Machine Learning-Based Control Methods for an Optimal AMG Setup Strategy with Massive Distributed Parallelism

Andrés Casillas García de Presno

Born 2nd August 1999 in Distrito Federal, Mexico.

11th November 2025

Master's Thesis Mathematics

Advisor: Prof. Dr. Marc Alexander Schweitzer

Second Advisor: Dr. Sebastian Gries

Institut für Numerische Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

A mi familia y a México.

Science is what we understand well enough to explain to a computer; art is everything else.

-Donald E. Knuth

Acknowledgments

I would like to use this space to thank all of the people and institutions that have made this work possible.

First and foremost I want to thank my family: my father Juan, my mother Laura, and my brother Juan, for being the greatest support in every aspect of my life, even when living on opposite sides of the world. I simply could not imagine my life without them. These words also apply to Brenda Aguilar Saavedra, who has become a part of my family throughout the years.

Secondly, I want to thank my friends -both my lifelong friends and the friends that I made in Bonn- for always being there for me: you have made this experience worth living.

I also want to thank the "Fondo para el Desarrollo de Recursos Humanos" (FIDERH) from "Banco de México" for the financial aid that they have given me to complete my graduate studies in Germany.

Moreover, I want to thank my two advisors: Prof. Dr. Marc Alexander Schweitzer and Dr. Sebastian Gries, for their helpful advice and guidance throughout the making of this work. I would also like to thank them for giving me the opportunity to be part of the *Schnelle Lösungen* team at Fraunhofer's SCAI.

Lastly, a big thank you to Bonn University and to the City of Bonn, for being my home for the last two years.

Andrés Casillas García de Presno

Contents

1 Introduction

The aim of this thesis is to investigate the potential of machine learning methods to make performance-optimal decisions regarding the setup strategy -specifically the coarsening strategy- of algebraic multigrid methods, with a focus on massively parallel applications. We demonstrate that this approach enables near-optimal coarsening choices in practical applications, yielding average improvement factors of 41% over default coarsening settings -see Subsection 4.2.

On the one hand, machine learning methods (ML) refer to a family of statistical techniques which have been developed, in conjunction with mathematical models and algorithms, in order to predict an outcome based on previously gathered data. These techniques have seen significant progress in the last decade and have led to remarkable advancements in almost all areas of science and technology [10, 11, 14, 25]. In this context, the application of ML techniques to algorithm optimization represents a fruitful direction to explore. On the other hand, algebraic multigrid methods (AMG) refer to a family of algorithms that build a hierarchy of so-called "grids" -which originally arose from the discretization of elliptic and parabolic partial differential equations, but have since been applied to many other problems [1, 10]- in order to systematically and efficiently solve an algebraic linear system of equations of the form $Ax = b$.

The problem of solving an algebraic system of linear equations might seem theoretically trivial, but in practice dimensionality, matrix properties, numerical instability, and the lack of a universal solver are serious challenges which drive cutting-edge research to this day. Moreover, efficient solution of these systems is critical in many areas of applied mathematics, as solving the linear system often constitutes the most time-consuming phase of a simulation [1, 13, 29, 31].

This thesis focuses on massively large problems -involving hundreds of millions or even billions of degrees of freedom- in which parallelization techniques are inevitable even for the most efficient algorithms. To this day, AMG is among the most efficient numerical solvers for sparse linear systems of algebraic equations [1, 29, 31]. It has two fundamental parts: the setup phase, in which many of its components are defined and structured, and the solution phase, in which these components come together to actually solve the problem at hand. One of the fundamental steps in the setup phase is the so-called "coarsening strategy selection", in which, as its name suggests, a subset of grid points is selected for membership of the subsequent grid in the hierarchy. The choice of coarsening strategy strongly affects perfomance and scalability -see Subsection 2.4- and thus choosing an optimal coarsening strategy is of utmost importance.

This thesis addresses this challenge by employing several ML methods -namely multinomial logistic regression, support vector classification, k-nearest neighbors classification, random forest classification, and extreme gradient boosting classification-, trained on a substan-

tial amount of generated data, to predict the optimal coarsening strategy. The candidate strategies considered include standard coarsening, two types of aggressive coarsening, CLJP, PMIS, and HMIS. Existing work on ML-based algorithmic tuning for parameter optimization includes the use of classical genetic algorithms for linear systems arising from reservoir simulations [7, 21], tree-based approaches to narrow down the parameter space of genetic algorithms [8, 15], and surrogate models to incorporate information from previous simulation runs into a genetic algorithm accelerated with decision trees [25].

The present work is organized as follows: Section 2 presents a technical overview of AMG, with an emphasis on massively parallel coarsening strategies, namely CLJP, PMIS, and HMIS. Subsection 2.1 begins with a mathematically thorough exposition of its formal components, while Subsections 2.2 and 2.3 provide a detailed exposition of all coarsening strategies involved. A short summary of all the presented coarsening strategies is given in Subsection 2.4, while in Subsection 2.5 all of the exposed coarsening strategies are tested on different problems in order to understand their advantages and uses as problem sizes and processor counts grow.

Section 3 presents a thorough theoretical description of all ML methods used, starting with a strategy overview in Subsection 3.1 which covers how data is created, pre-processed, used for training and validation, and how the ML methods are trained and used to make predictions. Subsection 3.2 provides a detailed mathematical exposition of all algorithms and statisticall aspects of the ML methods involved, with its corresponding summary being presented in Subsection 3.3.

Section 4 presents my implementation of these ideas in Fraunhofer's *Schnelle Löser* team's code and my findings using this approach. Subsection 4.1 deals with matrix properties and learner selection. Subsection 4.2 presents the results of comparing this method both with Fraunhofer's most novel ML method -called Autonomous Solver Control- and with AMG's default coarsening settings, while Subsection 4.3 deals with the actual implementation workflow.

We finalize this work with some conclusions and outlooks in Section 5.

2 Algebraic Multigrid

Algebraic Multigrid (AMG) is an iterative technique which creates an algebraic coarse representation of the geometrical discretization -so-called "grids"- in order to solve large sparse linear systems of algebraic equations, say

$$Au = f \quad \text{or} \quad \sum_{j=1}^{n} a_{ij} u_j = f_i \quad \text{for} \quad i \in \{1, \ldots, n\} \tag{2.1}$$

with solution vector $u \in \mathbb{R}^n$, a sparse, symmetric, positive definite M-matrix $A \in \mathbb{R}^{n \times n}$, and a right hand side vector $f \in \mathbb{R}^n$ [10]. While these conditions of the matrix A are required for the theory, in practical problems -such as those presented in Section 4- they are often violated. These systems originally arose from discretizations of elliptic partial differential equations, but AMG's range of applicability has extended far beyond this setting. In contrast with Geometric Multigrid (GMG), where the unknown variables u_i are defined on known spatial locations (grid points), AMG operates from a purely algebraic viewpoint –hence its name– based on the adjacency graph induced by the matrix A [1].

On the one hand, AMG can adapt itself to specific requirements of the problem, making it robust and flexible. On the other hand, this flexibility implies that a *setup phase* -where the coarse grids are constructed and all operators are assembled- must finish before the actual *solution phase* begins [29]. The present work focuses on the coarsening strategy of the setup phase, namely, the step where a subset of grid points is selected for membership of the subsequent grid in the hierarchy with the goal of balancing the robustness and solving speed of the solver.

In Subsection 2.1 we present the formal components of AMG in a mathematically thorough manner, with an emphasis on coarsening strategies employed in the coarsening phase. Subsections 2.2 and 2.3 are dedicated to expositions of the different families of coarsening strategies -namely classical coarsening strategies and natively parallel coarsening strategies-, including the complete algorithmic description of each coarsening method. Lastly, numerical results are presented in Subsection 2.4 in order to compare the different coarsening strategies exposed.

