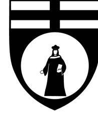
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Kernels for High Performance Distributed Computing

Gabriele Caletti

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/



Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

Kernels for High Performance Distributed Computing

Gabriele Caletti

Advisor: Daniele D'Agostino, Lorenzo Rosasco
Examiner: Matteo Dell'Amico

December, 2023

Table of Contents

Chapter 1 Introduction **6**

Chapter 2 Background **9**

2.1 Representer Theorem 9

2.2 Kernel Functions 10

2.3 Kernel Trick 11

Chapter 3 State of Art **13**

3.1 Scikit-learn 13

3.2 Kernelmethods 0.2 14

Chapter 4 Falkon Library **16**

4.1 Background 17

4.2 Falkon Pseudocode 19

Chapter 5 Divide and Conquer **21**

5.1 Empirical Motivations 22

5.2 Pseudocode 24

Chapter 6 Falkon DC **26**

6.1 FalkonDC Implementation Overview 26

6.1.1 Pseudocode 29

6.1.2 Conductor 31
6.2 How Split the Dataset 32
6.2.1 Naive Splitting 32
6.2.2 Split with Overlap 34
6.2.3 Partition not Split 34
6.3 The Preconditioner 36

Chapter 7 Tests and Results **39**
7.1 Experimental Environment and Settings 40
7.2 FalkonDC Performance 40
7.2.1 Execution Time 41
7.2.2 Accuracy 43
7.2.3 Time with n/m constant 43
7.2.4 Memory Usage 48
7.2.5 Speedup 48
7.3 Datasets Splitting Results 50
7.4 Preconditioner Results 52
7.5 Combinational Parameters 54

Chapter 8 Conclusions **56**
8.1 Future Works 57
8.2 Acknowledgements 59

Appendices **61**

Appendix A Docker Ecosystem Overview **62**
A.1 Docker Components and Tools 62
A.2 Docker's Advantages 64

Appendix B Falkon Docker Image **66**

B.1 Download the Falkon Docker Image 66
B.2 Run a Falkon Container 66
B.3 Create a Distributed Falkon Model with Majority Voting 67

Appendix C Speedup and Efficiency **68**

Appendix D Other Experiments **69**
D.1 Synthetic Binary Classification Dataset 69
D.2 Airlines Regression Dataset 72
D.3 Airlines Binary Classification Dataset 75
D.4 Airlines Multiclass Classification Dataset 78

Bibliography **81**

Chapter 1

Introduction

In recent years, Kernel Methods have emerged as a dominant and transformative paradigm in the field of machine learning, playing a pivotal role in addressing intricate problems characterized by non-linear data relationships. At the heart of Kernel Methods lies a fundamental concept: the transformation of data into a higher-dimensional feature space, enabling linear algorithms to operate with remarkable efficiency and effectiveness. What sets Kernel Methods apart from traditional explicit feature engineering is their innate ability to implicitly capture intricate patterns and latent structures within data, all thanks to the ingenious utilization of kernel functions.

Figure 1.1 serves as an illustrative example of how the application of a kernel can facilitate the discovery of an effective linear separator for two distinct classes, represented by green circles and red squares. In the source space of the problem, identifying a suitable linear separator, such as a straight line minimizing classification errors for both classes, proves impossible. All potential straight lines can be categorized into two groups: those that bisect the dataset, akin to flipping a coin, and those that align with one end of the subset of red squares, correctly classifying them but leading to the misclassification of many circles as squares. Therefore, the application of a kernel proves to be a superior choice for achieving a robust classifier.

One distinguishing feature of Kernel Methods is the concept of a feature map within an infinite-dimensional space. Intriguingly, these methods only require a finite-dimensional matrix as input, in accordance with the Representer theorem. At the core of Kernel Methods lies the ingenious utilization of kernel functions, empowering them to function within a high-dimensional and implicit feature space without the need for explicit computation of data point coordinates within that space. Instead, they rely on calculating inner products between pairs of data points in the feature space, a technique frequently more computa-

tionally efficient than explicit coordinate computation. This innovative approach is aptly known as the Kernel trick.

Regrettably, the execution of kernel-based methods is frequently hindered, even with datasets of moderate size, owing to prolonged execution times or resource limitations such as insufficient RAM. Addressing this challenge, my implementation called FalkonDC provides a remedy by introducing a distributed solution tailored for high-performance computing and cloud architectures. This approach aims to mitigate the computational constraints associated with kernel methods, ensuring scalability and efficiency.

This thesis is meticulously organized to establish the theoretical foundation necessary for comprehending the functioning of kernels, as elucidated in Chapter 2. Following this, Chapter 3 provides an overview of the current state of the art in kernel methods, highlighting the most prominent and extensively utilized libraries in the field.

Chapter 4 serves as a pivotal juncture where the Falkon library [Fal], forming the basis of my work, is introduced. Motivated by the factors expounded upon in Chapter 5, inspired by the article *Divide and Conquer* [ZDW14], I proceed to implement a series of solutions aimed at enhancing the performance of Falkon and adapting it for deployment on distributed High-Performance architectures. Chapters 6 and 7 then delve into comprehensive explanations of the implemented solutions and present the results of the conducted tests, shedding light on the discernible impact in terms of acceleration and other noteworthy aspects of these implementations.

This detailed exploration not only contributes to an understanding of the optimizations made to Falkon but also underscores the substantial potential that kernel methods possess in addressing a diverse array of real-world challenges. Tasks such as classification, regression, and beyond are within the purview of this research, emphasizing the broad applicability of kernel methods in tackling a multitude of computational problems.

In summary, Kernel Methods represent a groundbreaking approach within the realm of machine learning, offering a potent means to address complex problems by harnessing the capabilities of high-dimensional feature spaces and kernel functions. Their adaptability and ability to unveil concealed patterns render them invaluable tools in a myriad of real-world applications.

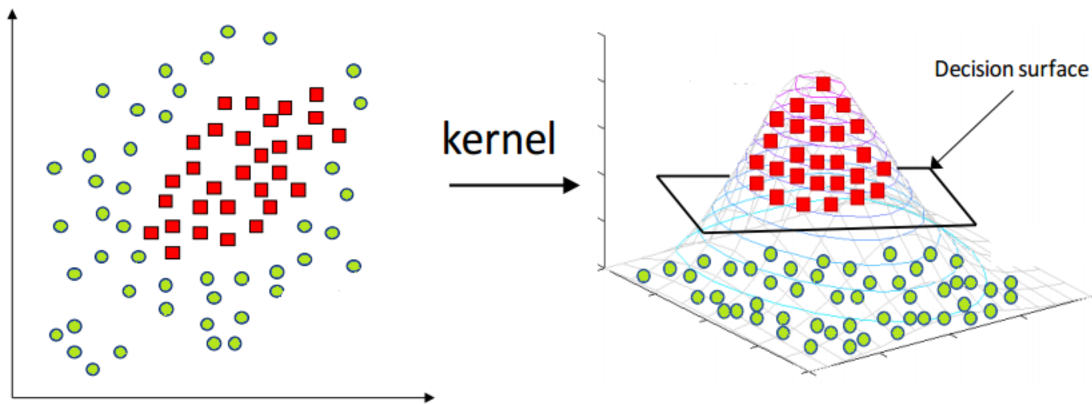


Figure 1.1: Visual representation of how the application of a kernel makes it easy to solve a classification problem that cannot be solved linearly in the source space.

Chapter 2

Background

This chapter delves into three fundamental components that shape the landscape of kernel methods: the Representative Theorem, Kernel Functions and the Kernel Trick. This brief introduction to the foundations of kernel methods is important in order to better understand the solutions adopted by Falkon and to fully comprehend the results that emerged from the tests I carried out.

2.1 Representer Theorem

Kernels stand as a fundamental concept in the realm of machine learning, assuming a central role in a variety of algorithms, particularly within the context of support vector machines (SVMs) and kernel methods. Essentially, kernels are mathematical functions that facilitate the implicit transformation of data into a higher-dimensional feature space. This transformation is realized through the computation of inner products between data points, effectively capturing intricate relationships and patterns that may remain hidden within the original input space. A cornerstone theorem in the field of kernel methods is the Representer Theorem. This theorem posits that for any optimization problem involving regularized empirical risk minimization, the solution can be expressed as a linear combination of the images of the training data points in the feature space. In mathematical terms, this can be represented as:

$$f(x) = \sum a_i * K(x_i, x) \tag{2.1}$$

Here, $f(x)$ represents the decision function, x_i denotes the training data points, a_i are the weight coefficients, and $K(x_i, x)$ signifies the kernel function. This theorem significantly simplifies the optimization problem, reducing it from potentially infinite dimensions to the

number of training examples.

Moreover, the Kernel Trick serves as an ingenious computational strategy that leverages the properties of kernels. Rather than explicitly computing the transformation into the high-dimensional feature space, it directly calculates the inner product between data points in the input space. This approach offers computational efficiency and enables the application of kernel methods even in cases where the feature space is of infinite dimension. In combination, kernels, the Representer Theorem, and the Kernel Trick comprise a potent toolbox within the realm of machine learning, affording us the means to address complex problems by effectively capturing non-linear data relationships while maintaining computational efficiency.

2.2 Kernel Functions

Kernel functions are a crucial component of Kernel Methods, as the name suggests. Here are some of the most common kernel functions, along with their descriptions:

1. Linear Kernel (Identity Kernel)
   - Formula: $K(x, y) = x^T * y$
   - Description: The linear kernel is the simplest and most basic kernel. It computes the inner product of the input vectors, effectively measuring the linear relationship between them. It's useful when data is already linearly separable in the feature space.

2. Polynomial Kernel:
   - Formula: $K(x, y) = (a * x^T * y + c)^d$
   - Description: The polynomial kernel is used to capture non-linear relationships by mapping the input data into a higher-dimensional space. The degree d controls the degree of the polynomial transformation, while c is a constant term. It is effective for non-linear classification problems.

3. Gaussian Kernel:
   - Formula: $K(x, y) = exp(-\gamma * ||x - y||^2)$
   - Description: The RBF kernel is one of the most commonly used kernels in various machine learning algorithms. It transforms data into an infinite-dimensional space. The parameter γ controls the shape of the decision boundary, with smaller values resulting in a smoother boundary and larger values making it more complex.

4. Sigmoid Kernel:
   - Formula: $K(x, y) = tanh(a * x^T * y + c)$
   - Description: The sigmoid kernel resembles the sigmoid activation function. It

maps data into a higher-dimensional space, and the parameters a and c control the steepness and shift of the sigmoid curve, respectively.

5. Laplacian Kernel:
   - Formula: $K(x, y) = exp(-a * ||x - y||)$
   - Description: The Laplacian kernel is similar to the gaussina kernel but has a different shape. It assigns more weight to data points that are closer to each other, making it robust to outliers. The parameter a controls the kernel's width.

6. Bessel Kernel:
   - Formula: $K(x, y) = J_v(\beta * ||x - y||)$
   - Description: The Bessel kernel employs Bessel functions to map data into a higher-dimensional space. It is often used in applications involving circular symmetry, such as image processing and signal analysis. The parameters β and v influence the shape of the kernel.

7. String Subsequence Kernel:
   - Formula: $K(x, y) = \sum \delta(x_i, y_i)$
   - Description: The string subsequence kernel measures the similarity between sequences, such as DNA sequences or text documents, by counting common subsequences. It is particularly useful in natural language processing and bioinformatics.

These are some of the most common kernel functions used in machine learning. Choosing the appropriate kernel function depends on the nature of the data and the specific problem being addressed, as different kernels have varying properties and suitability for different types of data and tasks.

2.3 Kernel Trick

The Kernel Trick stands as a fundamental technique in the field of machine learning, particularly in the domains of classification and regression, where it finds extensive use in Support Vector Machines and other kernel-based algorithms. This ingenious approach enables the transformation of a non-linear feature space into a higher-dimensional realm, where the data may become linearly separable. In practical terms, the Kernel Trick hinges on the definition of a kernel function, often denoted as "K," which computes the dot product between two vectors in the transformed feature space without the need to explicitly perform this transformation.

The application of the Kernel Trick yields distinct advantages by circumventing the explicit expansion of the high-dimensional feature space, resulting in savings in both time

and computational resources. Furthermore, it effectively addresses non-linear classification problems by facilitating the discovery of an optimal hyperplane that demarcates class boundaries in the transformed feature space. Nonetheless, the choice of the kernel and its associated parameters necessitates judicious consideration and experimentation, as these decisions can wield significant influence over the model's performance.

The essence of the Kernel Trick lies in its ability to implicitly map input data into a higher-dimensional space without explicitly computing the transformed feature vectors. Mathematically, this can be expressed as follows: let ϕ be the mapping function that transforms the input data x into a higher-dimensional space, and $K(x, x')$ be the kernel function representing the inner product of the transformed data points. The key insight is that, rather than explicitly computing $\phi(x)$ and $\phi(x')$ and then taking their inner product, one can directly compute $K(x, x')$ without explicitly knowing ϕ.

In summary, the Kernel Trick emerges as a potent tool that extends the modeling capabilities of SVM classifiers and other algorithms, empowering them to adeptly navigate and process complex and non-linear data with success.

Chapter 3

State of Art

Many popular machine learning libraries commonly implement kernel methods as presented in the previous chapter, which can lead to algorithmic complexity scaling with n, where n represents the number of samples in the dataset. This approach necessitates the computation of an nxn matrix, and when dealing with very large datasets (with n reaching 10^5 or higher), the associated costs in terms of memory usage and execution time can quickly become prohibitive for many hardware systems. As a result, this either restricts the utility of these libraries to datasets with lower dimensionality or compels practitioners to manually implement dimensionality reduction techniques in order to enhance performance while minimizing the trade-offs in terms of model accuracy.

In the subsequent sections, I delve into a more detailed analysis of the two most commonly used libraries for leveraging kernel methods.

3.1 Scikit-learn

Within the domain of kernel methods, scikit-learn provides an efficient implementation of Kernel Ridge Regression (KRR), which seamlessly integrates ridge regression with the kernel trick. This approach enables the learning of a linear function in a space defined by a user-selected kernel and the given dataset. When dealing with non-linear kernels, it extends its capabilities to learn complex non-linear functions within the original feature space. KRR shares a resemblance with Support Vector Regression (SVR) in terms of the resulting model but diverges in its choice of loss functions. While KRR opts for the squared error loss, SVR employs epsilon-insensitive loss, both complemented by L2 regularization. Notably, KRR stands out for its ability to achieve an efficient closed-form fitting process, making it highly suitable for medium-sized datasets compared to SVR. However, it's cru-

cial to acknowledge that the model learned through KRR tends to be non-sparse, which results in somewhat slower prediction times. On the other hand, SVR, particularly when configured for $\epsilon > 0$, yields a sparse model that significantly expedites predictions.

Examinations carried out in scikit-learn that KRR exhibits a speed advantage of approximately sevenfold over SVR when employed in tandem with grid-search. This efficiency enables KRR to effectively handle training sets of medium size. By medium size, scikitlearn's documentation means datasets on the order of 10^3, so significantly smaller than can be used with other solutions. This prevents the use of this technique in some use cases. In terms of prediction times, SVR consistently outperforms KRR across various training set sizes, primarily due to its sparse model. Also, scikit-learn does not natively allow sample reduction via a parameter to the model creation function. For more details I refer to the official Sciki-learn documentation [Sci]. In summary, scikit-learn's KRR implementation emerges as a highly efficient choice for kernel-based regression, particularly tailored for datasets of moderate size. Its adaptable nature allows users to choose from different loss functions to suit specific task requirements.

3.2 Kernelmethods 0.2

The Kernelmethods 0.2 library is a notable contribution to the realm of kernel methods in the Python-based machine learning ecosystem. While many existing libraries offer predefined kernels, Kernelmethods 0.2 sets itself apart by providing modular classes that empower users with the ability to compute various kernel functions on a given sample, offering greater flexibility. This library caters to a wide range of applications, acknowledging the diversity and performance demands that different tasks require. At its core, the KernelMatrix class serves as a pivotal bridge between input data and various kernel learning algorithms. Designed for usability and extensibility across different applications and data types, this class enables the application of basic kernels to generate KernelMatrices. Furthermore, Kernelmethods 0.2 offers a spectrum of kernel operations, including normalization, centering, product computation, alignment evaluation, linear combination, and ranking of kernel matrices based on multiple performance metrics.

To enhance the management of a large collection of kernels, the library introduces convenient classes like KernelSet and KernelBucket. These classes are particularly valuable in scenarios necessitating automatic kernel selection and optimization, as found in Multiple Kernel Learning (MKL) applications. Beyond the conventional numerical kernels like Gaussian and Polynomial, Kernelmethods 0.2 stands out for its flexibility. It facilitates the development of categorical, string, and graph kernels with an intuitive and highly-testable API. The library's scope extends to providing native implementations for non-numerical

kernels and aims to be a user-friendly and highly extensible framework for various input data types, including sequences, trees, and graphs, supported by data structures like pyradigm.

In addition to these foundational contributions, Kernelmethods 0.2 offers drop-in Estimator classes, such as KernelMachine, which harnesses the power of SVM. This seamless integration within the scikit-learn ecosystem ensures that users can leverage SVM for their tasks efficiently. Moreover, the library introduces the OptimalKernelSVR class, designed to identify the most optimal kernel function for a given sample and subsequently trains the SVM using the optimal kernel. This not only streamlines the process but also empowers users to achieve better performance and results. Overall, Kernelmethods 0.2 is a valuable asset in the domain of kernel methods, addressing the need for flexibility, extensibility, and ease of use, making it a significant advancement in the machine learning landscape. For more details I refer to the official Kernelmetohds 0.2 documentation [Ker].

Thus Kernelmethods 0.2 is certainly a far more powerful and comprehensive tool than Scikit-learn, and in combination with the latter it is possible to meet the needs of most use cases.

Chapter 4

Falkon Library

Falkon is a versatile library scikit-learn like, developed by the MALGA research group at the University of Genoa. Its primary aim, as written in [MCRR20], is to tackle the challenges posed by large datasets when working with kernel methods. Traditional implementations of kernel methods can become impractical when confronted with datasets exceeding a dimensionality greater than 10^5. This is primarily due to the substantial computational and memory demands they place. The matrix required in such scenarios grows to sizes that surpass the available RAM memory, even on exceptionally powerful machines, as it scales with $O(n^2)$.

To surmount these limitations, Falkon harnesses a spectrum of innovative dataset approximation and reduction techniques, including Cholesky Decomposition and Nyström methods. Furthermore, Falkon is meticulously optimized with CUDA, capitalizing on GPU acceleration for matrix operations. This optimization renders Falkon exceedingly well-suited for high-performance servers equipped with robust GPUs. It's worth noting that Falkon is not confined exclusively to Nvidia GPUs; it can also utilize Intel MKL on multicore/multi CPU machines lacking Nvidia graphics cards.

It's essential to highlight that Falkon has predominantly been tested on Linux systems, as noted by the authors in the documentation [Fal], and its compatibility with Windows and Mac OS machines is not guaranteed. To ensure cross-platform accessibility, I have crafted a Docker image that facilitates hardware virtualization. This Docker image empowers Falkon to function on virtually any machine, irrespective of the underlying operating system and hardware configuration. For more comprehensive information on Docker and its utilization, please refer to Appendix A. Additionally, for guidance on how to obtain and employ the Falkon Docker image, please consult Appendix B.

