ENERGY, EMISSIONS AND PERFORMANCE: CROSS-LANGUAGE AND CROSS-ALGORITHM ANALYSIS IN MACHINE LEARNING

Leonardo Pampaloni
Department of Software Engeneering
University of Florence
Florence, Italy
leonardo.pampaloni1@edu.unifi.it

Niccolò Marini
Department of Artificial Intelligence
University of Florence
Florence, Italy
niccolo.marini@edu.unifi.it

Filippo di Martino
Department of information Technology
University of Florence
Florence, Italy
filippo.dimartino@edu.unifi.it

ABSTRACT

This study investigates the carbon emissions, execution time, and computational resource usage of various machine learning models implemented across multiple programming languages, including Python, MATLAB, R, Java, and C++. Using the CodeCarbon library, we measured and compared the environmental footprint and performance of these models, leveraging different ML frameworks. The algorithms tested include Logistic Regression, AdaBoost, Decision Tree, Random Forest, K-Nearest Neighbors, Support Vector Classifier (SVC), and Naive Bayes, applied to well-known datasets such as Breast Cancer, Wine, and Iris from UC Irvine. By analyzing emissions, energy efficiency, and runtime across languages and frameworks, this paper aims to provide a comprehensive understanding of the trade-offs between environmental sustainability and computational performance in machine learning applications. The results offer practical insights for developers and researchers seeking to reduce the environmental impact of machine learning technologies.

1 Introduction

As machine learning (ML) continues to gain prominence across diverse sectors such as healthcare, finance, and engineering, the computational demands of training and deploying these models have surged. While the development of sophisticated ML algorithms has driven innovation and efficiency, it has also introduced new challenges related to energy consumption and carbon emissions. In a world increasingly focused on sustainability, the environmental impact of ML has emerged as a pressing concern. The relationship between model complexity, computational cost, and environmental sustainability remains largely underexplored.

This study addresses the critical need to evaluate machine learning processes not only in terms of accuracy but also in terms of environmental footprint. The primary focus of our investigation is to assess the carbon emissions, execution time, and resource usage associated with training and testing machine learning models across multiple programming languages—Python, MATLAB, R, Java, and C++. Each language offers distinct performance characteristics, and through various machine learning frameworks, we aim to quantify their environmental efficiency. Popular algorithms such as Logistic Regression, AdaBoost, Decision Tree, Random Forest, K-Nearest Neighbors, Support Vector Classifier (SVC), and Naive Bayes are tested on three standard datasets: Breast Cancer, Wine, and Iris.

Our solution involves leveraging the CodeCarbon library to track energy consumption in real-time across different programming environments and machine learning frameworks. This comprehensive analysis enables us to provide insights into the trade-offs between execution time, accuracy, and emissions, contributing valuable information to the field of sustainable machine learning.

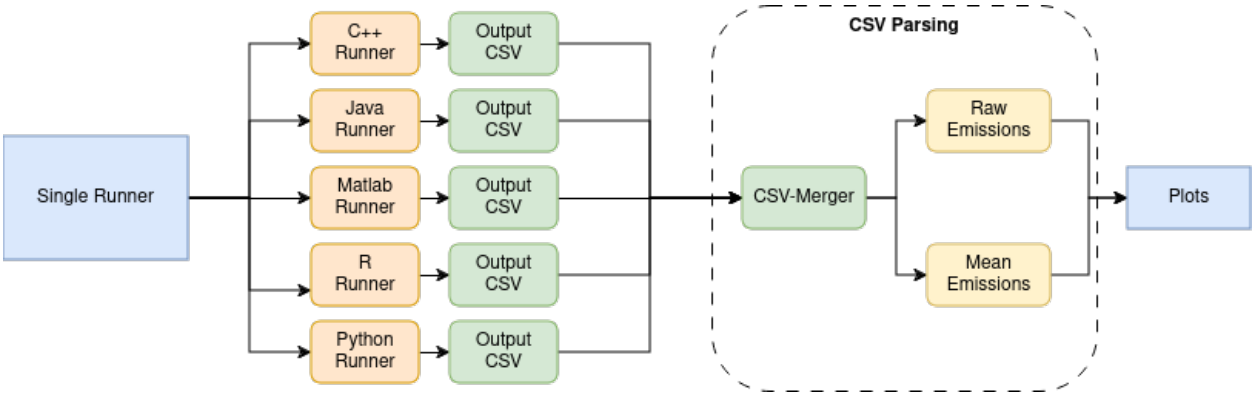

Figure 1: Pipeline outlining the data collection process, where multiple Train/Test runs of various algorithms are executed through a single .sh runner. The resulting output CSV files are then consolidated into two files to streamline data analysis.

The main **contributions** of this study are:

- A detailed comparison of carbon emissions across multiple programming languages and frameworks for common machine learning models.
- An evaluation of execution time and resource consumption as factors in the environmental cost of ML workflows.
- Practical recommendations for selecting programming languages and frameworks based on both performance and sustainability.

This research is intended for data scientists, machine learning practitioners, software engineers, and researchers interested in the environmental impact of their work. It also targets organizations looking to balance machine learning efficiency with sustainability. The outcomes of this study provide actionable insights to help minimize the carbon footprint of machine learning, fostering the development of eco-friendly computational solutions without compromising performance.

2 Related Work

Numerous studies have explored the efficiency of machine learning (ML) algorithms in different programming languages, emphasizing various factors such as computational complexity, optimization techniques, and the role of the programming language itself.

Chen et al. [1] analyzed competitive programming data and highlighted the difficulty developers face in producing efficient code, emphasizing the significant performance gains achievable through machine learning-guided code optimizations. They found that ML techniques, such as VQ-VAE for learned discrete transformations, pointer networks for variable handling, and Transformers pre-trained on large code corpora, could suggest improvements in code efficiency, leading to faster execution times and better resource utilization.

Additionally, Gao et al. [2] discussed the potential of Julia, a relatively new programming language, in striking a balance between simplicity and performance. Julia's high-performance capabilities, particularly for ML algorithms, were demonstrated, making it a strong competitor to more widely used languages like Python and C++.

Aremu and Salako [3] conducted a comparative analysis of several programming languages (e.g., C, C++, Pascal, Java, and Visual BASIC) regarding their efficiency in implementing specific algorithms, such as tree search algorithms. They found that the choice of language could significantly impact algorithm performance, with Pascal outperforming others in Breadth-first searches, while Java was more efficient in Depth-first searches.

