J. Innovative Eng. Nat. Sci., vol. 5, no. 2, pp. 704-719, 2025.
http://doi.org/10.61112/jiens.1639780

JIENS
Journal of Innovative Engineering and Natural Science
(Yenilikçi Mühendislik ve Doğa Bilimleri Dergisi)
https://dergipark.org.tr/en/pub/jiens
Original Paper

PWFS: A scalable parallel Python module for wrapper feature selection

Hakan Alp Eren[a,*], Savaş Okyay[b] and Nihat Adar[b]

[a]Eskişehir Osmangazi University, Faculty of Engineering and Architecture, Department of Software Engineering, 26040, Eskişehir, Türkiye.
[b]Eskişehir Osmangazi University, Faculty of Engineering and Architecture, Department of Computer Engineering, 26040, Eskişehir, Türkiye.

ARTICLE INFO

Article history:
Received 14 February 2025
Received in revised form 6 April 2025
Accepted 27 April 2025

Available online

Keywords:
Parallel programming
Open-source software
Feature selection
Exhaustive search
Message passing
Python
Machine learning

ABSTRACT

In the field of machine learning, the feature selection process is a crucial step, and it can significantly impact the performance of predictive models. Despite the existence of various time-efficient algorithms, the only method that guarantees problem optimization is exhaustive search, but it requires an enormous computational load. Although the exhaustive search ensures the best feature selection, a lifetime would not be enough after certain large feature counts. This study proposes a generic, scalable open-source parallel Python module to find the best wrapper feature subset in a fully optimized execution time, especially for reasonable feature counts. This parallel wrapper feature selection module, PWFS, is independent of machine learning algorithms and can function with user-defined methods. The framework promises maximum benefit on the machine learning side by empowering parallel performance and efficiency. The system design is built on the most efficient message-passing communication, where the framework distributes the computational load equally among the parallel agents via feature masking. The module is validated on two workstations, one of which is hyper-threading capable. An overall performance gain of 19.77% is achieved with hyper-threading. Various scenarios and experiments yield different speedups and efficiencies up to 96.74%, validating the flexible design of the proposed parallel framework. The source code of the module is available at https://github.com/haeren/parallel-feature-selector and https://pypi.org/project/parallel-feature-selector/.

I. INTRODUCTION

Despite advancements in hardware and software technologies that have greatly improved the processing speed of general-purpose computers, there remain certain computational tasks where performance is lacking. This is particularly accurate for areas such as medical applications, artificial intelligence, scientific simulations, and brute force algorithms. Moreover, the size of the data affects the total runtime in addition to the complexity of the problem. High-performance computing (HPC) is operated to complete such time-consuming problems by parallelizing the tasks. Supercomputers with many computational nodes can achieve HPC; nevertheless, this choice is not always efficient due to cost and accessibility. Alternatively, it is possible to compute in parallel by dividing the task into subtasks and distributing the workload among multiple processors [1].

Machine learning techniques require extensive mathematical operations and a large amount of data. A model utilizing lots of data is more effective in making predictions more accurate; building a model concurrently working on all available processors determines the achievable highest performance. Employing all features when training a model may not always be efficient in any research, especially when working with a considerable number of features. The size of the dataset can be reduced by evaluating which features should be used to achieve the highest gain.

*Corresponding author. Tel.: +90-222-239-3750 / 6936; e-mail: hakan.eren@ogu.edu.tr

Machine learning is utilized in many fields of data science effectively, especially for detecting the most efficient characteristics of data, i.e., feature subsets. There are various feature selection algorithms, such as forward selection, backward elimination, and exhaustive search [2]. Even though the former two are time-efficient, there is no guarantee of acquiring the best result. Besides, testing different combinations of features is a wise strategy when it comes to determining the contribution of each individual feature to general success. The exhaustive search [3] considers all possible feature combinations to achieve the best result. But the operation is costly and time-consuming since the number of subsets increases exponentially for each additional feature. A dataset with f features has 2^f-1 feature subsets in total. The brute force search for large f values can only be completed in a reasonable time with a parallel approach.

In filter feature selection approaches, such as regression, etc., the techniques are independent of classifiers and run more quickly. On the other hand, the wrapper approaches are integrated with learning algorithms for the most optimized classification performance [4].

In this study, we offer a PWFS (Parallel Wrapper Feature Selection) framework that applies brute force methods for selected NP-complete problems. The user can determine the NP-complete problem by providing appropriate algorithms and datasets. While brute force is typically inefficient for large problem sizes, it provides a baseline for evaluating performance and can serve as a fallback when more efficient algorithms are not available. Thus, PWFS can extend the capabilities of brute force methods for NP-complete problems by leveraging the technological limits of current state-of-the-art parallel architectures. By distributing the workload and exploring solution space in parallel, parallel computing offers the potential to tackle larger instances and achieve performance improvements, albeit within the inherent complexity limitations of NP-complete problems. This parallel exploration of different solution paths can potentially lead to the discovery of solutions or acceptable approximations in a reasonable time frame.

The source code of the PWFS module is available at https://github.com/haeren/parallel-feature-selector and https://pypi.org/project/parallel-feature-selector/. The proposed parallel wrapper framework aims that users can operate parallel feature selection on tabular datasets using any machine learning algorithm. The main operation is independent of such parameters and distributes the computational load among the processes. A simple algorithmic flow of the generic framework is illustrated in Figure 1. Each bullet in the figure matches one step in the overall flow. The first step is to choose the dataset that will be used to train the machine learning algorithm. In this parallel machine learning process, the masked features are distributed across multiple machines or cores. The next step is to choose the machine learning algorithm that will be used to learn from data. Different machine learning algorithms may be affected by the parallel process in dissimilar ways. The third step involves configuring the parameters of the machine learning algorithm and the parallel run. This may include setting the number of machines or cores to use, specifying how data should be distributed, and tuning other parameters that affect the performance of the parallel run. The penultimate step involves implementing the proposed algorithm using the chosen machine learning algorithm and parallel run configuration. The final step is to analyze the results of the parallel workflow.

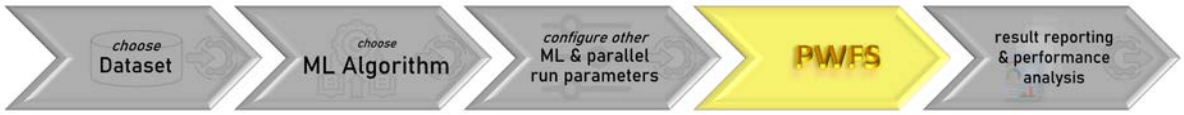

Figure 1. Simple flow of the PWFS framework

The processes operate on their separate data while running the same program. This approach is categorized as a single program multiple data (SPMD) and supports both distributed and shared memory architectures. Machine learning models are trained with different subsets in parallel, and the most successful results are reported together with the corresponding subsets. The developed module has been tested on two workstations, one of which is hyper-threading capable. The parallel algorithm was analyzed with various classifiers on datasets with different feature counts. The high-*efficiency* values observed in the results validate the effectiveness of the algorithm design. A graphical abstract indicating the paper structure and containing a detailed flowchart is in Figure 2.