Throughout this thesis several diagrams will be shown representing coarsening procedures. In all of them, red nodes represent C-points -points chosen as members of the subsequent level of the grid-hierarchy-, blue nodes represent F-points -points which are not chosen as members of the subsequent level of the grid-hierarchy-, and white nodes represent undecided points -points whose membership of the subsequent level of the hierarchy has not yet been decided at that stage of the coarsening algorithm. These labels and concepts will be made clear to the reader as they appear in the following sections.

2.1 Formal Components

Following the exposition of AMG given in [1] and [29], we present AMG's complete functionality.
Let h and H correspond to fine and coarse levels, respectively. We begin by rewriting Equation (2.1) as

$$A_h u^h = f^h \quad \text{or} \quad \sum_{j \in \Omega^h} a_{ij}^h \, u_j^h = f_i^h \quad \text{for} \quad i \in \Omega^h, \tag{2.2}$$

where $\Omega^h = \{1, \ldots, n\}$, simply an index set, acts as a *fictitious grid*.

As stated earlier, AMG is constituted by two main phases: the setup phase and the solution phase.

2.1.1 Setup Phase

The setup phase of AMG is divided into three steps:

- Coarsening (or C/F splitting)
  The index set Ω^h is partitioned into C^h-points and F^h-points based on connectivity requirements.
- Construction of the interpolation (and restriction) operator I_h^H
  The interpolation operator I_h^H for transferring from coarse-level variables to fine-level variables is constructed, as well as the restriction operator I_H^h for transferring from fine-level variables to coarse-level variables.
- Calculation of the coarse-level operator A_H
  Based on the interpolation operator I_h^H and the corresponding restriction operator I_H^h, the coarse-level operator A_H is calculated as the so-called Galerkin product $A_H = I_H^h A_h I_h^H$ of interpolation and restriction.

Coarsening

The first step of the setup phase is coarsening, in which we aim to make a disjoint partition of the index set $\Omega^h = C^h \cup F^h$, where C^h is the set of coarse variables and $F^h := \Omega^h \setminus C^h$. There are several strategies to make this partition, each with different characteristics and performance impacts on the algorithm. Since the present work deals precisely with this topic, we refer the reader to Subsections 1.2 and 1.3 for a detailed exposition of coarsening strategies.

Construction of the interpolation and restriction operators

As previously mentioned, we wish to construct an interpolation operator I_h^H which maps coarse-grid vectors to fine-grid vectors, thus $I_h^H \in \mathbb{R}^{n_H \times n_h}$ where n_H, n_h denote the dimensions of the coarse- and fine-grid, respectively. Throughout this thesis we will take the restriction operator as the transpose of the interpolation operator i.e. $I_H^h = (I_h^H)^T$. This is a common procedure, as this ensures a symmetric coarse level operator A_H and, hence, a symmetric cycling in the solution phase [10]. The entries of the interpolation operator $I_h^H = (w_{i,j})_{i \in \{1,\ldots,n_h\}, j \in \{1,\ldots,n_H\}}$ are called *weights* and are specified as follows:

$$v_i^h = I_h^H v^H = \begin{cases} v_i^H, & \text{if } i \in C \\ \sum_j w_{i,j} v_j^H, & \text{if } i \in F \end{cases} = \begin{pmatrix} I_{HH} \\ I_{hH} \end{pmatrix} \tag{2.3}$$

where $v^H \in \mathbb{R}^{n_H}, v^h \in \mathbb{R}^{n_h}$ are coarse and fine grid vectors, respectively. This is called standard or direct interpolation [29].

In order to achieve convergence of the multigrid method, there are certain properties that the interpolation and restriction operators must have. We summarize these below, and refer the reader to [29] for their proofs.

Definition 2.1.1 *Consider Equation (2.2)*

$$A_h u^h = f^h$$

on an arbitrary coarse level h, and let v^h be an approximate solution vector. Let $r^h := f^h - A_h v^h$ be the residual and $e^h := u^h - v^h$ the error. An operator $\mathcal{L}$ satisfies the smoothing property with respect to a symmetric and positive definite matrix $A \in \mathbb{R}^{n \times n}$ if

$$||\mathcal{L} e^h||_1^2 \leq ||e^h||_1^2 - \sigma ||e^h||_2^2 \tag{2.4}$$

holds with $\sigma \in \mathbb{R}$ being independent of $e^h \in \mathbb{R}^n$.

This property essentially ensures that high-frequent error components are efficiently reduced by the smoother [10]. We wish to have an interpolation operator which effectively reduces low-frequency error components, since high-frequency error components are handled by the smoother. Denote by $\mathcal{T} := \mathbb{1}_h - I_H^h A_H^{-1} I_h^H$ the *coarse-level correction* operator. Our next theorem states that operator $\mathcal{T}$ should give an error that is suitable for post-smoothing [10].

Theorem 2.1.1 *Let A be a symmetric positive definite matrix and $\mathcal{L}$ be a smoothing operator which fulfills the smoothing property (2.4). If the C/F splitting and the interpolation operator fulfill*

$$||\mathcal{T} e||_1^2 \leq \tau ||\mathcal{T} e||_2^2$$

with $\tau > 0$ independent of $e \in \mathbb{R}^{n_h}$, then $\tau > \sigma$ and $||\mathcal{L}\mathcal{T}||_1 \leq \sqrt{1 - \frac{\sigma}{\tau}}$.

The interpolation formula (2.3) satisfies Theorem (2.1.1).

We refer the reader to [31] for a detailed description of different approaches to construct the interpolation and restriction operators.

Construction of the coarse-level operator

Defining $\Omega^H := C^h$ we can construct coarse-level AMG systems

$$A_H u^H = f^H \quad \text{or} \quad \sum_{j \in \Omega^H} a_{ij}^H u_j^H = f_i^H \quad \text{for} \quad i \in \Omega^H \tag{2.5}$$

as the Galerking product of the interpolation and restriction operator as follows

$$A_H := I_h^H A_h I_H^h \tag{2.6}$$

where I_h^H maps coarse-level vectors into fine-level ones and I_H^h maps fine-level vectors into coarse-level ones. By recursively applying a coarsening and a calculation of interpolation/restriction operators, a full matrix hierarchy of Galerkin operators $A_2, \ldots, A_{max}$ can be constructed based on the fine-level matrix $A_1 := A$, where A is the coefficient matrix from Equation (2.1) and A_{max} can be solved using a direct solver or a given stopping criterion has been reached [10].

2.1.2 Solution Phase

Two-Level Scheme

Due to the recursive nature of AMG, it is worth presenting a two-level correction scheme and then proceed recursively. Consider Equation (2.2)

$$A_h u^h = f^h$$

on an arbitrary level h, and let v^h be an approximate solution vector. Let $r^h := f^h - A_h v^h$ be the residual and $e^h := u^h - v^h$ the error. Our goal is to iteratively calculate u^h, for which we need to solve

$$A_H e^H = I_H^h r^h \tag{2.7}$$

for e^H on the coarse level H [10]. We then correct our approximate solution vector v^h by interpolating back to the fine grid h as

$$v^h = e^h + I_h^H e^H.$$

This so-called correction procedure, combined with a smoothing process on the fine level h -using an iterative solver such as Gauss-Seidel- is a powerful combination for approximating the solution of Equation (2.1): high-frequency error components are handled on the fine level, while low-frequency components are handled on the coarse level [10].