Finally, Mitra et al. [4] explored energy-efficient algorithms for machine learning on edge devices, analyzing how computational limitations impact algorithmic performance and energy usage. The study emphasizes optimizing algorithms for low-power devices and introduces JEPO, an Eclipse plugin designed to assist software developers in writing energy-efficient code both dynamically and statically, while also providing measurements for energy consumption at the method level.

These studies collectively underline the importance of both algorithmic complexity and language efficiency in machine learning, with insights into how different languages optimize performance in various computational contexts.

3 Approach

3.1 Datasets Used

For this study, three well-known datasets were selected from the UC Irvine Machine Learning Repository to

evaluate the performance and carbon emissions of different machine learning algorithms. These datasets are:

Breast Cancer Dataset: This dataset is widely used for binary classification tasks. It contains features computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. The features describe the characteristics of the cell nuclei present in the image.

Wine Quality Dataset: This dataset contains physicochemical properties of red and white wines, along with their quality ratings. The task is to predict the quality of the wine based on these properties. The dataset includes both classification and regression problems.

Iris Dataset: One of the most famous datasets in the machine learning community, the Iris dataset is used for multiclass classification. It includes three classes of iris plants, with four features describing the physical dimensions of the flowers.

3.2 Implemented Algorithms

The following machine learning algorithms were implemented to compare their carbon emissions and performance. The selection of these algorithms was determined by their widespread use, enabling implementation across all the chosen programming languages and allowing for a more realistic comparison with real-world scenarios.

Logistic Regression: A statistical model used for binary classification, which predicts the probability of a binary outcome.

AdaBoost: An ensemble learning method that combines multiple weak classifiers to create a strong classifier. It focuses on instances that are misclassified by previous classifiers, improving the model's accuracy.

Decision Tree: A non-parametric supervised learning method used for classification and regression. It splits the data into subsets based on the value of input features.

Random Forest: An ensemble learning method that operates by constructing multiple decision trees during training. It outputs the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

K-Nearest Neighbors (KNN): A simple, instance-based learning algorithm where the prediction for a new instance is determined by the majority class among its k-nearest neighbors in the training data.

Support Vector Classifier (SVC): A supervised learning model that analyzes data for classification and regression analysis. It works by finding the hyperplane that best separates the classes in the feature space.

Naive Bayes: A probabilistic classifier based on Bayes' theorem, assuming independence between features. It is particularly effective for large datasets and text classification tasks.

3.3 Tools and Programming Languages

The full list of libraries and functions for Python, Java, Matlab and R can be viewed in the file `requirementsAll.txt`, the requirements for C++ can be viewed in `requirementsCpp.txt` in the GitHub repository.

The project utilized the following programming languages:

MATLAB: Used for implementing and executing most machine learning algorithms, MATLAB's flexibility in handling different data types and powerful computational capabilities were leveraged through the **Statistics and Machine Learning Toolbox**, which provides an extensive range of built-in functions.

Python: Employed alongside other languages to manage carbon emissions tracking. Scripts automated the starting and stopping of the *CodeCarbon* tracker and processed output data. The **scikit-learn** library was essential for implementing various machine learning algorithms, facilitating seamless experimentation and evaluation.

Java: Utilized for performance-sensitive tasks, particularly where robust, cross-platform support is essential. The **WEKA** library was employed for implementing most machine learning algorithms, while **XGBoost** was used for AdaBoost due to its superior performance and ability to mimic scikit-learn's implementation with decision trees as base learners.

C++: Applied in performance-critical situations requiring fine-grained control over memory management and execution speed. The **mlpack** library was used to implement nearly all machine learning algorithms, while **OpenCV** was chosen for the Naive Bayes algorithm due to its robust implementation.

R: Primarily utilized for its statistical computing capabilities, particularly in analyzing and visualizing results. The tool used to track the emissions is **CodeCarbon:** A Python package used to estimate the carbon footprint of cloud computing and software development processes. It tracks the power usage and converts it into CO_2 emissions.

3.4 Modifications and Integrations

Effectively monitoring and managing carbon emissions during the execution of machine learning models required several key modifications and integrations. A significant challenge was the use of the `CodeCarbon` library, which is specifically designed for Python. This posed difficulties when integrating it with other programming languages.

Initially, the approach involved starting the `CodeCarbon` tracker before running the entire script in each programming language. However, this method introduced bias because it included emissions from additional opera-

tions such as loading, splitting, and defining the datasets, which are not part of the core model development phases. This made it difficult to isolate the emissions attributable solely to the training and testing stages, which are the primary areas of interest.

To address this issue, a more refined solution was implemented. Instead of tracking emissions for the entire script, two Python functions, `start_tracker()` and `stop_tracker()`, were developed to manage emissions tracking. These functions were invoked directly within each language's script to precisely start and stop `CodeCarbon` tracking around the specific lines of code responsible for training and testing the models. This ensured that only the emissions from these core phases were monitored, excluding all side operations and providing a much more accurate measurement of the carbon footprint associated with the machine learning models.

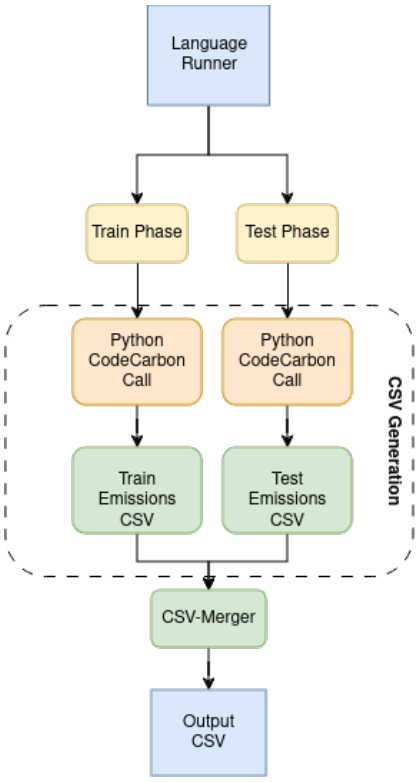

Figure 2: Pipeline illustrating the process of calculating carbon emissions for each programming language used in the experiments.

This method streamlined the emissions tracking process while ensuring consistency and accuracy across different stages of model development. By isolating the emissions data to the training and testing phases alone, the approach minimized potential biases and offered clearer insights into the environmental impact of the machine learning algorithms. In doing so, it provided a more realistic and precise assessment of their carbon emissions, enabling better comparisons between the models.

