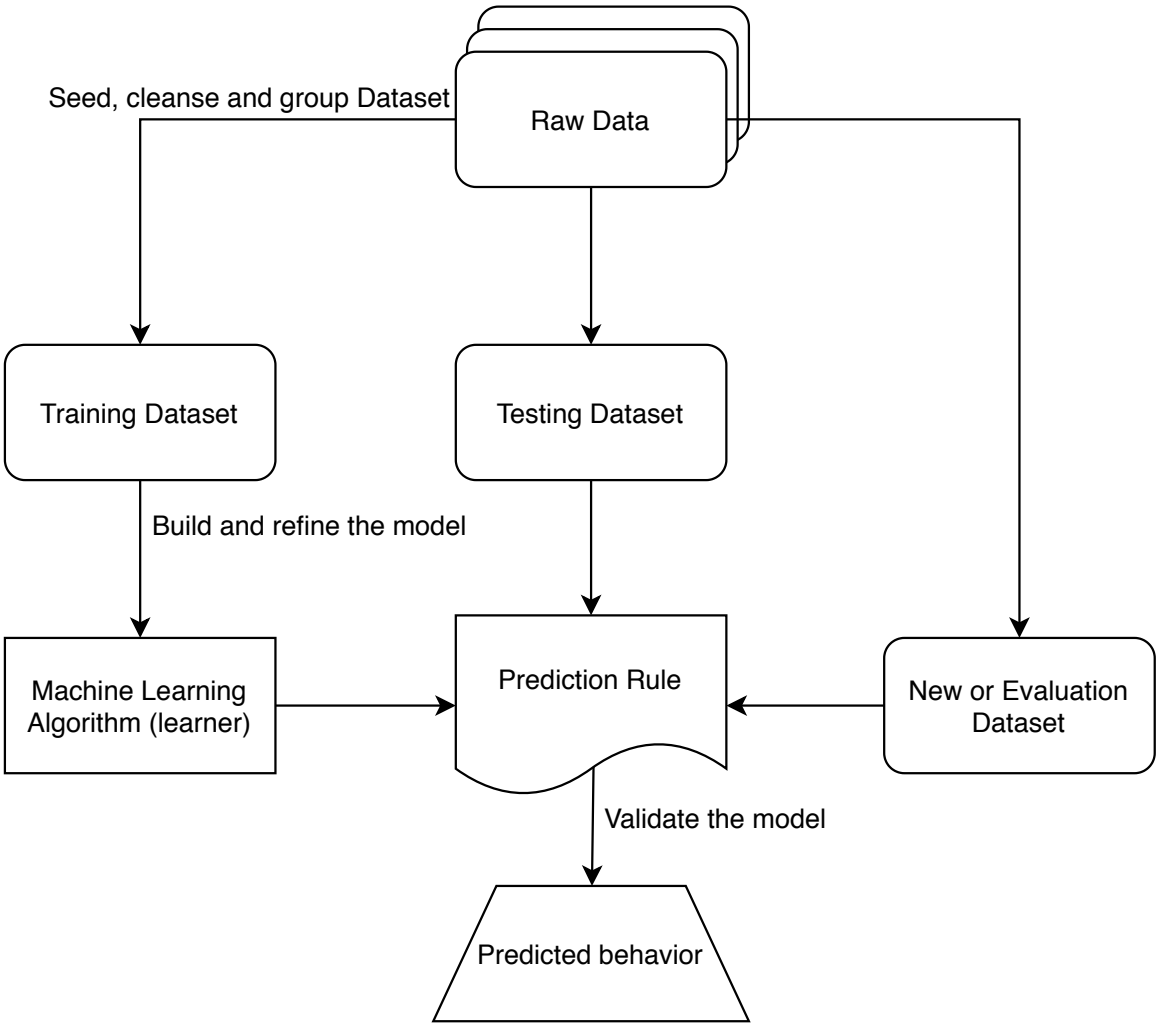

Figure 4.6 – Learning process example.

Table 4.2 presents five different predictions and exemplifies how the regression maps parameters to execution time. The model considers an x-axis composed of 4 variables. The predictions achieved a MSE equal to 6.7905e-05, where each x element is compose as follows: ['window', 'negative', 'batch-size', 'threads'].