Multilevel Scheme

In order to solve Equation (2.7), we can apply the two-level scheme recursively on each constructed level $h \in \{h_1, \ldots, h_{max}\}$, where h_{max} denotes the coarsest level constructed. The resulting multilevel scheme is called V-cycle. We refer the reader to [31] for variations of the V-cycle -such as as the W-cycle or F-cycle-, and we present the V-cycle algorithm as in [1]:

Algorithm 1: V-cycle algorithm $\mathbf{V}(\mathbf{u}, \mathbf{f})$

Input: $A \in \mathbb{R}^{n \times n}$, $f \in \mathbb{R}^n$, initial guess $u_0 \in \mathbb{R}^n$
Data: smoother G, restriction I_H^h, prolongation I_h^H, maximum coarsening level h_{max}
Output: $u \in \mathbb{R}^n$ approximate solution of $Au = f$
if $h = h_{\max}$ **then**
  go to post-smoothing
else
  pre-smoothing: $u^h \leftarrow G(u^h, f^h, A_h)$
  restrict residual: $f^{h+1} \leftarrow I_{h+1}^h (f^h - A_h u^h)$
  initial guess on coarse level: $u^{h+1} \leftarrow 0$
  coarse grid correction: $u^{h+1} \leftarrow \mathbf{V}(u^{h+1}, f^{h+1})$
correct solution: $u^h \leftarrow u^h + I_h^{h+1} u^{h+1}$
post-smoothing: $u^h \leftarrow G(u^h, f^h, A^h)$
return u^h;

2.2 Classical Coarsening Methods

In order to achieve fast convergence, as shown by Theorem (2.4), a balance should be made between the number of coarse variables per level C^h -which impacts memory requierements- and a good-enough approximation by the interpolation operator, which is largely impacted by the $F-\text{to}-C$ connectivity. Precisely how to achieve this balance and what is meant by *good-enough* $F-\text{to}-C$ connectivity is handled differently by different coarsening strategies. Classical coarsening methods are designed to work sequentially, making them suitable for small to medium-sized problems where paralellization is impractical or unnecesary [29]. The following exposition of classical coarening methods -namely Standard/Ruge-Stüben and Aggresive Standard/Ruge-Stüben coarsening- closely follow the exposition presented in [29].

For the sake of simplicity, we make the following exposition without explicitly mentioning the level index h. Let $\Omega = C \cup F$ denote the set of variables on the fine level, where C is the set of variables to be transfered to the coarse level and $F := \Omega \setminus C$ is the complementary set of fine variables which remain on the fine level [10]. For this reason coarsening strategies are also called C/F splittings. In the following exposition, $a_{i,j}$ are entries of the matrix $A \in \mathbb{R}^{n\times n}$ from Equation (2.1) for all $i, j \in \{1, \dots, n\}$.

In order to achieve said balance, we must introduce the concept of *coupled variables*.

Definition 2.2.1 *Let $\Omega = \{1, \dots, n\}$ be an index set and $i, j \in \Omega$. We say that i and j are coupled if $a_{i,j} \neq 0$.*

Moreover, we define the neighborhood of a point i as the set of all its couplings, namely:

Definition 2.2.2 *Let $\Omega = \{1, \dots, n\}$ be an index set and $i \in \Omega$. We define*

$$N_i := \{j \in \Omega : i \neq j, a_{i,j} \neq 0\}$$

as the neighborhood of i.

Furthermore, the concept of *strongly coupled variables* plays a central role in the description of most coarsening strategies:

Definition 2.2.3 *Let $\Omega = \{1, \dots, n\}$ be an index set and $i, j \in \Omega$. We say that i is strongly coupled to j if*

$$-a_{i,j} \geq \alpha \max_{a_{i,k}<0} |a_{i,k}| \tag{2.8}$$

for a fixed $\alpha \in (0, 1)$.

In practical applications $\alpha = 0.25$ has been established as a reasonable threshold value [10].

Definition 2.2.4 *We denote by S_i the set of strong couplings of i, namely*

$$S_i = \{j \in N_i : i \text{ is strongly coupled to } j\}$$

and by S_i^T the set of transpose strong couplings of i, namely

$$S_i^T = \{j \in \Omega : i \in S_j\}.$$

The couplings that are not strong are called weak couplings.

We are now in a position to present Standard/Ruge-Stüben coarsening.

2.2.1 Standard/Ruge-Stüben

The Ruge-Stüben coarsening aims to achieve a C/F splitting with the following properties [24]:

- (RS1) The coarse-level variables C form a maximal independent set of variables in terms of strong connectivity, i.e., two coarse-level variables have no strong coupling between them.

- (RS2) For every fine-level variable $i \in F$ and each strongly connected variable $j \in S_i$, the variable j is either a coarse-level variable or strongly coupled to another coarse-level variable.

(RS1) ensures that the set of coarse grid points C is not too large -thus handling memory resources- while (RS2) ensures interpolation accuracy. One might think of a greedy approach for ensuring both constraints are satisfied, as follows:

Algorithm 2: Greedy approach to Standard/Ruge-Stüben coarsening

$C \leftarrow \emptyset, F \leftarrow \emptyset$
while $\Omega \setminus (C \cup F) \neq \emptyset$ **do**
  Choose $i \in \Omega \setminus (C \cup F)$
  $C \leftarrow C \cup \{i\}$
  foreach $j \in S_i^T$ **do**
    $F \leftarrow F \cup \{j\}$
return C, F

The problem with the above algorithm is that one might end up with a non-uniform distribution of C and F variables, which is not desirable [29]. In order to overcome this problem, we need to make a better choice of each variable in C.

Definition 2.2.5 *We define the "measure of importance" λ_i of an undecided variable $i \in U := \Omega \setminus (C \cup F)$ as*

$$\lambda_i = |S_i^T \cap U| + 2|S_i^T \cap F|$$

where U is the set of undecided variables at any stage of the algorithm.

Notice that λ_i acts as a measure of importance of $i \in U$ to become a C variable: the more F and U points are in the set of transpose couplings of i, the more attractive the undecided variable i is to becoming a C-point. Notice also that λ_i can be computed globally only once at the beginning of the algorithm and locally updated throughout its execution [29], making it computationally inexpensive to incorporate to Algorithm (2). We now present the Ruge-Stüben coarsening algorithm:

Algorithm 3: Standard/Ruge-Stüben coarsening

Input: Ω
Output: a $C \setminus F$ splitting of Ω
$\Omega \leftarrow \emptyset, C \leftarrow \emptyset, F \leftarrow \emptyset$
foreach $i \in U$ **do**
  define λ_i
while $\Omega \setminus (C \cup F) \neq \emptyset$ **do**
  Choose $i \in \Omega \setminus (C \cup F)$ with max λ_i
  $C \leftarrow C \cup \{i\}$
  $U \leftarrow U \setminus \{i\}$
  foreach $j \in S_i^T \cap U$ **do**
    $F \leftarrow F \cup \{j\}$
    $U \leftarrow U \setminus \{i\}$
return C, F