4 Methodology

4.1 Experimental Setup: Machine Specifications

All experiments were conducted on a MacBook equipped with an Apple M2 chip. The M2 chip, built on Apple's second-generation silicon architecture, is known for its energy efficiency and high computational power, making it an ideal platform for evaluating both the performance and environmental impact of machine learning algorithms. The key specifications of the machine used in this study are as follows:

- **Processor:** Apple M2 chip with an 8-core CPU (4 performance cores and 4 efficiency cores).
- **GPU:** Integrated 10-core GPU.
- **Memory:** 8 GB unified memory.
- **Storage:** 256 GB SSD.
- **Operating System:** macOS Sonoma.

The machine's architecture provides an excellent balance between performance and power consumption, allowing for precise tracking of energy usage during the training and testing phases of machine learning models. This setup ensures that the measured carbon emissions are reflective of a widely accessible computing environment, thereby improving the study's relevance for practitioners working on consumer-grade hardware.

Apple Silicon chips, which integrate both the CPU and GPU, offer an efficient and unified architecture for energy tracking. CodeCarbon monitors the energy consumption on Apple Silicon chips by utilizing `powermetrics`, a native macOS tool that provides detailed power usage statistics for the chip. This ensures accurate and consistent tracking of energy consumption on these machines, further enhancing the precision of the carbon emission measurements.

Note: To use `powermetrics`, sudo privileges are required, meaning that during execution, the system will prompt for the user's password. If you wish to automate the password input, a guide is available in the official documentation.

4.2 Algorithm Testing

For each algorithm-dataset combination, 30 run for both train and test were performed as follows:

Training: The model was trained using 80% of the dataset, and the process was monitored for energy consumption.
Prediction: The trained model was then tested on the remaining 20% of the dataset, with emissions recorded during this phase.

4.3 Data Collection

During the experiments, emissions data were collected using the **CodeCarbon** library. The emissions data were saved separately for each algorithm and dataset combination, allowing for a detailed analysis of the environmental impact associated with different machine learning tasks.

The experiments were repeated 30 times to ensure the reliability of the results, with each run contributing to the final emission and accuracy measurements.

4.4 Merging and Analysis of Results

After completing the experimentation, the emissions data from the training and prediction phases were consolidated into a single comprehensive dataset, named `raw_merged_emissions.csv`. From this dataset, a processed version will be created, called `meanEmission.csv`, which will contain the average emissions for each combination across the 30 runs.

Subsequently, these datasets were combined into a unified dataset, allowing for the comparison of emissions across different algorithms, datasets, languages, and phases of execution. This merged dataset was pivotal in evaluating the carbon footprint of each model.

4.5 Comparison Metrics

The key metrics compared across experiments were:

Duration (T): The total time taken for model training and prediction, measured in seconds (s). This metric provides insight into the computational efficiency of each algorithm. It is calculated as:

$$T = t_{\text{end}} - t_{\text{start}}$$

where t_{start} and t_{end} represent the start and end times of the model execution, respectively.

Energy Consumed (E): The total energy consumption during the model training and prediction phases, measured in kilowatt-hours (kWh). This is derived using the integral of power consumption ($P(t)$) over the duration T:

$$E = \int_0^T P(t)\,dt$$

where $P(t)$ is the instantaneous power consumption at time t.

CO_2 Emissions (C): The carbon emissions produced during the training and prediction phases, measured in kilograms of CO_2 equivalent ($kgCO_2e$). This is proportional to the energy consumed and the carbon intensity factor γ, expressed as:

$$C = \gamma \cdot E$$

where γ represents the amount of CO_2 emitted per kWh of energy consumed ($kgCO_2$/kWh).

Emissions Rate (r_C): The rate of CO_2 emissions relative to the duration of the model's execution, measured in kilograms of CO_2 per second ($kgCO_2$/s). This metric normalizes emissions over the execution time and is given by:

$$r_C = \frac{C}{T}$$

CPU Energy (E_{CPU}): The total energy consumed by the CPU during the model's execution, measured in watt-hours (Wh). This is calculated by integrating the CPU power over time:

$$E_{\text{CPU}} = \int_0^T P_{\text{CPU}}(t)\,dt$$

These metrics provide insights into the efficiency and environmental sustainability of each algorithm in different programming environments, enabling a detailed comparison between the algorithms' performance and their carbon footprint.

5 Results

The results section presents the findings extracted from CodeCarbon. Emission data for each programming language were initially recorded in a CSV file, which was subsequently consolidated into a single comprehensive CSV. This decision was made to improve the clarity of the results without the need for constant recompilation.. From this file, two datasets were derived. The first dataset represents the average emissions values, calculated from 30 runs for each combination of language, dataset, algorithm and ML phase. The second dataset comprises all individual runs. To provide a comprehensive understanding of the emissions associated with different programming languages and algorithms, we present two key visualizations. These plots illustrate the variations in emissions across different languages and datasets, as well as the performance of various algorithms.

5.1 CPU energy results overview

These visualizations offer insights into the environmental impact of different algorithms and help in understanding the overall performance across various programming languages and datasets.

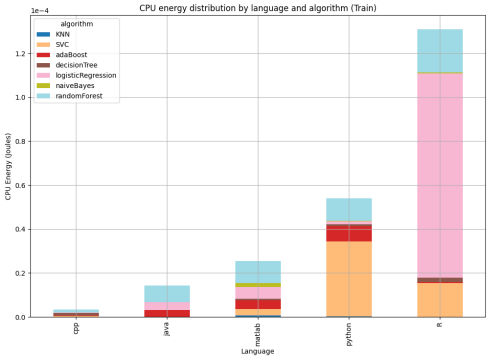

Figure 4: The figure illustrates the behavior of the CPU energy variable across different programming languages. Each language is broken down into its respective components, specifically the algorithms used.

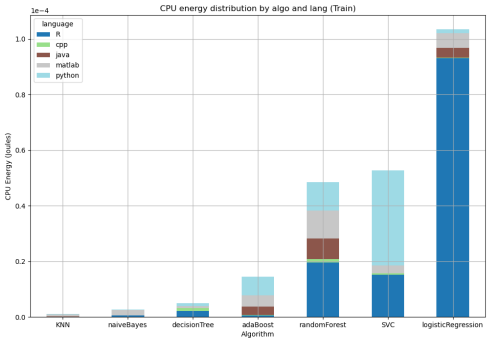

Figure 5: The figure illustrates a different perspective related to the previous image.