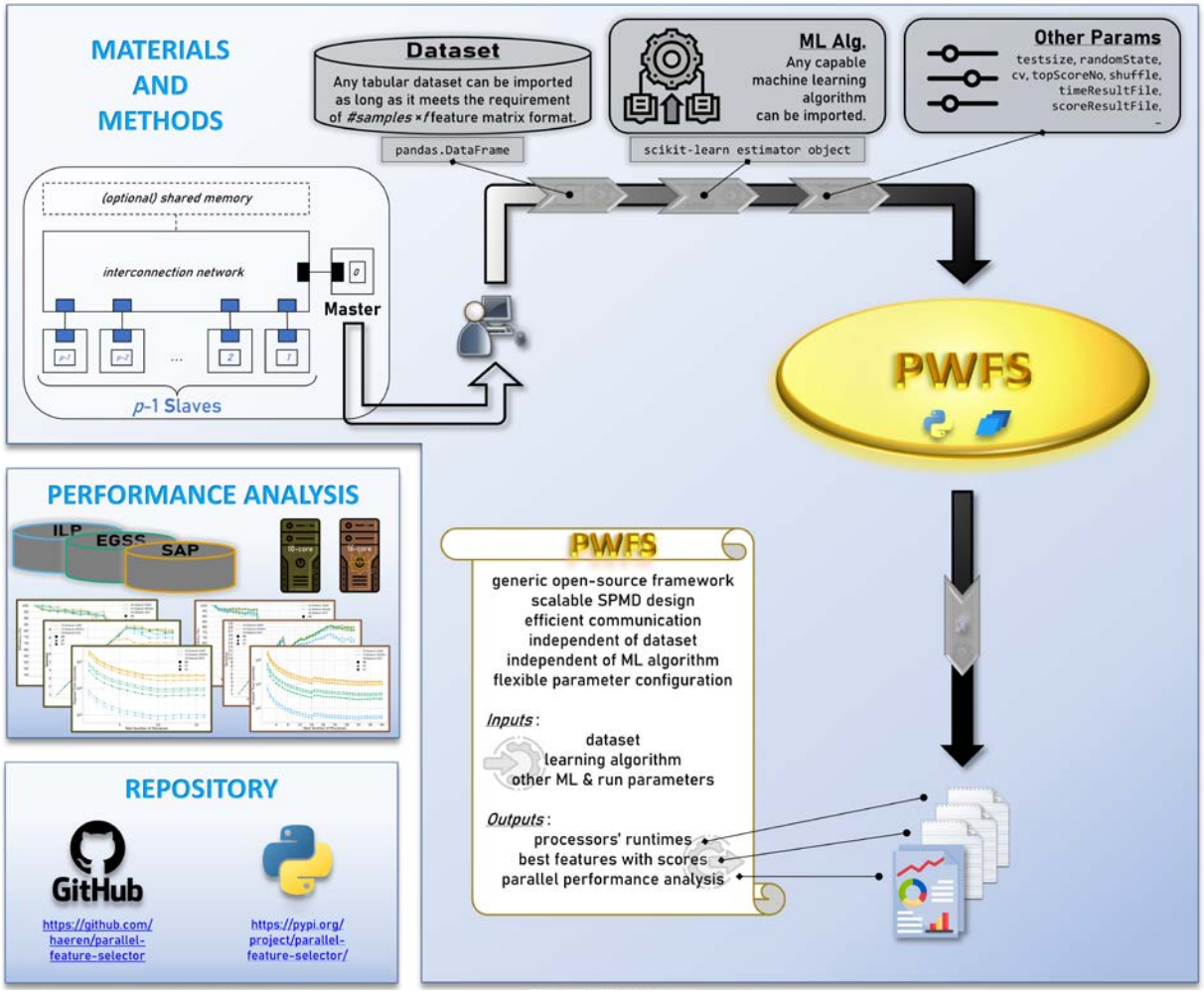

Figure 2. Paper structure and detailed flowchart

There are many studies on feature processing in the field of machine learning. The correct use of features is one of the essential topics, undoubtedly. Acquiring high performance and eliminating the curse of dimensionality issues might be the most noteworthy. For this purpose, some researchers have developed feature selection libraries and tools. Some other researchers have parallelized the feature subset search with various methods to achieve high performance in a shorter time.

Nersisyan et al. [3] proposed ExhauFS, a command-line implementation of the exhaustive search approach for classification and survival regression, supporting *Python* multiprocessing module parallel execution, in which each process performs a search on its own set of *k*-element feature combinations; however, the focus was not parallelization and the hardware performance analysis. Bolón-Canedo, Sánchez-Marono, and Cervino-Rabunal [5] proposed a general framework for feature selection by partitioning the data vertically and parallelizing the process. In this approach, the data is partitioned by features, and stable feature sets are obtained with filters. The features obtained from partitions were combined into a subset. Improvement in runtime is observed in the experiments using three different datasets. Roffo [6] developed a feature selection library named *FSLib*, available for *MATLAB*. Yu, Ding, and Wu [7] presented an online streaming feature selection library for *MATLAB* and *Octave*. Horn, Pack, and Rieger [8] proposed the *autofeat Python* library for automated feature engineering and selection. The *scikit-learn* style linear regression and classification models are available in the library. Masoudi-Sobhanzadeh, Motieghader, and Masoudi-Nejad [9] developed a software named *FeatureSelect* for feature selection. The software is compatible with. mat, *.xls, .txt* input file extensions. Data normalization and fuzzification can be applied to the data. Available learning algorithms consist of support vector machines, decision trees, and artificial neural networks. Users can select a filter or wrapper method. The software generates a detailed report about performance metrics. Pilnenskiy and Smetannikov [10] emphasized that the *Python* programming language was becoming increasingly popular in the field of machine learning, and there was a need for tools. They presented an open-source feature selection library named *ITMO FS*, compatible with *scikit-learn*.

Zhao et al. [11] used variance analysis for large-scale feature selection and implemented two parallel modes named symmetric multiprocessing and massive parallel processing. The algorithm has support for supervised and unsupervised feature selection. Stojanovski [12] analyzed the performance of the parallel approach for exhaustive search. The importance of mutual assistance between parallel agents and dynamic load balancing was emphasized. Sun and Li [13] proposed a data-intensive parallel feature selection method based on the *MapReduce* model. Zhou et al. [14] introduced a parallel feature selection method inspired by group testing. The method easily scales to millions of features and instances by taking advantage of parallelism. El-Alfy and Alshammari [15] used a parallel genetic algorithm in *MapReduce* for feature subset selection. The dataset was divided into smaller parts to distribute the workload among the nodes. Gieseke et al. [16] implemented parallel subset selection for linear regression and a brute-force method to select an optimal feature subset with few features. The implementation was analyzed on different processor types. Li et al. [17] proposed a parallel feature selection method based on *MapReduce* for text classification. The method was used to reduce the computational cost of mutual information to find the correlations between variables. The results showed that it could be applied to large-scale problems in many fields. González-Domínguez et al. [18] introduced a parallelized minimum redundancy maximum relevance (mRMR) implementation called *fast-mRMR-MPI*. Distributed memory clusters were used with the message passing interface (*MPI*) and *OpenMP*. The parallel implementation allowed feature analysis that takes hours using *mRMR* to be completed in seconds with the same result. Nguyen et al. [19] used the trace criterion in linear

discriminant analysis to parallelize the feature selection process. Vivek, Ravi, and Krishna [20] proposed a scalable feature selection method for big data using parallel hybrid wrappers based on evolutionary algorithms.