Table 4.2 – Multilayer Perceptron predictions.

x=[window, neg, batch, threads]	Prediction (sec)
[20, 7, 33, 59]	63.496098
[5, 25, 43, 31]	146.0617
[90, 50, 10, 8]	3021.2434
[23, 25, 24, 40]	236.5051
[42, 37, 24, 26]	629.6549

Metaheuristic suggestion

Since our x-axis is composed of multiple variables, to make suggestions that lead to faster executions consists of identifying the influence of each x-variable over the output. The Simulated Annealing Algorithm, as previously mentioned, is a metaheuristic based on the metal cooling process technique. This algorithm performs searches over combinatorial solutions usually applied to optimization problems. Algorithm 4.1 presents the pseudo-code for the SA algorithm in agreement to the six steps mentioned in Section 2.4. In early iterations, the algorithm utilize random x-variables combinations aiming to explore and understand the path to minimize the output cost. Additionally, over the cycles, we choose the parameter to be updated and, if it will be incremented or decremented, randomly. This procedure influences the convergence to global optima.

Select an initial solution $\omega \in \Omega$;
Select the temperature change counter k = 0;
Select a temperature cooling schedule, t_k;
Select an initial temperature $T = t_0 \geq 0$;
Select M_k iterations for each t_k temperature;
while *Stopping criterion is not met* **do**
    Set repetition counter m = 0;
    while $m \neq M_k$ **do**
        $\Delta_{\omega,\omega'} \leftarrow f(\omega') - f(\omega)$;
        if $\Delta_{\omega,\omega'} \leq 0$ **then**
            $\omega \leftarrow \omega'$;
        end
        if $\Delta_{\omega,\omega'} > 0$ **then**
            $\omega \leftarrow \omega'$ with probability $exp(-\Delta_{\omega,\omega'}/t_k)$;
        end
        $m \leftarrow m + 1$;
    end
    $k \leftarrow k + 1$;
end
Where k corresponds to the value for t_k.

Algorithm 4.1 – Simulated annealing pseudo-code [13, 5].

Table 4.3 presents SA outputs for a scenario where the algorithm is allowed to predict values for any x-variable. Due to the higher probability of worse solutions acceptance, early iterations presents worse outputs. As the execution proceeds and the temperature cools down, the acceptance probability of worse solutions decreases, leading final iterations to global optima. The computational testbed utilized for this experiment was the same presented in Section 3.1. Another considered feature is the possibility to lock up the x-variables. Thus, the algorithm will consider those variables as static values over the iterations. Table 4.4

presents the SA predictions considering the negative x-variable locked to 15. Therefore the user may predefine x-variables values as its necessity.

Table 4.3 – Simulated Annealing predictions.

Cycle	Temperature	x=[window, neg, batch, threads]	Prediction (sec)
0	100	[9, 10, 10, 10]	186.49
1	98.0	[8, 17, 10, 13]	204.15
2	96.04	[8, 22, 17, 13]	194.28
3	94.12	[7, 12, 10, 18]	109.95
4	92.24	[8, 21, 14, 12]	232.65
n	...	...	...
349	0.0	[5, 5, 20, 48]	**33.15**

The results presented in Table 4.3 and Table 4.4 shows how our parameter tuning model works and how such a model may be used to find efficient parameter combinations. Associated with the wrapper library previously presented, the tuning model can make suggestions. It is noteworthy that the learning process and predictions initially will consider the available resources as equal to the testbed utilized for the dataset generation. However, as new executions occur, the registry updates the dataset, so the latest predictions will eventually start considering new computational environments. The suggestion of parameters, despite been based only on the pWord2vec history profiling, can also be used over other WE algorithms, being constrained to the same adjustments necessaries for new environments to be considered by the neural network.

Table 4.4 – Simulated Annealing predictions for locked variables.

Cycle	Temperature	x=[window, neg, batch, threads]	Prediction (sec)
0	100	[11, 15, 10, 15]	200
1	98.0	[9, 15, 11, 15]	179.33
2	96.04	[5, 15, 20, 24]	91.88
3	94.12	[8, 15, 18, 21]	102.08
4	92.24	[8, 15, 10, 12]	197.29
n	...	...	...
349	0.0	[5, 15, 18, 48]	**62.37**

In a real scenario, the executions will take longer than our last experiment, as demonstrated in Section 3.1, they can take several hours. However, considering the proportionality between the size of the input files, it is possible to estimate the execution time for different inputs. Otherwise, to make more precise estimates with file size variation, it would be necessary to update the dataset considering executions with different file sizes.