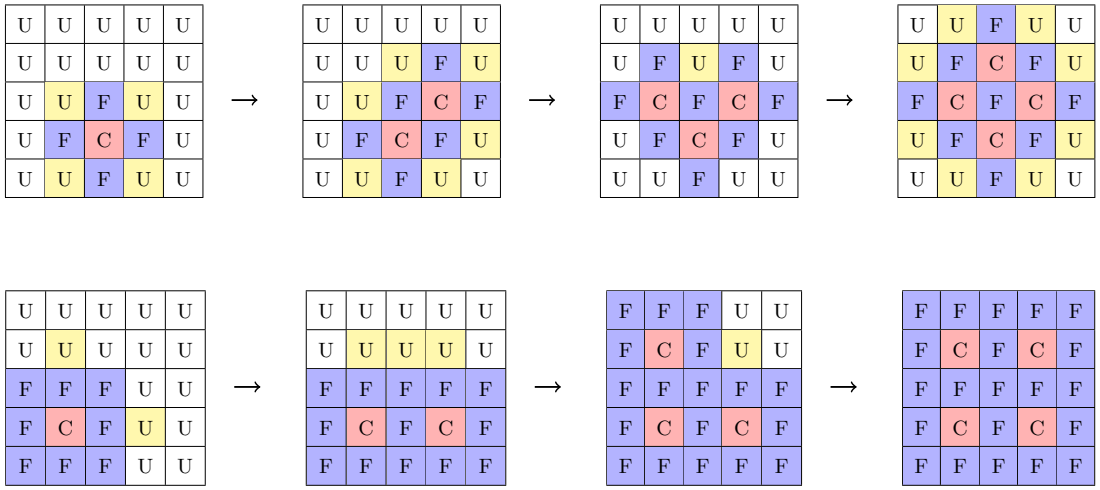


Table 2.1: First steps of the standard coarsening process in the case of isotropic 5-point (top) and 9-point stencils (bottom). At each stage, the undecided points with the highest λ value are shown in yellow. Adapted from [29].

2.2.2 Aggresive Standard/Ruge-Stüben

When dealing with small discretization stencils of PDEs, standard coarsening might cause a relatively high complexity in terms of memory requirement due to denser and denser Galerkin products [29]. In order to address this problem, aggressive standard coarsening deals with *long-range connections* instead of direct connections. With that being said, we present the following definitions:

Definition 2.2.6 *Let $\Omega = \{1, \ldots, n\}$ be an index set. A variable $i \in \Omega$ is said to be strongly connected to a variable $j \in \Omega$ along a path of length l if there exists a sequence of variables $i_0, \ldots, i_l$ with $i_0 = i$ and $i_l = j$ such that $i_{k+1} \in S_{i_k}$ for $k \in \{0, \ldots, l-1\}$.*

Furthermore, we present the concept of *strong connections along paths* as follows:

Definition 2.2.7 *Let $\Omega = \{1, \ldots, n\}$ be an index set and $i, j \in \Omega$. We define i to be strongly connected to j w.r.t (p, l) if at least $p > 1$ paths of length at most l exist such that i is strongly connected to j along each of these paths.*

Algorithm (3) immediately carries over if we apply it to the set

$$S_i^{p,l} = \{j \in \Omega : i \text{ is strongly connected to } j \text{ w.r.t. } (p, l)\}$$

In most applications, however, the cases $p \in \{1, 2\}$ and $l = 2$ turn out to be the most useful [29].

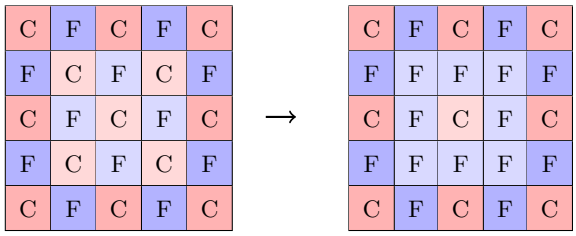


Table 2.2: Result of aggressive coarsening using $S_i^{2,2}$ with an isotropic 5-point stencil. Light colors depict the range of strong connectivity. Adapted from [29].

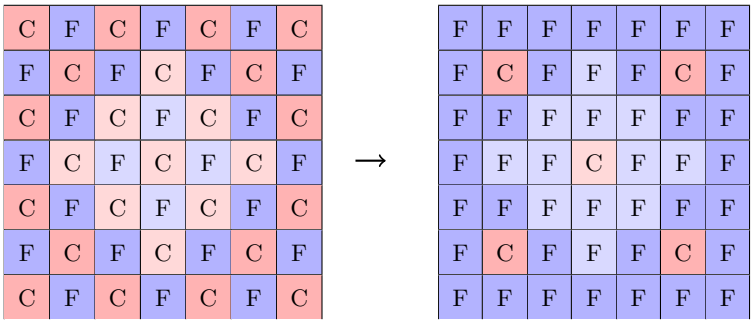


Table 2.3: Result of aggressive coarsening using $S_i^{1,2}$ with an isotropic 5-point stencil. Light colors depict the range of strong connectivity. Adapted from [29].

Numerically, this method's aggressiveness leads to a higher number of cycles in order to reach a given convergence threshold. This is due to the fact that strong connections are captured indirectly through paths rather than directly as in classical Ruge-Stüben coarsening, making the interpolation less accurate [29].

2.3 Natively Parallel Coarsening Methods

Many practical applications require solving extremely large linear systems of equations - involving millions or even billions of unknowns-, for which implementing AMG on massively parallel computers with distributed memory hierarchies becomes of utmost importance [13]. The coarsening methods presented above are of sequential nature, making them challenging to implement in parallel environments. As an alternative, coarsening strategies based on independent set algorithms have been recently developed: namely CLJP, PMIS and HMIS. The exposition of the CLJP strategy follows [13] closely, while the expositions of PMIS and HMIS follow [6].

2.3.1 Cleary–Luby–Jones–Plassman (CLJP)

The Cleary–Luby–Jones–Plassman (CLJP) parallel coarsening algorithm was proposed by Cleary et al. [5] and is based on parallel graph partitioning algorithms. It makes use of

the influence matrix S which is derived from the matrix A involved in the original linear system.

Definition 2.3.1 *The auxiliary influence matrix $S \in \mathbb{R}^{n\times n}$ is defined as*

$$S_{i,j} := \begin{cases} 1 & \text{if } j \in S_i \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

for any $i, j \in \Omega = \{1, \ldots, n\}$.

Notice that the i-th row of S is the incidence vector of S_i -the set of dependencies of i- while the i-th column of S is the incidence vector of S_i^T -the set of influences of i. We can then form the directed graph induced by S, where a directed edge from node i to node j exists only if $S_{i,j} \neq 0$. In order to determine which point has the most influences, we assign a weight to each point as follows

$$w(i) = |S_i^T| + r_i \tag{2.10}$$

for all $i \in \Omega$, where $r_i \in (0, 1)$ is a random number used to break ties. We then form an independent set $D \subseteq \Omega$ as follows

$$i \in D \iff w(i) > w(k) \quad \forall k \in S_i \cup S_i^T \tag{2.11}$$

i.e. a point i is in the independent set D if and only if its measure is larger than the measures of all the points that either influence or depend on i. Notice that D is an independent set by construction: if $i, j \in D$ and $(i, j) \in E$ then $w(i) > w(j)$ -by definition of D- and $w(j) > w(i)$ -since $i \in S_j \cup S_j^T$-, a contradiction. Once the independent set has been chosen, we modify the graph according to the following heuristics, which are meant to control the mesh size and quality of interpolation:

- (CLJP1) Neighbors that influence a C-point are less valuable as potential C-points themselves.
- (CLJP2) If k and j both depend on i, a given C-point, and j influences k, then j is less valuable as a potential C-point, since k can be interpolated from i.

Notice that, in this manner, all C points can be chosen without sequential passes through the graph, but rather by evaluating its local interactions. This makes CLJP, as opposed to Standard Coarsening, natively suitable for a parallel environment -where each processor handles a section of the mesh-.
For clarity, we expose the core ideas of of the algorithm, followed by its exposition as pseudo-code [13]:

1. Throughout the algorithm, edges are removed to indicate that certain influences have been taken into account.
2. Initially, we find an independent set in the graph G based on the assigned weights. All elements of the independent set become C-points -since they have the largest measure and are thus good candidates for interpolating the values of its neighbors.
3. Every node that strongly influences an element of the independent set (set of C-points) is less valuable as a C-point itself, so we lower its measure (heuristic (CLJP1)).
4. Ensure heuristic (CLJP2).
5. If at any point of the algorhtm the weight of a node is less than 1, label it as an F-point.