4.1 Background

Supervised learning is a foundational machine learning paradigm wherein an algorithm is tasked with the crucial role of learning to make predictions or decisions about new, unseen data by drawing insights from a curated training dataset. This training dataset is composed of input-output pairs where each input is associated with a corresponding label, and it serves as the foundation upon which the algorithm builds its predictive capabilities. The overarching objective in supervised learning is to craft a predictive function that effectively maps inputs to their respective labels, thereby enabling the algorithm to generalize its understanding and deliver accurate predictions for novel, unseen data points that were not part of the training process.

To achieve this, the algorithm embarks on a journey of learning, iteratively adjusting its internal parameters to optimize its predictive performance. A crucial component of this optimization is the minimization of what is often referred to as the "expected loss" or "cost." The expected loss quantifies how well the predictive function aligns with the ground truth labels in the training data. In essence, it measures the dissimilarity between the algorithm's predictions and the actual outcomes, and the learning process revolves around minimizing this discrepancy.

The pursuit of a "good" predictive function, denoted as 'f,' is thus characterized by its ability to navigate through the complex landscape of data and relationships within it. This function should not merely memorize the training examples but should distill the underlying patterns and regularities inherent in the data, allowing it to make accurate predictions when faced with previously unseen instances. In essence, the hallmark of a successful supervised learning algorithm is its capacity to strike a harmonious balance between overfitting (capturing noise in the training data) and underfitting (oversimplifying the relationships), ultimately yielding a predictive function that exhibits high accuracy and robust generalization capabilities.

In summary, supervised learning is a pivotal paradigm within machine learning, where algorithms are tasked with constructing predictive functions capable of translating input data into meaningful predictions. The journey involves optimizing the expected loss, leading to the creation of a powerful tool for solving diverse real-world problems across domains such as image recognition, natural language processing, and more.

Thus, a good function f must minimize the expected loss

$$L(f) = \int l(f(x), y) dp(x, y) \tag{4.1}$$

Equation 4.1 can be resolved using empirical risk minimization, which is based on the notion of substituting an empirical mean for the expectation. The search for a solution must be limited to an appropriate hypothesis space, e.g., for linear functions $f(x) = w^T x$, while kernel methods also consider a nonlinear feature map $f(x) = w^T \Phi(x)$. The Hilbert space of reproductive kernel is the set of functions H as previously established, and if we designate by $|| \ f \ ||_H$ its norm, then the regularized empirical risk minimization is given by

$$\hat{f}_\lambda = arg_{f \in H} \, min \frac{1}{n} \sum_{i=1}^{n} l(f(x_i), y_i) + \lambda \ || \ f \ ||_H^2 \tag{4.2}$$

where the penalty term $|| \ f \ ||_H$ is meant to prevent instabilities and $\lambda \geq 0$ is a hyper-parameter. Under basic assumption, for $\lambda = \mathcal{O}(1/\sqrt{n})$, it holds with high probability that

$$L(\hat{f}_\lambda) - inf_{f \in H} L(f) = \mathcal{O}(n^{-1/2}) \tag{4.3}$$

From the computational point of view, the key fact is that it is possible to compute a solution even if $\Phi(x)$ is an infinite feature vector, as long as the kernel $k(x, x') = \Phi(x)^T \Phi(x')$ can be computed. According to the represented theorem $\hat{f}_\lambda(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i)$ then the problem can be replaced with a finite dimensional problem on the coefficients. The solution depends on the loss considered, but in general it involves handling the matrix K_{nn} which becomes forbidding with n about 10^5. Using Nystrom approximation we obtain a function with the form

$$f(x) = \sum_{i=1}^{m} \alpha_i k(x,) \tag{4.4}$$

this approach brings an immediate advantage. If we consider the case where we use Square Loss, both loss and penalty are quadratic, so it all comes down to solving a linear system. In particular, we get

$$(K_{nm}^T K_{nm} + \lambda n K_{nm}) \alpha = K_{nm}^T y \tag{4.5}$$

The original problem can be solved in $O(n^3)$ time and $O(n^2)$ space while Equation 4.5 can be solved directly in $O(nm^2 + m^3)$ time and $O(m^2)$ space. Moreover, the convergence of the latter can be greatly improved by considering the preconditioner.

The naive preconditioner P for problem 4.5 is such that $PP^T = (K_{nm}^T K_{nm} + \lambda n K_{mm})^{-1}$ and costs computationally the same as the original problem, but we can use Nystrom again and get:

$$\tilde{P} \tilde{P}^T = (\frac{n}{m} K_{mm}^2 + \lambda n K_{mm})^{-1} \tag{4.6}$$

Then, we follow [ARR17] again and combine the precondition above with the conjugate gradient (CG) method. The pseudocode of the complete procedure is provided in Algorithm 4.2. With this approach, the total computational cost to achieve the optimal statistical bounds is $O(n\sqrt{n} \, log \, n)$ in time and $O(n)$ in memory, making it ideal for large-scale scenarios.

4.2 Falkon Pseudocode

The pseudo-code presented in Figure 4.2 provides a comprehensive insight into the inner workings of Falkon. It is crucial to grasp this section thoroughly since my subsequent contributions, discussed in Chapters 5 and 6, build upon this codebase and introduce optimizations to enhance Falkon's efficiency in specific scenarios.
In the next rows, the pseudo-code is dissected step by step, shedding light on its key components and algorithms.

1. The Falkon algorithm is defined as 4.5, so as a function that takes several input parameters:
   - X: Input data, typically a matrix of shape (n, d) where n is the number of samples, and d is the number of features.
   - y: Output labels or target values, typically a vector of shape $(n, 1)$.
   - λ, m, and t: Scalars representing various parameters for the algorithm.

2. Initialize X_m by randomly subsampling m data points from X using the *RANDOM-SUBSAMPLE* function.

3. Calculate two matrices T and A using the *PRECONDITIONER* function, where:
   - T is the Cholesky decomposition of K_{mm}, where K_{mm} is computed using the kernel function λ applied to the subsampled data X_m.
   - A is the Cholesky decomposition of K_{mm} after being updated with identity matrix I_m.

4. Define a function *LINOP(B)* that takes an input matrix B.

5. Calculate w as the product of the inverse of matrix A and matrix B.

6. Calculate c as the result of the kernel computation between X_m and X and use it in further matrix operations.

7. Return a matrix $A^{-T}T^{-T}c + \lambda nv$.

8. End the *LINOP* function.

9. Calculate R as the result of $A^{-T}T^{-T}k(X, X_m)y$.

10. Compute β using the *CONJUGATEGRADIENT* function with *LINOP* as the linear operator and the result R as the right-hand side, with a termination parameter t.

11. Return the result of $T^{-1}A^{-1}B$.

Algorithm 1 Pseudocode for the Falkon algorithm.

```
1: function FALKON(X ∈ ℝ^{n×d}, y ∈ ℝ^n, λ, m, t)
2:     X_m ← RANDOMSUBSAMPLE(X, m)
3:     T, A ← PRECONDITIONER(X_m, λ)
4:     function LINOP(β)
5:         v ← A^{-1}β
6:         c ← k(X_m, X)k(X, X_m)T^{-1}v
7:         return A^{-⊤}T^{-⊤}c + λnv
8:     end function
9:     R ← A^{-⊤}T^{-⊤}k(X, X_m)y
10:    β ← CONJUGATEGRADIENT(LINOP, R, t)
11:    return T^{-1}A^{-1}β
12: end function
13: function PRECONDITIONER(X_m ∈ ℝ^{m×d}, λ)
14:    K_mm ← k(X_m, X_m)
15:    T ← chol(K_mm)
16:    K_mm ← 1/m TT^⊤ + λI
17:    A ← chol(K_mm)
18:    return T, A
19: end function
```

12. End the Falkon algorithm.
13. Define a function *PRECONDITIONER* that takes in input, X_m and λ.
14. Calculate K_{mm} as the result of applying the kernel function k to the matrix X_m.
15. Compute the Cholesky decomposition of Kmm and store it in the matrix T.
16. Update Kmm with $1/mTT^T + \lambda I$
17. Compute another Cholesky decomposition of Kmm and store it in the matrix A.
18. Return two matrices, T and A, as the result of the *PRECONDITIONER* function.
19. End the *PRECONDITIONER* function.

It is important to pay particular attention to the Xm matrix and the preconditioner function, as they will be among the central elements of my modifications and analysis. In particular, m, as one can easily guess, is extremely influential on runtime, since Falkon unlike traditional kernel implementations, goes to compute an mxm matrix with $m << n$. This matrix constitutes the preconditioner as described by lines 2 and 3 of the pseudocode, so m affects execution time more than n.

Chapter 5

Divide and Conquer

In a typical implementation of Kernel Ridge Regression (KRR), a crucial step involves inverting the kernel matrix, a process that demands $O(n^3)$ time and consumes $O(n^2)$ memory. Such computational complexities can be prohibitive when dealing with large sample sizes, denoted as n. Consequently, various approximation methods have been devised to alleviate the computational burden associated with finding an exact minimizer. One approach within this family of techniques revolves around low-rank approximations of the kernel matrix. Notable examples include Kernel Principal Component Analysis (PCA), the Incomplete Cholesky Decomposition, or Nyström sampling. The Falkon library, in particular, incorporates a combination of these techniques to minimize the model training execution time effectively. These methods effectively reduce the time complexity to $O(dn^2)$ or $O(d^2n)$, where $d << n$ represents the preserved rank.

In the scientific paper referenced in [ZDW14], a decomposition-based approach is explored. This algorithm possesses an appealing simplicity: it randomly partitions the dataset of size n into p equally sized subsets and computes the Kernel Ridge Regression estimate $\hat{f}_i$ independently for each of the $i = 1, ..., p$ subsets, employing a carefully chosen regularization parameter. The estimates are subsequently averaged using Equation 5.1.

$$\bar{f} = \frac{1}{m} \sum_{i=1}^{m} \hat{f}_i. \tag{5.1}$$

This approach results in time and memory complexities scaling as $O(n^3/p^2)$ and $O(n^2/p^2)$, respectively.

The partitions consist of the same number of elements and each subset is disjointed from the others. This mode of splitting is the basic idea from which I started. Furthermore,

this approach naturally aligns with parallel and distributed computation paradigms, as it allows up to superlinear speedup with the utilization of p parallel processors. However, it necessitates communication of the function estimates from each processor.

One challenge when solving each of the sub-problems independently is the selection of the appropriate regularization parameter. Due to the infinite-dimensional nature of non-parametric problems, the choice of the regularization parameter must be made judiciously. An interesting consequence of the theoretical analysis is that, even though each partitioned sub-problem is based only on a fraction of n/p samples, it is essential to regularize the partitioned sub-problems as if they had access to all n samples. Consequently, from a local perspective, each sub-problem is under-regularized. This "under-regularization" keeps the bias of each local estimate minimal but can lead to a detrimental increase in variance. However, as proven, the m-fold averaging method mitigates the variance sufficiently to ensure that the resulting estimator $\bar{f}$ still attains the optimal convergence rate.

5.1 Empirical Motivations

In the scientific paper cited as [ZDW14], heuristics derived from theory are substantiated through practical experiments conducted in paragraph 5 of the article. Figure 5.1 illustrates a comparison of three implementations of Kernel Ridge Regression (KRR): the green line represents KRR with Random Feature Approximation, the blue line represents KRR with Nystrom Sampling, and the red line represents the Fast-KRR model. The graph demonstrates that both Fast-KRR and Nystrom outperform the model employing random feature approximation in terms of achieving a lower mean square error. However, it's worth noting that the Nystrom approach takes longer to train the model, whereas Fast-KRR significantly reduces the runtime. Therefore, the Divide and Conquer approach presented in the article proves to be exceptionally compelling, especially considering its relatively straightforward implementation.

Figure 5.2 compares the Fast-KRR model with a standard model (without any optimization) trained on a dataset size of $1/m$, equivalent to the amount of data used by a single process of the Fast-KRR model. The comparison reveals that as the number of partitions increases, the mean square error also rises in both cases. However, the error for the standard model increases more significantly and rapidly. This is primarily due to under-regularization when using a small dataset, which results in higher variance. In the case of Fast-KRR, this issue is mitigated by aggregating the results through majority voting among all trials. As a result, Fast-KRR achieves significantly lower error rates than KRR using only a fraction of the data. Furthermore, the averaging process stabilizes the estimators, with negligible standard deviations compared to standard KRR.

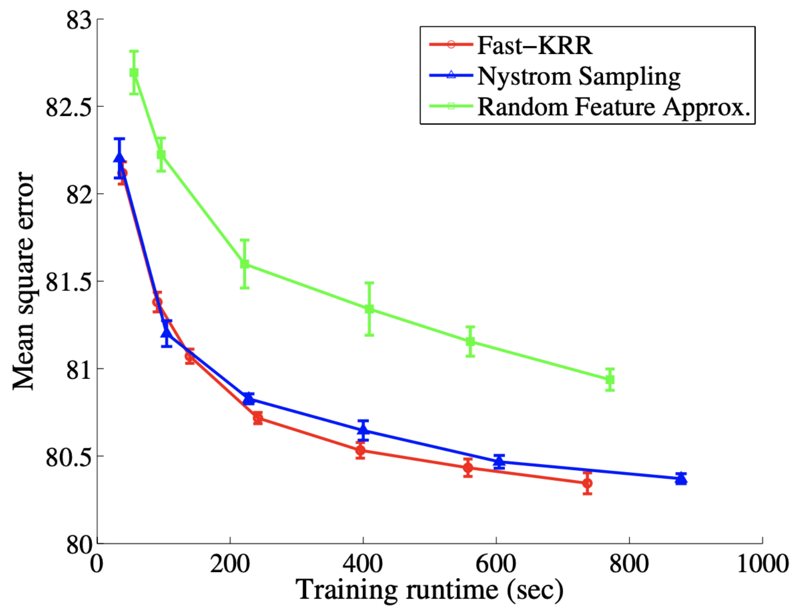


Figure 5.1: Results on year prediction on held-out test songs for Fast-KRR, Nystrom sampling, and random feature approximation. Error bars indicate standard deviations over ten experiments.

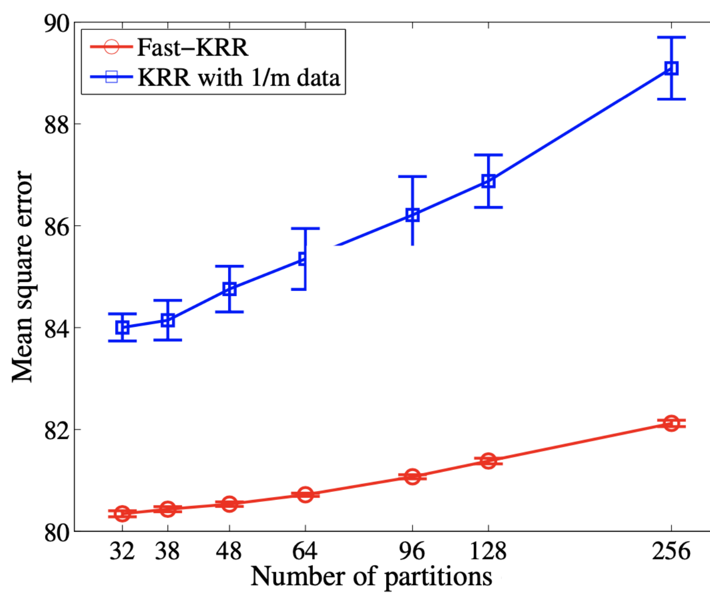


Figure 5.2: Comparison of the performance of Fast-KRR to a standard KRR estimator using a fraction $1/p$ of the data.

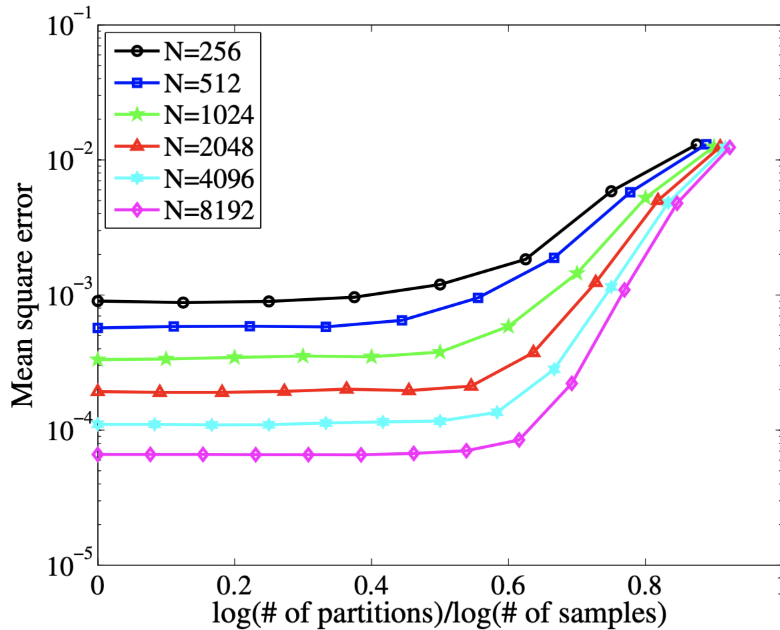


Figure 5.3: The mean-square error curves for fixed sample size but varied number of partitions. We are interested in the threshold of partitioning number m under which the optimal rate of convergence is achieved.

Additionally, it is valuable to explore the number of partitions, denoted as p, into which a dataset of size n can be divided while maintaining good statistical performance. According to Corollary 4 in the Divide and Conquer paper. [ZDW14] with $\nu = 1$, for the first-order Sobolev kernel, the performance degradation should be limited as long as $m \lesssim n^{1/3}$. To test this prediction, Figure 5.3 depicts the mean square error as a function of the ratio $log(p)/log(n)$. The theory predicts that even as the number of partitions p grows polynomially with n, the error should only increase beyond a certain constant value of $log(p)/log(n)$. As shown in Figure 5.3, the point at which the mean square error begins to increase appears to be around $log(p) \approx 0.45log(n)$ for reasonably large values of n. This empirical performance is slightly better than the threshold of (1/3) predicted by Corollary 4 but confirms that the number of partitions p can scale polynomially with n while maintaining minimax optimality.

5.2 Pseudocode

In this algorithm, the set of data samples is divided into disjoint subsets, each of which is processed independently. By computing local Kernel Ridge Regression (KRR) estimates

for each subset and then averaging them, the algorithm produces a global estimate, f, that captures the underlying patterns in the data. This approach not only simplifies the computation but also allows for parallel processing, making it a valuable tool for handling large datasets. Fast-KRR showcases the power of divide-and-conquer techniques in machine learning, offering an efficient way to perform regression tasks while maintaining accuracy and generalization.

The divide-and-conquer algorithm Fast-KRR is easy to describe.

1. Divide the set of samples $\{(x_1, y_1), ..., (x_n, y_n)\}$ evenly and uniformly at random into the m disjoint subsets $S_1, ..., S_p \subset XxR$, such that every subset contains n/p samples.
2. For each $i = 1, 2, ..., p$, compute the local KRR estimate $\hat{f}_i := argmin_{f \in H}\{\frac{1}{|S_i|}\sum_{(x,y)\in S_i}(f(x) - y)^2 + \lambda||f||_H^2\}$
3. Average together the local estimates and output $\bar{f} = \frac{1}{p}\sum_{i=1}^{m}\hat{f}_i$

In conclusion, the Fast-KRR algorithm offers a practical and scalable solution to regression problems by dividing the workload into smaller, manageable parts and then combining the results. This divide-and-conquer strategy is a testament to the adaptability of computer science techniques in tackling complex machine learning challenges, and it exemplifies the importance of optimizing computations for large-scale datasets. Fast-KRR stands as an insightful example of how algorithms can be designed to balance efficiency and accuracy in solving real-world problems.