5. WRAPPER LIBRARY AND PARAMETER TUNING EVALUATION

This chapter presents an evaluation of our proposed wrapper library, explaining how the tool changes the steps of running the WE algorithms in different computational environments for non-expert users. Furthermore, we present an evaluation of our parameter tuning method, comparing results with the default values from the regular NLP tool analyzed.

5.1 Wrapper library evaluation

As early mentioned, the wrapper library allows executions through different computational environments. Thus, the user is able to previously define machines that support larger workloads and work on bigger corpora, generating models with different settings. Regarding local executions, the user only needs to make a call for the respective algorithm method. Additionally, for remote executions, as the library aims to improve usability, and manages, among other things, context switches, taking charge of processes related to resource provisioning, the user may switch environments also by a function call. Without the wrapper aid, for remote executions, the user is responsible for a series of resource provisioning issues, once it is necessary to manage procedures through the network. Follows the most typical required steps to make such execution:

1. Allocate environment informing specific settings OR define batch-job request;
2. Transfer data to external environments utilizing FTP (File Transfer Protocol);
3. Start remote execution;
4. Transfer the output to the local environment utilizing FTP;
5. Release allocated resources.

Through the wrapper library, the usage of HPC optimizations is made directly from usual Python scripts. Thus, the actual execution time and resource consumption may reduce expressively. In this case, a local execution may be satisfactory. Although, if the local machine still can not hold the workload, the steps to execute the algorithms remotely are much more simplistic, once the library is in charge of making all resource management steps, as presented below:

1. Define environment (optional);
2. Start executions.

Another feature of the wrapper library is parameter tuning. To inform a setup of parameters that do not fit the computational environment, may directly influence execution time as well as resource consumption. Therefore, our parameter tuning model suggestions may be an aid to reduce several hours of execution time. We believe that this interdisciplinary work approximates the research areas of NLP and HPC. The wrapper library improves the usability of the parallel and distributed WE tools since it enables the achievement of performance gains for non-expert users. The previous analysis allows us to estimate that Python versions may need up to 40 GB more than the optimizations in C language for its execution. The C versions, on the other hand, can execute consuming less memory and perform faster time. The Listing 2 presents an example of local execution through the library and demonstrates the steps for calling the algorithms. Follows the required steps to make such execution:

```
1    import wrapper
2
3    emb = wrapper.Embeddings()
4
5    emb.Wang2vec()
6    emb.Word2vec()
7    emb.pWord2vec(train = 'InputFile.txt', window = 8)
```

Listing 2: Algorithms execution through the wrapper library.

Listing 3 details the steps for calling the algorithms for remote executions through the library. As the user may intend to collect results from the different algorithms, the example below shows how to execute the algorithms simultaneously through the network, and how the switch between environments can be easily accomplished. As the environment changes by the method call, all new executions start considering the new context as the current environment, that so, the library deals with the topics related to it transparently for the user.

```
1    import wrapper
2
3    emb = wrapper.Embeddings()
4
5    emb.setEnv('remote_env_1')
6    emb.pWord2vec()
7    emb.setEnv('remote_env_2')
8    emb.pWord2vec()
```

Listing 3: Remote execution through the wrapper library.

To figure out the best embedding to be used as input for real-world NLP applications, researchers need to explore input parameters over different algorithms approaches. Such a situation implies several executions over parameter combinations in order to analyze

the output differences running in real-world applications. Perform such an experiment on a single machine would take several hours to be concluded, making it unfeasible. With this work contribution, users are able to simultaneously start executions in different computational environments utilizing optimizations with much minor processing time. Furthermore, researchers are no longer responsible for several resource provisioning steps.

Despite the very similar quality on the output models presented in Section 3.3, during our research, we observed some specific characteristics among the WE algorithms. Given the obtained results, we conclude that there are relevant differences related to resource consumption between them. Also, it was possible to identify how parameters are related to resource requirements, indicating the relevance of utilizing appropriate parameter sets. As presented in previous Sections, depending on the inputs, an ordinary machine may not support the workload due to the massive resource consumption. The considered Python implementations require an amount of memory over ten times the input data size.