Algorithm 4: CLJP Coarsening - Theoretical Structure

Input: Ω, G directed graph of $S_{i,j}$
Output: a $C \setminus F$ splitting of Ω
$\Omega \leftarrow \emptyset, C \leftarrow \emptyset, F \leftarrow \emptyset$
while $\Omega \setminus (C \cup F) \neq \emptyset$ **do**
  Pick an independent set D of the graph G as in (2.11)
  foreach $i \in D$ **do**
    foreach j *that influences* i **do**
      decrement $w(j)$
      $S_{i,j} \leftarrow 0$ (remove edge (i, j) from the graph)
    foreach j *that depends on* i **do**
      $S_{j,i} \leftarrow 0$ (remove edge (j, i) from the graph)
      foreach k *that depends on* j **do**
        if k *depends on* i **then**
          decrement $w(j)$
          $S_{k,j} \leftarrow 0$ (remove edge (k, j) from the graph)
  if $w(j) < 1$ **then**
    $F \leftarrow F \cup \{j\}$
  $C \leftarrow C \cup D$
  Communication step required: each processor updates its weights w
  $G \leftarrow g(S)$
return C, F

The parallelization aspects demand more care on the so-called *halo areas*, which can be thought of as the boundaries of the processors.

Definition 2.3.2 *The halo area of processor p is defined as*

$$H_p = \{j \in \Omega : j \in S_i \cup S_i^T \text{ , } i \text{ belongs to processor } p\}$$

for all $p \in \{0, \ldots, N\}$, where N in the total number of processors.

One must take care of the necessary communication steps in order to efficiently implement CLJP coarsening in a parallel setting. Despite the description of the algorithm being simple, its implementation requires further work and can be cumbersome. For example, one of the main difficulties of its implementation is that nodes that depend on or influence each other may reside on different processors. Since the selection of F-points is based on weights -which reflect number of influences-, a flagging system must be used in order to keep track of visited nodes and their influences and dependencies. Overall, careful design of communication and bookkeeping mechanisms are essential to achieve a correct and efficient implementation of the CLJP algorithm. I implemented this algorithm in a parallel environment at Fraunhofer Institute for Algorithms and Scientific Computing (SCAI) as part of the *Schnelle Löser* team, addressing the aforementioned problems through careful ordering and iteration over the sets of influences and dependencies of elements in the independent set as follows:

Algorithm 5: CLJP Coarsening - Practical Implementation

Input: Ω, $G = (\Omega, E)$ directed graph of $S_{i,j}$
Output: a $C \setminus F$ splitting of Ω
$\Omega \leftarrow \emptyset, C \leftarrow \emptyset, F \leftarrow \emptyset, H \leftarrow \emptyset$ an auxiliary set
foreach $i \in \Omega$ **do**
  if $i \in D$ **then**
    $C \leftarrow C \cup \{i\}$
    foreach $j \in S_i$ *s.t.* $(i,j) \in E$ **do**
      $E \leftarrow E \setminus \{(i,j)\}$
      $w(j) \leftarrow w(j) - 1$
  else
    foreach $j \in S_i$ **do**
      if $j \in C$ **then**
        if $(i,j) \in E$ **then**
          $E \leftarrow E \setminus \{(i,j)\}$
        $H \leftarrow H \cup \{j\}$
      foreach $k \in S_j$ **do**
        if $k \in H$ **then**
          if $(i,j) \in E$ **then**
            $E \leftarrow E \ \{(i,j)\}$
          $w(j) \leftarrow w(j) - 1$
          break
  foreach $j \in S_i$ **do**
    $H \leftarrow H \setminus \{j\}$
return C, F

As stated in [13], the main advantage of CLJP coarsening is that it is entirely parallel, and results in the same selection of coarse-grid points (given the same global set of random numbers) regardless of the number of processors involved [6]. Experience shows that CLJP tends to select more C points than, for example, Standard Coarsening, leading to large memory complexities [13]. Heuristically, CLJP flags nodes as "potential F-points" by lowering their measure, rather than directly labeling them as F-points if certain connectivity requirements are met, resulting in a cardinality-wise smaller set of F-points (and thus a larger set of C points). This drawback, however, is complemented with its low time complexity on large problem sizes because of its parallel structure, as shown by the numerical experiments in Subsection 2.4.

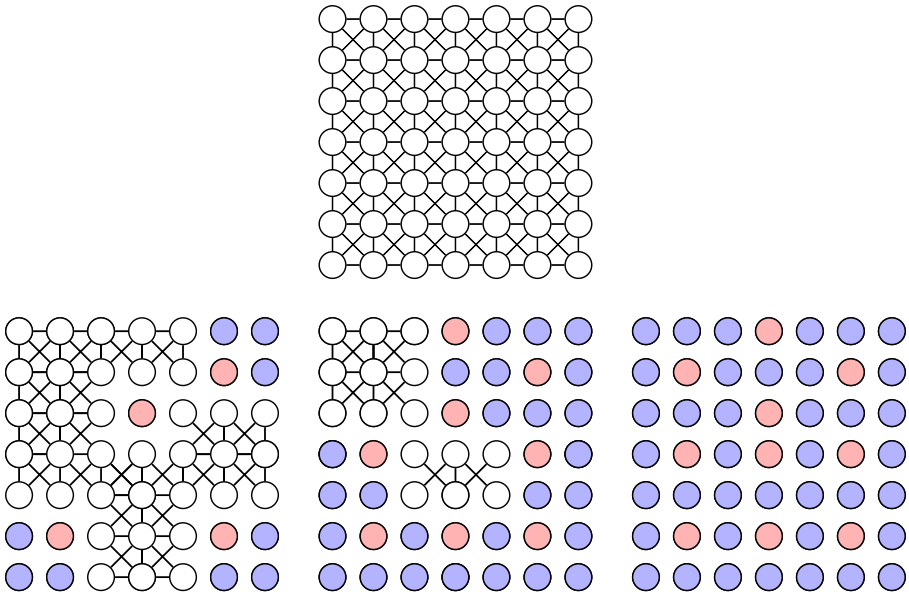


Table 2.4: Sequence of coloring steps using the CLJP algorithm for the nine-point Laplacian on a uniform grid. Top-center: the original dependence graph; bottom: application of heuristics in lexicographical order. Adapted from [13].

2.3.2 Parallel Modified Independent Set (PMIS)

The Parallel Modified Independent Set (PMIS) coarsening strategy was proposed by Yang et al. [6] and is based on Luby's parallel maximal independent set algorithm [19]. Like CLJP, it makes use of the auxiliary influence matrix (2.9), the weights assigned to each node (2.10), and the construction of an independent set (2.11).

In this case, we create the undirected graph $g(S) = (V, E)$ from the influence matrix S as $V = \Omega$ and $E = \{\{i, j\} : S_{i,j} \neq 0 \text{ or } S_{j,i} \neq 0\}$ and adhere to the following heuristics [6]:

- (PMIS1) Each F-point needs to strongly depend on at least one C-point.
- (PMIS2) The set of C-points needs to form a maximal independent set in the induced graph of the matrix (with only strong connections retained in the graph).

(PMIS1) ensures a good interpolation of values at F-points by C-points, while (PMIS2) intends to strike a balance between having enough C-points -intended by the use of a maximal set- but not too many -enforcing it via an independent set.