Chapter 6

Falkon DC

In this chapter, I will elucidate the intricacies of FalkonDC's workflow and the decisions made during its implementation. Following a general overview of how FalkonDC operates in the next sub-section, I will delve into the examination of specific components within the implementation of this model.

By providing a comprehensive explanation of FalkonDC's underlying mechanics and my rationale for certain implementation choices, this chapter aims to offer a clear and insightful understanding of how the model functions and how it has been adapted for practical use. The subsequent analysis of key implementation aspects will shed light on the core components, their roles, and the considerations that influenced their design, ultimately contributing to a more in-depth comprehension of FalkonDC's inner workings.

6.1 FalkonDC Implementation Overview

FalkonDC is an extension of the Falkon library in Python, tailored for distributed architectures. Beyond capitalizing on existing optimizations like OMP and CUDA, and incorporating efficient algorithms such as Cholesky Decomposition and the Nyström method, FalkonDC introduces the capability to harness the power of clusters or multiple separate computers. This extension enables the concurrent execution of multiple machine learning models on different machines, resulting in substantial time savings while maintaining model accuracy. In contrast to a traditional Falkon-based model, FalkonDC combines the outputs of all models trained across the nodes through an averaging process for regression problems or a majority vote for classification problems. This approach enhances scalability and efficiency in distributed machine learning tasks.

When comparing Falkon to a kernel-based model provided by a commercial library like Scikit-learn, it's clear that Falkon holds a significant advantage in terms of algorithmic complexity. The traditional Falkon algorithm demonstrates a time complexity of $O(dn^2)$ and a space complexity of $O(d^2n)$, offering a considerable improvement over the $O(n^3)$ time complexity and $O(n^2)$ space complexity of conventional methods.
However, the innovation introduced by FalkonDC takes this reduction in complexity even further. With FalkonDC, the algorithmic complexity is further diminished, resulting in an impressive time complexity of $O(dn^2/p^2)$ and a space complexity of $O(d^2n/p^2)$. This is a noteworthy enhancement that not only maintains the efficiency of Falkon but also scales exceptionally well when dealing with large datasets preserving the accuracy of the model, making it a robust choice for machine learning tasks. The division of the algorithm into smaller, more manageable chunks (indicated by m) is a key factor contributing to this increased efficiency, ensuring that FalkonDC can process data with both speed and accuracy.

FalkonDC offers not only a remarkable speedup compared to its basic version but is also exceptionally well-suited for specific scenarios where dealing with a centralized dataset on a single machine doesn't make practical sense due to time, or privacy considerations. It addresses use cases where data distribution is inherent, and the owners of the data either cannot or should not share it.
Consider a real-world scenario involving a distributed dataset owned by multiple parties, such as a hospital. In this context, the privacy and confidentiality of personal data are of utmost importance. The data cannot be easily merged into a single location, and sharing it isn't a viable option.
FalkonDC presents an elegant solution for such scenarios because it allows for the creation and training of multiple models, one for each hospital's data. This approach circumvents issues related to personal data dissemination. Essentially, each hospital can utilize FalkonDC to independently train a model on its own data without compromising the privacy of the individuals in the dataset. However, by aggregating the results of all the models, the result we obtain is very close to the one we would have obtained using the complete dataset.
This not only simplifies the practical aspects of model training but also ensures that sensitive data remains secure, making FalkonDC a valuable tool in situations where data privacy and distribution are paramount concerns.

FalkonDC employs Docker containers for its implementation, providing several advantages. This approach simplifies the installation process on any machine, regardless of its underlying hardware and operating system. It ensures seamless compatibility and facilitates deployment, making FalkonDC accessible across diverse environments. Moreover, Docker containers offer the benefit of isolation, enhancing system security by creating a shielded environment.
At the heart of operationalizing the FalkonDC model lies the pivotal FalkonDC.sh file, serving as the central orchestrator for the entire algorithmic process. This file assumes

a critical role by providing a platform to configure all parameters and options associated with FalkonDC, thereby offering a high degree of flexibility to tailor the model to specific requirements. The key parameters are delineated below, elucidating their respective functionalities and highlighting the user-friendly nature of the configuration process. This adaptability not only streamlines the utilization of the FalkonDC algorithm but also underscores its versatility in accommodating diverse modeling needs.

FalkonDC parameters:

1. **P (Number of Nodes):** This parameter allows you to specify the number of nodes or machines on which the model will be deployed. The choice of P influences the size of the data subsets assigned to each node, impacting parallel processing.

2. **Mode (Dataset Partitioning Algorithm):** Mode is a crucial parameter that permits you to select the algorithm for dividing the dataset into subsets. The available modes for dataset partitioning will be elaborated on in the following section, offering options to cater to different dataset structures.

3. **Overlap:** Overlap parameter enables you to introduce a degree of overlap between the data subsets. Overlapping subsets can be useful in specific scenarios and can be customized based on requirements.

4. **N (Total Subset Size):** 'N' allows you to define the total number of elements within each data subset, thereby controlling the granularity of the partitioned data.

5. **Model (Machine Learning Algorithm):** The Model parameter empowers you to select and switch between various machine learning algorithms to align with your specific use case. Options may include Kernel Ridge Regression, Logistic Kernel, and others, ensuring adaptability to diverse modeling requirements.

6. **Type (Regression/Binary Classification/Multi Class Classification):** The Type parameter serves as a crucial indicator for the system, informing it whether the model is intended for regression or classification tasks. This distinction is vital in configuring the model correctly to suit the target problem.

Further insight into the functioning of FalkonDC's code, particularly the script responsible for orchestrating processes, is elaborated upon in the subsequent sections. These detailed discussions aim to provide a comprehensive understanding of the algorithm's operational intricacies, shedding light on the script's functionality and its role in ensuring the seamless execution of FalkonDC.

In summary, FalkonDC leverages Docker containers for cross-platform compatibility and enhanced security. The FalkonDC.sh script streamlines the entire process by segmenting data (when necessary), distributing the workload across machines, and consolidating results to evaluate the model's accuracy, making it a versatile and efficient tool for large-scale distributed machine learning tasks.

6.1.1 Pseudocode

The pseudocode underlying FalkonDC is straightforward to articulate and can be summarized as follows:

1. Divide the set of samples into p parts using a partitioning criterion of your choice. Then you get p train sets of the same size and a single test set.
   - Splitting
   - Partitioning
2. For each training subset p, train a model based on Falkon, so execute p time the algorithm 4.2.
   - If the *GlobalPreconditioner* option is turned on it trains each model with its respective subset of data but the preconditioner is calculated on the entire original data set.
   - Otherwise calculate the preconditioner on the same training dataset.
3. Aggregates the outputs of all P models.
   - If it is a classification problem it calculates the output of the FalkonDC algorithm through a majority vote.
   - If it is a regrassion problem it calculates the output of the FalkonDC algorithm by averaging between the results of each model.

To better understand how FalkonDC works, I made a diagram 6.1 that allows you to visualize the various components of the model and also allows you to understand how I used docker to facilitate the management and deployment of my code. The countless benefits of docker, but of containerization in general, are discussed in more detail in Appendix A.

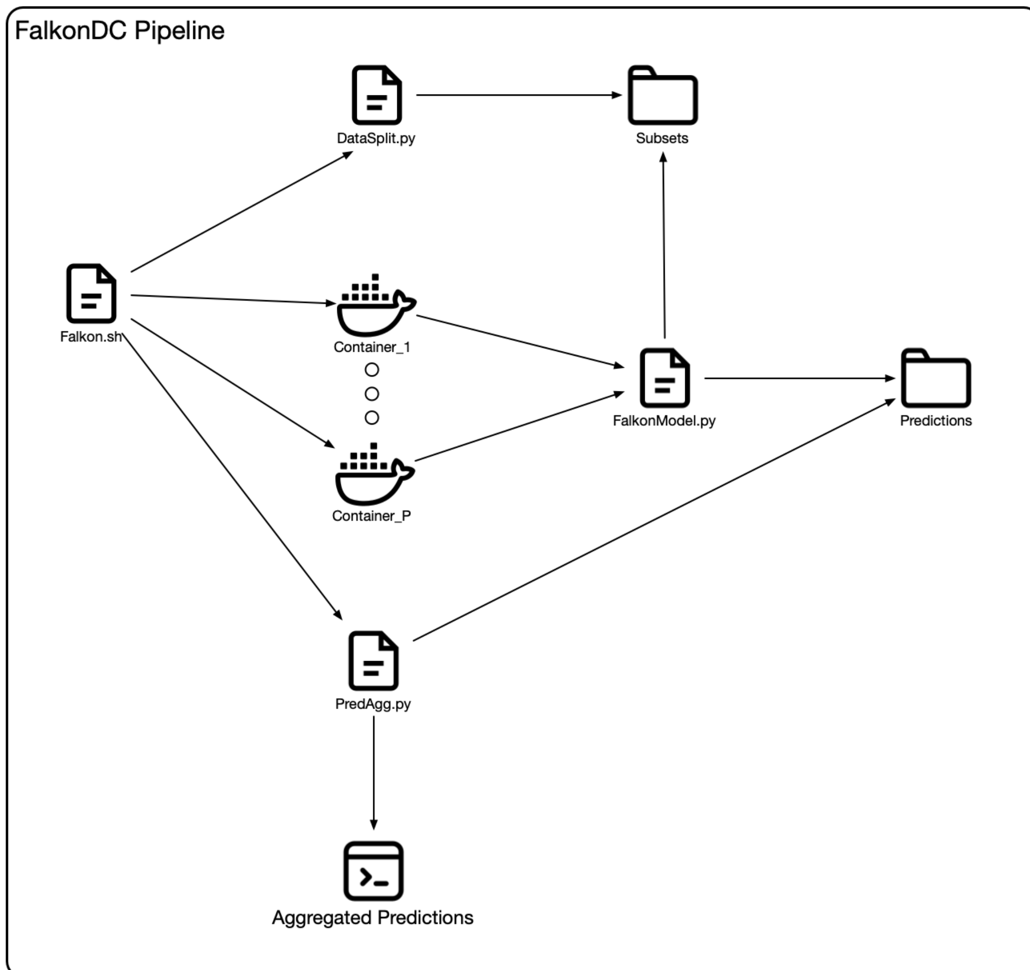


Figure 6.1: FalkonDC pipeline, is the graphical representation of what is instantiated when the bash script Falkon.sh is executed.

6.1.2 Conductor

The pivotal Falkon.sh file, illustrated on the left in Image 6.1, serves as the orchestrator for the comprehensive management and organization of FalkonDC. At the script's inception, users have the flexibility to fine-tune the algorithm's behavior by adjusting a set of variables. Additionally, the script mandates the specification of the dataset path, yet to undergo segmentation.

The subsequent segment of the script is dedicated to executing a container responsible for partitioning the dataset and generating individual p files, each tailored for a specific node. All resulting files are meticulously organized within the *Subsets* folder, facilitating subsequent model training to selectively access the desired dataset components.

The third segment of the code orchestrates the instantiation of Docker containers across all nodes, each equipped with its subset file and a standardized test set shared among all nodes. Post-training, each container independently evaluates the model on the collective test set, producing an output file conveniently located in the shared *Predictions* folder.

The fourth component of the script initiates another container tasked with aggregating results and computing key metrics, such as accuracy and RMSE. This container, synchronized with the Result folder, houses Python code capable of amalgamating results across files through either a majority vote for classification problems or an average for regression problems. The consolidated result is then presented in the terminal upon execution of the bash file.

Concluding the script, the last segment takes charge of a meticulous cleanup, systematically removing superfluous files that are no longer essential to the algorithm's execution. This sophisticated script encapsulates the entire lifecycle of FalkonDC, from data preparation to distributed model training, aggregation, and result analysis, underscoring its comprehensive utility in handling complex machine learning workflows.

Image 6.1 provides a visual representation that aids in comprehending the structure and functioning of FalkonDC, offering a clear and intuitive overview of the system's operation. For a more profound comprehension of the FalkonDC algorithm, it is recommended to invest time in immersing oneself in the code provided in the repository FalkonDC, accessible at the following address. This hands-on exploration serves as a valuable opportunity to gain firsthand insight into the intricacies of the algorithm's implementation, fostering a deeper understanding of its functionalities and nuances.

6.2 How Split the Dataset

The dataset splitting phase constitutes the initial and essential step in the execution of FalkonDC. There are scenarios where data splitting is implicit, such as when multiple entities wish to collaborate on training a shared model without openly sharing their raw data. In these situations, FalkonDC can be employed to allow each entity to train a model on their individual data, with the aggregated results leading to the creation of a much more potent distributed model. In cases like the one described, data splitting across p nodes is a implicit procedure.

However, even in cases where a cluster comprises p machines and a single dataset, it's crucial to explore how dividing the data and training each node can impact the model's performance. This impact can manifest in terms of accuracy and execution speed. The effectiveness of the data partitioning method plays a pivotal role in achieving the desired results. Consequently, it is imperative to carefully consider the data splitting strategy based on the specific use case and the available computational resources.

In the subsequent sub-sections, I will delve into the implementation of three distinct methods for splitting datasets. Moreover, in the following chapter, I will conduct a thorough analysis of the performance of models trained using these techniques. This analysis will serve the purpose of determining the optimal methodology based on the context of usage, shedding light on which data splitting approach is most suitable for a given scenario. By doing so, we aim to provide practical insights into optimizing the use of FalkonDC within distributed computing environments.

6.2.1 Naive Splitting

One approach to partitioning the dataset involves a straightforward random division into equal parts, ensuring that each element of the dataset exclusively belongs to one of the subsets. In terms of implementation, I employed the $df.sample()$ method from the Pandas library to accomplish this task. This method facilitates dataset shuffling, enabling the subsequent selection of the first m samples for the initial subset, where m equals n/p. This process is repeated for each subset, iterating p times, with p representing the desired number nodes to be utilized.

This method of dataset division is characterized by its simplicity in implementation and swift execution. It provides an efficient means of creating disjoint subsets from the dataset, making it suitable for various parallel processing scenarios. However, it's essential to note that this random division approach may not optimize data distribution based on inherent patterns or structures within the dataset.

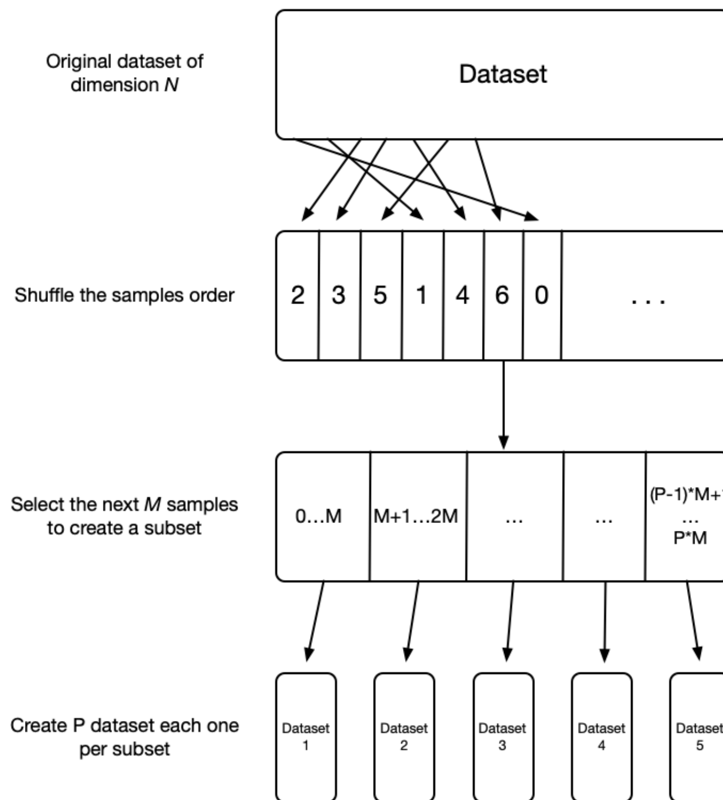


Figure 6.2: Naive Split Algorithm

Figure 6.2 shows the splitting procedure, which I call *naive*, selectable in FalkonDC from the configuration file by setting the $MODE = -s$ parameter. The picture shows perfectly all the steps to obtain p disjoint subsets from one large initial dataset.

6.2.2 Split with Overlap

A variation of the dataset partitioning method described in the previous paragraph involves introducing overlap between subsets. Unlike the previous method, each element of the original dataset can be part of multiple subsets, but it must be part of at least one subset. From an implementation perspective, after shuffling the dataset, it is divided into p parts, each of size $m+2o$, where o equals the number of elements shared with the previous and next block. This approach to dataset division remains relatively straightforward to implement, albeit slightly more time-consuming than the method outlined in the preceding paragraph.

One notable advantage of this approach is its ability to generate p subsets with a significantly large value of p, while maintaining subsets of size $m + 2o$ that are sufficiently substantial. This can be particularly advantageous when dealing with datasets that may not be exceptionally large. However, it's essential to note that this advantage may not be fully leveraged in use cases where Falkon is employed, as the dataset dimensionality typically ranges on the order of $10^5/10^6$ or even higher.

While introducing overlap between subsets can enhance the flexibility of data distribution, it's important to carefully consider its applicability based on the specific dataset and computational resources at hand. In scenarios with extremely high-dimensional datasets, alternative partitioning strategies or optimizations may be more suitable to exploit the full potential of Falkon and achieve efficient processing.

Figure 6.3 shows the partitioning procedure with overlap. This mode can be set from the FalkonDC configuration file, by putting the parameter $MODE = -s$ and $O = o$ with o small being the overlap value between the chunks, so it must be an integer between 0 and n.