There are several scenarios where to execute those implementations would be unfeasible. We could consider a user desktop machine with 16 GB of main memory and an input file of 10 GB. Considering only results from the execution time analysis and the same parameter set, the estimated time would be approximately 3 hours. However, the memory footprint, as the analysis presented to us, would be equal to 100 GB. The operating system, in these conditions, would be obliged to utilize swap memory, which would considerably increase the execution time. Utilizing Gensim, a possible alternative would be split the input file into several batches. In this case, the tasks would run in serial, which also would require considerable processing time. In the same scenario running the optimized versions, as also demonstrated in the analysis, the memory footprint is very close to the input file size. In that case, the memory required would be equal to 10 GB. Therefore, this regular machine would support the optimization workload with minor execution time close to 30 minutes using the pWord2vec, considering this context of a desktop machine.

As the processing tools wildly used by the NLP community to generate WE models are available for the Python language, we assume that to enable high-performance environments to be easily accessed by an integration of the mentioned tools as a Python library, would allow the NLP researchers to solve bigger problems since they could use heavier parameters in bigger corpora entries. The comparison between the speed-up metrics of the algorithms included in the wrapper library and the implementation of the Gensim shows that by using our wrapper library to make the executions in proper environments, users that are not HPC experts may achieve better results in their experiments and could be able to use parameters that generate heavier workloads. Utilizing these optimizations, as shown in Chapter 4, it is possible to achieve an average performance gain of 15x for multicores and 105x for multinodes compared to the original version. There is also a big reduction in the memory footprint compared to the most popular Python versions.

5.2 Parameter Tuning Evaluation

As previously discussed, resource availability does not imply that programs will perform in the best way possible. Input parameters strongly influence resource consumption and also programs behavior at runtime. Therefore, to utilize parameter tuning considering testbed specificities is an alternative to improve resource utilization. Section 4.2 presents how our parameter tuning model learns from an input dataset containing parameters as well as the respective execution time. Our neural network generates a model able to predict precise outputs for any input set, which is used by the SA algorithm to find the best parameter combination for that context. The table below shows the comparison between default parameters applied to our model trained with the pWord2vec history dataset and the suggested parameters.

Table 5.1 – Comparison between default parameters and suggested parameters.

	x=[window, neg, batch, threads]	**Prediction (sec)**
Gensim.Word2vec	[11, 15, 10, 15]	200
Word2vec	[8, 5, 17, 24]	56.34
Suggestion	[5, 5, 20, 48]	33.15

The simple usage of generic default parameters generally is not the best option in terms of resource consumption, as demonstrated in Table 5.1. In our work, we consider processing time as our goal, therefore, our parameter tuning model searches for parameters that leads us to a minor processing time. It is noteworthy that the predictions would be different if our model were trained considering another metric as the main goal. Also, if our dataset considered other algorithms history data, the values to be found by the SA algorithm would be different, thus the suggestions itself. In the same way, different environments would make the SA conclusions change, which reinforces the need for parameter sets that take into account the computational environments in which they are inserted so that they can use their resources efficiently. Figure 5.1 presents how SA searches for x-variables combinations that will eventually lead to global optima. As previously mentioned, aiming to explore and understand the path to minimize the output cost, the algorithm compares neighbors values. As the temperature cools down and the probability of acceptance of worse solutions decreases, the algorithm accepts x candidates that eventually will converge to global optima.

Resource underutilization may be a problem if the specific context is not considered. Therefore, a parametrization aware of the computational environment brings resource usage improvements. In this work, our parameter tuning experiment considers only one algorithm and restrict parameters running in a single testbed. An ideal scenario should consider all the algorithms present in the wrapper. In that case, the predictions would probably be different for each algorithm, once the study discussed in Section 3.1 shows that some of