Definition 2.3.3 *Given an undirected graph $G = (V, E)$ of an influence matrix S, we denote the subgraph $G' = (V', E')$ of G induced by the vertex set $V' \subset V$ as $g(V', S)$, where $E' = \{\{i.j\} \in V' \times V' : \{i, j\} \in E\}$.*

For clarity, we expose the core ideas of the algorithm followed by its pseudocode:

1. Initially, all points that do not influence any other point are made F-points -since they would not be useful for interpolation.
2. All elements of the independent set are labeled as C-points, since they have the largest measure and are thus good candidates for interpolating the values of its neighbors.

3. All neighbors that are strongly influenced by a newly assigned C-point are labeled as F-points, since we can interpolate their values from them -ensuring (PMIS1).

Algorithm 6: PMIS Coarsening

Input: Ω, $G = (\Omega, E)$ undirected graph of $S_{i,j}$
Output: a $C \setminus F$ splitting of Ω
$\Omega \leftarrow \emptyset, C \leftarrow \emptyset, F \leftarrow \{i \in \Omega : |S_i^T| = 0\}, V \leftarrow \Omega \setminus F$
while $V \neq \emptyset$ **do**
  Choose an independent set D of V as in Equation 2.11.
  $C \leftarrow C \cup C_{new}$ where $C_{new} = D$
  $F \leftarrow F \cup F_{new}$ where $F_{new} = \{j \in (V \setminus D) : \exists\, i \in D \; s.t. \; i \in S_j\}$
  $V \leftarrow V \setminus (C_{new} \cup F_{new})$
  $G = g(V, S)$
return C, F

While (PMIS1) is strictly ensured by point 3, (PMIS2) is not strictly fulfilled: by turning an isolated F-point into a C-point the maximality of the set is violated while preserving independence; moreover, since the notion of dependence and influence is not symmetric, it might be the case that the resulting set of C-points is not independent [6]. In many applications, however, such notions are indeed symmetric and in general these slight violations of heuristic (PMIS2) represents no compromise for the resulting splitting.

It is worth noting that PMIS is naturally parallel and thus its parallelization is straightforward, only requiring communication steps for the so-called halo areas. The algorithm will also generate the same C/F-splitting independently of the number of processors and of the distribution of points per processor -given that the random numbers are independent of these characteristics and are the same for different runs [6].

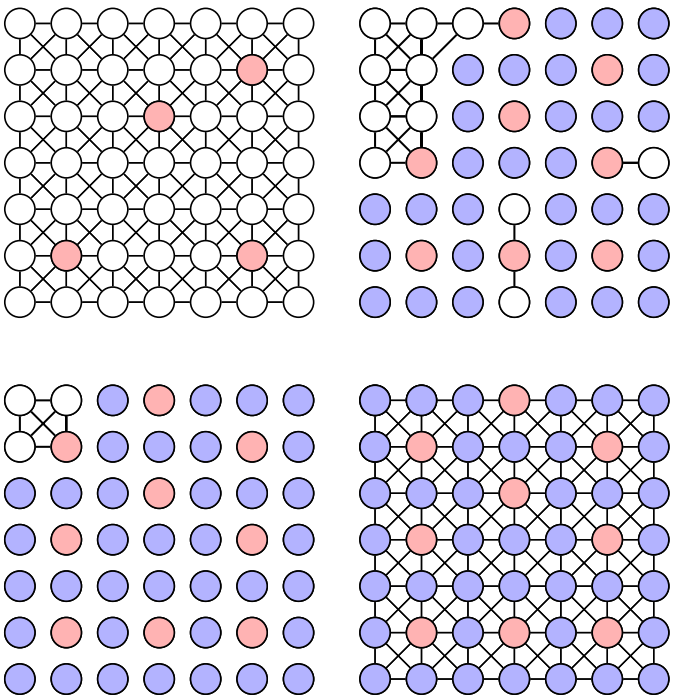


Table 2.5: PMIS coarsening performed for a 2D 9-point Laplace operator. Order of images: top-left, top-right, bottom-left, bottom-right. Adapted from [6]

2.3.3 Hybrid Modified Independent Set (HMIS)

Hybrid Modified Independent Set (HMIS) coarsening was proposed by Yang et al. [6] using ideas from Falgout coarsening [12]. The core idea of the algorithm is simple: first, apply classical Standard/Ruge-Stüben coarsening independently on each processor. Once completed, the C-points that are not in the halo area (or boundary) of any processor become the initial C-points of the PMIS algorithm, which is ran with such set as the initial set of C-points. The resulting coarsening scheme is called Hybrid Modified Independent Set (HMIS) coarsening scheme, since it combines two algorithms: one-pass of Standard coarsening on each processor, and the PMIS algorithm [6].

An example of the HMIS algorithm applied on a 2-D 5-point Laplace operator across 4 processors is shown below, with red nodes denoting C-points which were produced by the first pass of RS coarsening, and orange nodes denoting C-points which were produced by the PMIS algorithm in the second phase.

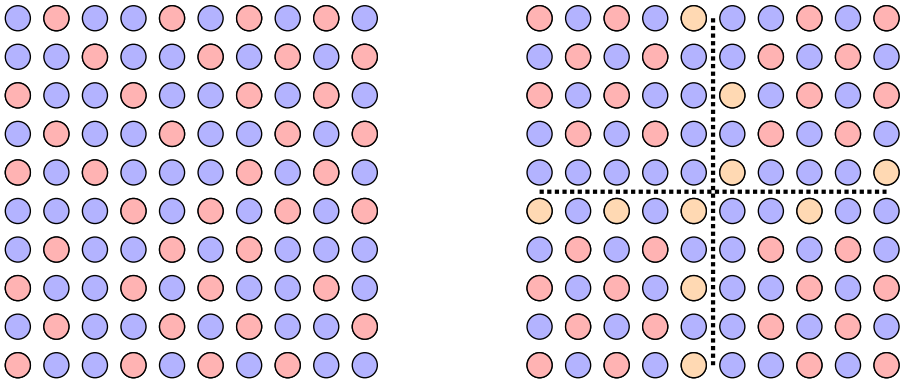


Table 2.6: PMIS (left) and HMIS (right) final coarsening after being applied on a 2-D 5-point Laplace operator across 4 processors. This example already shows that HMIS tends to produce coarser grids [6]. Adapted from [6].

2.4 Summary of Coarsening Strategies

For clarity of exposition and ease of consultation, the following table provides an overview of the different coarsening strategies presented in the current section. Based on this section's contents and closely following [9, 13, 29, 31], the main advantages and disadvantages of each coarsening strategy are summarized, providing the reader with a panorama of coarsening strategies.

Table 2.7: Summary of coarsening strategies.

Coarsening Strategy	**Type**	**Advantages**	**Disadvantages**
Standard / Ruge-Stüben	Classical	• Direct couplings provide good interpolation quality. • Easy to implement in serial.	• Challenging to parallelize. • High complexity on small stencils.
Aggressive Standard / Ruge-Stüben	Classical	• Reduces complexity on small stencils.	• Indirect couplings may reduce interpolation quality, increasing iteration counts.
Parallel Modified Independent Set (PMIS)	Natively Parallel	• Well-suited for parallel computations. • Easier to implement than CLJP.	• Can be less robust than Standard Coarsening.
Cleary-Luby-Jones-Plassmann (CLJP)	Natively Parallel	• Parallel coarsening independent of domain decomposition.	• Implementation complexity. • May choose too many coarse grid points.
Hybrid Modified Independent Set (HMIS)	Natively Parallel	• Combines advantages of Standard and PMIS.	• Requires multiple communication steps, adding overhead.