6.2.3 Partition not Split

Building upon the simpler implementations elucidated in the preceding paragraphs, a notable evolution in the algorithm is introduced, inspired by insights gleaned from the article [CVCR21]. The decision to harness the capabilities of a clustering algorithm for strate-

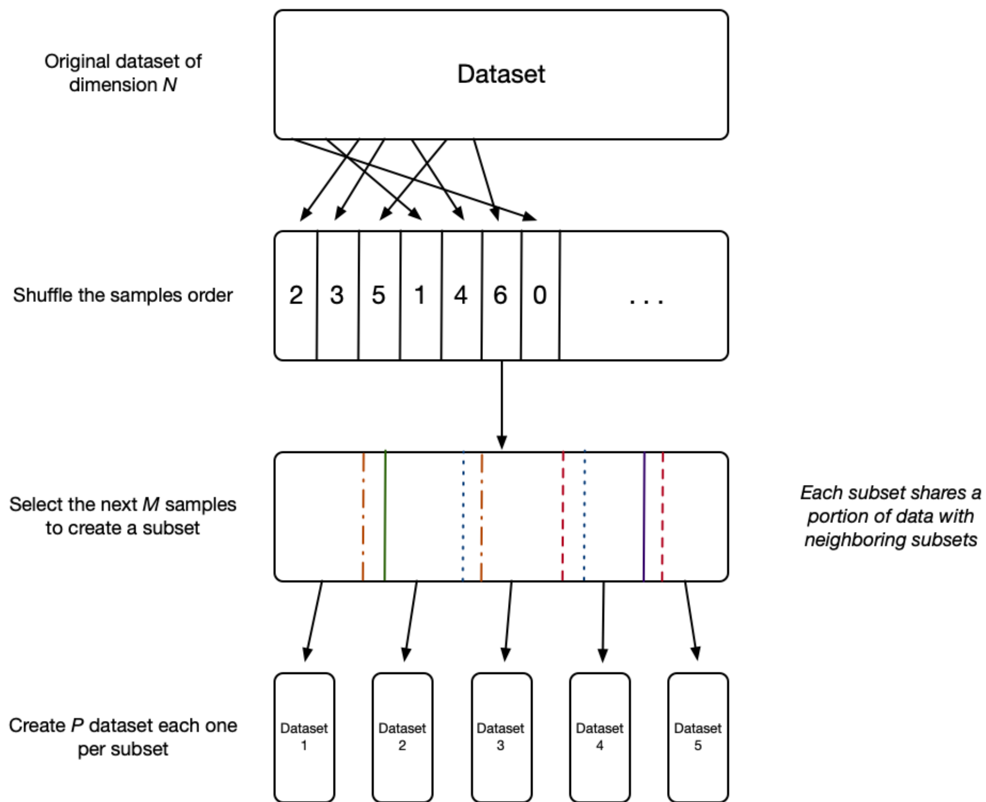


Figure 6.3: Naive Split Algorithm with overlap between neighboring subsets

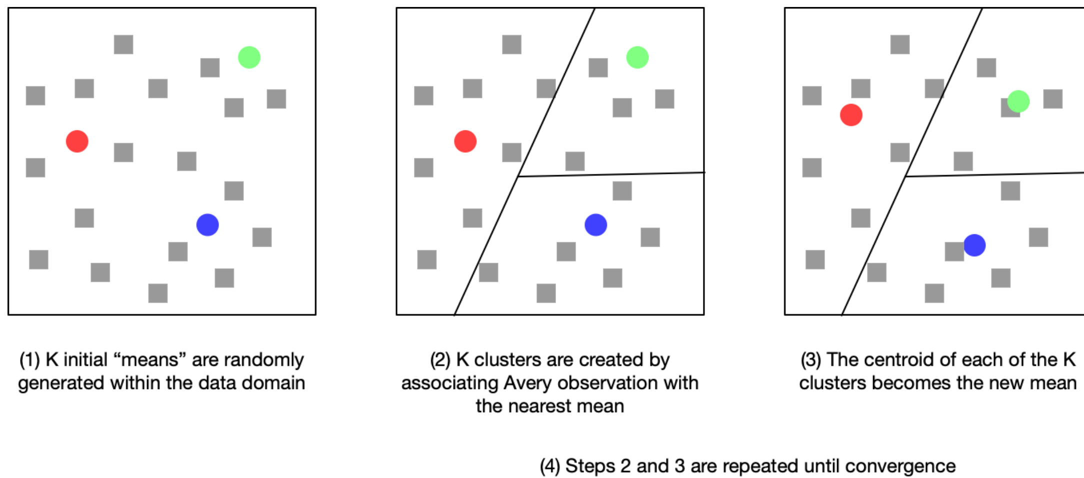


Figure 6.4: K-means for partitioning a dataset.

gic dataset partitioning represents a departure from conventional practices. Notably, this novel technique chooses to define partitions within the feature space rather than the input space, a strategic shift aimed at enhancing the orthogonality between local estimators.

This method presents a departure from convention and introduces various advantages, placing emphasis on preserving critical factors such as local effective size and bias. The results obtained through this innovative approach exhibit promising potential for advancing the accuracy of large-scale data analysis in the field of computer science. This strategic integration of clustering algorithms into the algorithmic framework opens new avenues for research, suggesting a more nuanced and effective methodology for handling intricate datasets.

The algorithm used to partition the dataset in my experiments is briefly illustrated by image 6.4. The results obtained through the experiments, influenced by the ParK methodology, are presented in the next chapter or in the appropriate appendices. This collaborative exploration emphasizes the value of incorporating innovative methodologies from research articles into practical experiments, enhancing the depth and applicability of the findings in the context of large-scale data analysis.

6.3 The Preconditioner

A pivotal element distinguishing Falkon lies in its incorporation of the Preconditioner, setting it apart from commercial counterparts. Section 4.2 delves into the specifics of the Preconditioner and its calculation through pseudocode, elucidating its role in Falkon's

functionality. This component assumes a critical function in the algorithm's initial stages, particularly in the selection of a data subset from the input dataset.

In the FalkonDC implementation, each model receives a subset of the initial dataset as input, prompting the Preconditioner to sample elements from this subset. To assess potential impacts on model accuracy, an alternative approach for Preconditioner calculation, termed the GlobalPreconditioner, is introduced. This variant involves computing the Preconditioner on the entire dataset and then passing it directly to models training on specific subsets without recalculating the Preconditioner on the limited dataset.

Implementation-wise, the class falkon.Falkon is modified, giving rise to the class GlobalPreconditioner.Falkon. While inheriting all fields and methods of the original class, this modification allows the model to train on a subset using a Preconditioner calculated on the entire original dataset. The objective is to achieve execution times comparable to the original implementation with an enhancement in model accuracy.

It is imperative to acknowledge that the essence of ensemble methods, exemplified by FalkonDC, lies in the diversification of sub-processes to mitigate variance, contrasting with the generation of similar processes. The upcoming chapter delves into an in-depth exploration of FalkonDC's performance, including an evaluation of the Preconditioner's impact. For a detailed examination of these results, please refer to the corresponding section, shedding light on the intricate trade-offs involved in optimizing Falkon.

The code depicted in below constitutes the focal point of my modifications to the $fit()$ method within the $Falkon$ class of the Falkon library. This particular modification addresses the capability to utilize a global preconditioner, even when a subset of the original dataset is provided to the model. In essence, this modification enables the model to employ the specified subset for training purposes while utilizing the entire original dataset for preconditioner calculation.

As elucidated in the comments of the original code, this segment is responsible for the selection of Nystrom centers. In the original version, the X parameter was passed to $senter_selection.select()$. In my implementation, however, the original dataset file is directly provided as a parameter. Here, X represents the tensor of training data, structured as $[num_samples, num_dimensions]$. In the global preconditioner version, X is consistently employed for all operations except the specifically indicated one. This adaptation not only extends the functionality of the Falkon algorithm but also streamlines the process of Nystrom center selection, enhancing the algorithm's flexibility and performance. These modifications are essential to accommodate the unique requirements of the global preconditioner, aligning with the broader objective of optimizing Falkon for diverse use cases.

Listing 6.1: GlobalPreconditione.Falkon

```
# Pick Nystrom centers
if self.weight_fn is not None:
    #ny_points, ny_indices = self.center_selection.select_indices
        (X, None)
    ny_points, ny_indices = self.center_selection.select_indices(
        torch.load("/root/Data/X_train.pt"), None)
else:
    #ny_points: Union[torch.Tensor, falkon.sparse.SparseTensor] =
         self.center_selection.select(X, None)
    ny_points: Union[torch.Tensor, falkon.sparse.SparseTensor] =
        self.center_selection.select(torch.load("/root/Data/
        X_train.pt"), None)
    ny_indices = None
num_centers = ny_points.shape[0]
```

The code above presents the two original lines commented on by me and the corresponding lines that implemented the changes described above.

Chapter 7

Tests and Results

In this chapter, I conducted a comprehensive evaluation of my implementation of FalkonDC. The primary objective was to evaluate the performance of the model under different configurations and conditions. Initially, I examined the performance of the model by applying the basic settings. Subsequently, I went into depth by analysing the behaviour of all the variants and settings described in the previous chapter. This chapter provides a detailed account of the experimentation and results, shedding light on the various factors influencing the performance of the FalkonDC algorithm.

I conducted an array of tests on multiple datasets, encompassing diverse scenarios such as a Synthetic dataset (binary classification), the Diabetes dataset (binary classification), and three versions of the Airlines dataset involving regression, binary classification, and multiclass classification. Remarkably, the results across these distinct datasets exhibited a consistent pattern.
So, the plots presented below offer a representative snapshot of FalkonDC's performance, with a specific focus on the Diabetes dataset. Additional comprehensive test results conducted on other datasets can be referenced in Appendix D, providing a view of FalkonDC's performance across various data domains and problem types.

The Diabetes dataset employed in these tests comprises a substantial 10^6 samples, each characterized by 8 features, as visually depicted in the accompanying figure. This dataset, as its name implies, is tailored for the task of classifying patients into those with or without diabetes, making it a binary classification problem. Common to many medical datasets, it exhibits a slight class imbalance, with a bias towards patients without diabetes. More precisely, patients without diabetes make up about 60 per cent of the total while the remainder are patients with the disease. However, it's noteworthy that despite this imbalance, the dataset exhibits behavior similar to perfectly balanced datasets, even synthetic ones, which

	Gender	Age	Hypertension	Heart_disease	Bmi	HbA1c_level	Blood_glucose_level	Diabetes
0	0	80	0	1	25.19.00	6.6	140	-1
1	0	54	0	0	27.32.00	6.6	80	-1
2	1	28	0	0	27.32.00	5.7	158	-1
3	0	44	1	0	36.08.00	6.5	126	1

Figure 7.1: The first elements of the Diabetes dataset.

underscores the versatility of FalkonDC across different data distributions.

The choice of the Diabetes dataset for these tests aligns with the prior discussions in earlier chapters, highlighting FalkonDC's aptness for applications in the medical domain. FalkonDC's distributed architecture presents an ideal solution in this context, where each computation node could correspond to a hospital or research center. This setup allows institutions to train their models on local, sensitive data without the need for centralized sharing. Subsequently, the results can be aggregated to enhance accuracy, a critical factor in the medical field where precision is paramount.

7.1 Experimental Environment and Settings

The experiments were conducted on hardware featuring an Intel x86 CPU, specifically the *Intel i9-10980 XE* with 18 cores and 36 threads at 3.00GHz, complemented by 120GB of RAM. This robust hardware configuration provided the necessary capacity to conduct tests on large datasets. Experiments were carried out up to a maximum of 15 nodes and for voting reasons an odd number of machines were always used.

For consistency across experiments, unless explicitly specified otherwise, basic settings were applied. These included the use of naive dataset splitting without overlap, the utilization of local rather than global preconditioning. These basic settings provide a standardized foundation for analysis, allowing for clear comparisons and assessments of algorithmic performance across diverse datasets and scenarios.

7.2 FalkonDC Performance

To assess the effectiveness of FalkonDC's approach, a comprehensive evaluation was undertaken, considering key metrics such as execution time, model accuracy and speedup. Among these, speedup stand as fundamental measurements in the realm of High-Performance

Computing. Speedup quantifies the degree to which the parallel model surpasses the performance of its sequential counterpart. For a more detailed exposition on speedup, readers are encouraged to refer to Appendix C.

Within this section, I will present the results of tests conducted on four distinct types of models, each providing unique insights into FalkonDC's performance:

1. Falkon "out of the box"
2. My Implementation of FalkonDC
3. FalkonN/P Trained on n/p Data
4. Standard Implementation of a Kernel Library

All the results showcased in this section are the outcome of averaging the results of 10 individual runs. This practice aims to provide more consistent and reliable values, thereby reducing the influence of random fluctuations and ensuring a more robust evaluation of the algorithm's performance. This systematic approach allows us to draw more meaningful conclusions and insights from the data, free from the idiosyncrasies of any single run.

7.2.1 Execution Time

In our exploration of the Falkon library, we sought to gauge its performance in training models under the specific dataset and hardware configurations outlined earlier. Remarkably, the average training time for a model utilizing the Falkon library out of the box clocked in at a mere 53 seconds.

Given that these models inherently involve the computation of an nxn matrix, with n representing the dataset's dimensionality, one might intuitively assume that training the same model with $n/2$ data would yield a runtime reduction approximating $\sqrt{(t)}$. Yet, the reality is more nuanced. Several other factors come into play that influence the overall execution time, and it's crucial to highlight that Falkon facilitates the computation of a submatrix with a dimensionality of mxm, where m can be tuned using an appropriate parameter. To maintain consistency across all our tests, we set m to 10^4, as mentioned in the preceding subsection.
However, I also carried out some tests by keeping the ratio between n and m constant and then adjusting m according to the dimensionality of the dataset in question. These tests are specified in the caption of the image, in case of no clarification it is assumed that the test was performed with the default parameters described above.

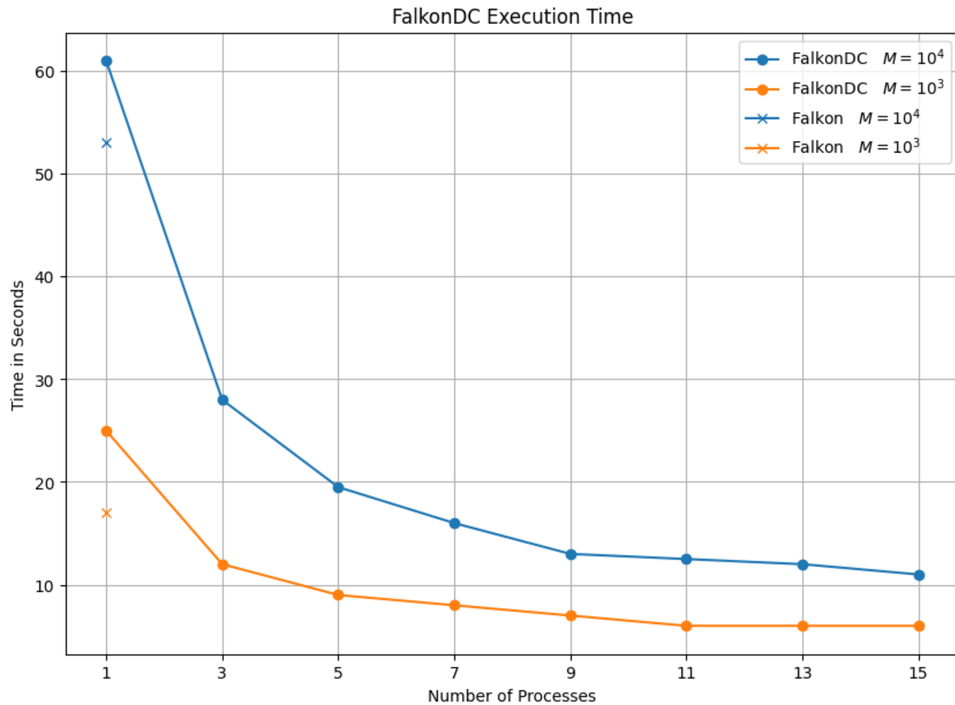


Figure 7.2: The graph of FalkonDC execution times as the number of nodes used changes.

For an in-depth analysis of FalkonDC's execution time in relation to the number of nodes employed, we've generated the plot 7.2. The x-axis illustrates the count of nodes or processes, while the y-axis presents the time in seconds. Our tests encompassed odd numbers of nodes ranging from 3 to 15 inclusively, thus avoiding the pitfalls associated with parity in the majority vote. Plot 7.2, depicted below, illustrates an intriguing trend: as the number of processes increases, the execution time per machine decreases, albeit not as significantly as one might have initially anticipated. This behavior can be attributed to the factors elucidated earlier. Notably, with the availability of 15 machines, it becomes feasible to attain an execution time approximately tenfold faster than the classical counterpart on a single node.

The Falkon-N/P variant essentially mirrors the FalkonDC version in terms of execution time, as each node in this setup is a Falkon model trained on n/p data. However, it's vital to underline that, despite similar runtimes, the accuracy of Falkon-N/P diverges. We will delve deeper into this disparity in the subsequent subsection.

Finally, we endeavored to train a model using the scikit-learn library. Regrettably, our attempts were met with an abrupt crash and an error message after several minutes of execution, highlighting potential challenges and limitations in this approach.

7.2.2 Accuracy

The primary objective of FalkonDC is twofold: first, to significantly reduce runtimes, as elucidated in the preceding subsection, and second, to strive for an accuracy score that closely approaches that of the original model. Boxplot 7.3, a visual representation that follows, contrasts the accuracy trends among three models: Falkon, FalkonDC, and Falkon-N/P. The x-axis of the plot delineates the models, while the y-axis portrays the accuracy trends.

From the plot, it becomes evident that the FalkonDC model achieves accuracy levels slightly lower than the standard model but significantly superior to that of the individual model trained on n/p data. Moreover, the boxplot underscores a crucial distinction: the Falkon-N/P model, owing to its relatively smaller dataset, exhibits notable variations in accuracy scores across different runs, a characteristic not shared by the other two models.

Subsequent to the initial comparative analysis of the diverse approaches, my focus shifted to examining the accuracy trends under varying numbers of processes.

To provide a more insightful visualization of this aspect, an additional boxplot was generated 7.4. In this visualization, the x-axis signifies the numbers of processes or nodes, while the y-axis represents the accuracy trends.

The graph reveals a compelling trend: as the number of processes increases, accuracy also sees a proportional improvement, and the variance between the results of the same model diminishes. This phenomenon aligns with theoretical expectations, with an upper limit dictated by the condition $m <= n/p$.

In essence, the data and visualizations offered in this section illustrate the delicate balance struck by FalkonDC. It manages to reduce runtimes significantly while maintaining accuracy levels that are impressively close to those of the original model, all while showcasing the benefit of parallel processing. The observed trend in accuracy underlines the algorithm's adaptability to a varying number of processes, with the potential to yield higher accuracy as computational resources expand.

7.2.3 Time with n/m constant

In situations where achieving even shorter execution times is imperative, it becomes viable to diminish the size of parameter m, albeit at the cost of some accuracy. The key innovation lies in maintaining a constant n/m ratio for processes executed on the nodes, enabling a substantial reduction in execution time while preserving the accuracy of the final model.