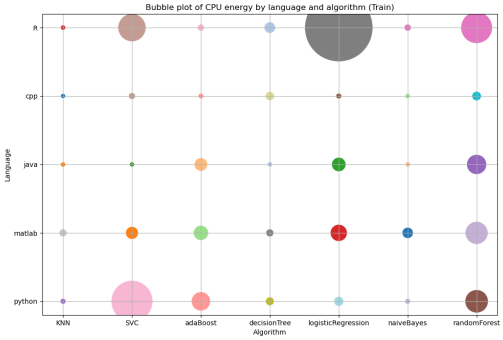

Figure 6: The figure illustrates the bubbleplot of the CPU energy variable across different algorithms.

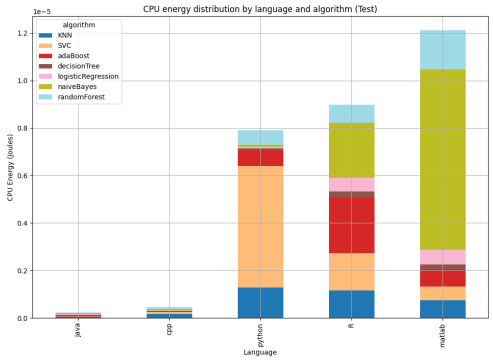

Figure 8: For clearer representation, the data is divided into the two separate phases: training and testing and in this figure we can observe the test phase. The separation by dataset will be addressed later.

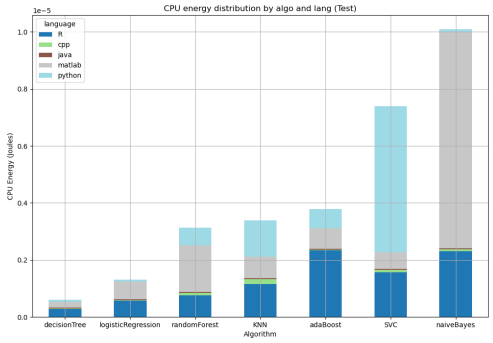

Figure 9: As used before we can observe the division with algorithms on the axis

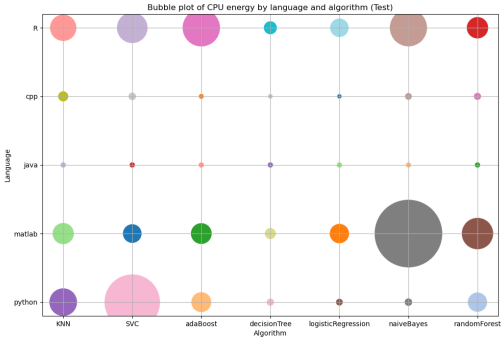

Figure 10: Bubbleplot for the test phase

As observed, contrary to what might have been expected, Python is significantly slower both during the training phase and, even more so, during the testing phase. Although we know that C++ and Java are much faster languages, the popularity of Python for machine learning and deep learning might suggest a higher level of

optimization. At this point, we questioned whether some datasets might have a greater impact than others, given that the Wine Quality dataset contains approximately ten times as many rows as the breast cancer dataset and fourty times the iris dataset.

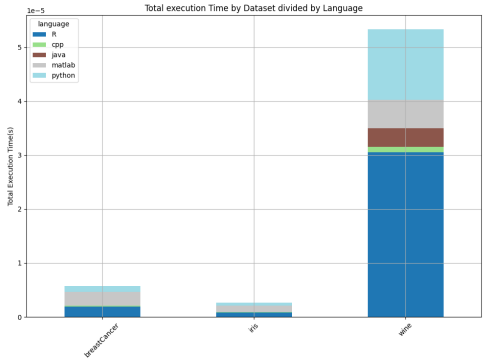

Figure 11: Execution time divided by dataset and language.

The figure above presents a graph illustrating the execution time of each dataset, grouped by programming language. As expected, the majority of computation time is attributed to the Wine dataset, as it is the largest. Multiple datasets were used to assess whether different types of variables could influence the results.

5.2 Heatmaps

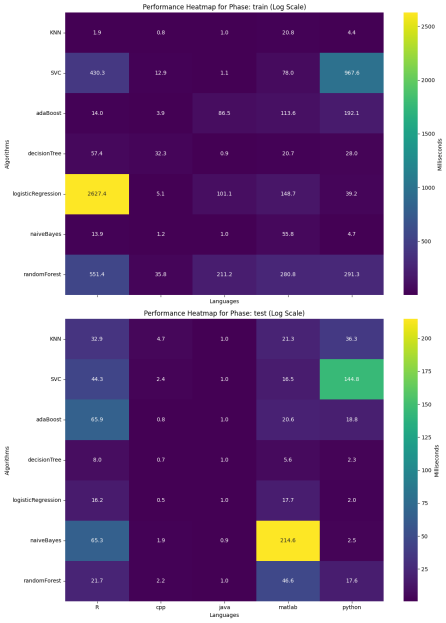

Figure 12: The heatmaps illustrate the combinations that differ from the others. By separating the two phases, we can identify which combinations exhibit less variation. The heatmap uses a scale to better visualize the wide range of execution times.

Heatmaps are a powerful visualization tool used to represent data in a matrix format, where individual values are depicted by varying colors. This type of plot is particularly useful for identifying patterns, trends, and anomalies in complex datasets. By mapping data values to a color scale, heatmaps provide an intuitive and immediate understanding of the intensity or magnitude of the data points across different categories. In our analysis, heatmaps are employed to compare performance metrics across various languages and algorithms, enabling us to visually discern how these combinations impact the overall performance. This visualization helps in quickly spotting areas of significant difference and understanding how different factors interact within the data.

At a glance, the heatmaps reveal that the DecisionTree and KNN algorithms consistently exhibit the smallest execution time across various languages, both in the training and testing phases. This noticeable pattern prompted us to investigate further to understand the extent of this observed difference.

While Logistic Regression also demonstrated relatively good performance in terms of execution time, especially compared to more complex algorithms, it was not given primary focus in our analysis for two reasons. Firstly, Logistic Regression is a well-known, commonly used algorithm, and therefore does not offer significant new insights in the context of energy efficiency analysis. Its frequent use in real-world applications is well-documented, which limits its novelty in this specific comparison.

Secondly, its performance on the R language appeared anomalous. In R, the execution times for Logistic Regression were unusually high compared to the results in other languages, raising questions about potential inefficiencies or differences in implementation. This irregularity made it difficult to directly compare its results across all languages in a consistent manner. Thus, while Logistic Regression remains an important baseline model, our focus naturally shifted to other algorithms, such as DecisionTree and KNN, which showed more consistent and competitive results across the board.