2.5 Numerical results

To better understand the effect of each coarsening strategy on the performance of AMG, as well as to highlight the different settings in which one might choose one coarsening strategy over another, we visualize their performance for increasing levels of parallelism on several problems.

As starting point, let us consider the Poisson problem

$$-\Delta u = 0 \quad \text{in } \Omega \tag{2.12}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{2.13}$$

where $u \in H_0^1(\Omega)$ is the unknown solution, $\Omega = [0, 1]$ is the unit square, and $\Delta u = \nabla \cdot \nabla u$ denotes the Laplace operator. The resulting linear system $Ax = b$ to be solved comes from the finite difference discretization of (2.12) when partitioning Ω in n_x, n_y regions in the vertical and horizontal axis respectively, so grid points are numbered $0, 1, \ldots, n_x$ and $0, 1, \ldots, n_y$ on each axis. Naturally, as n_x and n_y grow larger we obtain a finer grid and thus a larger matrix A, in which case more computational power is needed to solve the problem efficiently. The figures on the next page show the plots of processor count against the total runtime of AMG -keeping all aspects of AMG unchanged expect for the coarsening strategy- for different values of n_x and n_y.

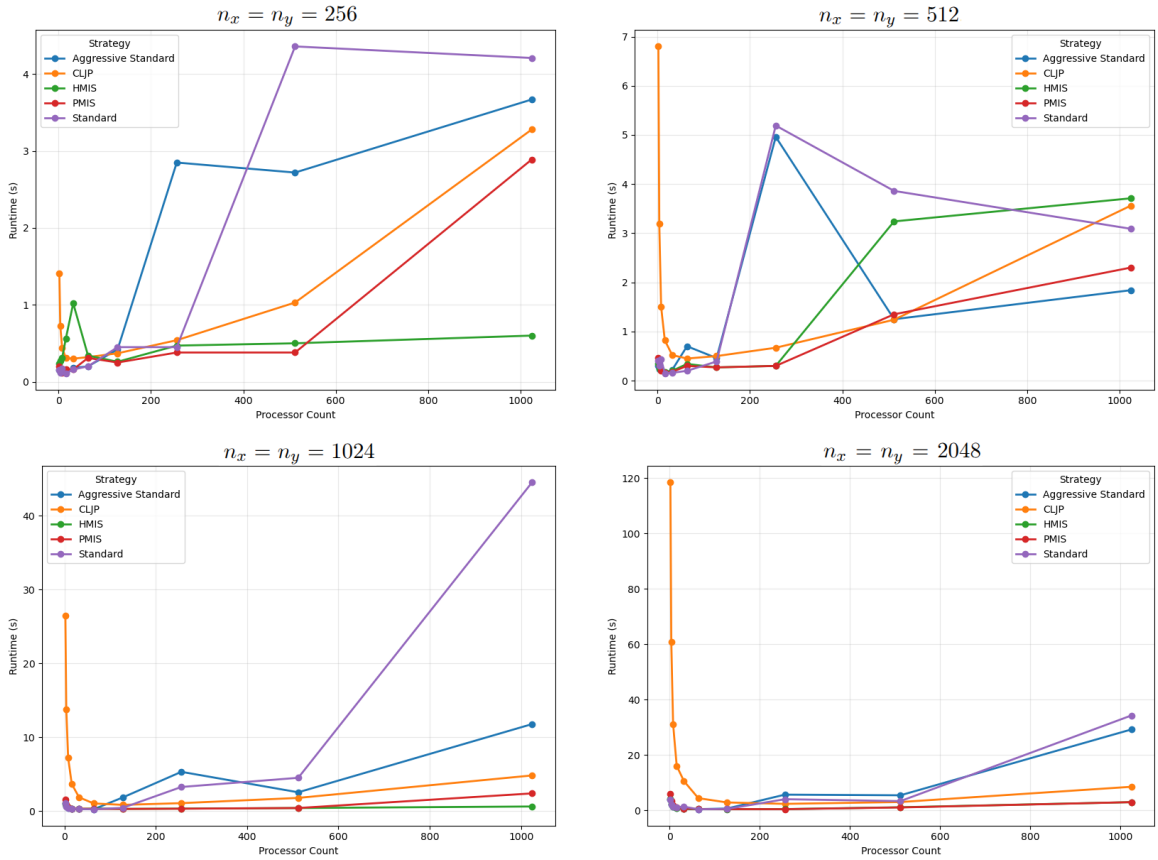


Figure 2.1: Comparison of coarsening strategies as grid size grows. In lexicographical order (top-to-bottom and left-to-right): $n_x = n_y = 256$, $n_x = n_y = 512$, $n_x = n_y = 1024$, $n_x = n_y = 2048$. In all cases the processor count takes values 2^n for $n = 1, \ldots, 10$ and runtime is measured in seconds.

Notice how for $n_x = n_y = 256$, $n_x = n_y = 512$ both the classical and the natively parallel coarsening methods achieve similar running times, while for $n_x = n_y = 1024$ and $n_x = n_y = 2048$ natively parallel coarsening methods largely outperform classical coarsening methods. In general, natively parallel coarsening methods outperform classical coarsening methods as the processor count grows -for large enough problem sizes. For smaller problem sizes, numerical aspects such as robustness of classical coarsening methods and convergence per cycle make them good contenders against natively parallel coarsening methods, where the advantages of parallelism become more pronounced as problem size grows. It is also worth noticing how CLJP coarsening (orange) achieves a drastically high running time for low processor counts, even when compared with other natively parallel coarsening methods. The reason behind this is that, as previously mentioned, CLJP tends to have a higher selection of C-points [6], having a coarse level which is about 75% of the fine level, while, for example, PMIS has a coarse level which is about 36% of the fine level, thus having to make less levels in order to achieve convergence, resulting in a faster running time. This phenomenon has to be accepted as it is inherit to the coarsening algorithm itself, and can not be controlled by a parameter on the fine-to-coarse level ratio. This same attribute suggests that CLJP might pay off in massive settings, where the parallelism of the algorithm outperforms the high fine-to-coarse level ratio, as shown in Figure (2.3).

We now consider four representative problems that arise in the context of physics and

engineering simulations:

- **Electro Magnetic (EM):** This problem models potential equalization and correction within the electromagnetic field of an electric engine, where the underlying physics is governed by Maxwell-type equations.
- **Groundwater North (GN):** Based on the "Modflow" simulation package by the U.S. Geological Survey (USGS), this problem involves multiple pressure corrections in a groundwater basin simulation in the context of the effects of wells, rivers, or other hydrologic stresses on an aquifer system [18].
- **Reservoir SPE10 (SPE10):** This problem is the standard benchmark test case in reservoir simulation, originally introduced by Christie and Blunt [4]. It features a highly heterogeneous permeability field and is visualized as a long bar-shaped domain, where challenge lies on the pressure correction problem within the complex reservoir geometry.
- **Meshfree Watercrossing (MW):** In automotive simulations, water–air interactions are studied to assess risks like water entering the engine or AC intake, vehicle soiling, and sensor contamination. This problem simulates a car driving through a pool of water in order to analyze real-world scenarios and manage risks of damage.