This approach holds notable relevance in diverse usage contexts, particularly when multiple machines are available, and there is no imperative need to enhance model accuracy through ensemble methods. To explore the efficacy of this strategy, a series of tests were

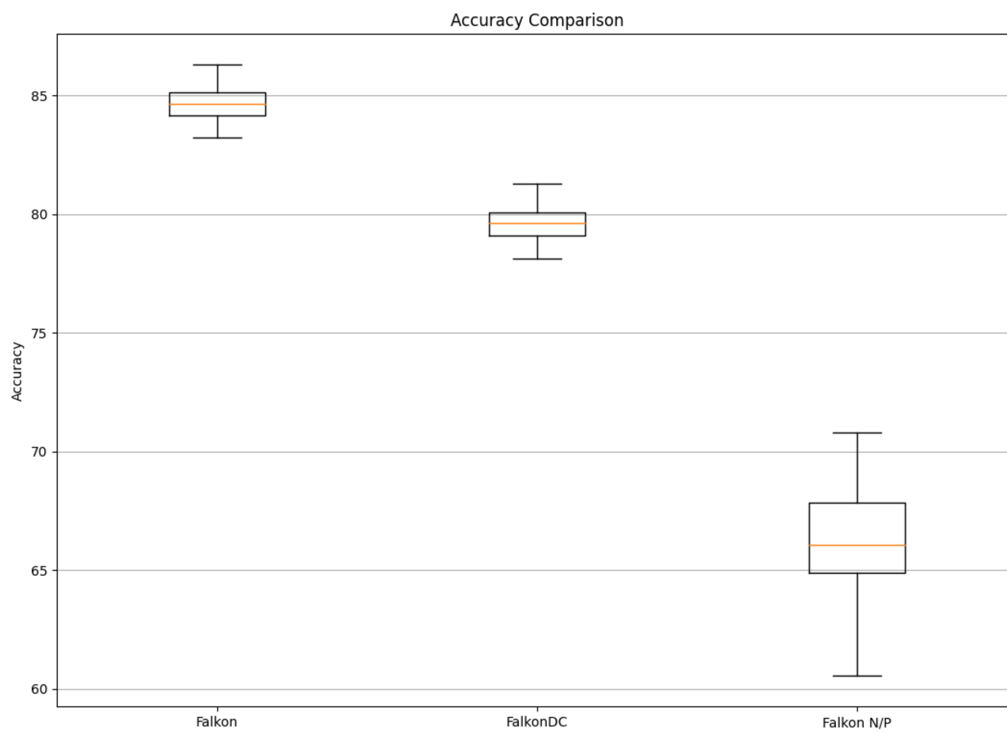


Figure 7.3: The accuracy distribution (in %) of Falkon, FalkonDC and FalkonN/P

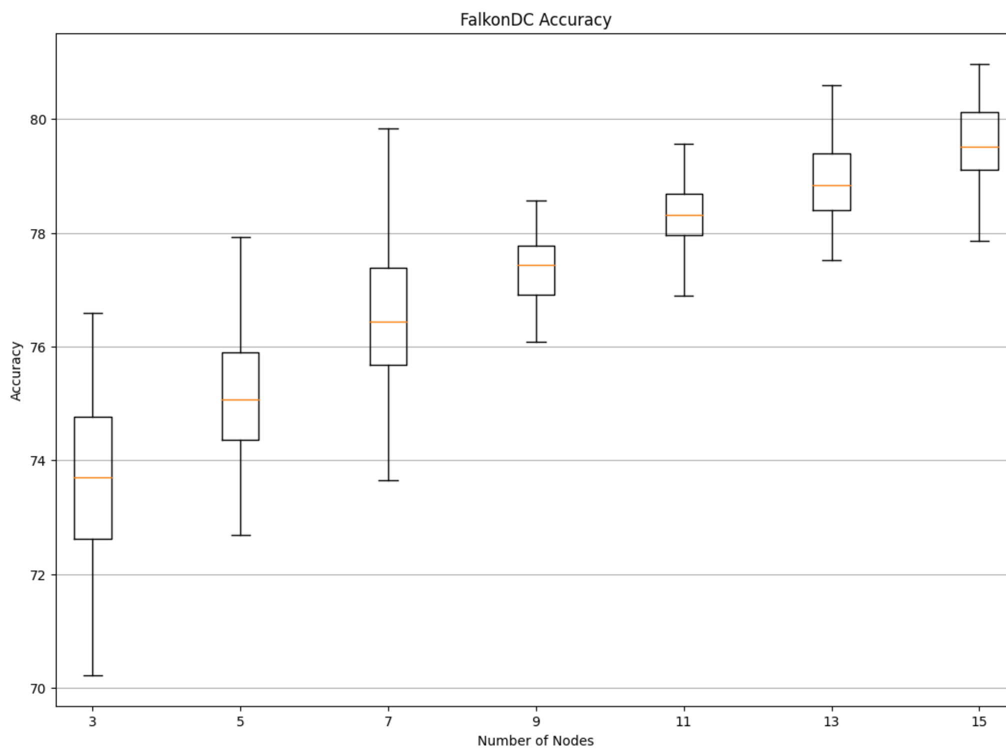


Figure 7.4: The accuracy (in %) and variance trend of FalkonDC as the number of nodes used changes.

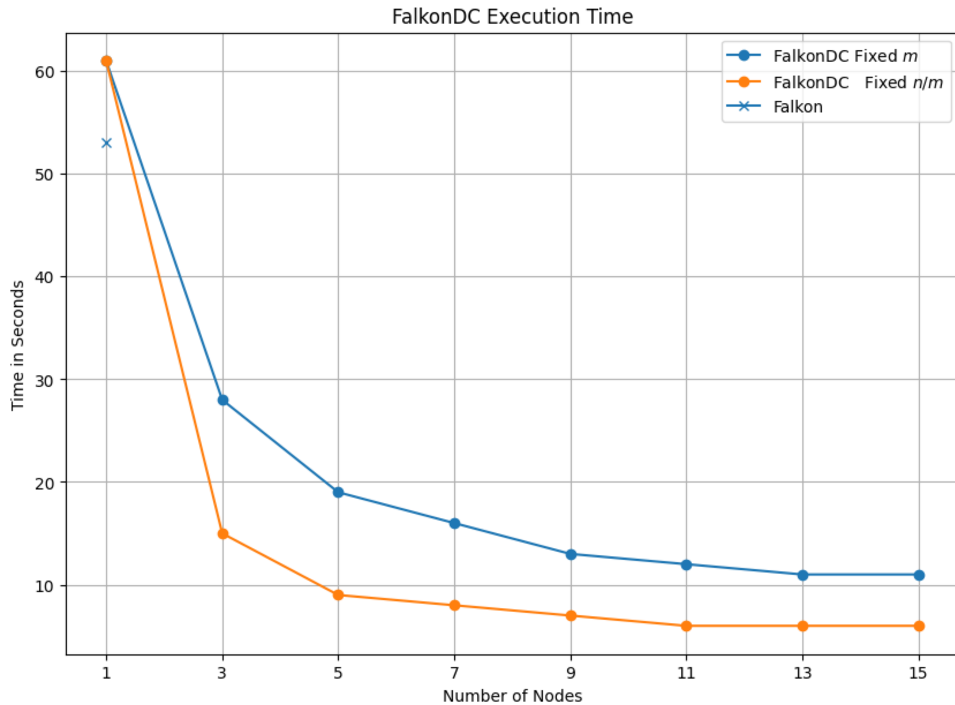


Figure 7.5: The comparison between the execution time of FalkonDC with fixed m and FalkonDC with fixed n/m ratio.

conducted, adhering to a consistent 100-to-1 ratio between n and m. Initiating with a configuration of $m = 10^4$ and $n = 10^6$, the size of m was then systematically adjusted to remain proportional to the number of samples per node as the node count increased. Consequently, this led to a progressive reduction in the size of m concurrently decreasing the execution time.

As illustrated in Plot 7.5, there is a significant reduction in execution times. It is noteworthy that the chosen example case may not be optimal due to its relatively small scale in relation to the number of nodes employed. Certainly with larger datasets the difference would be more pronounced.

Plot 7.6 reinforces the robustness of this technique, demonstrating that, even as the size of m decreases, the accuracy of the model remains constant. This trade-off between execution time and accuracy offers a valuable alternative for scenarios prioritizing computational speed over marginal improvements in model precision.

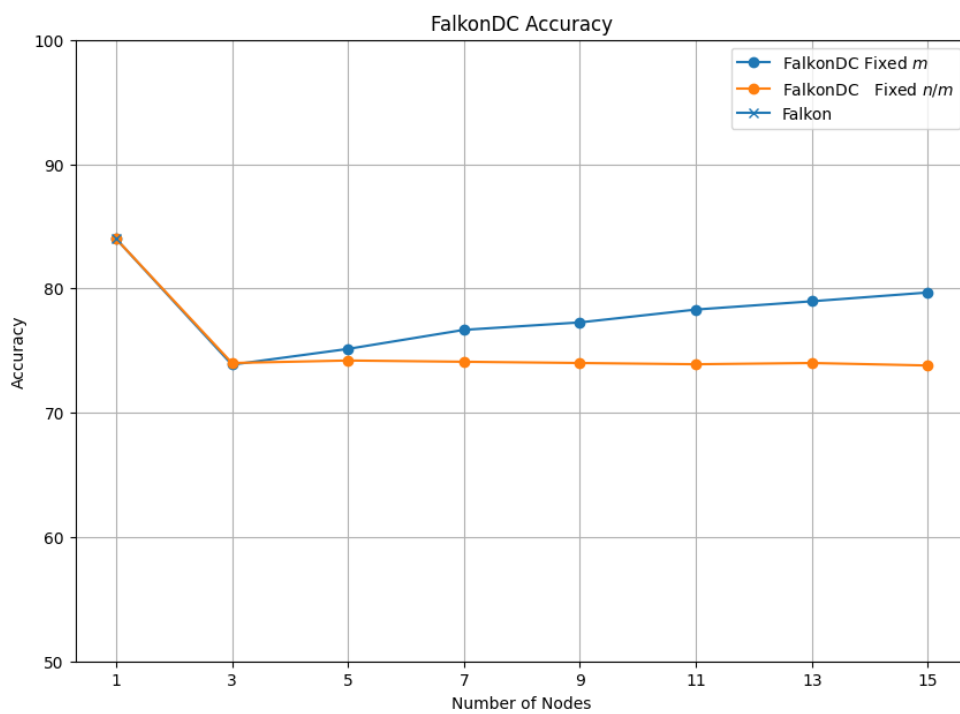


Figure 7.6: The comparison between the accuracy (in %) of FalkonDC with fixed m and FalkonDC with fixed n/m ratio. Accuracy expressed as a percentage.

7.2.4 Memory Usage

This metric holds immense significance as it offers valuable insights into the reasons behind the unsuccessful execution of the scikit-learn model implementation. The primary issue stems from the dataset's high dimensionality, which leads to the creation of an nxn matrix that surpasses the available RAM memory capacity.
In stark contrast, Falkon and, by extension, FalkonDC, employ innovative solutions that enable the reduction of this matrix, albeit at the cost of a slight compromise in model accuracy. As depicted in barplot 7.7, the x-axis portrays various models with different parameters, while the y-axis illustrates the percentage of RAM memory utilization. Specifically, the blue portion represents the memory currently in use, while the orange portion signifies the remaining free memory. The plot reveals that the memory usage of FalkonDC model, with fixed m remains consistent with Falkon. This phenomenon arises from the fact that Falkon, when provided with n samples as input, effectively mitigates the dimensionality challenge by computing a reduced matrix of size mxm, with m predetermined. Throughout my experiments, I employed a fixed value of $m = 10^4$, both for the standard Falkon model and the subprocesses of FalkonDC.
The last column shows FalkonDC executed with a fixed n/m value. This allows the value of m to be progressively reduced as the number of nodes increases. As a result, there is a net decrease in the RAM memory required to execute the processes. Thus, FalkonDC also offers significant improvements in this respect compared with the already good results achieved by Falkon.

In stark contrast to these optimized approaches, conventional implementations that necessitate the computation of an nxn matrix tend to exhaust available memory resources, rendering them incapable of completing the task when handling extensive datasets. This underscores the critical role of memory-efficient algorithms, like Falkon, in addressing the challenges posed by large-scale data processing.

7.2.5 Speedup

The concept of speedup is a vital metric in assessing the performance of a parallel algorithm in comparison to its sequential counterpart. Plot 7.8, displayed below, offers a visual representation of the speedup dynamics. On the x-axis, we can observe the numbers of processes or nodes, while the y-axis illustrates the speedup achieved. This plot vividly demonstrates the direct relationship between the number of nodes and the speedup, showcasing the algorithm's improved efficiency as parallelism is harnessed. Moreover, the plot incorporates the ideal speedup trend, which serves as a theoretical benchmark, representing the maximum attainable speedup. While the ideal speedup is challenging to reach in practical scenarios, it provides a crucial reference point.

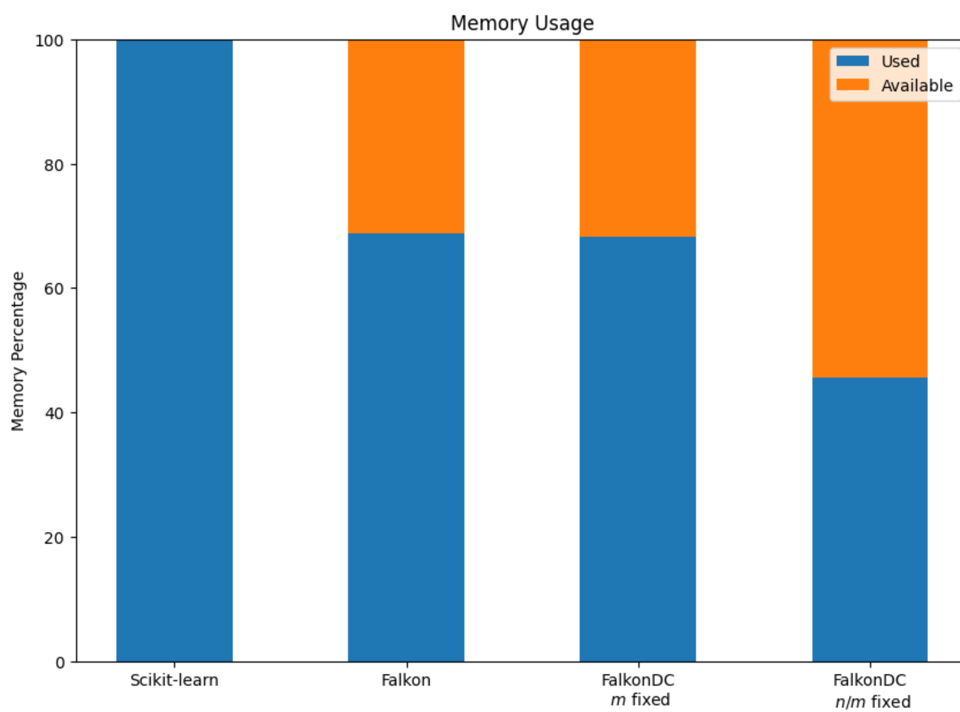


Figure 7.7: The average value of the percentage of RAM utilisation available from a FalkonDC container, Falkon and Scikit-learn.

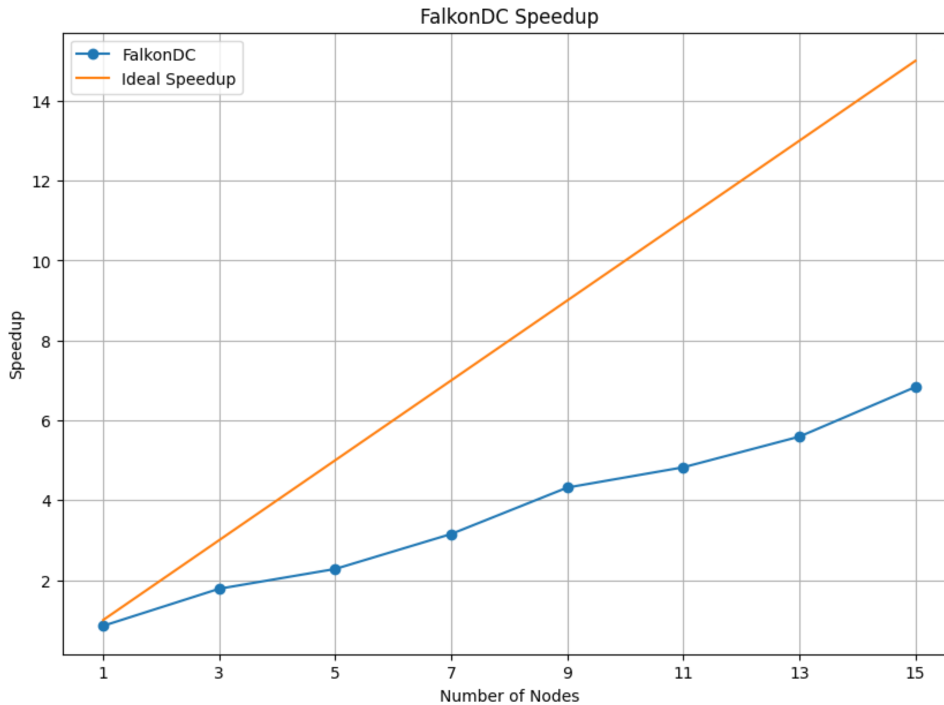


Figure 7.8: The speed of FalkonDC compared to the ideal one.

Notably, the speedup trend displayed in the plot remains consistently positive as the number of processes increases. This consistent upward trajectory underscores the pronounced advantage of FalkonDC over its sequential counterpart. The increase in speedup not only signifies improved performance but also reflects the algorithm's ability to harness the available computational resources effectively. This trend aligns with the core objective of parallel algorithms – to enhance efficiency and reduce execution time by exploiting multiple processing units in a coordinated fashion.
In summary, the data presented in Plot 7.8 underscores the success of FalkonDC in achieving significant speedup as more processes or nodes are engaged, emphasizing its superiority in parallel execution. This outcome is pivotal in the context of optimizing computational workloads, particularly when dealing with large datasets and complex computations, and it showcases the practical advantages of employing parallel algorithms.

7.3 Datasets Splitting Results

To establish the optimal configuration for FalkonDC and determine the most suitable dataset splitting strategy, I conducted a series of comprehensive tests targeted at these

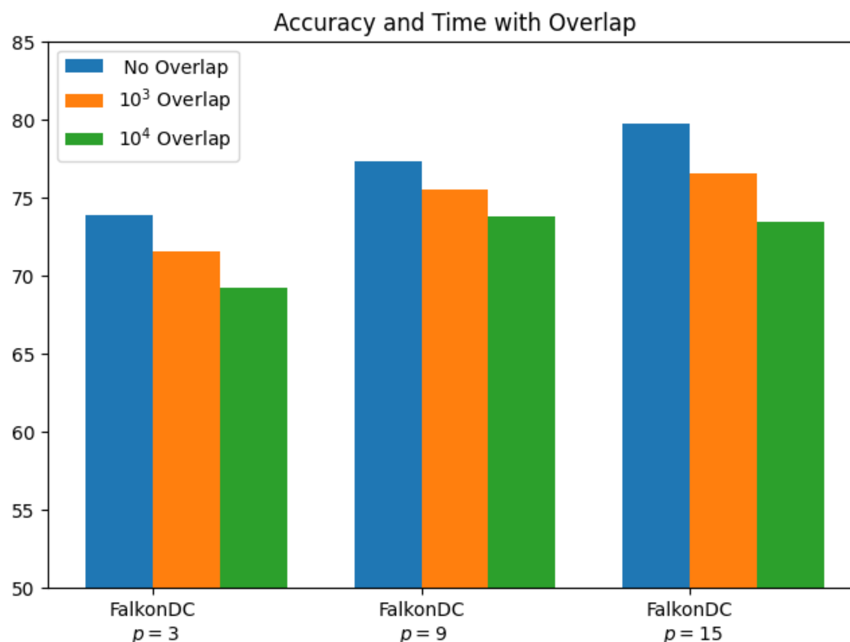


Figure 7.9: A comparison of the accuracy (in %) of different FalkonDC configurations with different amounts of overlap.

critical parameters. Specifically, I delved into a comparative analysis of performance concerning both accuracy and execution time. The tests encompassed various scenarios, including a baseline setup with no overlap, followed by the introduction of overlap settings at magnitudes of 10^3 and 10^4. The results are vividly illustrated in Plot 7.9, offering a visual representation of the aforementioned experiments. Notably, regardless of the number of nodes and dataset partitions, the accuracy of the models exhibits an inverse relationship with the amount of shared samples between subsets. It became evident that for significantly smaller datasets, introducing overlap could potentially enhance model accuracy. However, as the volume of data increases, the issue of decorrelation between the constituent models within FalkonDC starts to significantly impact overall performance. Regarding execution time, the introduction of overlap does not introduce any substantial delays in the process. It's only with larger overlaps, reaching the size of the subset itself, that perceptible slowdowns become apparent. These findings shed light on the intricate trade-offs involved in optimizing FalkonDC's configuration.