This study presents an exhaustive wrapper feature selection method that exploits parallel processing as a fully optimized *Python* module. It aims to compare the performances of all feature subsets in a reasonable time with high efficiency. The module is designed to be compatible with *scikit-learn* and user-defined machine learning algorithms. Comprehensive design is not limited to a specific problem and can be used for general purposes. The design spends minimum time on interprocessor communication and achieves the best use of parallelism by distributing the computational load equally to the workers.

II. EXPERIMENTAL METHOD

The proposed PWFS framework is discussed in two subsections: design and implementation. The first subsection focuses on the parallel structure of the algorithm, and the second explains the implementation parameters.

2.1 Design

The problem is optimized in parallel through the functions provided by the *MPI* communication standard. There are various commands for this purpose, such as *send*, *receive*, *scatter*, and *gather*. These commands are utilized to provide communication between processes, especially between the main process (master) and the remaining processes (slaves). Point-to-point (P2P) communication is performed with the *send* and *receive* commands. The *scatter* and *gather* commands allow collective communication between all processes. In the proposed method, the feature subset indices assigned to each process are distributed with the *scatter* operation. The results of each process are collected by the master process with the *gather* operation. The method is designed to have the least number of communications between processes by using the most appropriate parallel operations. Hence, the effect of communication on the runtime is minimal compared to the computation and can be ignored. The relationship between processes, i.e., communication scheme, is listed step by step in Table 1. The visualized version can be followed in Figure 3 in the same order.

Table 1. Communication and computation tasks

	MASTER	**SLAVEs**
	MPI **Initialize**	
1.	Calculate the feature subset interval for each process Each process will have $\sim(2^l\text{-}1)/p$ subsets to analyze	
2.	**Scatter** p intervals	**Scatter** p intervals
3.	Compute *cvAccuracy* and find top n subsets	Compute *cvAccuracy* and find top n subsets
4.	**Gather** $p \times n$ *cvAccuracy* and subsets	**Gather** $p \times n$ *cvAccuracy* and subsets
5.	Sort the $p \times n$ *cvAccuracy* to find top n subsets	
6.	Find the accuracies for top n subsets	
7.	**Gather** p execution times	**Gather** p execution times
8.	Write the results into file	
	MPI **Finalize**	

The PWFS framework is specifically designed to avoid redundant evaluations of feature subsets. Each possible subset is uniquely represented by a binary index corresponding to its feature mask, and the total search space is partitioned into non-overlapping intervals that are distributed across processes using MPI scatter operations. This

exhaustive yet coordinated division ensures that each subset is evaluated exactly once during execution, eliminating redundancy and enabling efficient use of computational resources throughout the parallel process.

The number of computations increases exponentially with the number of features. Each feature subset has a unique index between 1 and 2^f-1, where f is the number of features. The master process adjusts feature subset intervals for each subtask. Thus, feature subsets are distributed evenly among the p processes. Chunk size is $(2^f\text{-}1)/p$ and equal for all processes when 2^f-1 is divisible by p. If not, an arrangement is made for the smallest multiple of p greater than 2^f-1.

Figure 3. Visualized algorithm steps

The binary form of an index value is used to detect favorable features to include. Each binary digit represents a feature. The features included in a subset can be expressed with 1, similar to bit masking. Each process converts the indices to binary form to fetch feature masks. Table 2 shows a case of five processes using feature masks to find which feature(s) to include among four features. In this example, the number of feature subsets is perfectly divisible by the number of processors. In other cases, the balanced distribution settings are adjusted.

Table 2. Example of 4-feature masking for five processes

Process	Interval	Index	Mask	f_4	f_3	f_2	f_1
		1	0001				✓
1	[1, 3]	2	0010			✓	
		3	0011			✓	✓
		4	0100		✓		
2	[4, 6]	5	0101		✓		✓
		6	0110		✓	✓	
		7	0111		✓	✓	✓
3	[7, 9]	8	1000	✓			
		9	1001	✓			✓
		10	1010	✓		✓	
4	[10, 12]	11	1011	✓		✓	✓
		12	1100	✓	✓		
		13	1101	✓	✓		✓
5	[13, 15]	14	1110	✓	✓	✓	
		15	1111	✓	✓	✓	✓

A machine learning model is trained with each feature subset to get accuracy scores. The models' performances are calculated using *k*-fold cross-validation (CV) for each subset. The algorithm checks whether a subset has fewer features besides the validation score during the search. The processes find the candidate solutions which are the most successful subsets in the given local index interval. The master process collects, sorts, and analyzes the candidate solutions during the test phase to get the results. The pseudocode of the module is given in Algorithm 1.

Algorithm 1. Pseudocode of the algorithm

```
Input :  p = size
         f = number of features
         n = output size
Output : bestScores = list of zeros with the length of n initially

if Master then
   length = 2^f−1
   Divide the subset index interval [1, length] into equal p
end
scatter(p intervals)
for i = intervalStart to intervalEnd do
   subset = Fetch feature mask (binarized i)
   cvAccuracy = Train a model with the masked features
   if cvAccuracy > min(bestScores) then
      Replace subset with the minimum scored subset
      (removing subsets having more features is prioritized)
   else if cvAccuracy = min(bestScores) then
      if subset has fewer features then
         Replace subset with a subset having min(bestScores)
      end
   end
end
gather(p × n subsets)
if Master then
   Take top n subsets based on cvAccuracy
   Compute test accuracies for n subsets
end
gather(p elapsed times)
if Master then
   Write test results and performance statistics to output files
end
```

2.2 Implementation

As previously mentioned, the framework is implemented as an open-source *Python* module. The module has several dependencies, such as *MPI for Python* (*mpi4py*), *scikit-learn*, and *pandas*. *mpi4py* package allows *Python* programs to exploit multiple processors [21]. *scikit-learn* provides functions for machine learning algorithms [22], and tabular data can be easily manipulated with *pandas* [23]. The feature selection function in the module has nine parameters in total, as shown in Table 3.

The PWFS framework is designed to be model-agnostic and dataset-independent for structured tabular data. It supports any dataset that can be represented as a *pandas DataFrame*, and any estimator that implements the standard scikit-learn interface—i.e., with *fit()* and *predict()* methods. This includes both built-in scikit-learn models and custom user-defined estimators. By decoupling the feature selection mechanism from the internal structure of the estimator, PWFS ensures broad flexibility and applicability across various use cases, provided that the estimator adheres to the required interface.