5.3 KNN & DecisionTree

We present a more detailed analysis of the various languages with a focus on these two specific algorithms. This analysis includes a breakdown of the performance across different datasets.

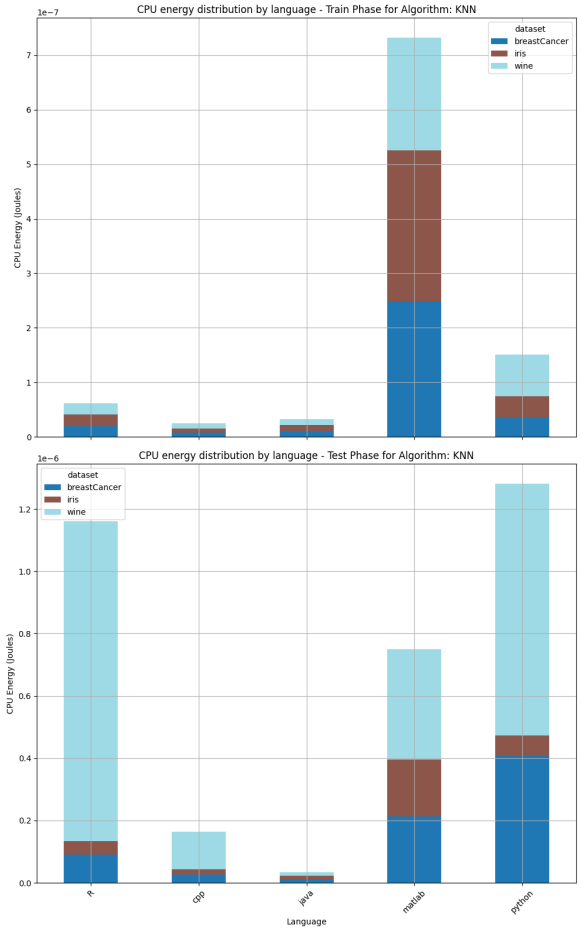

Figure 13: Average CPU energy for KNN for each language divided by dataset

Although KNN might be fairly consistent across all languages during the training phase, except for MATLAB, the testing phase remains as initially observed, with C++ and Java being significantly faster.

The situation is interesting during the testing phase, where Python is slower by a factor of 2 to 3 times compared to java and c++, rather than 50 times as with other algorithms. This configuration could represent an optimal tradeoff for those who need to use Python.

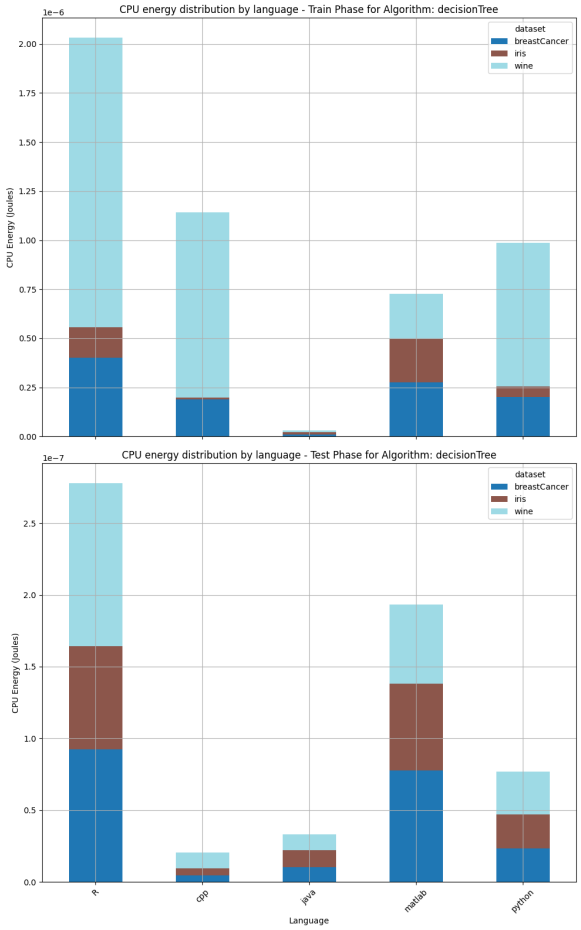

Figure 14: Average CPU energy for decisionTree for each language divided by dataset

5.4 Correlations

At this point, it seemed interesting to explore the correlations between the various features, specifically how emissions, duration, and CPU energy relate to each other. In addition to generating visual plots, we calculated several correlation measures, including Pearson, Spearman, Kendall, and R-squared.

While Pearson's correlation requires certain assumptions, such as the normality of the data, which can be assessed using the Shapiro-Wilk or Kolmogorov-Smirnov tests, and homoscedasticity, determined through the Breusch-Pagan test or by visually inspecting residual plotsour analysis did not strictly adhere to these criteria. This decision was driven by the recognition that exploring data in a less rigid framework can yield valuable insights.

We believe that, even in the absence of these stringent assumptions, the analysis can reveal interesting patterns worth further investigation. This exploratory approach allows for flexibility, as non-parametric tests can also

provide meaningful results despite potential violations of the underlying assumptions of parametric tests.

Additionally, real-world data often do not conform to theoretical distributions. Thus, analyzing the data as they are can offer a more authentic perspective on the phenomena under study. The results obtained here can serve as a foundation for subsequent analyses, where we might apply more rigorous methodologies to confirm and deepen our findings.

Below, we explain each of these correlation measures and provide their mathematical formulas.

5.4.1 Pearson Correlation Coefficient

The Pearson correlation coefficient measures the linear relationship between two continuous variables. It ranges from -1 to 1, where 1 indicates a perfect positive linear relationship, -1 indicates a perfect negative linear relationship, and 0 indicates no linear relationship. The formula for Pearson's correlation coefficient r is:

$$r = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{1}$$

where $\text{cov}(X, Y)$ is the covariance of X and Y, and σ_X and σ_Y are the standard deviations of X and Y, respectively. Covariance is calculated as:

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \tag{2}$$

where x_i and y_i are the values of the variables X and Y, and $\bar{x}$ and $\bar{y}$ are their means.

5.4.2 Spearman's Rank Correlation Coefficient

Spearman's rank correlation coefficient assesses how well the relationship between two variables can be described by a monotonic function. Unlike Pearson's coefficient, it does not assume a linear relationship. The formula for Spearman's rank correlation coefficient ρ is:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{3}$$

where d_i is the difference between the ranks of each pair of values, and n is the number of observations.