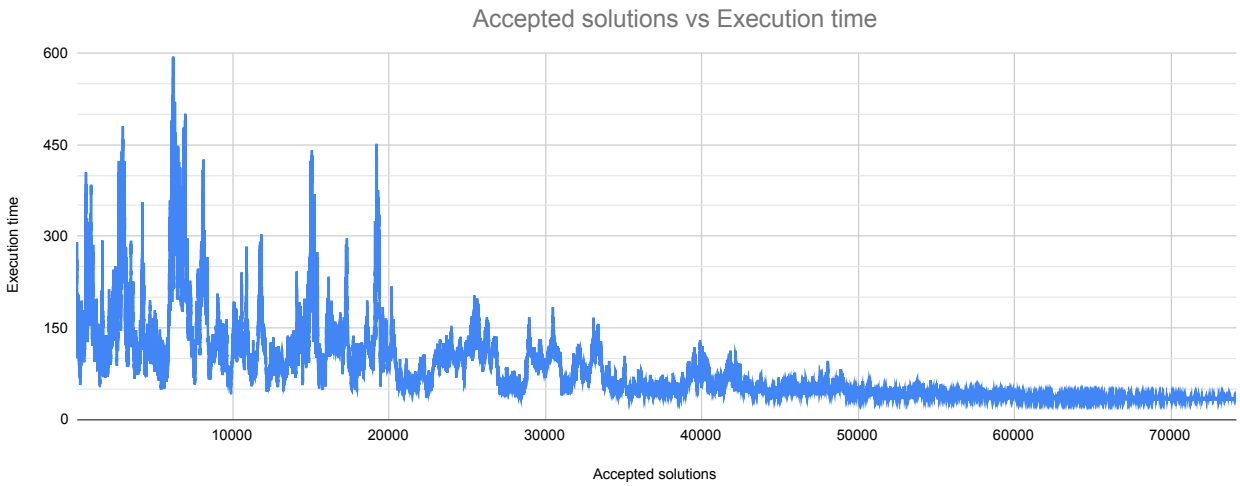

Figure 5.1 – Accepted solutions vs Execution time.

them do not scale to the max process number for example. Furthermore, to change parameters aiming to decrease execution time, may influence the quality of the models. Therefore, the possibility to lock parameters in static values may be useful in that context. Although our predictions are initially restricted to the parameters and the testbed that we examined. Our experiment shows how resource management can be improved by learning from past executions.

6. RELATED WORK

In this chapter we discuss the related work. Since we have divided our efforts into correlated areas, we consider two relate work branches. Firstly we review works which address frameworks containing different WE implementations, aiming to simplify the training process for the user perspective and performance improvements. Furthermore, we review works that adopted neural networks and SA as a parameter adjustment technique.

6.1 High-Performance Computing and Natural Language Processing focused frameworks

Akbik A. et al. [2] proposed FLAIR, a framework in which the core idea is to present a simple and unified interface for conceptually different types of word and document embeddings, facilitating the training and distribution of state-of-the-art sequence labeling, text classification, and language models. Furthermore, the framework aims to abstract specific engineering challenges that different types of WE raises by presenting a unified interface for all WE and arbitrary combinations of embeddings. As described, FLAIR is a framework designed to facilitate experimentation with different embedding types, as well as training and distributing sequence labeling and text classification models.

Exner et al. [10] present an end-to-end framework to process unstructured natural language content of multilingual documents called KOSHIK. The authors used the Hadoop distributed computing infrastructure to build the framework, aiming to improve scalability. The authors designed an annotation model that allows the processing algorithms to incrementally add layers of annotation without modifying the original document. As the paper describes, by using the framework, it is possible to complete the semantic parsing of the English edition of Wikipedia in less than 20 days and the syntactic parsing of the Swedish one in less than 15 minutes.

Nesi et al. [37] presents a distributed framework based on the Apache Hadoop ecosystem for crawling web documents, running Natural Language Processing tasks in parallel. The system uses a MapReduce parallel programming paradigm. The authors highlight as main contributions offered by this work the capability of executing general purpose GATE softwares and open source software toolkits for text processing problems, in a distributed design. As the paper describes, the evaluation process performing on different cluster configurations (from 2 to 5 nodes) have shown nearly linear scalability, which is an encouraging result for future assessments on even larger datasets and cluster configurations.

The works mentioned above present frameworks that intent to improve tasks for the NLP area. The relation with our work is the attempt to optimize such processes, allow-

ing complex tasks to be solved more easily through frameworks. Such efforts may also be understood as usability improvements, simplifying operability in the usage of WE tools. The first work presented focuses on embeddings tasks. The last two presents frameworks focused on parallel and distributed infrastructures. In our work, differently from the considered related work, we focus on resource management, intending to mitigate issues to achieve performance gains.