Table 3. The list of parameters in the feature selection function

Parameter	Description
data	Dataset as `pandas.DataFrame`
estimator	scikit-learn estimator object
testSize	Test size between 0 and 1 for train/test split
randomState	Random state for train/test split
cv	Number of folds for cross-validation
topScoreNo	Number of the output feature subsets
shuffle	To shuffle the dataset before train/test split
timeResultFile	Output file path for elapsed time per process
scoreResultFile	Output file path for scores and feature subsets

The first parameter *data* is a *pandas* data frame containing the dataset. The second parameter *estimator* is a *scikit-learn* estimator object. User-defined learning algorithms can be used as an estimator. The first two parameters have no default value. The third parameter *testSize* is used to split the dataset into train and test sets with the given ratio. The ratio must be between 0 and 1. The fourth parameter *randomState* controls the random number generator while shuffling the dataset before train/test split. Integer values can be used to get reproducible output. The fifth parameter *cv* is the number of folds for cross-validation. The function applies 5-fold cross-validation by default. The sixth parameter *topScoreNo* specifies the number of feature subsets in the output. The seventh parameter *shuffle* is a boolean variable that controls shuffling before data split. By default, the dataset is shuffled. The last two parameters define the paths of the output files. The function saves the best feature subsets and the execution times of the processes in two separates *.csv* files.

III. RESULTS AND DISCUSSIONS

The proposed framework was tested on two workstations with the highest number of available cores. To conduct, analyze, and report the experiments within a reasonable time frame, datasets with an appropriate number of features were selected, as shown in Table 4. Although the framework is fully capable of operating on datasets with significantly higher dimensionality, the experiments focused on these selected datasets to ensure the timely and reproducible completion of all scenarios. The inherent scalability of the framework allows it to be extended to high-dimensional data using more powerful computing infrastructures, such as HPC clusters.

Table 4. Dataset characteristics used in the experiments

Dataset Name	Number of Features Used	Number of Samples Used	Presence of Missing Values	Number of Classes
Indian Liver Patient	10	570	N/A	2
Electrical Grid Stability Simulated	13	1500	N/A	2
Student Academic Performance	15	480	N/A	3

Datasets with different numbers of features and similar numbers of samples were collected from the *UCI Machine Learning Repository* to analyze the effect of the number of active processes and different numbers of features on the performance. The first dataset *Indian Liver Patient* (*ILP*) has 10 features, and the second dataset *Electrical Grid Stability Simulated* (*EGSS*) consists of 13 features. The last dataset *Students' Academic Performance* (*SAP*), in which categorical attributes were encoded, has 15 features. Since the main motivation is to analyze the effect of the number of active processes and there is more or less the same number of instances for the aforementioned datasets, the minor effect in the sample count variations affecting the completion time and the negligible parallel performance loss resulting from the instance counts were ignored. To interpret the effect over the runtime of

different kinds of learning approaches, various learning algorithms were selected in the experiments, namely, probabilistic classifier *Naïve Bayes* (*NB*), linear classifier *logistic regression* (*LR*), and tree-based classifier *decision tree* (*DT*). Nevertheless, the comprehensive design is not limited to a specific selection and can work with similar learning algorithms. A custom machine learning algorithm can also be integrated into the module, and the PWFS framework guarantees to find the best feature subset for given machine learning parameters. To test this, experiments were repeated multiple times with each learning algorithm and dataset combination. All individual test cases with the same machine learning parameters in the parallel workload have identical classification results, which validates the machine learning side. In detail, all random seeds were set to the same value for reproducibility. Accordingly, the effect of tested machine learning algorithms on the runtime shows similar characteristics in both systems. The shortest runtime is generally observed with the *Naïve Bayes* due to its computation complexity. Since the maximum benefit is achieved within the scope of machine learning, to reveal parallel performance and efficiency, cost analysis within the scope of parallel computing is structured based on the parallel environment and hardware parameters selected during runtime.

The number of cores in the system is an important parameter that affects the parallel runtime in addition to the dataset and the machine learning algorithm. A physical processor can be used as two logical processors with hyper-threading [24]. In this way, the performance of some *MPI* applications can be improved. The performance gain varies depending on the nature of the application. Since computational-intensive applications are likely to utilize CPU resources highly, the chance for performance improvement with hyper-threading is lower [25]. It is possible to increase performance by up to 30% with Intel® hyper-threading technology [24]. For this purpose, two workstations, specifically one of which is hyper-threading capable, were selected to perform the experiments. The first system has dual Intel Xeon E5-2620 v4 @ 2.10 *GHz* processors with eight cores and sixteen threads (a total of sixteen cores and thirty-two threads), 16 GB 2133 *MHz* DDR4 RAM. The second system has Intel Xeon Bronze 3104 1.7 *GHz* processor with twelve cores and twelve threads (ten cores participated in the experiments), 32 GB 2666 *MHz* DDR4 RAM.

Regarding reliability, the procedure was repeated multiple times to mitigate the randomness effect and provide stable analyses. In each independent analysis, the folds were shuffled, and the experiments were repeated five times in total on both workstations. The results obtained were evaluated together, and the average of all individual test results was reported finally.

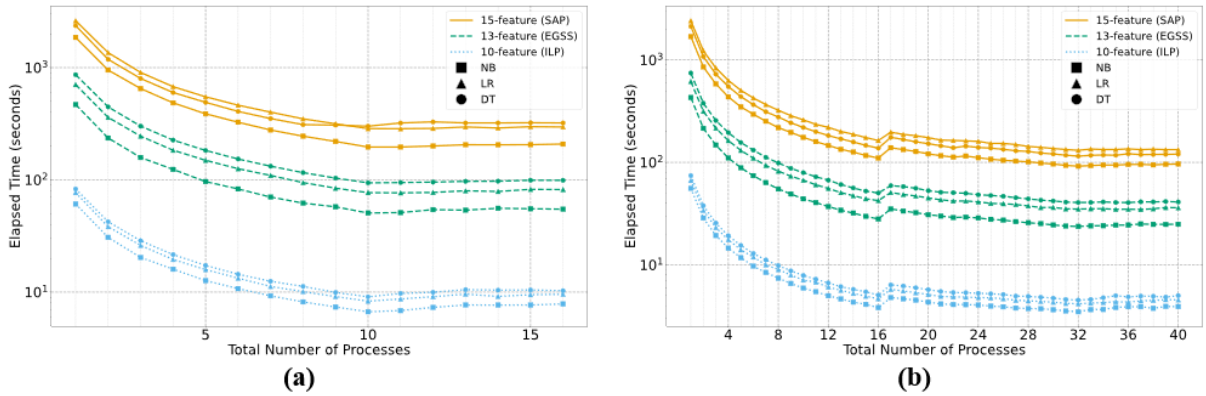

Figure 4. Elapsed time for the **(a)** 10-core and **(b)** 16-core systems

The execution times obtained with different numbers of processes are shown in Figure 4 (see the tables in Appendix A for detailed information about the values). In all plot-related visuals, the (a) parts symbolize the 10-core workstation while (b) is for the hyper-threading capable 16-core workstation. Tests performed on the 10-core system show improvement in runtime up to ten processes. The same outcome applies to the 16-core system up to the full load. The notable effect of hyper-threading is evident between 17 and 32 processes. The minimum runtime is achieved when all processes are loaded with hyper-threading.