5.4.3 Kendall's Tau

Kendall's Tau is another measure of correlation that assesses the strength of association between two variables. It is based on the concept of concordant and discordant pairs. The formula for Kendall's Tau τ is:

$$\tau = \frac{C - D}{\frac{1}{2} n(n-1)} \tag{4}$$

where C is the number of concordant pairs, D is the number of discordant pairs, and n is the number of observations.

5.4.4 Coefficient of Determination (R-squared)

The coefficient of determination R^2 measures the proportion of variance in the dependent variable that is predictable from the independent variable. It is used to assess the goodness of fit of a regression model. The formula for R^2 is:

$$R^2 = \frac{\text{SS}_{\text{reg}}}{\text{SS}_{\text{tot}}} \tag{5}$$

where SS_{reg} is the sum of squares due to regression, and SS_{tot} is the total sum of squares:

$$\text{SS}_{\text{reg}} = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 \tag{6}$$

$$\text{SS}_{\text{tot}} = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{7}$$

where $\hat{y}_i$ are the predicted values, y_i are the actual values, and $\bar{y}$ is the mean of the actual values.

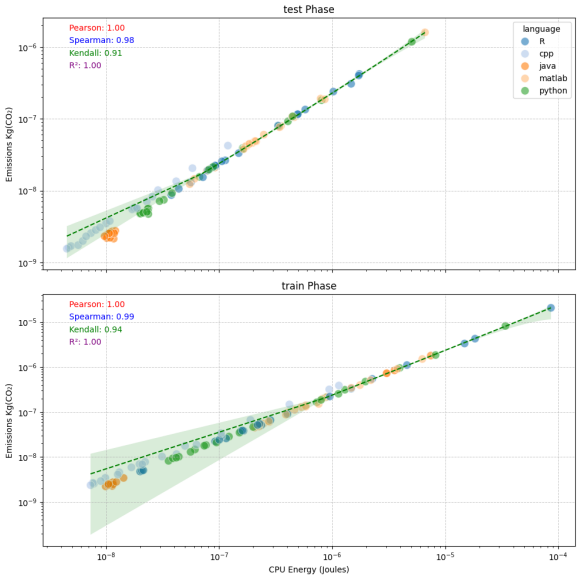

Figure 15: Correlation between emissions and CPU energy

Almost a perfect correlation with these 2 variables.

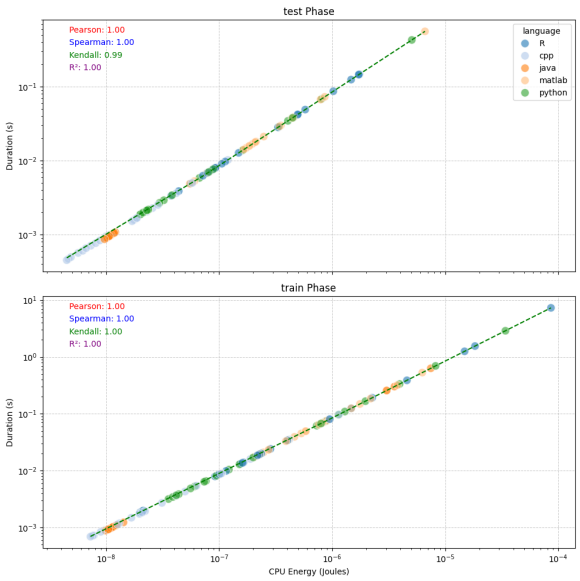

Figure 16: Correlation between time and CPU energy

Even better correlation was observed with time and emissions. With this result, we can assert that duration, emissions, and CPU energy follow almost the same pattern. However, it is worth noting that the correlation measures we employed earlier did not yield additional insights, as the relationships remained predominantly linear, with correlation coefficients close to 1. This reinforces the idea that the three variables are closely interconnected and behave similarly in relation to each other.

6 Tables

Table 1 summarizes the average performance metrics by language and phase. It is evident that C++ and Java consistently outperform other languages across various metrics. For instance, C++ shows the lowest duration and energy consumption in the test phase, and Java exhibits minimal emissions and emissions rate in both test and train phases. These results indicate that C++ and Java are superior in terms of performance efficiency.

Similarly, Table 2 highlights the performance metrics for different algorithms. KNN and decision tree algorithms generally demonstrate the best performance, with lower duration, energy consumption, and emissions compared to other algorithms. This trend reinforces the efficiency of KNN and decision tree algorithms across the evaluated metrics.

In summary, both C++ and Java emerge as the most efficient languages, while KNN and decision tree are the most effective algorithms based on the observed performance metrics.

7 Real Case Scenario: Inference Time Analysis

In this section, we assess the comparative performance of machine learning algorithms in Python and C++ by evaluating inference times for both simple and complex models. Although simpler models, such as Logistic Regression, are generally faster to execute than more complex ones like Decision Trees, we consistently find that C++ outperforms Python across all levels of model complexity.

To deepen our understanding of the parameters, we also considered the time taken for each model. Since time and energy consumption have a linear relationship, this approach allows us to draw conclusions about energy efficiency without directly measuring energy usage. Thus, the results regarding performance and efficiency remain consistent, reinforcing the insights gained from analyzing inference times.

7.1 Inference Time Comparison

We measure the time required to make a single inference using the Wine dataset for both Logistic Regression and Decision Tree models. Here are the results for both C++ and Python implementations:

Logistic Regression in C++: 6.03×10^{-4} seconds per inference.
Logistic Regression in Python: 2.17×10^{-3} seconds per inference.
Decision Tree in C++: 9.95×10^{-4} seconds per inference.
Decision Tree in Python: 2.69×10^{-3} seconds per inference.

Despite Logistic Regression being easier to run due to its simpler structure, C++ still outperforms Python, providing faster inference times for both algorithms. This advantage remains even when moving to the more computationally intensive Decision Tree model.

7.2 Requests Processed per Day

To further illustrate the performance difference, we calculate how many requests each model can handle within 24 hours, based on their respective inference times.

Number of Requests (C++ Logistic Regression):

$$\text{NR}_{\text{C++}} = \frac{24 \times 60 \times 60}{0.000603} \approx 143{,}454{,}125$$

Number of Requests (Python Logistic Regression):

$$\text{NR}_{\text{Python}} = \frac{24 \times 60 \times 60}{0.002174} \approx 39{,}358{,}681$$

Number of Requests (C++ Decision Tree):

$$\text{NR}_{\text{C++}} = \frac{24 \times 60 \times 60}{0.000995} \approx 86{,}406{,}251$$

Table 1: Average performance summary by language and phase . Data are shown by mean due to uniform distribution in same language.