6.2 Simulated Annealing algorithm applied to parameter tuning problems

Mollee et al. [34] employed parameter tuning techniques through the Simulated Annealing algorithm to explore how parameter tuning can be used to analyze the behavior of computational models and to investigate the reasons behind the observed behavior. The authors established that the usage of parameter tuning techniques to analyze and better understand the behavior of dynamic computational models has indeed the potential to provide insights into the structural properties of the models.

Matuszyk et al. [29] investigated the applicability of optimization strategies to recommender systems. The comparative approaches nine different algorithms in an extensive evaluation on real-world datasets. Regarding the best-performing optimization algorithm, the authors recommend the random search for optimizing hyperparameters in the domain of recommender systems. In application scenarios, where the parameters are numerical and parallelization is not necessary, the authors recommend the application of the Nelder-Mead [36] algorithm or of Simulated Annealing.

Mundada et al. [35] presented the effort made to optimize the surface roughness value in the context of optimization of process parameters not only increases the efficacy for machining economics. The authors applied Simulated Annealing and Artificial Neural Network Algorithms to optimize the response. Moreover, the work presents a comparison and analyzes among predicted values using a genetic algorithm, simulated annealing and, neural network algorithm. The SA algorithm was applied to find the best cutting conditions leading to minimum surface roughness. From the optimization results, it was possible to select an appropriate combination of input parameters to obtain minimum roughness. As described, the methodology of prediction of optimum cutting conditions and tool geometry using surface roughness model can be made use of in computer aided process planning stage of computer aided manufacturing.

The works described above presents parameter tuning problems solved by using the SA algorithm, which is based on the cooling process technique to get the ground state of metals. As previously discussed, this algorithm accepts worse solutions in early iterations, which is used to scape from local optima. Despite the focus of these works are not specifically on the NLP parametrization, nor resource consumption and HPC problems, the

process to find the optimum cutting conditions to optimize a specific task is similar to ours. Once as the iterations proceeds and the temperature cools down, the algorithm intents to find the best combination available, considering the possible states. In our work, we utilize the SA algorithm, aiming to find the best parameter combination in a specific scenario, where we first train an MLP with resource consumption history. Then, similarly to the related works, we search for the best parameter combination inside the vector space.

7. CONCLUSION

Aiming to promote a better understanding of the available implementations of WE algorithms and aid researchers to make more consistent choices regarding their applicability, we present in this master thesis a detailed performance evaluation that evidences their specific characteristics. We analyze performance, efficiency, memory consumption, models quality, and usability, in multicore and multinode architectures. Thus, it can become a strong ally to users and researchers, serving as an aid for choosing the most appropriate version for the desired output model, as well as the available computational resources.

In this interdisciplinary work, we also developed a wrapper library that integrates the more optimized parallel and distributed versions of the WE algorithm with the most popular NLP tools, improving their usability. This resulted in an average performance gain of 15x for multicores and 105x for multinodes compared to the original version. There is also a big reduction in the memory footprint allowing the execution of bigger models. Furthermore, we present a parameter tuning model to automatically suggest the best parameter setup to further optimize the execution of the parallel and distributed implementations presented in this work. Our parameter tuning model utilizes an MLP neural network and the Simulated Annealing algorithm. Associated with the wrapper library, previously presented, the tuning model can be used to make suggestions and to find efficient parameter combinations.

In this master thesis, we present parallel programming as an alternative to achieve high-performance and scalability over NLP tools, approximating the research areas of NLP and HPC. We believe that this work improves understandability, learnability, and operability in the usage WE tools, allowing non-expert users to achieve performance gains. The scripts and source codes used in our experiments are available on GitHub [3] for reproducibility purposes. The preliminary results of this work has been presented at the 28th Euromicro International Conference on Parallel Distributed and Network-based Processing (PDP 2020) [44]. The main focus of the article was to present a study which addresses resource utilization and performance aspects of known WE algorithms found in the literature. Furthermore, to present our wrapper library for local and remote execution environments containing a set of optimizations.

As extensions and future work, we intend to improve the neural network training to make more specific parameter suggestions by reviewing methods to contemplate diverse input sizes and the remaining parameters not considered in our proof of concept experiment. Furthermore, we intend to extend the neural network model to support all algorithms present in the wrapper library.

[3] https://github.com/mmatheuslyra/Wrapper

REFERENCES

[1] AI Explosion. “Industrial-strength natural language processing networks”. Source: https://spacy.io, Apr 2019.