Subsequently, I delved into a comprehensive performance analysis, comparing the partitioning version with the splitting approach. The findings, as illustrated in Figure 7.10, revealed that there were no discernible differences in terms of accuracy; the results were remarkably consistent and superimposable. These outcomes were consistently replicated

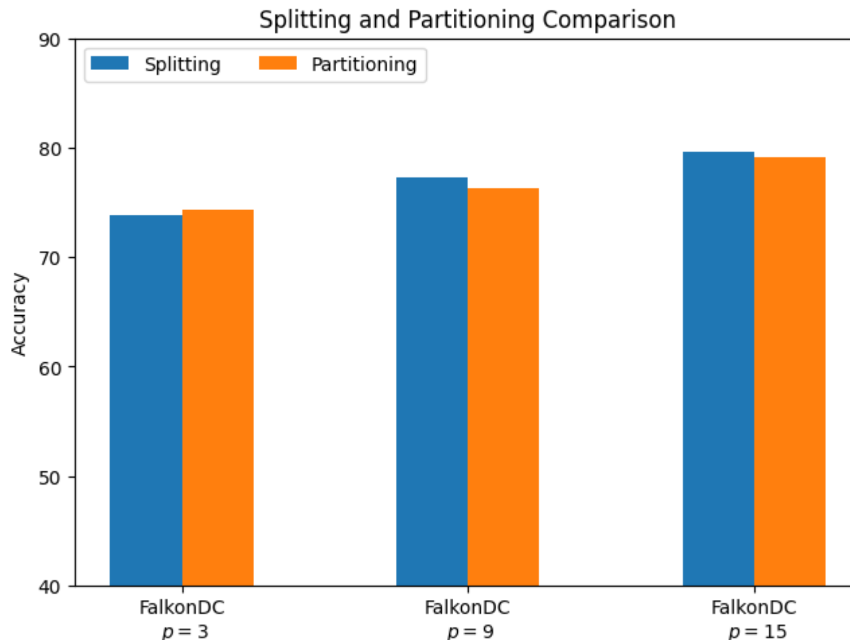


Figure 7.10: FalkonDC accuracy (in %) comparison performed by splitting or partitioning the dataset.

when experimenting with other real datasets.

However, some intriguing exceptions surfaced, notably in the case of a synthetic dataset showcased in Figure 7.11. I did reverse engineering to obtain a suitable dataset. In this instance, samples within the black square were labeled as 1, while the rest received a -1 label. The visual representation of the dataset demonstrated its natural division into three distinct clusters. In this specific scenario, the partitioning strategy consistently outperformed the splitting approach. Out of 10 tests conducted, an accuracy rate of 3 percentage points was achieved for the partitioning version. Furthermore, it's worth noting that the variance among the results obtained through splitting was significantly higher, underscoring the potential advantages of partitioning in certain data configurations. These observations emphasize the nuanced decision-making process required to optimize performance in different scenarios.

7.4 Preconditioner Results

The global preconditioner stands out as a configurable option within the FalkonDC setup, as elaborated in more depth in the dedicated section. This feature enables the computation of the preconditioner on the original dataset, which is of size n, while the actual model

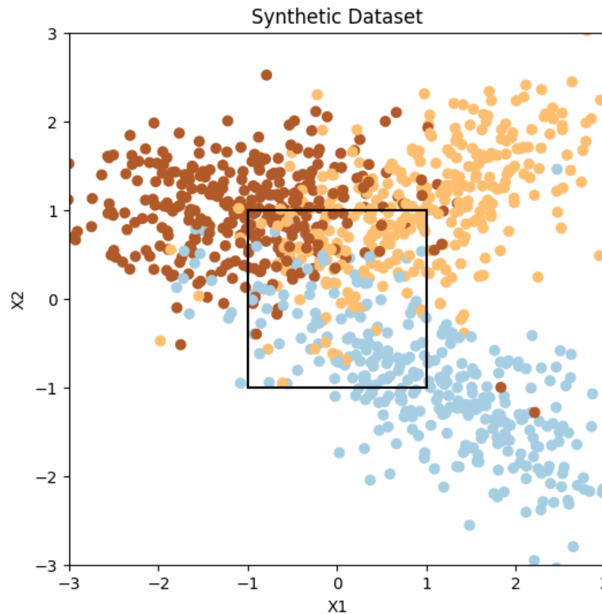


Figure 7.11: Synthetic dataset specifically created from three clusters in order to benefit from partitioning versus splitting.

training takes place on the assigned subset, sized at n/p. This approach paves the way for remarkably similar run times between the two available options. However, as depicted in Plot 7.13, the global preconditioner offers distinct advantages in certain scenarios.
The Plot provides a clear visualization of the accuracy of three FalkonDC models, each executed with different values of p (equal to 3, 9, and 15, respectively), with and without the global preconditioner. Notably, the performance of the global preconditioner becomes particularly apparent when the number of processes is relatively low.
Conversely, when the number of processes is substantial (as illustrated in our example, exceeding 9), the accuracy obtained with the global preconditioner may slightly lag behind that of the classical counterpart. This discrepancy likely arises from the fact that all processes draw from the same set of points for the preconditioner, resulting in a reduced level of diversification. Diversification, indeed, is a key element that empowers FalkonDC to deliver impressive results.
Nonetheless, in scenarios where diversification is inherently limited due to a small number of processes, the advantages brought about by the global preconditioner remain significant. This underscores the value of tailoring the choice of preconditioner to the specific context of the task and the available computational resources.
In summary, Plot 7.13 effectively illustrates the impact of the global preconditioner on the accuracy of FalkonDC models. It highlights the trade-offs involved and emphasizes the need to consider the number of processes and the nature of the dataset when configuring

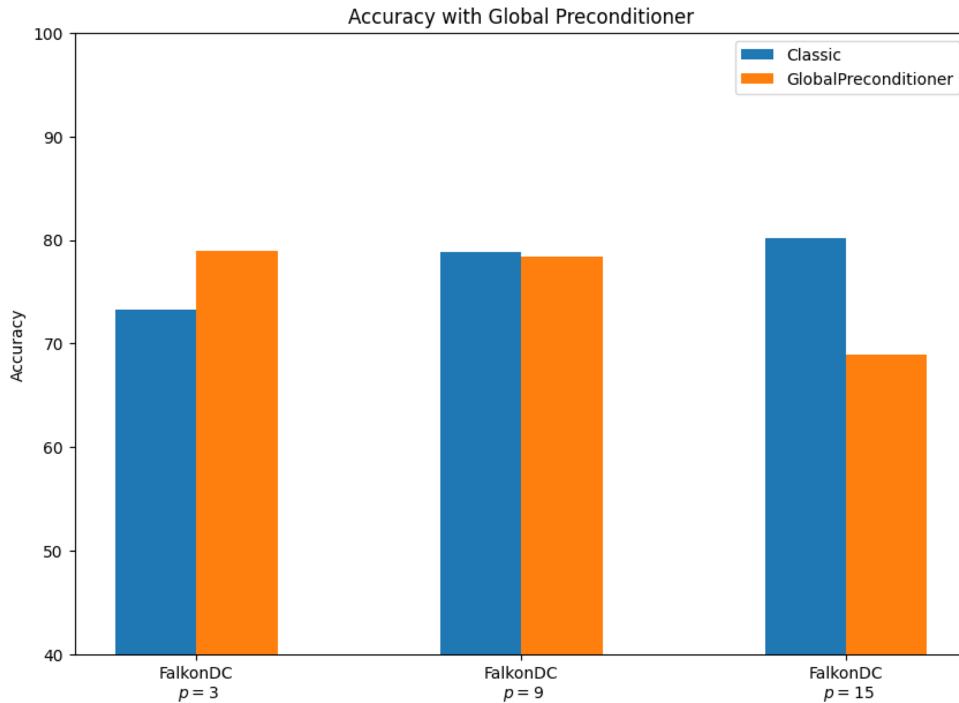


Figure 7.12: A comparison of the accuracy (in %) of FalkonDC using the GlobalPreconditioner in comparison with the classic version.

FalkonDC for optimal results.

7.5 Combinational Parameters

Motivated by the outcomes observed in prior experiments, I have introduced a modified parameter setting for FalkonDC in this iteration. Addressing concerns raised by both global preconditioner tests and tests involving splitting with overlap, it became evident that certain scenarios led to decreased accuracy due to diminished diversification among models trained on individual nodes. In response, a configurable option was incorporated into FalkonDC to enhance diversification among models.

This enhancement involves partitioning the dataset into disjoint subsets of varying sizes. Notably, the size parameter (m) is now dynamically adjusted for each model, independent of the assigned subset. The results of this configuration are depicted in plot 7.13. For instance, when FalkonDC was executed on 9 nodes in the traditional setup, each node was trained on 10^5 samples with $m = 10^4$. In contrast, the combinational version featured

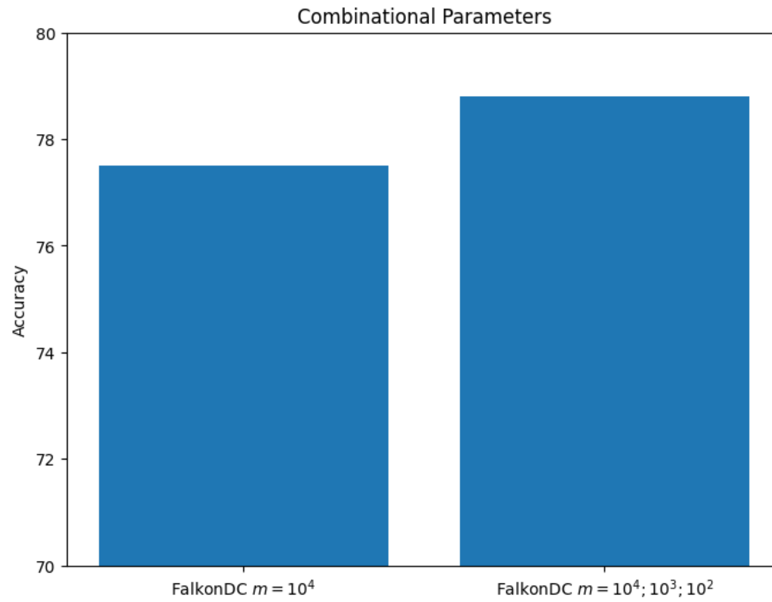


Figure 7.13: A comparison of the accuracy (in %) of FalkonDC and FalkonDC with Combinational Parameters, in both cases $p = 9$.

subsets of disparate sizes, with three nodes configured with $m = 10^4$, another three with $m = 10^3$, and the remaining three with $m = 10^2$. This deliberate diversification strategy aims to compensate for potentially suboptimal individual models, ultimately yielding a more robust final classifier.

As illustrated in the barplot, the Combinational version exhibits a slight improvement in accuracy, albeit by a few percentage points. The execution times of both versions remain identical, as the system adapts to slower nodes.

Chapter 8

Conclusions

In conclusion, the enhancements introduced by the DC version of Falkon stand as a compelling testament to its algorithmic prowess. Through my comprehensive studies, I have observed substantial gains in terms of execution time without compromising excessively accuracy. Notably, with datasets reaching a size of 10^6, the DC version demonstrates a remarkable speed-up of nearly tenfold when compared to its classical counterpart. It's important to note that this speed-up varies with the dimensionality of the dataset, inversely related to nxn, where n signifies the size of the matrix based on the dataset.

The true strength of FalkonDC becomes apparent when dealing with datasets of considerable size that standard implementations struggle to handle, primarily due to memory constraints and prohibitively long computation times. FalkonDC excels in this domain, enabling the execution of machine learning models trained on multiples of classical Falkon based on the number of available machines.

Beyond its exceptional performance, FalkonDC is equally fascinating for its capacity to provide a simple yet distributed solution that allows diverse entities to collaborate without the need to share sensitive data. A prime use case that comes to mind is the scenario involving confidential, patient-related information. Hospitals and research centers can readily embrace FalkonDC to enhance their own machine learning models while simultaneously facilitating the progress of others in the field.

Moreover, my implementation of FalkonDC via a container system offers the added benefit of effortless and rapid installation on a wide array of hardware configurations and operating systems. This approach mitigates many of the challenges encountered with the standard version, ensuring a smoother and more accessible experience for users across diverse platforms.

FalkonDC offers a high degree of flexibility by allowing users to customize the underlying models and various parameters to align with the specific characteristics of their datasets. This adaptability enables users to tailor FalkonDC to their unique use cases. Plot 8.1 below serves as a valuable tool for visualizing the impact of different FalkonDC parameters on both time and accuracy improvements.

The initial pair of plots highlights a common scenario where minimizing execution time is paramount. By setting the parameter "m" proportionally to the number of samples, users can efficiently reduce execution time with only a marginal sacrifice in accuracy, as demonstrated in the first set of plots.

The second pair of plots addresses situations where entities cannot share their data. In such cases, maintaining consistent execution times, as seen on the left plot, while rapidly increasing accuracy with the addition of nodes becomes crucial.

The third plot pair illustrates that, particularly with small datasets or a high node-to-dataset dimensionality ratio, introducing overlap between subsets can yield a notable increase in accuracy without a significant impact on execution time.

Moving to the fourth pair of plots, the benefits of partitioning and not splitting the dataset become apparent. This approach showcases an accuracy improvement with only a modest increase in execution time.

In the fifth pair of plots, the introduction of a global preconditioner is explored, particularly when the number of nodes is limited compared to the dataset's dimensionality. Here, a slight increase in accuracy is achieved without a notable rise in execution time.

In summary, a clear understanding of the specific use case is paramount for effectively defining parameters and obtaining optimal results. The nature of the dataset and the available hardware are pivotal considerations that should guide parameter customization to meet specific requirements.

8.1 Future Works

The FalkonDC algorithm presents a fertile ground for further enhancement and exploration in the context of computer science. One promising avenue for refinement is the introduction of novel techniques for dataset partitioning aimed at maximizing diversity and independence among individual subsets. This pursuit of improved data distribution

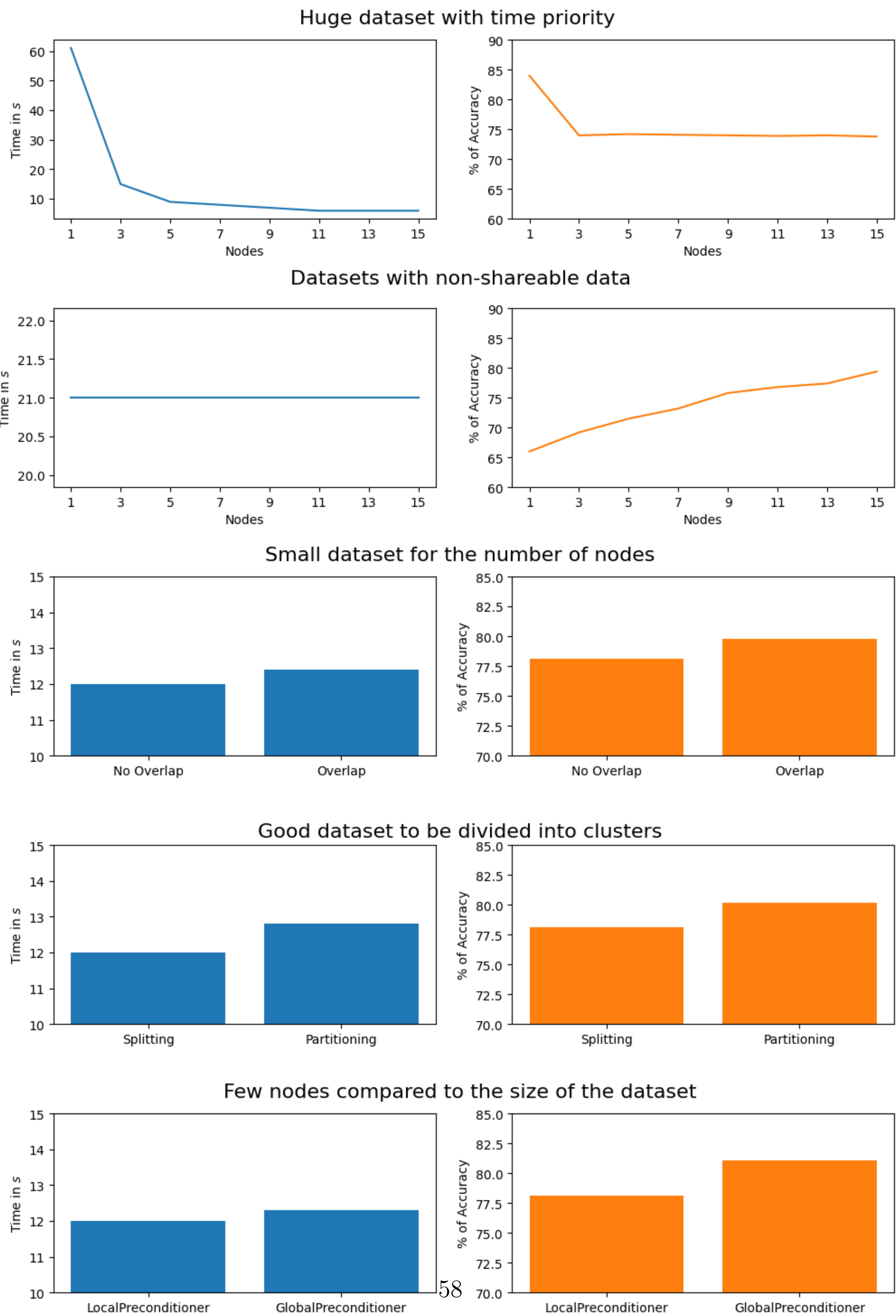


Figure 8.1: An overview of the best FalkonDC settings for specific use cases.

holds the potential to advance the algorithm's overall performance.

Another noteworthy dimension for investigation is the optimization of the preconditioner calculation process. In addition to the existing Falkon library's approach, a novel global preconditioner has been introduced, demonstrating the capacity to enhance accuracy in specific scenarios without imposing a substantial burden on execution time.

Crucially, owing to its deployment via Docker containers, FalkonDC offers a seamless transition towards becoming a cloud-compatible application. This evolution could seamlessly extend the algorithm's reach to popular cloud platforms such as Amazon's AWS, Google's GCP, and Microsoft's Azure. By doing so, we enable the creation of software that operates within a web browser environment, eliminating the need for complex installations and granting accessibility to a wider audience.

In the pursuit of broadening Falkon and FalkonDC's accessibility, a user-friendly interface could be designed using technologies like Angular. This approach aligns with the overarching vision of making these powerful algorithms available to a broader spectrum of users, facilitating their utilization with ease and convenience.

8.2 Acknowledgements

This thesis represents a significant milestone in my educational journey, one that has not only enriched my academic knowledge but has also provided me with valuable insights that have paved the way for early entry into the professional world. Throughout this challenging undertaking, I am deeply indebted to my university peers who have been unwavering sources of support. In particular, I extend my heartfelt gratitude to Gagandeep Singh and Alessandro Penco, with whom I shared countless rigorous study sessions.