Language	Phase	Duration (s)	Energy Consumed (J)	Emissions (kg CO_2)	Emissions Rate	CPU Energy (J)
R	Test	0.0363	4.57×10^{-7}	9.90×10^{-8}	2.73×10^{-6}	4.27×10^{-7}
R	Train	0.5281	6.67×10^{-6}	1.49×10^{-6}	2.73×10^{-6}	6.23×10^{-6}
C++	Test	**0.0019**	**2.26** $\times 10^{-8}$	**7.19** $\times 10^{-9}$	3.67×10^{-6}	**2.13** $\times 10^{-8}$
C++	Train	0.0131	1.65×10^{-7}	5.22×10^{-8}	3.83×10^{-6}	1.54×10^{-7}
Java	Test	**0.0010**	**1.15** $\times 10^{-8}$	**2.45** $\times 10^{-9}$	2.57×10^{-6}	**1.09** $\times 10^{-8}$
Java	Train	0.0575	7.26×10^{-7}	1.63×10^{-7}	2.61×10^{-6}	6.79×10^{-7}
Matlab	Test	0.0490	6.17×10^{-7}	1.37×10^{-7}	2.74×10^{-6}	5.77×10^{-7}
Matlab	Train	0.1026	1.30×10^{-6}	2.87×10^{-7}	2.73×10^{-6}	1.21×10^{-6}
Python	Test	0.0320	4.03×10^{-7}	8.79×10^{-8}	2.64×10^{-6}	3.77×10^{-7}
Python	Train	0.2182	2.76×10^{-6}	6.15×10^{-7}	2.74×10^{-6}	2.57×10^{-6}

Table 2: Average Performance Summary by Algorithm and Phase.

Algorithm	Phase	Duration (s)	Energy Consumed (J)	Emissions (kg CO_2)	Emissions Rate	CPU Energy (J)
KNN	Test	0.0192	2.41×10^{-7}	5.32×10^{-8}	2.94×10^{-6}	2.26×10^{-7}
KNN	Train	**0.0058**	**7.13** $\times 10^{-8}$	1.52×10^{-8}	2.81×10^{-6}	**6.68** $\times 10^{-8}$
SVC	Test	0.0418	5.27×10^{-7}	1.13×10^{-7}	2.83×10^{-6}	4.92×10^{-7}
SVC	Train	0.2980	3.77×10^{-6}	8.37×10^{-7}	2.99×10^{-6}	3.52×10^{-6}
adaBoost	Test	0.0214	2.69×10^{-7}	5.99×10^{-8}	2.90×10^{-6}	2.52×10^{-7}
adaBoost	Train	0.0820	1.04×10^{-6}	2.34×10^{-7}	2.98×10^{-6}	9.67×10^{-7}
decisionTree	Test	**0.0035**	**4.26** $\times 10^{-8}$	**9.48** $\times 10^{-9}$	2.81×10^{-6}	**4.01** $\times 10^{-8}$
decisionTree	Train	0.0279	3.51×10^{-7}	8.30×10^{-8}	2.91×10^{-6}	3.28×10^{-7}
logReg	Test	**0.0075**	**9.29** $\times 10^{-8}$	2.06×10^{-8}	2.79×10^{-6}	**8.70** $\times 10^{-8}$
logReg	Train	0.5843	7.38×10^{-6}	1.67×10^{-6}	2.93×10^{-6}	6.90×10^{-6}
naiveBayes	Test	0.0570	7.20×10^{-7}	1.62×10^{-7}	2.89×10^{-6}	6.72×10^{-7}
naiveBayes	Train	0.0153	1.92×10^{-7}	4.11×10^{-8}	2.88×10^{-6}	1.80×10^{-7}
randomForest	Test	0.0178	2.23×10^{-7}	4.89×10^{-8}	2.93×10^{-6}	2.09×10^{-7}
randomForest	Train	0.2741	3.46×10^{-6}	7.70×10^{-7}	2.99×10^{-6}	3.24×10^{-6}

Number of Requests (Python Decision Tree):

$$\text{NR}_{\text{Python}} = \frac{24 \times 60 \times 60}{0.00269} \approx 31{,}958{,}577$$

7.3 Performance Gap and Conclusion

Even though Logistic Regression is inherently faster and simpler to run than Decision Tree, C++ offers a significant advantage in inference speed, consistently handling more requests per day than Python even if we use a more demanding model for this specific task as the Decision Tree.

Percentage Increase in Requests for Logistic Regression:

$$\text{PI} = \left(\frac{143{,}454{,}125 - 39{,}358{,}681}{39{,}358{,}681} \right) \times 100 \approx 264.56\%$$

The performance gap becomes even more pronounced with the Decision Tree model. Despite its complexity, C++ handles nearly three times as many requests as Python.

This comparison highlights that while simpler algorithms like Logistic Regression are faster to execute, C++ remains ahead of Python in both simple and complex cases, demonstrating its consistent efficiency.

8 Conclusion

This study provides valuable insights into the energy and environmental costs associated with machine learning across various algorithms and programming languages. While Python is predominantly used not only for prototyping but also in production environments, this preference stems from several key factors. Python offers rapid prototyping capabilities and ease of implementation, which allow developers to quickly iterate and test models. Its extensive libraries and frameworks, such as TensorFlow and PyTorch, further enhance productivity and streamline the development process.

However, this convenience comes with trade-offs. Languages like C++ may provide higher execution efficiency and better performance, particularly for resource-intensive tasks, but they often require more complex code and a steeper learning curve. Consequently, while Python may be less efficient in terms of execution speed and energy consumption, its accessibility and versatility make it the language of choice for many machine learning practitioners.

Building more sustainable AI systems will necessitate a deeper awareness of these trade-offs, especially as machine learning continues to permeate various industries.

Developers and organizations must balance the benefits of rapid development with the need for energy efficiency and sustainability.

To enhance the scope and accuracy of this study, future work could incorporate a broader range of machine learning models, especially in the context of deep learning frameworks, along with more diverse datasets. Additionally, placing a larger emphasis on cloud-based environments would provide critical insights into the long-term viability of machine learning models at scale, especially given the growing reliance on cloud infrastructure for large-scale computations. Understanding how these factors interplay will be essential for creating efficient and sustainable AI systems that can thrive in real-world applications.