[2] Akbik, A.; Bergmann, T.; Blythe, D.; Rasul, K.; Schweter, S.; Vollgraf, R. “Flair: An easy-to-use framework for state-of-the-art nlp”. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations), 2019, pp. 54–59.

[3] Bevan, N.; Azuma, M. “Quality in use: Incorporating human factors into the software engineering lifecycle”. In: Proceedings of the IEEE International Symposium on Software Engineering Standards, 1997, pp. 169–179.

[4] Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. “Enriching word vectors with subword information”, *Transactions of the Association for Computational Linguistics*, vol. 5, Jun 2017, pp. 135–146.

[5] BoussaïD, I.; Lepagnot, J.; Siarry, P. “A survey on optimization metaheuristics”, *Information Sciences*, vol. 237, Jul 2013, pp. 82–117.

[6] Cambria, E.; White, B. “Jumping nlp curves: A review of natural language processing research”, *IEEE Computational Intelligence Magazine*, vol. 9, May 2014, pp. 48–57.

[7] Choi, J. D.; Tetreault, J.; Stent, A. “It depends: Dependency parser comparison using a web-based evaluation tool”. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2015, pp. 387–396.

[8] Computing, A. “Torque resource manager documentation”. Source: http://www.adaptivecomputing.com/, Jul 2019.

[9] De Mulder, W.; Bethard, S.; Moens, M.-F. “A survey on the application of recurrent neural networks to statistical language modeling”, *Computer Speech & Language*, vol. 30, Mar 2015, pp. 61–98.

[10] Exner, P.; Nugues, P. “Koshik-a large-scale distributed computing framework for nlp.” In: Proceedings of the International Conference on Pattern Recognition Applications and Methods, 2014, pp. 463–470.

[11] Fonseca, E. “nlpnet — natural language processing with neural networks”. Source: http://nilc.icmc.usp.br/nlpnet/, Apr 2019.

[12] Gil, D.; Girela, J. L.; De Juan, J.; Gomez-Torres, M. J.; Johnsson, M. “Predicting seminal quality with artificial intelligence methods”, *Expert Systems with Applications*, vol. 39, Nov 2012, pp. 12564–12573.

[13] Glover, F.; Kochenberger, G. A. “Handbook of metaheuristics”. Kluwer Academic Publishers, 2003, 557p.

[14] Gollapudi, S. “Practical machine learning”. Packt Publishing Ltd, 2016, 470p.

[15] Gropp, W.; Lusk, E.; Skjellum, A. “Using MPI: Portable Parallel Programming with the Message-Passing Interface (Scientific and Engineering Computation)”. The MIT Press, 2014, 336p.

[16] Guha, D. R.; Patra, S. K. “Cochannel interference minimization using wilcoxon multilayer perceptron neural network”. In: Proceedings of the International Conference on Recent Trends in Information, Telecommunication and Computing, 2010, pp. 145–149.

[17] Hartmann, N. S.; Fonseca, E. R.; Shulby, C. D.; Treviso, M. V.; Rodrigues, J. S.; Aluísio, S. M. “Portuguese word embeddings: Evaluating on word analogies and natural language tasks”. In: Anais do XI Simpósio Brasileiro de Tecnologia da Informação e da Linguagem Humana, 2017, pp. 122–131.

[18] Hirschberg, J.; Manning, C. D. “Advances in natural language processing”, *Science*, vol. 349, Jul 2015, pp. 261–266.

[19] Ji, S.; Satish, N.; Li, S.; Dubey, P. K. “Parallelizing word2vec in shared and distributed memory”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, Sep 2019, pp. 2090–2100.

[20] Joulin, A.; Grave, E.; Bojanowski, P.; Mikolov, T. “Bag of tricks for efficient text classification”. In: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2, Short Papers), 2017, pp. 427–431.

[21] Jurafsky, D.; Martin, J. H. “Speech and language processing”. Pearson Prentice Hall, 2009, 988p.

[22] Kijsipongse, E.; Vannarat, S. “Autonomic resource provisioning in rocks clusters using eucalyptus cloud computing”. In: Proceedings of the International Conference on Management of Emergent Digital EcoSystems, 2010, pp. 61–66.

[23] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. “Optimization by simulated annealing”, *Science*, vol. 220, Nov 1983, pp. 671–680.