I am also immensely appreciative of the dedicated teaching staff whose guidance and willingness to offer clarifications have been indispensable. Among them, special thanks are due to Daniele D'Agostino, my lecturer, and Lorenzo Rosasco, both of whom exhibited exceptional kindness and readiness to assist throughout this academic journey. Additionally, I must express my gratitude to Giacomo Meanti for his invaluable assistance with the Falkon library.

Last but certainly not least, I owe a debt of gratitude to my family, whose unwavering support has been a constant source of strength and encouragement. In particular, I would like to acknowledge my brother Simone, whose assistance and engaging discussions on various IT topics have been both enjoyable and enlightening. Their support has been pivotal

in shaping my academic and professional pursuits, and for that, I am truly grateful.

Appendices

Appendix A

Docker Ecosystem Overview

This appendix provides an in-depth look at the Docker ecosystem, elaborating on key components and tools that play a crucial role in Docker's functionality and widespread adoption. Docker, a containerization technology, has revolutionized the way applications are packaged, deployed, and managed. This section offer a comprehensive understanding of Docker's various facets.

A.1 Docker Components and Tools

Docker encompasses a rich array of components and tools that collectively enable the efficient utilization of container technology. Below, we delve deeper into these essential elements:

Docker Engine

The Docker Engine serves as the foundation of Docker's containerization capabilities. It manages the intricate processes involved in constructing container-based applications. This core component establishes a server-side daemon process responsible for hosting images, containers, networks, and storage volumes. Simultaneously, it provides users with a client-side Command-Line Interface (CLI) that allows interaction with the daemon through Docker's Application Programming Interface (API). Docker containers are created using Dockerfiles, while Docker Compose files specify the composition of components within a Docker container.

Docker Hub

Docker Hub stands as a pivotal software-as-a-service tool within the Docker ecosystem. It facilitates the publication and sharing of container-based applications through a centralized repository. Docker Hub boasts a vast library comprising over 100,000 publicly available applications, alongside provisions for both public and private container registries.

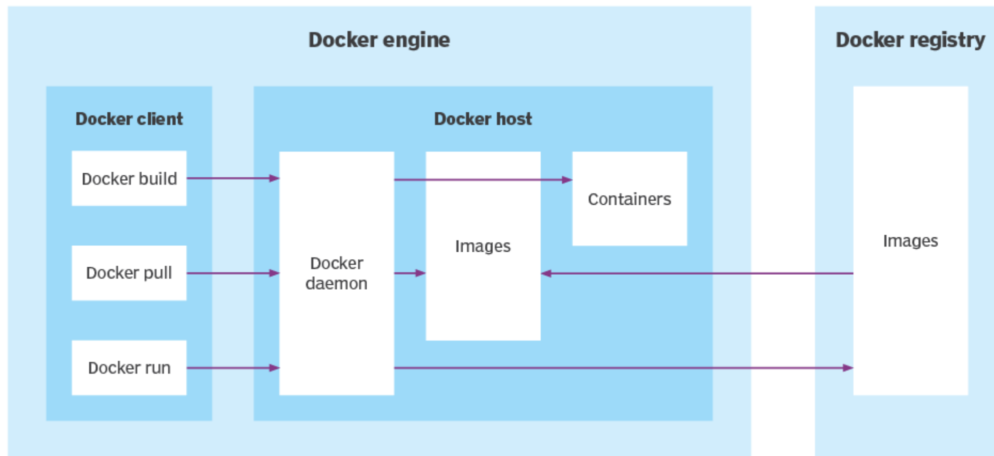


Figure A.1: Essential elements of the Docker architecture

This platform plays a vital role in promoting collaboration, distribution, and accessibility of containerized software.

Trusted Registry

In parallel with Docker Hub, the Trusted Registry serves as a repository for container image storage and distribution. It incorporates an added layer of control and ownership, enhancing security and management capabilities for organizations seeking greater autonomy over their container assets. This feature is particularly valuable for enterprises and businesses requiring strict governance of their containerized applications.

Docker Swarm

Docker Swarm is an integral component of the Docker Engine, designed to support cluster load balancing for Docker containers. By pooling resources from multiple Docker hosts into a cohesive unit, Docker Swarm enables users to seamlessly scale container deployments across various hosts. This clustering mechanism enhances high availability, fault tolerance, and efficient resource allocation for containerized applications.

Universal Control Plane

The Universal Control Plane (UCP) offers a unified, web-based interface for managing both clusters and applications. UCP simplifies the administration of Docker environments, providing a centralized platform for orchestrating containerized workloads, monitoring performance, and ensuring security across the entire infrastructure.

Compose

Docker Compose is an indispensable tool for configuring multi-container application services. It offers the capability to view container statuses, stream log output, and execute single-instance processes. Compose simplifies the orchestration of complex applications, making it easier to define and manage interconnected containers.

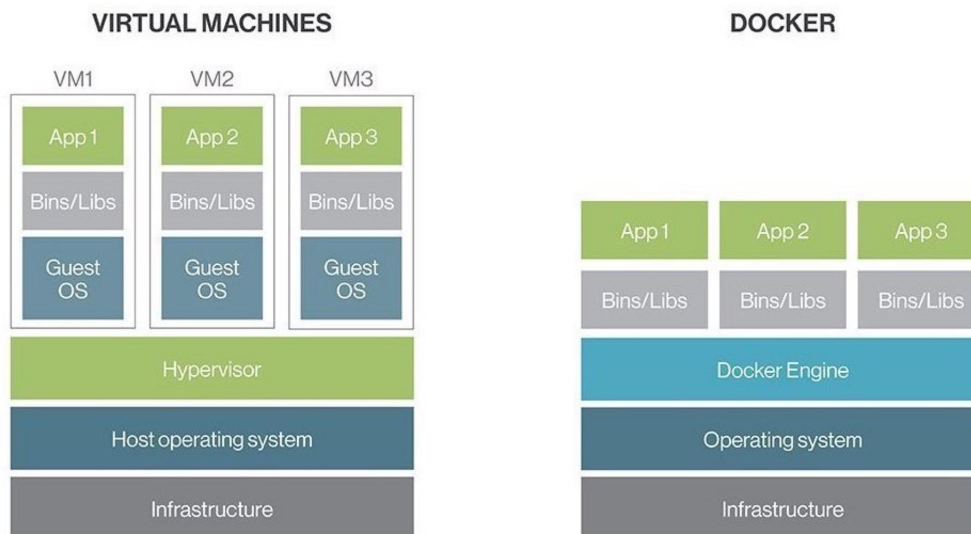


Figure A.2: Virtual machines versus Docker containers

Content Trust
Content Trust serves as a vital security feature within Docker, aimed at verifying the integrity of remote Docker registries. It achieves this by employing user signatures and image tags to establish trustworthiness. Content Trust bolsters the security of containerized applications, assuring users that the images they pull from registries are authentic and unaltered.

A.2 Docker's Advantages

Docker's widespread adoption is underpinned by several compelling advantages, as outlined below:
Portability
One of Docker's standout features is its exceptional portability. Containers encapsulate an application, along with all its dependencies, libraries, and configurations, into a single, self-contained unit. This portability enables users to effortlessly register and share containers across diverse hosts, regardless of the underlying infrastructure. This flexibility promotes consistency and ease of deployment in multi-cloud and hybrid environments.
Resource Efficiency
Compared to traditional virtual machines (VMs), Docker containers are incredibly resource-efficient. They share the services of a single underlying operating system, resulting in significantly reduced overhead. This efficient resource utilization translates into cost savings,

improved system performance, and optimized resource allocation for applications.

Rapid Deployment

Docker's containerization technology facilitates rapid deployment, far outpacing the time-consuming process associated with VMs. Containers can be spun up and taken down swiftly, enabling quick development, testing, and scaling of applications. This agility is a fundamental advantage for agile development practices and DevOps workflows.

In conclusion, Docker's ecosystem comprises a robust set of components and tools that empower users to harness the benefits of container technology. From efficient resource management to streamlined application deployment, Docker has emerged as a de facto standard platform for modern software development and infrastructure management.

Appendix B

Falkon Docker Image

This appendix provides a detailed guide on how to use the *falkon* Docker image to create a complete environment for running models based on the Falkon library. The falkon image is available on Docker Hub and offers a simple and quick way to set up an environment with all the necessary libraries to run Falkon models successfully.

You can find all the details at this link: falkon image.

B.1 Download the Falkon Docker Image

To get started, you need to download the *falkon* Docker image to your local environment. Open the terminal and execute the following command:

Listing B.1: bash

```
docker pull falkon
```

This command will download the *falkon* image from Docker Hub to your system.

B.2 Run a Falkon Container

Once the download of the 'falkon' image is complete, you can launch a container based on this image to create a Falkon development environment. Use the following command to start the container in an interactive mode:

Listing B.2: bash

```
docker run -it falkon
```

This instruction will start a container based on Debian with the Falkon environment correctly configured. Inside the container, you will have access to C++, Torch, Scikit-learn, and, of course, the Falkon library.

B.3 Create a Distributed Falkon Model with Majority Voting

To create a distributed Falkon model with majority voting, you can create a new Docker image based on the *falkon* image, adding the necessary dataset and model. The exact procedure for creating this image will depend on the specific configuration and data used in your experiments. We recommend referring to the GitHub repository mentioned in the main text of the thesis, where the data and code used for the experiments are made available. To access the data and code used in the experiments described in this thesis, please visit the following GitHub repository:

GitHub Repository for Data and Code

The repository will contain all the resources needed to reproduce the experiments and further details on using the Docker images created for Falkon. With this guide, you will be able to use the *falkon* Docker image to create a ready-to-use Falkon development environment and easily experiment with distributed Falkon models.

Appendix C

Speedup and Efficiency

In High Performance Computing to evaluate the goodness of the optimizations implemented compared to a sequential version of the same program , certain metrics such as Speedup and Efficiency are used. Speedup, often measured as a function of the number of processors, is central to quantifying performance improvements in parallel computing systems. It can be expressed as:

$$S = \frac{T_1}{T_P} \tag{C.1}$$

Where S represents speedup, T_1 is the execution time on a single processor, and T_P is the execution time on p processors. However, the quest for ever-increasing speedup should be tempered by the notion of Amdahl's Law, which articulates a limit on achievable speedup in parallel computing, defined by:

$$S \leq \frac{1}{F + \frac{1-F}{p}} \tag{C.2}$$

Where F is the fraction of the program that is inherently sequential.
Efficiency is a measure of how effectively a parallel system utilizes its resources and is expressed as:

$$E = \frac{S}{p} \tag{C.3}$$

Appendix D

Other Experiments

D.1 Synthetic Binary Classification Dataset

The synthetic dataset employed for these extensive tests is nothing short of impressive, featuring an astounding 10^7 samples, each defined by just 2 distinctive features, as thoughtfully illustrated in the accompanying figure D.1. It's noteworthy that this dataset was meticulously crafted by the researcher, designed specifically for the initial phase of testing. The deliberate choice of a synthetic dataset is not merely a matter of convenience; rather, it's a strategic decision borne out of necessity. Finding and managing datasets of this magnitude can be an arduous task, and custom generation becomes the pragmatic approach to meet the requirements of large-scale experimentation. Despite its synthetic nature, this dataset stands poised for binary classification challenges, setting the stage for rigorous evaluation and showcasing the adaptability of FalkonDC to handle massive datasets in the pursuit of machine learning excellence.

Plot D.2 provides a crucial insight into the performance of the FalkonDC algorithm, particularly in terms of container setup operations. It is evident that these operations introduce a measure of overhead, impacting the execution time when compared to the default Falkon

	X1	X2	Y
0	1,05E+18	-8,22E+16	-1,00E+18
1	9,66E+17	-1,66E+17	-1,00E+18
2	5,96E+17	-5,71E+17	1,00E+18
3	3,31E+17	-7,23E+17	1,00E+18

Figure D.1: The first 4 elements of the synthetic dataset.

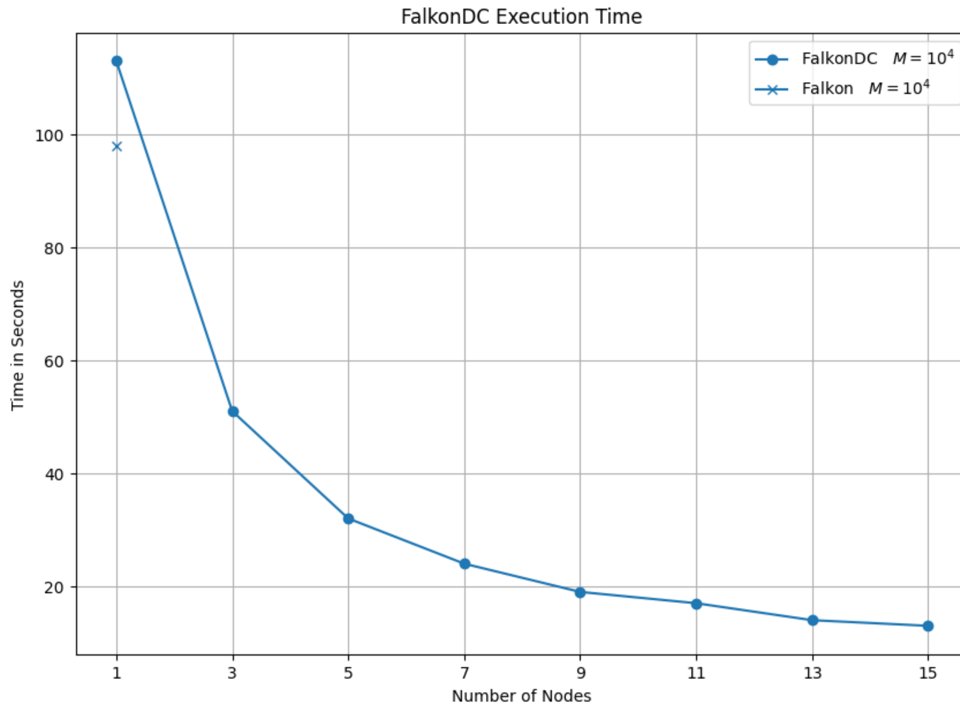


Figure D.2: Execution time of the synthetic dataset trained on P nodes with 10^7 samples.

implementation. However, it's important to note that FalkonDC is explicitly designed to leverage the capabilities of a multi-node architecture. Even with a limited number of machines, the algorithm demonstrates its potential by delivering a notable improvement in code execution speed, leading to substantial reductions in overall execution times. It's worth emphasizing that the specific execution times are highly dependent on the chosen parameters and the size of the training dataset, underlining the importance of these factors in optimizing FalkonDC's performance.

Plot D.3 provides a comprehensive analysis of FalkonDC's accuracy, offering a comparative view against both the standard Falkon and Falkon trained on N/P samples. The boxplot visualization illustrates that the standard version of Falkon achieves higher accuracy levels than the alternative implementations. However, FalkonDC excels by delivering commendable accuracy while significantly reducing the time required for the computations. Notably, Falkon running on N/P samples exhibits lower accuracy than the DC version, reaffirming the effectiveness of the latter in terms of accuracy-time trade-offs. Additionally, the boxplot conveys how FalkonDC manages to reduce the variance in its results, ensuring more consistent and dependable outputs, ultimately enhancing the algorithm's overall reliability and performance.

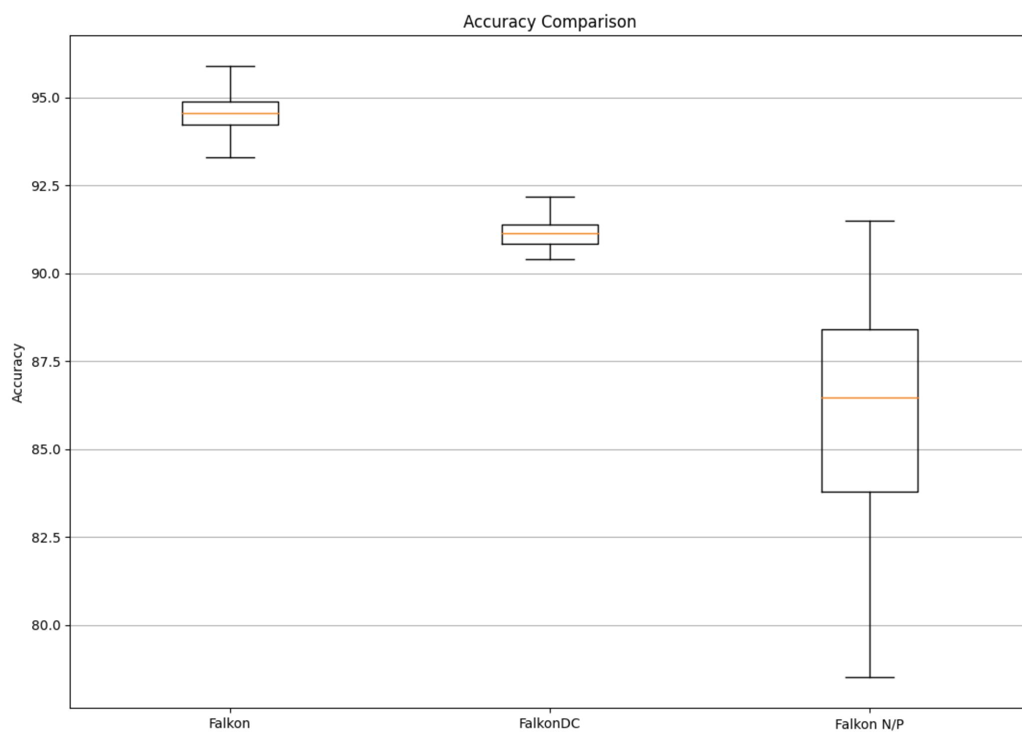


Figure D.3: Distribution of the Accuracy (in %) for Falkon, FalkonDC and Falkon N/P with the synthetic dataset.

	Month	DayOfMonth	DayOfWeek	PlaneAge	Distance	AirTime	DepTime	ArrTime	ArrDelay
0	1.0	3.0	4.0	10.0	810.0	116.0	2003.0	2211.0	-14.0
1	1.0	3.0	4.0	10.0	2283.0	314.0	734.0	958.0	-22.0
2	1.0	4.0	5.0	10.0	239.0	48.0	1338.0	1440.0	10.0
3	1.0	4.0	5.0	10.0	781.0	100.0	1509.0	1807.0	12.0

Figure D.4: The first 4 elements of the dataset Airlines Regression.

D.2 Airlines Regression Dataset

The airlines dataset consists of 10^6 samples and 9 features, as shown in the figure D.4. This dataset contains airline flights and the difference between the actual arrival time versus the predicted time. Therefore, it can be used to predict aircraft delays. This is the original dataset on which the binary classification and multiclass classification datasets are based.

This particular test case represents a departure from the previous scenarios, as it addresses a regression problem rather than a classification task. Consequently, a distinct model was employed for training, and all model files utilized in the experiments can be accessed in the FalkonModel folder within the associated GitHub repository.

In alignment with previous results, the outcomes obtained from this regression problem remain consistent. Plot D.8 offers a compelling visualization, showcasing a reduction in execution time as the number of nodes in the multi-node architecture increases. However, it is important to note that this reduction in time is influenced by the Docker-based infrastructure, setting an upper limit on the performance gains achievable.