Supplementary Material

The supplementary material for this study includes additional datasets, source code, and further details of the experimental setup along with additional tests and results on the data (sec B). These materials are provided to enhance the reproducibility and transparency of the findings(sec A).

A Reproducibility

This section details the steps necessary to reproduce the results of our experiments with various machine learning algorithms implemented in different programming languages. Each language and framework has its unique considerations for ensuring reproducibility which led to creating multiple environments.

Python excels in reproducibility due to its extensive ecosystem of libraries and tools. To replicate our experiments, create an isolated environment using ‘virtualenv‘ or ‘conda‘, and ensure the exact package versions are installed as specified in the ‘requirements.txt‘. It is important to note that when using hardware accelerators such as GPUs, driver versions and hardware configurations can influence performance metrics. For practically all languages, the following environment will be used; however, C++ was particular about which version of the Python libraries it needed.

MATLAB supports strong reproducibility through its integrated environment. All scripts and data files are bundled within MATLAB Live Scripts or saved as ‘.mat‘ files, ensuring that experiments can be rerun with the same data and code. MATLAB’s version control and toolboxes are specified to match those used in our experiments.

R offers good reproducibility with the use of ‘packrat‘ or ‘renv‘, which manage project environments and lock package versions. R scripts are accompanied by R Markdown files that document the code and results. Differences in system configurations and external dependencies can affect reproducibility; therefore, ensuring the same system setup and package versions as specified is crucial.

Java presents some challenges for reproducibility due to the manual configuration of environments and dependencies. However, using Maven can significantly enhance reproducibility by managing dependencies and build configurations. Maven’s ‘pom.xml‘ file allows for precise specification of library versions and build settings, which simplifies the replication of the experimental environment. Ensure that the same version of the Java Virtual Machine (JVM) and Maven are used, as differences in JVM versions or Maven configurations can affect performance. Detailed notes on system configuration, JVM version, and Maven build settings are provided in the GitHub Repository. Maven’s centralized dependency management helps mitigate issues related to external dependencies, making it easier to achieve consistent results across different setups.

C++ provides a low level of abstraction, which offers greater control but complicates reproducibility. Implementing machine learning algorithms in C++ involves manual configuration of libraries (e.g., OpenCV, MLPack, Armadillo) and compiler settings which are strictly bond to the machine being used. Detailed build scripts (e.g., ‘CMakeLists.txt‘) and documentation on library versions and hardware specifications are provided in the GitHub Repository to ensure the same build environment is used (On arm MacOS). Variations in hardware architecture, compilers, and compiler flags can significantly affect performance metrics, so adhering to the provided setup is crucial for replication. In addition to these difficulties, C++ necessitated the usage of a distinct Python environment with different versions of the previously utilized libraries.

By following the above guidelines and utilizing the code in the GitHub Repository, researchers can reproduce our experiments and verify the results accurately.

B Additional Results

B.1 Timeseries

We present one of the possible combinations of time series to observe how the same runs varied over time. Our goal was to determine whether runs conducted consecutively could influence subsequent runs, potentially affecting performance due to thermal throttling of the device, which does not have active cooling.

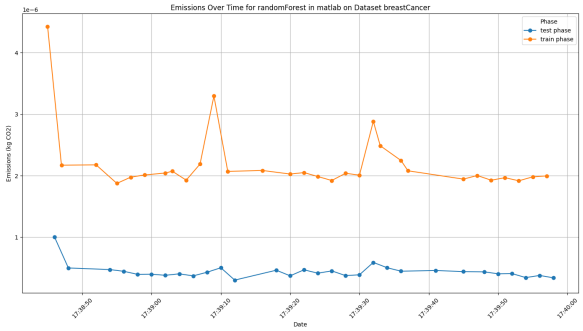

Figure 17: Time series in matlab during the 30 runs of breast cancer dataset

We observe that the execution time between the two phases remains consistent. We also present the time series analysis of emissions for different machine learning algorithms. The plot illustrates how emissions evolve over time for each algorithm.

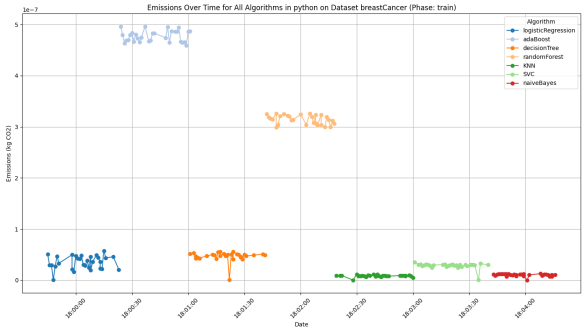

Figure 18: Time series in python during the 30 runs of breast cancer dataset for all algorithms

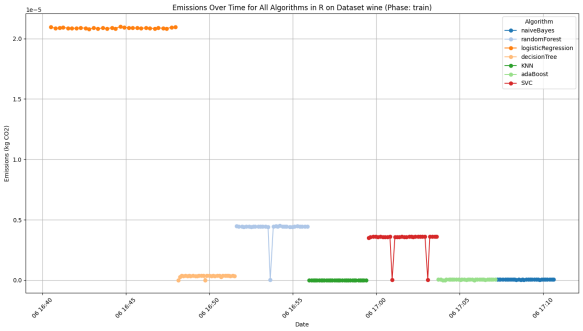

Figure 19: Time series in R during the 30 runs of wine dataset for all algorithms

The plots provide insights into the efficiency of each algorithm in terms of energy consumption during the training phase. By examining the emissions over time, we can compare the environmental impact of the algorithms and evaluate their performance in training scenarios. The visual representation helps in understanding how the energy footprint of each algorithm varies. The time series analysis of emissions for various machine learning algorithms reveals an important insight: the emissions remain relatively constant across different algorithms and do not significantly influence each other.

References

[1] Chen, B., Tarlow, D., Swersky, K., Maas, M., Heiber, P., Naik, A., Hashemi, M., & Ranganathan, P. (2022). Learning to Improve Code Efficiency. *ArXiv*

[2] Gao, K., Tu, J., Huo, Z., Mei, G., Piccialli, F., & Cuomo, S. (2020). Julia Language in Machine Learning: Algorithms, Applications, and Open Issues. *Comput. Sci. Rev.*, 37

[3] Aremu, & Salako. (2014). Efficiency Comparison of Some Selected Programming Languages. *Ilorin Journal of Science*.

[4] Mitra, Arjun and Thomas, Lily. *Energy-Efficient Algorithms for Machine Learning on Edge Devices*. IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 3, 2021, pp. 1102-1115.