[24] Klusáček, D.; Chlumský, V.; Rudová, H. “Planning and optimization in torque resource manager”. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, 2015, pp. 203–206.

[25] LALIC. “Repositório de word embeddings do nilc”. Source: http://nilc.icmc.usp.br/embeddings, Nov 2018.

[26] Levy, O.; Goldberg, Y. “Dependency-based word embeddings”. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2014, pp. 302–308.

[27] Ling, W.; Dyer, C.; Black, A. W.; Trancoso, I. “Two/too simple adaptations of word2vec for syntax problems”. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2015, pp. 1299–1304.

[28] Mahdavi, S.; Shiri, M. E.; Rahnamayan, S. “Metaheuristics in large-scale global continues optimization: A survey”, *Information Sciences*, vol. 295, Oct 2015, pp. 407–428.

[29] Matuszyk, P.; Castillo, R. T.; Kottke, D.; Spiliopoulou, M. “A comparative study on hyperparameter optimization for recommender systems”. In: Proceedings of the Workshop on Recommender Systems and Big Data Analytics, 2016, pp. 13–21.

[30] Meyer, V.; Kirchoff, D. F.; Silva, M. L.; César A. F., D. R. “An interference-aware application classifier based on machine learning to improve scheduling in clouds”. In: Proceedings of the 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2020, pp. 80–87.

[31] Meyer, V.; Xavier, M.; Kirchoff, D.; RIGHI, R.; De Rose, C. A. F. “Performance and cost analysis between elasticity strategies over pipeline-structured applications”. In: Proceedings of the 9th International Conference on Cloud Computing and Services Science (Volume 1: CLOSER), 2019, pp. 404–411.

[32] Mikolov, T.; Chen, K.; Corrado, G. S.; Dean, J. “Efficient estimation of word representations in vector space”, *CoRR*, vol. abs/1301.3781, Sep 2013, pp. 1–12.

[33] Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; Dean, J. “Distributed representations of words and phrases and their compositionality”. In: Proceedings of the Advances in Neural Information Processing Systems, 2013, pp. 3111–3119.

[34] Mollee, J. S.; Araújo, E. F.; Klein, M. C. “Exploring parameter tuning for analysis and optimization of a computational model”. In: Proceedings of the International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2017, pp. 341–352.

[35] Mundada, V.; Narala, S. K. R. “Optimization of milling operations using artificial neural networks (ann) and simulated annealing algorithm (saa)”, *Materials Today*, vol. 5, Dec 2018, pp. 4971–4985.

[36] Nelder, J. A.; Mead, R. “A simplex method for function minimization”, *The Computer Journal*, vol. 7, May 1965, pp. 308–313.

[37] Nesi, P.; Pantaleo, G.; Sanesi, G. “A distributed framework for nlp-based keyword and keyphrase extraction from web pages and documents”. In: Proceedings of the Distributed Multimedia Systems, 2015, pp. 155–161.

[38] OpenMP. “Openmp: The openmp api specification for parallel programming”. Source: https://www.openmp.org/, Jul 2019.

[39] Quinn, M. J. “Parallel Programming in C with MPI and OpenMP”. McGraw-Hill Education Group, 2004, 529p.

[40] Řehůřek, R.; Sojka, P. “Software Framework for Topic Modelling with Large Corpora”. In: Proceedings of the LREC Workshop on New Challenges for NLP Frameworks, 2010, pp. 45–50.

[41] Rengasamy, V.; Fu, T.-Y.; Lee, W.-C.; Madduri, K. “Optimizing word2vec performance on multicore systems”. In: Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, 2017, pp. 3.

[42] Rere, L. R.; Fanany, M. I.; Arymurthy, A. M. “Simulated annealing algorithm for deep learning”, *Procedia Computer Science*, vol. 72, Nov 2015, pp. 137–144.

[43] Secretary, I. C. “Software engineering – product quality, part 1”, Standard, International Organization for Standardization, 2001, 12p.

[44] Silva, M. L.; Meyer, V.; Kirchoff, D. F.; Joaquim Neto, F. S.; Vieira, R.; César De Rose, A. F. “Evaluating the performance and improving the usability of parallel and distributed word embeddings tools”. In: Proceedings of the 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2020, pp. 201–206.

[45] Řehůřek, R. “Gensim: Topic modelling for humans”. Source: https://radimrehurek.com/gensim/, Apr 2019.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br