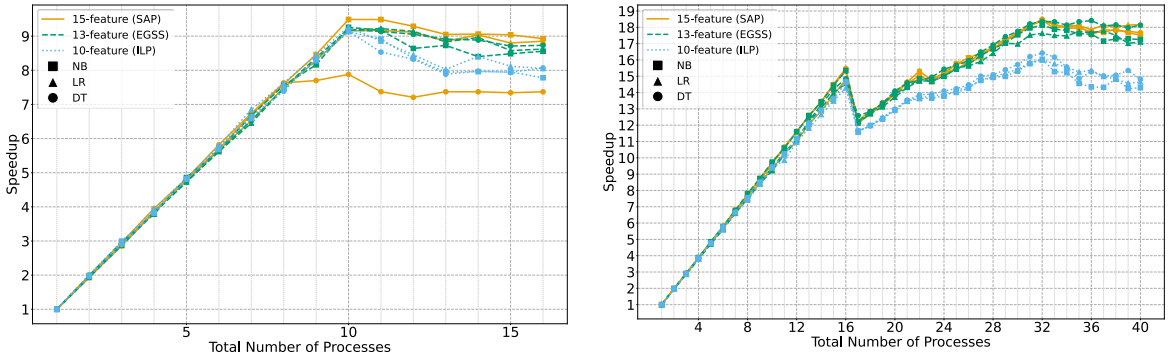

Figure 5. Speedup for the **(a)** 10-core and **(b)** 16-core systems

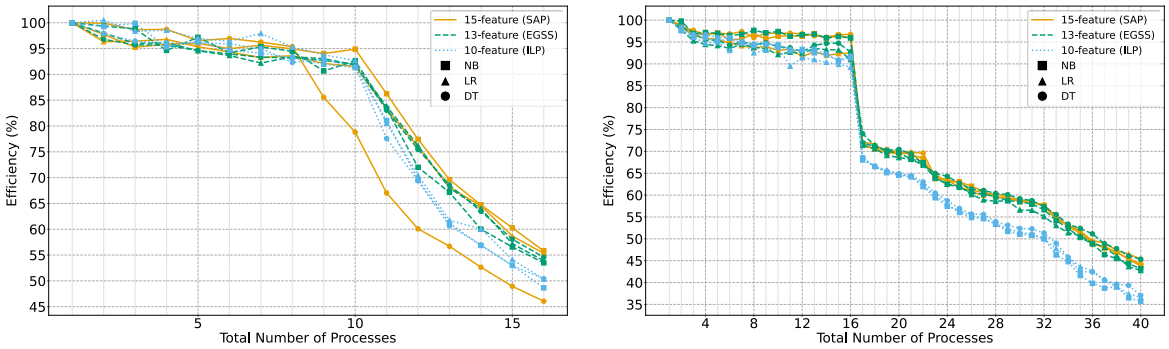

Figure 6. Efficiency for the **(a)** 10-core and **(b)** 16-core systems

$$S(p) = \frac{T_s}{T_p} \tag{1}$$

$$E(p) = \frac{S(p)}{p} \tag{2}$$

Two important performance metrics are frequently discussed in literature to measure the effect of a parallel design. In Eq. (1), speedup gives the ratio of the computation time for the sequential algorithm utilizing one processor to the time for the parallel algorithm utilizing p processors. If the speedup factor is p, then it is called to have a p-fold speedup (i.e., the highest performance is achieved) [26]. In Eq. (2), efficiency indicates how efficiently p processors are utilized [26].

The speedup values for both systems are shown in Figure 5. In theory, a linear increase is expected till reaching the full load. However, the behavior of the linear increase changes in the orange line with circles on the 10-core system (with a lower RAM capacity) due to the computation complexity and memory usage of decision tree algorithm and increased number of features. Besides, there is a linear increase for all lines on the 16-core system. There is still an increase in the speedup after the full-core-load processes with hyper-threading; this is a solid indicator to interpret the contribution of hyper-threading. In the experiment with the highest speedup value, a 16.29% performance gain is achieved with hyper-threading. In general, there is a performance increase of up to 19.77%.

Efficiency graphs obtained from the speedup values are given in Figure 6. The efficiency is around 95% for the full load on the 10-core system. On the 16-core system, the efficiency is over 90% without hyper-threading. Although there is a sudden decrease in efficiency at 17 processes, the hyper-threading impact is reflected in the performance afterward. Notably, higher efficiency is achieved for a higher number of features.

IV. CONCLUSIONS

In order to construct better machine learning models, it is necessary to identify the features that will provide the highest success. Although there are time-efficient algorithms for this process, exhaustive search is the only method that guarantees finding the best results. In this study, a novel parallel framework, PWFS, is developed as a *Python* module to find the best feature subset(s) of a dataset in a reasonable time. The framework promises the maximum benefit on the machine learning side by underlining parallel performance and efficiency. PWFS has a considerable number of programmatic features. While developing the parallel algorithm, a limited set of *MPI* instructions ensuring efficient communication were used so that the proposed algorithm can easily be adapted to Hadoop clusters on cloud systems. The proposed method aims to reduce the overall runtime in exhaustive feature search to an acceptable level proportional to the number of cores with a scalable SPMD design. For performance analysis, different scenarios are realized on two different workstations, one of which is hyper-threading capable. An overall performance gain of 19.77% is achieved with hyper-threading. Various experiments yield vivid speedups and efficiencies up to 96.74%. The high-*efficiency* values obtained in the experimental results confirm that this method is suitable for runtime-optimized feature selection. In addition, a custom machine learning algorithm might also be executed in the flexible module. Since the generic open-source module is independent of the dataset and any machine learning algorithm, researchers working on different subjects can benefit. New program features, considering more effective hardware utilization and more efficient dynamic load balancing, while incorporating advanced feature engineering techniques along with new performance evaluations, are planned for future releases.

While GPU-based solutions are known to accelerate massively parallel numerical computations, the PWFS framework was deliberately implemented using CPU-based parallelism due to practical, architectural, and compatibility considerations. The primary target of the framework is traditional machine learning algorithms—such as decision trees, logistic regression, and Naïve Bayes—which typically do not benefit significantly from GPU acceleration, as they rely more on conditional logic and branching structures than on large-scale matrix operations. Additionally, PWFS is built upon the scikit-learn ecosystem, which is inherently optimized for CPU usage and lacks native support for GPU execution. Ensuring compatibility with scikit-learn was a key design goal, as it allows seamless integration with a wide range of existing models and promotes broad accessibility. Future

work may consider GPU-supported extensions, especially for use cases that involve deep learning or large-scale numerical operations.

Compared to existing feature selection approaches, PWFS offers several notable advantages. Filter-based methods are typically faster but ignore the interaction between features and classifiers. Wrapper methods, while more accurate, are computationally expensive. PWFS bridges this gap by offering an exhaustive wrapper method enhanced with parallel processing. Unlike most existing tools, it is fully compatible with scikit-learn and supports any model implementing standard *fit()* and *predict()* methods. Its open-source implementation, ease of integration, and model-agnostic design make it highly adaptable for various real-world use cases. While this study focuses on implementation and performance within parallel environments, future work may include direct benchmarking with other popular feature selection tools to further validate these advantages.