Contrastingly, Plot D.9 shifts its focus from accuracy to RMSE, with a boxplot that exhibits a behavior akin to that observed in prior datasets. It is crucial to highlight that lower RMSE values, approaching zero, signify the superiority of the models. In this context, Falkon outperforms the DC version, albeit in considerably larger datasets. Nevertheless, the distinct feature of the DC version is its ability to diminish the variance in results, a trait that renders it particularly favorable for achieving homogeneity and stability, especially in domains where reliability is paramount, such as the sensitive field of medical research. This emphasizes the potential of FalkonDC as a robust and dependable choice in scenarios with stringent requirements.

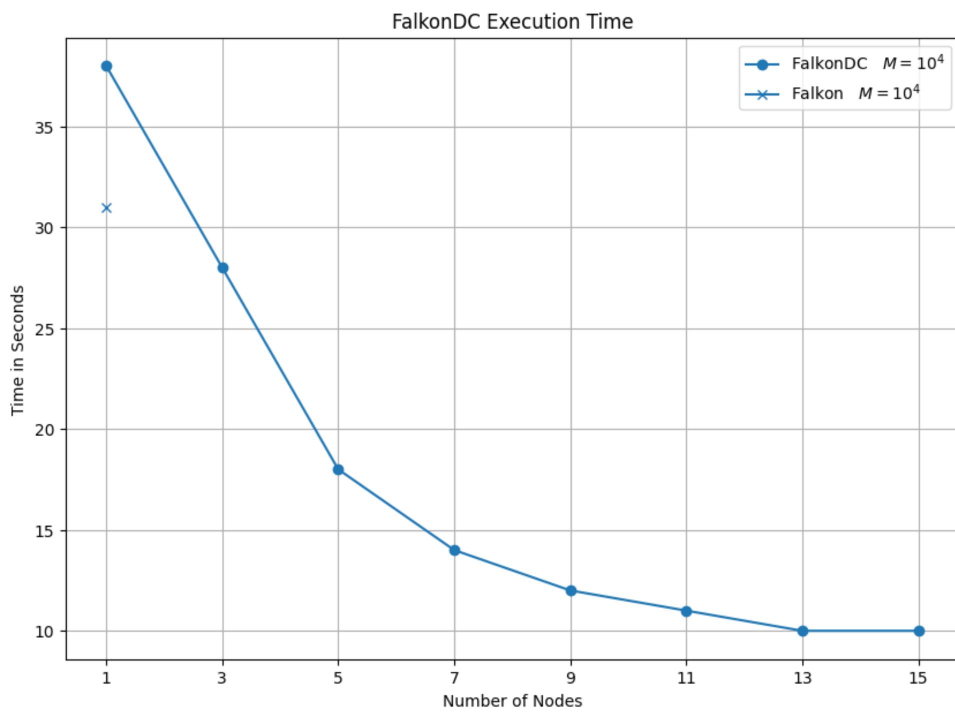


Figure D.5: Execution time of the Airlines dataset trained on P nodes with 10^7 samples.

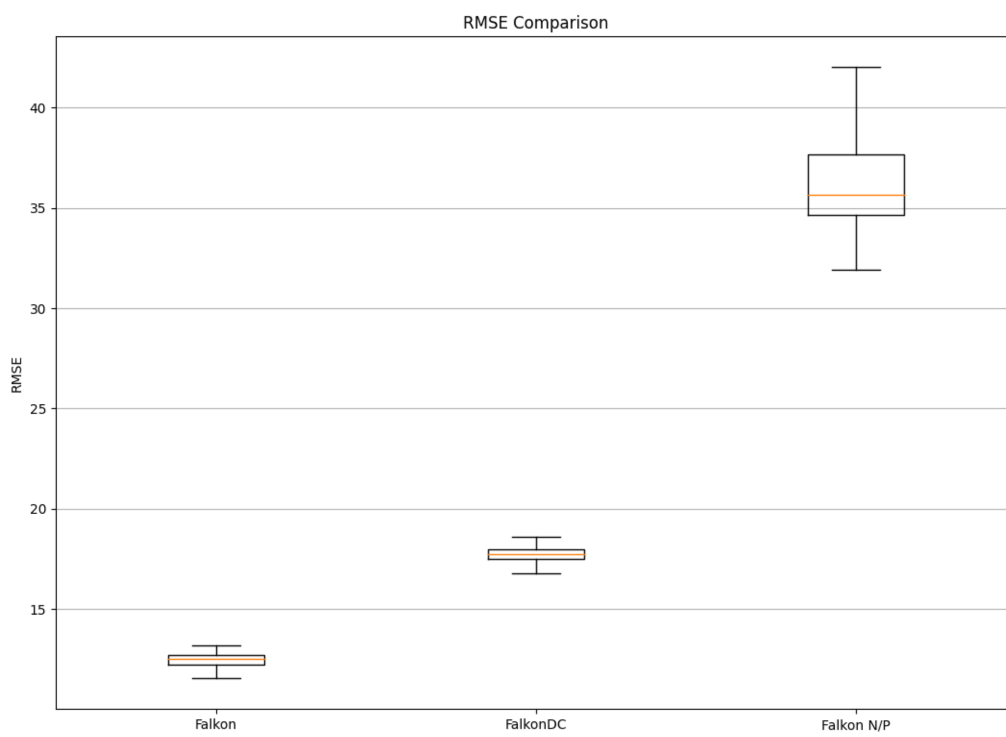


Figure D.6: Distribution of the RMSE for Falkon, FalkonDC and Falkon N/P with the Airlines dataset.

	Month	DayOfMonth	DayOfWeek	PlaneAge	Distance	AirTime	DepTime	ArrTime	ArrDelay
0	1.0	3.0	4.0	10.0	810.0	116.0	2003.0	2211.0	1
1	1.0	3.0	4.0	10.0	2283.0	314.0	734.0	958.0	1
2	1.0	4.0	5.0	10.0	239.0	48.0	1338.0	1440.0	-1
3	1.0	4.0	5.0	10.0	781.0	100.0	1509.0	1807.0	-1

Figure D.7: The first 4 elements of the dataset Airlines Binary Classification.

D.3 Airlines Binary Classification Dataset

The Airlines dataset is a collection comprising 10^6 samples, each characterized by 9 distinctive features, a visual representation of which is thoughtfully provided in the figure D.7. Primarily designed for binary classification, this dataset immerses us in the intricate world of airline routes and punctuality prediction. By scrutinizing the delays associated with these routes, it aspires to forecast whether a flight will arrive promptly or face tardiness. Notably, this dataset shares its lineage with the renowned Airlines dataset typically dedicated to regression challenges. The transformation applied to create this binary classification dataset is intriguing; it hinges on a well-defined threshold, separating flights into two distinct categories. Those that cross this predefined threshold are categorized as 'late,' while those below are deemed 'on time.' This unique dataset offers an exceptional platform for assessing FalkonDC's prowess in predictive modeling under real-world conditions.

In this dataset, the performance aligns with the trends observed thus far. However, this specific instance entails training using LogisticFalkon, offering an opportunity to explore the capabilities of the Falkon library and assess whether it can consistently deliver results across diverse model options.

Plot 5 presents a comprehensive analysis of the accuracy achieved by FalkonDC, providing a comparative perspective alongside the standard Falkon and Falkon trained on N/P samples. Remarkably, the accuracy pattern remains in line with the patterns observed in previous cases, reaffirming the algorithm's reliability and robustness. Once again, FalkonDC stands out as the optimal choice, striking a commendable balance between accuracy and execution time. It is noteworthy that FalkonDC consistently reduces the variance in performance, further underlining its suitability for applications where both precision and consistency are essential. This continued consistency across varied scenarios solidifies FalkonDC as a dependable and versatile algorithm for real-world applications.

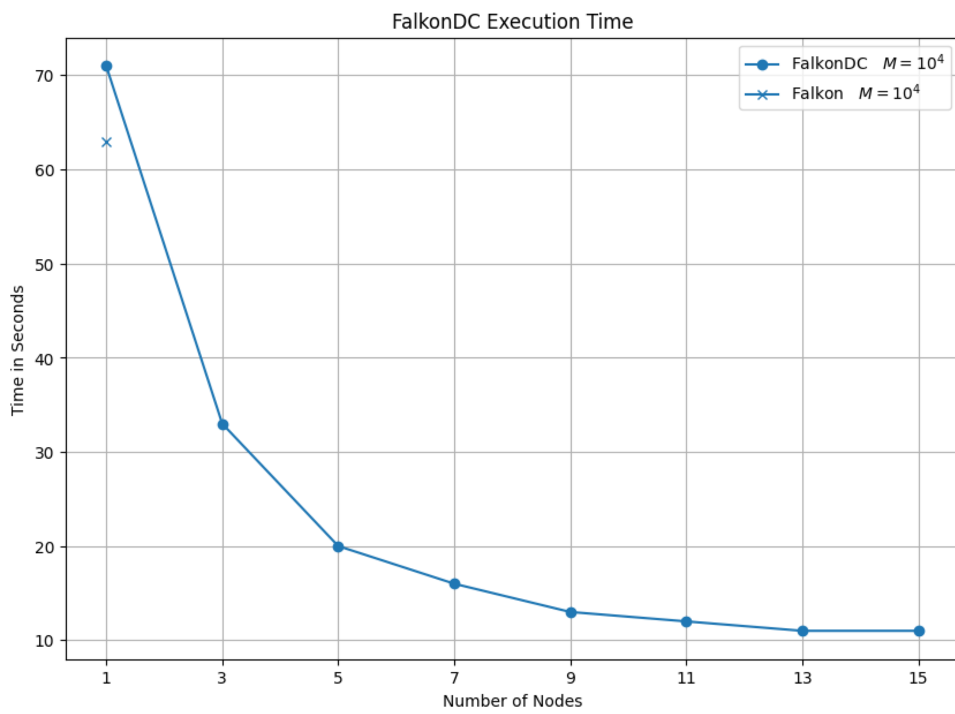


Figure D.8: Execution time of the Airlines dataset trained on P nodes with 10^6 samples.

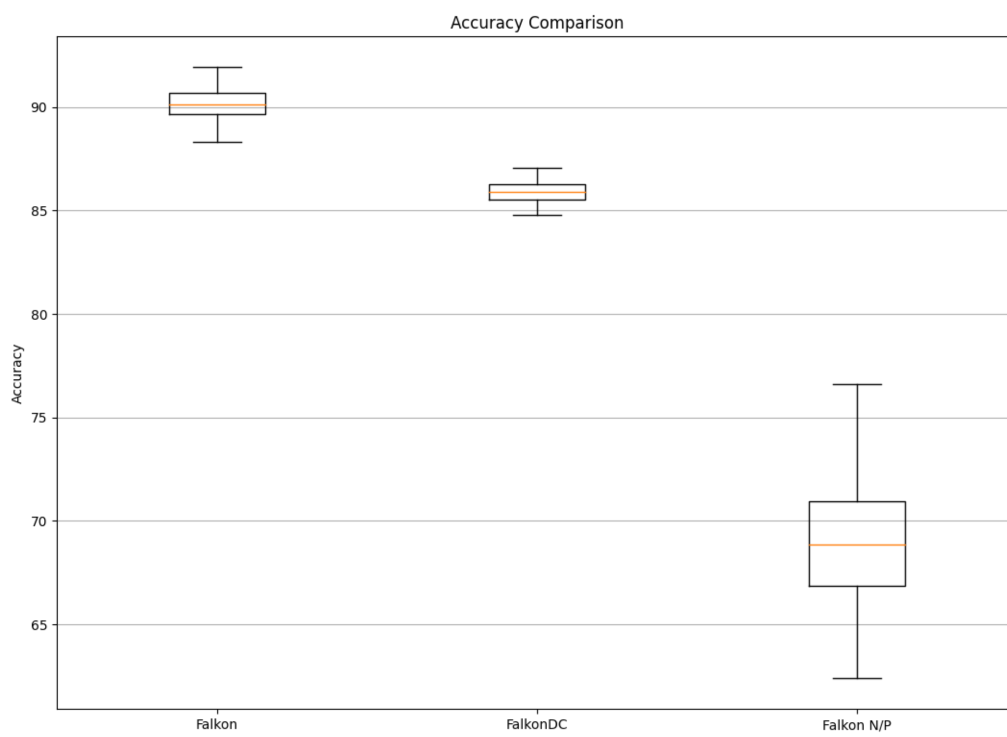


Figure D.9: Distribution of the Accuracy (in %) for Falkon, FalkonDC and Falkon N/P with Airlines dataset.

	Month	DayOfMonth	DayOfWeek	PlaneAge	Distance	AirTime	DepTime	ArrTime	Y
0	1.0	3.0	4.0	10.0	810.0	116.0	2003.0	2211.0	[1, 0, 0]
1	1.0	3.0	4.0	10.0	2283.0	314.0	734.0	958.0	[1, 0, 0]
3	1.0	3.0	4.0	10.0	577.0	79.0	1653.0	1932.0	[0, 1, 0]
13	1.0	5.0	6.0	10.0	507.0	77.0	2109.0	2249.0	[0, 0, 1]

Figure D.10: The first 4 elements of the dataset Airlines Multiclass Classification.

D.4 Airlines Multiclass Classification Dataset

The airlines dataset consists of 10^6 samples and 9 features, as shown in the figure D.10. This dataset contains airline flights and the difference between the actual arrival time versus the predicted time. Therefore, it can be used to predict aircraft delays. This dataset is based on its namesake dataset for regression, in order to generate a multiclass dataset, two thresholds were inserted so as to divide the planes that arrived on time, those that arrived late, and those that arrived early.

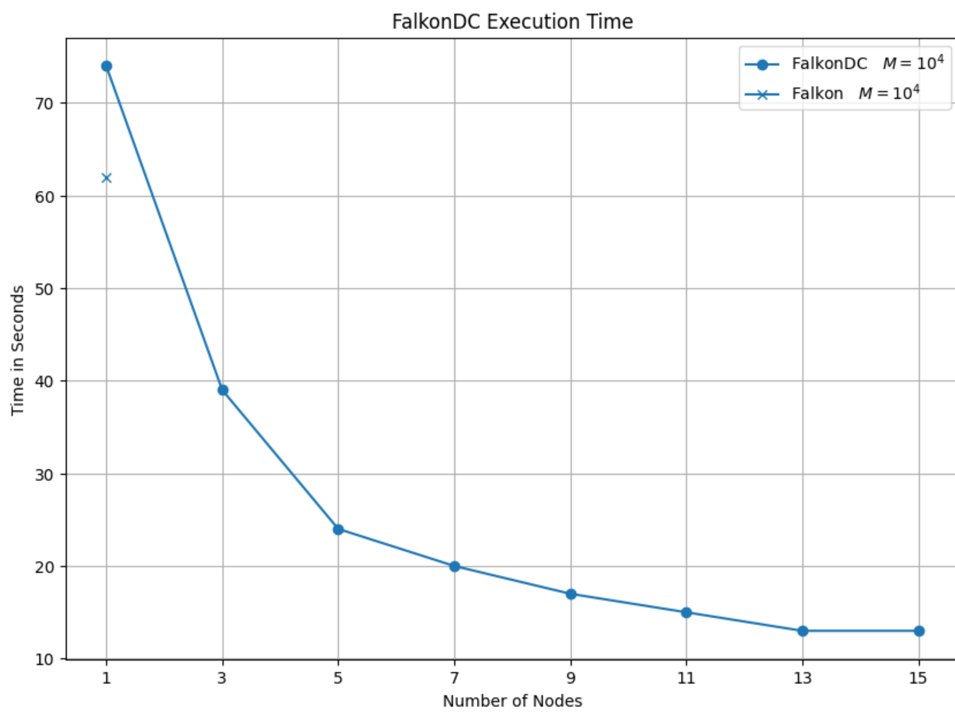


Figure D.11: Execution time of the Airlines dataset trained on P nodes with 10^7 samples.

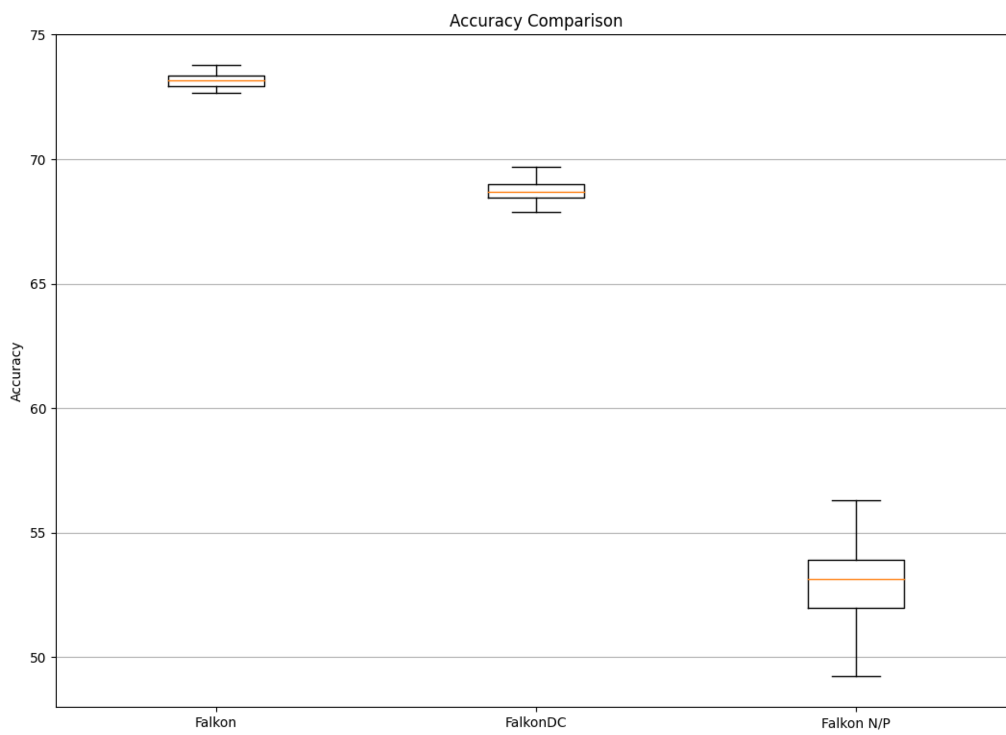


Figure D.12: Distribution of the Accuracy (in %) for Falkon, FalkonDC and Falkon N/P with the Airlines dataset.

Bibliography

[ARR17] Luigi Carratino Alessandro Rudi and Lorenzo Rosasco. Falkon: An optimal large scale kernel method. In *FALKON: An Optimal Large Scale Kernel Method*, 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/05546b0e38ab9175cd905eebcc6ebb76-Paper.pdf.

[Fal] Falkon. URL: https://falkonml.github.io/falkon/install.html.

[Ker] Kernelmethods 0.2. URL: https://raamana.github.io/kernelmethods/readme.html.

[MCRR20] Giacomo Meanti, Luigi Carratino, Lorenzo Rosasco, and Alessandro Rudi. Kernel methods through the roof: Handling billions of points efficiently, Nov 2020. URL: https://arxiv.org/abs/2006.10350.

[Sci] Sciki-learn. URL: https://scikit-learn.org/stable/modules/kernel_ridge.html.

[ZDW14] Yuchen Zhang, John C. Duchi, and Martin J. Wainwright. Divide and conquer kernel ridge regression: A distributed algorithm with minimax optimal rates, Apr 2014. URL: https://arxiv.org/abs/1305.5029.