REFERENCES

1. Okyay S, Adar N (2018) Parallel 3D brain modeling & feature extraction: ADNI dataset case study. 14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), Lviv-Slavske, Ukraine, Feb. 20-24. https://doi.org/10.1109/TCSET.2018.8336172
2. Jovi A, Brki K, Bogunovi N (2015) A review of feature selection methods with applications. 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, May 25-29. https://doi.org/10.1109/MIPRO.2015.7160458
3. Nersisyan S, Novosad V, Galatenko A, Sokolov A, Bokov G, Konovalov A et al (2022) ExhauFS: exhaustive search-based feature selection for classification and survival regression. PeerJ 10:e13200. https://doi.org/10.7717/peerj.13200
4. Okyay S, Adar N (2021) Filter Feature Selection Analysis to Determine the Characteristics of Dementia. Journal of Engineering and Architecture Faculty of Eskisehir Osmangazi University 29(1):20–7. https://doi.org/10.31796/ogummf.768872
5. Bolón-Canedo V, Sánchez-Marono N, Cervino-Rabunal J (2014) Toward parallel feature selection from vertically partitioned data. ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges, Belgium, Apr. 23-25.
6. Roffo G (2016) Feature selection library (MATLAB toolbox). arXiv preprint arXiv:160701327.
7. Yu K, Ding W, Wu X (2016) LOFS: A library of online streaming feature selection. Knowledge-Based Systems 113:1–3. https://doi.org/10.1016/j.knosys.2016.08.026
8. Horn F, Pack R, Rieger M (2019) The autofeat python library for automated feature engineering and selection. Machine Learning and Knowledge Discovery in Databases: International Workshops of ECML PKDD 2019, Würzburg, Germany, Sep. 16-20.
9. Masoudi-Sobhanzadeh Y, Motieghader H, Masoudi-Nejad A (2019) FeatureSelect: a software for feature selection based on machine learning approaches. BMC Bioinformatics 20:1–17. https://doi.org/10.1186/s12859-019-2754-0
10. Pilnenskiy N, Smetannikov I (2020) Feature selection algorithms as one of the python data analytical tools. Future Internet 12(3):54. https://doi.org/10.3390/fi12030054
11. Zhao Z, Zhang R, Cox J, Duling D, Sarle W (2013) Massively parallel feature selection: an approach based on variance preservation. Mach Learning 92:195–220. https://doi.org/10.1007/s10994-013-5373-4
12. Stojanovski TD (2014) Performance of exhaustive search with parallel agents. Turkish Journal of Electrical Engineering and Computer Sciences 22(5):1382–94. https://doi.org/10.3906/elk-1210-105
13. Sun Z, Li Z (2014) Data intensive parallel feature selection method study. International Joint Conference on Neural Networks (IJCNN), Beijing, China, Jul. 6-11. https://doi.org/10.1109/IJCNN.2014.6889409
14. Zhou Y, Porwal U, Zhang C, Ngo HQ, Nguyen X, Ré C et al (2014) Parallel feature selection inspired by group testing. Advances in Neural Information Processing Systems 27.
15. El-Alfy ESM, Alshammari MA (2016) Towards scalable rough set based attribute subset selection for intrusion detection using parallel genetic algorithm in MapReduce. Simulation Modelling Practice and Theory 64:18–29. https://doi.org/10.1016/j.simpat.2016.01.010
16. Gieseke F, Polsterer KL, Mahabal A, Igel C, Heskes T (2017) Massively-parallel best subset selection for ordinary least-squares regression. IEEE Symposium Series on Computational Intelligence (SSCI), Honolulu, HI, USA, Nov. 27 – Dec. 1. https://doi.org/10.1109/SSCI.2017.8285225

17. Li Z, Lu W, Sun Z, Xing W (2017) A parallel feature selection method study for text classification. Neural Computing and Applications 28:513–24. https://doi.org/10.1007/s00521-016-2351-3
18. González-Domínguez J, Bolón-Canedo V, Freire B, Touriño J (2019) Parallel feature selection for distributed-memory clusters. Information Sciences 496:399–409. https://doi.org/10.1016/j.ins.2019.01.050
19. Nguyen T, Phan N, Nguyen N, Nguyen BT, Halvorsen P, Riegler MA (2022) Parallel feature selection based on the trace ratio criterion. International Joint Conference on Neural Networks (IJCNN), Padua, Italy, Jul. 18-23. https://doi.org/10.1109/IJCNN55064.2022.9892181
20. Vivek Y, Ravi V, Krishna PR (2023) Scalable feature subset selection for big data using parallel hybrid evolutionary algorithm based wrapper under apache spark environment. Cluster Computing 26(3):1949–83. https://doi.org/10.1007/s10586-022-03725-w
21. Dalcin LD, Paz RR, Kler PA, Cosimo A (2011) Parallel distributed computing using Python. Advances in Water Resources 34(9):1124–39. https://doi.org/10.1016/j.advwatres.2011.04.013
22. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O et al (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12(85):2825–30.
23. McKinney W (2010) Data structures for statistical computing in Python. SciPy 445(1):51–6. https://doi.org/10.25080/Majora-92bf1922-00a
24. Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA et al (2002) Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal 6(1).
25. Tau Leng RA, Hsieh J, Mashayekhi V, Rooholamini R (2002) An empirical study of hyper-threading in high performance computing clusters. Linux HPC Revolution 45.
26. Eager DL, Zahorjan J, Lazowska ED (1989) Speedup versus efficiency in parallel systems. IEEE Transactions on Computers 38(3):408–23. https://doi.org/10.1109/12.21127

APPENDICES

APPENDIX A

Table A1. Elapsed time, speedup, and efficiency on the 10-core system using the 10-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	60.99	1.00	1.00	76.94	1.00	1.00	82.97	1.00	1.00
2	30.72	1.99	0.99	38.27	2.01	1.01	42.37	1.96	0.98
3	20.35	3.00	1.00	26.07	2.95	0.98	28.68	2.89	0.96
4	16.00	3.81	0.95	19.52	3.94	0.99	21.58	3.84	0.96
5	12.65	4.82	0.96	15.87	4.85	0.97	17.27	4.81	0.96
6	10.73	5.68	0.95	13.30	5.79	0.96	14.42	5.75	0.96
7	9.26	6.59	0.94	11.22	6.85	0.98	12.47	6.65	0.95
8	8.17	7.46	0.93	10.09	7.62	0.95	11.23	7.39	0.92
9	7.37	8.28	0.92	9.11	8.44	0.94	9.92	8.36	0.93
10	6.68	9.13	0.91	8.30	9.27	0.93	9.07	9.14	0.91
11	6.84	8.91	0.81	8.69	8.86	0.81	9.72	8.53	0.78
12	7.31	8.35	0.70	9.10	8.45	0.70	9.96	8.33	0.69
13	7.67	7.95	0.61	9.59	8.03	0.62	10.52	7.89	0.61
14	7.66	7.96	0.57	9.15	8.41	0.60	10.41	7.97	0.57
15	7.67	7.95	0.53	9.48	8.12	0.54	10.45	7.94	0.53
16	7.84	7.78	0.49	9.55	8.05	0.50	10.28	8.07	0.50

Table A2. Elapsed time, speedup, and efficiency on the 16-core system using the 10-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	55.84	1.00	1.00	66.59	1.00	1.00	74.31	1.00	1.00
2	28.62	1.95	0.98	33.95	1.96	0.98	37.76	1.97	0.98
3	19.37	2.88	0.96	23.10	2.88	0.96	25.61	2.90	0.97
4	14.54	3.84	0.96	17.57	3.79	0.95	19.22	3.87	0.97
5	11.74	4.76	0.95	14.01	4.75	0.95	15.48	4.80	0.96
6	9.71	5.75	0.96	11.92	5.58	0.93	12.90	5.76	0.96
7	8.42	6.63	0.95	10.00	6.66	0.95	11.15	6.66	0.95
8	7.41	7.54	0.94	9.00	7.40	0.93	9.83	7.56	0.95
9	6.57	8.50	0.94	7.95	8.38	0.93	8.74	8.51	0.95
10	5.93	9.42	0.94	7.10	9.38	0.94	7.87	9.44	0.94
11	5.46	10.23	0.93	6.77	9.84	0.89	7.24	10.26	0.93
12	4.99	11.20	0.93	6.07	10.97	0.91	6.66	11.16	0.93
13	4.64	12.03	0.93	5.63	11.83	0.91	6.12	12.14	0.93
14	4.32	12.92	0.92	5.26	12.65	0.90	5.77	12.89	0.92
15	4.11	13.59	0.91	4.94	13.48	0.90	5.45	13.63	0.91
16	3.82	14.62	0.91	4.66	14.28	0.89	5.06	14.69	0.92
17	4.80	11.64	0.68	5.76	11.57	0.68	6.38	11.65	0.69
18	4.66	11.98	0.67	5.55	11.99	0.67	6.21	11.96	0.66
19	4.51	12.37	0.65	5.33	12.48	0.66	5.97	12.44	0.65
20	4.33	12.91	0.65	5.14	12.95	0.65	5.73	12.98	0.65
21	4.14	13.47	0.64	4.92	13.52	0.64	5.49	13.53	0.64
22	4.10	13.63	0.62	4.87	13.67	0.62	5.36	13.86	0.63
23	4.09	13.66	0.59	4.83	13.78	0.60	5.34	13.91	0.60
24	4.05	13.78	0.57	4.79	13.89	0.58	5.27	14.09	0.59
25	3.96	14.12	0.56	4.75	14.01	0.56	5.22	14.24	0.57
26	3.89	14.35	0.55	4.67	14.26	0.55	5.13	14.47	0.56
27	3.79	14.74	0.55	4.50	14.79	0.55	4.95	15.01	0.56
28	3.74	14.93	0.53	4.45	14.95	0.53	4.92	15.10	0.54
29	3.73	14.99	0.52	4.38	15.19	0.52	4.83	15.40	0.53
30	3.64	15.32	0.51	4.32	15.42	0.51	4.73	15.72	0.52
31	3.54	15.77	0.51	4.20	15.84	0.51	4.59	16.20	0.52
32	3.48	16.04	0.50	4.16	16.00	0.50	4.52	16.42	0.51
33	3.65	15.28	0.46	4.22	15.77	0.48	4.59	16.17	0.49
34	3.67	15.22	0.45	4.28	15.56	0.46	4.77	15.57	0.46
35	3.84	14.56	0.42	4.36	15.26	0.44	5.00	14.87	0.42
36	3.89	14.34	0.40	4.33	15.37	0.43	4.86	15.28	0.42
37	3.90	14.32	0.39	4.44	14.99	0.41	4.95	15.02	0.41
38	3.77	14.83	0.39	4.49	14.82	0.39	4.93	15.07	0.40
39	3.92	14.24	0.37	4.57	14.57	0.37	4.84	15.35	0.39
40	3.91	14.29	0.36	4.54	14.66	0.37	5.01	14.82	0.37

Table A3. Elapsed time, speedup, and efficiency on the 10-core system using the 13-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	468.85	1.00	1.00	706.02	1.00	1.00	864.57	1.00	1.00
2	235.97	1.99	0.99	360.22	1.96	0.98	446.59	1.94	0.97
3	158.15	2.96	0.99	245.11	2.88	0.96	301.05	2.87	0.96
4	123.85	3.79	0.95	183.64	3.84	0.96	226.00	3.83	0.96
5	96.49	4.86	0.97	149.30	4.73	0.95	182.76	4.73	0.95
6	82.97	5.65	0.94	125.63	5.62	0.94	153.35	5.64	0.94
7	70.27	6.67	0.95	109.39	6.45	0.92	132.35	6.53	0.93
8	62.04	7.56	0.94	94.48	7.47	0.93	115.93	7.46	0.93
9	57.46	8.16	0.91	84.30	8.37	0.93	103.62	8.34	0.93
10	50.63	9.26	0.93	76.89	9.18	0.92	94.04	9.19	0.92
11	51.11	9.17	0.83	76.57	9.22	0.84	94.62	9.14	0.83
12	54.26	8.64	0.72	77.22	9.14	0.76	95.47	9.06	0.75
13	53.72	8.73	0.67	79.92	8.83	0.68	96.99	8.91	0.69
14	55.80	8.40	0.60	78.79	8.96	0.64	97.29	8.89	0.63
15	55.26	8.49	0.57	82.30	8.58	0.57	99.24	8.71	0.58
16	54.79	8.56	0.53	81.89	8.62	0.54	99.02	8.73	0.55

Table A4. Elapsed time, speedup, and efficiency on the 16-core system using the 13-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	429.53	1.00	1.00	616.98	1.00	1.00	745.55	1.00	1.00
2	215.20	2.00	1.00	315.54	1.96	0.98	378.44	1.97	0.99
3	147.72	2.91	0.97	215.89	2.86	0.95	256.45	2.91	0.97
4	110.49	3.89	0.97	163.34	3.78	0.94	195.15	3.82	0.96
5	88.60	4.85	0.97	131.02	4.71	0.94	156.31	4.77	0.95
6	74.17	5.79	0.97	109.94	5.61	0.94	131.78	5.66	0.94
7	63.42	6.77	0.97	93.25	6.62	0.95	112.14	6.65	0.95
8	54.97	7.81	0.98	82.23	7.50	0.94	98.74	7.55	0.94
9	49.16	8.74	0.97	73.36	8.41	0.93	87.36	8.53	0.95
10	44.10	9.74	0.97	66.98	9.21	0.92	79.38	9.39	0.94
11	40.54	10.59	0.96	60.52	10.20	0.93	72.35	10.31	0.94
12	37.04	11.60	0.97	55.66	11.08	0.92	67.00	11.13	0.93
13	34.14	12.58	0.97	50.74	12.16	0.94	60.84	12.25	0.94
14	31.97	13.43	0.96	47.26	13.06	0.93	56.24	13.26	0.95
15	29.75	14.44	0.96	44.14	13.98	0.93	52.47	14.21	0.95
16	27.97	15.35	0.96	42.38	14.56	0.91	50.25	14.84	0.93
17	35.17	12.21	0.72	50.82	12.14	0.71	59.23	12.59	0.74
18	33.53	12.81	0.71	48.56	12.70	0.71	58.11	12.83	0.71
19	32.28	13.30	0.70	47.02	13.12	0.69	55.87	13.34	0.70
20	30.48	13.93	0.70	44.98	13.72	0.69	52.90	14.09	0.70
21	29.96	14.34	0.68	43.10	14.32	0.68	51.10	14.59	0.69
22	28.96	14.83	0.67	41.95	14.71	0.67	50.60	14.74	0.67
23	29.25	14.68	0.64	41.94	14.71	0.64	49.90	14.94	0.65
24	28.64	15.00	0.62	41.15	14.99	0.62	48.32	15.43	0.64
25	27.81	15.44	0.62	39.77	15.52	0.62	47.56	15.67	0.63
26	27.29	15.74	0.61	39.48	15.63	0.60	46.72	15.96	0.61
27	26.40	16.27	0.60	38.80	15.90	0.59	45.22	16.49	0.61
28	25.74	16.68	0.60	37.61	16.41	0.59	44.09	16.91	0.60
29	25.22	17.03	0.59	36.24	17.03	0.59	42.78	17.43	0.60
30	24.49	17.54	0.58	36.34	16.98	0.57	42.00	17.75	0.59
31	23.92	17.96	0.58	35.21	17.53	0.57	40.95	18.21	0.59
32	23.69	18.13	0.57	35.00	17.63	0.55	40.60	18.36	0.57
33	24.01	17.89	0.54	35.25	17.50	0.53	40.66	18.34	0.56
34	24.12	17.81	0.52	35.31	17.47	0.51	41.11	18.13	0.53
35	24.39	17.61	0.50	34.79	17.74	0.51	40.65	18.34	0.52
36	24.43	17.58	0.49	34.91	17.67	0.49	40.50	18.41	0.51
37	25.03	17.16	0.46	34.71	17.78	0.48	41.22	18.09	0.49
38	24.84	17.29	0.45	35.24	17.51	0.46	41.08	18.15	0.48
39	24.84	17.29	0.44	36.26	17.02	0.44	41.48	17.97	0.46
40	24.90	17.25	0.43	36.08	17.10	0.43	41.11	18.13	0.45

Table A5. Elapsed time, speedup, and efficiency on the 10-core system using the 15-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	1860.31	1.00	1.00	2618.79	1.00	1.00	2368.78	1.00	1.00
2	951.50	1.96	0.98	1359.31	1.93	0.96	1185.19	2.00	1.00
3	650.94	2.86	0.95	905.17	2.89	0.96	800.31	2.96	0.99
4	485.09	3.83	0.96	676.45	3.87	0.97	599.86	3.95	0.99
5	387.57	4.80	0.96	549.24	4.77	0.95	490.72	4.83	0.97
6	326.42	5.70	0.95	462.69	5.66	0.94	407.17	5.82	0.97
7	277.84	6.70	0.96	401.10	6.53	0.93	351.56	6.74	0.96
8	244.92	7.60	0.95	349.65	7.49	0.94	310.42	7.63	0.95
9	219.81	8.46	0.94	315.90	8.29	0.92	307.62	7.70	0.86
10	196.06	9.49	0.95	286.10	9.15	0.92	300.58	7.88	0.79
11	196.08	9.49	0.86	285.59	9.17	0.83	321.26	7.37	0.67
12	200.25	9.29	0.77	287.32	9.11	0.76	328.57	7.21	0.60
13	205.56	9.05	0.70	295.80	8.85	0.68	321.35	7.37	0.57
14	205.32	9.06	0.65	289.79	9.04	0.65	321.40	7.37	0.53
15	205.79	9.04	0.60	297.59	8.80	0.59	322.57	7.34	0.49
16	208.41	8.93	0.56	295.83	8.85	0.55	321.43	7.37	0.46

Table A6. Elapsed time, speedup, and efficiency on the 16-core system using the 15-feature dataset

	Naïve Bayes			Logistic Regression			Decision Tree		
Total Process	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**	**Elapsed Time** *(s)*	**Speedup**	**Efficiency**
1	1691.89	1.00	1.00	2418.56	1.00	1.00	2127.95	1.00	1.00
2	855.21	1.98	0.99	1230.21	1.97	0.98	1080.38	1.97	0.98
3	583.84	2.90	0.97	834.93	2.90	0.97	726.27	2.93	0.98
4	436.66	3.87	0.97	631.12	3.83	0.96	550.11	3.87	0.97
5	348.40	4.86	0.97	505.65	4.78	0.96	438.75	4.85	0.97
6	295.40	5.73	0.95	425.31	5.69	0.95	366.18	5.81	0.97
7	252.58	6.70	0.96	366.98	6.59	0.94	312.17	6.82	0.97
8	218.24	7.75	0.97	322.76	7.49	0.94	277.06	7.68	0.96
9	196.42	8.61	0.96	285.90	8.46	0.94	243.65	8.73	0.97
10	175.60	9.64	0.96	259.94	9.30	0.93	219.33	9.70	0.97
11	159.54	10.60	0.96	235.40	10.27	0.93	199.41	10.67	0.97
12	146.30	11.56	0.96	219.89	11.00	0.92	182.97	11.63	0.97
13	134.28	12.60	0.97	200.53	12.06	0.93	169.73	12.54	0.96
14	126.20	13.41	0.96	187.88	12.87	0.92	158.16	13.45	0.96
15	116.85	14.48	0.97	174.61	13.85	0.92	146.72	14.50	0.97
16	110.30	15.34	0.96	163.48	14.79	0.92	137.47	15.48	0.97
17	139.12	12.16	0.72	197.26	12.26	0.72	175.63	12.12	0.71
18	133.26	12.70	0.71	187.65	12.89	0.72	167.38	12.71	0.71
19	127.77	13.24	0.70	182.32	13.27	0.70	158.96	13.39	0.70
20	121.16	13.96	0.70	174.31	13.87	0.69	152.33	13.97	0.70
21	116.23	14.56	0.69	165.11	14.65	0.70	145.35	14.64	0.70
22	112.51	15.04	0.68	163.79	14.77	0.67	138.92	15.32	0.70
23	115.36	14.67	0.64	162.19	14.91	0.65	144.17	14.76	0.64
24	111.15	15.22	0.63	160.37	15.08	0.63	139.79	15.22	0.63
25	107.42	15.75	0.63	153.21	15.79	0.63	137.86	15.44	0.62
26	104.90	16.13	0.62	153.23	15.78	0.61	134.11	15.87	0.61
27	103.00	16.43	0.61	149.28	16.20	0.60	129.83	16.39	0.61
28	101.24	16.71	0.60	143.01	16.91	0.60	127.94	16.63	0.59
29	98.46	17.18	0.59	140.31	17.24	0.59	123.50	17.23	0.59
30	96.37	17.56	0.59	136.67	17.70	0.59	120.84	17.61	0.59
31	93.45	18.10	0.58	133.63	18.10	0.58	117.62	18.09	0.58
32	91.94	18.40	0.58	131.16	18.44	0.58	115.07	18.49	0.58
33	93.13	18.17	0.55	134.91	17.93	0.54	117.19	18.16	0.55
34	94.33	17.94	0.53	133.61	18.10	0.53	118.13	18.01	0.53
35	94.21	17.96	0.51	133.55	18.11	0.52	117.47	18.11	0.52
36	95.43	17.73	0.49	135.43	17.86	0.50	121.35	17.54	0.49
37	95.65	17.69	0.48	133.31	18.14	0.49	119.08	17.87	0.48
38	94.59	17.89	0.47	134.45	17.99	0.47	119.64	17.79	0.47
39	95.77	17.67	0.45	133.32	18.14	0.47	119.86	17.75	0.46
40	96.55	17.52	0.44	133.17	18.16	0.45	120.42	17.67	0.44