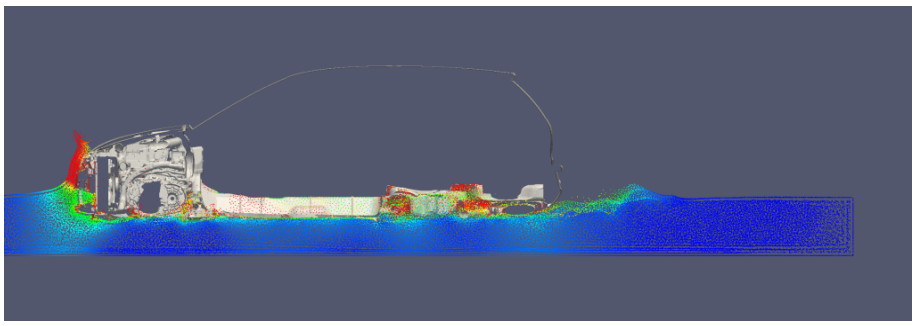


Figure 2.2: The MW problem simulates a car driving through a pool of water with the full engine compartment being modeled. Taken from [22].

Rather than the origin of the underlying PDE-discretized-system to solve, several mathematical properties of the resulting system $Ax = b$ become more relevant for the runtime and efficiency with which AMG solves the problem, as shown in Section 4. The following table summarizes some of the properties that characterize each problem:

Table 2.8: Matrix properties of benchmark problems

Property	**EM**	**GN**	**SPE10**	**MW**
Dimension (nnu)	$143,495$	$737,191$	$1,122,004$	$5,012,292$
Number of nonzero entries (nna)	$3,825,843$	$5,854,193$	$5,878,243$	$200,488,366$
Absolute average rowsum (relative)	0.27	0.7	0.3	0.11
Diagonal dominance over all rows (max)	7.3	2.6×10^4	1	2
L2 norm of diagonal vector (scaled)	5×10^{-14}	4.7	1.8×10^3	4.4×10^{-4}
Symmetry check (relative)	0	1	0.89	0.19

where the table properties are defined as:

- Dimension (nnu): dimension of the matrix A in the system to solve $Ax = b$.
- Number of nonzero entries (nna): number of nonzero entries of A.
- Absolute average rowsum (relative): $\frac{1}{nnu} \sum_{i=1}^{nnu} \sum_{j=1}^{nnu} \left| \frac{a_{ij}}{a_{ii}} \right|$
- Diagonal dominance over all rows (max): $\max_{i=1,\ldots,nnu} \left| \frac{\sum_{j=1, j\neq i}^{nnu} a_{ij}}{a_{ii}} \right|$
- L2 norm of diagonal vector (scaled): $\frac{||(a_{11},\ldots,a_{nnu,nnu})||_2}{nnu}$
- Symmetry check (relative): Let $E \in \mathbb{R}^{nnu \times nnu}$ be the matrix such that $e_{ij} = \frac{a_{ij} - a_{ji}}{a_{ii} + a_{jj}}$ for all $i, j = 1, \ldots, nnu$, $i \neq j$. Then the symmetrcy check (relative) of A is the Frobenius norm of E: $||E||_F := \left(\sum_{i=1}^{nnu} \sum_{j=1}^{nnu} |e_{ij}|^2 \right)^{\frac{1}{2}}$.

It is worth noticing that, despite the varying properties of each problem, natively parallel coarsening strategies outperform classical coarsening strategies in all cases as the problem size and processor count increase, as shown in the following plots:

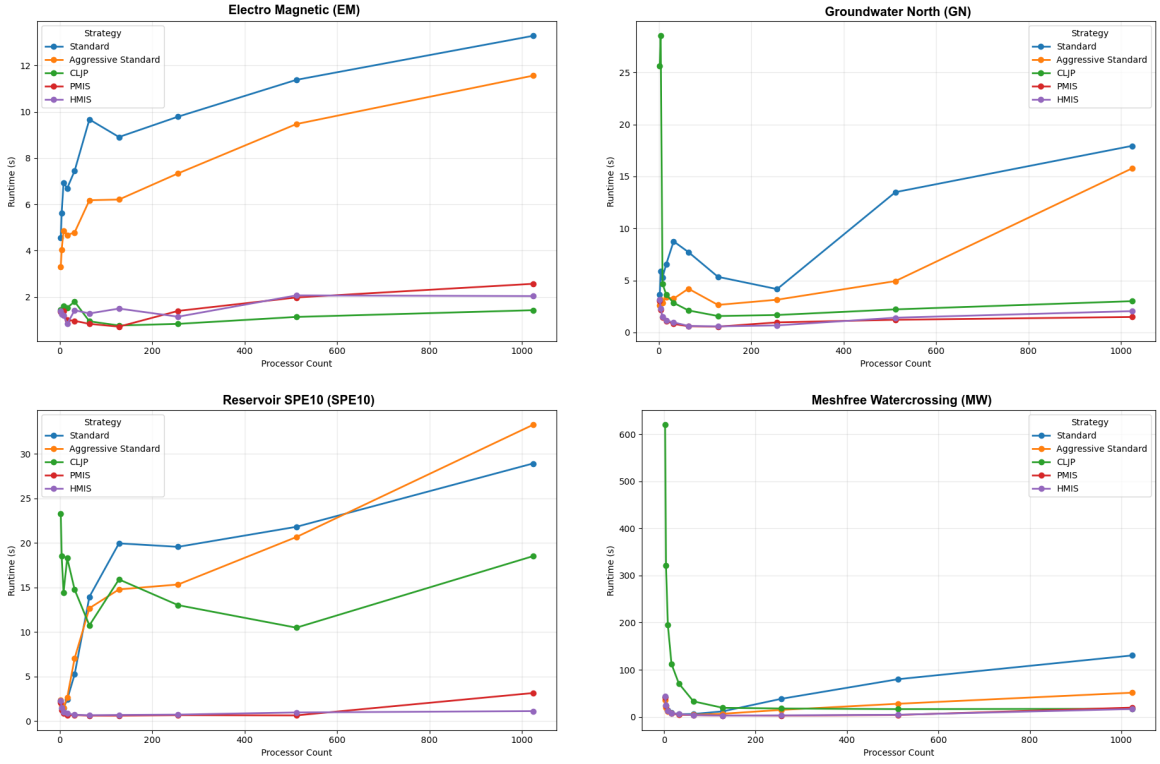


Figure 2.3: Comparison of running times for different coarsening strategies on benchmark problems. In lexicographical order (top-to-bottom and left-to-right): Electro Magnetic, Groundwater North, Reservoir SPE10, and Meshfree Watercrossing. In all cases the processor count takes values 2^n for $n = 1, \ldots, 10$ and runtime is measured in seconds. Despite the wildly different properties of each problem, natively parallel coarsening strategies (shown in green, red, and purple) outperform classical coarsening strategies (shown in blue and orange) as the processor count increases.

3 Machine Learning Methods for AMG

The past decade has seen substantial progress in Artificial Intelligence (AI), making it a key player in the digital era in almost every domain of human knowledge. The rapid development of AI has largely been due to the development and refinement of Machine Learning (ML) techniques [25]. Machine Learning can be defined as the process of gathering data, algorithmically building a mathematical model based on that dataset, and using this model to try to predict an outcome. In a typical scenario we have an *outcome* that we wish to predict based on a set of *features*. We have a *training set* of data, in which we observe the outcome and feature measurements for a set of objects. Using this data we build a *prediction model*, or *learner*, which will enable us to predict the outcome for new unseen objects. A good learner is one that accurately predicts such an outcome [11]. Previous work on applying ML techniques to optimize AMG's performance includes using single graph neural networks to learn prolongation operators [20], reinforcement learning to develop novel coarsening strategies [30], and genetic algorithms combined with tree-based accelerators for optimal parameter selection [25].

In this work we have used several ML methods to try to predict the most efficient coarsening strategy -in terms of runtime- for a given matrix A and a right-hand-side vector f given as in Equation (2.1). For clarity, we proceed to outline the elements of the ML models used in the present work:

- *Outcome (to predict): Coarsening Strategy based on lowest resulting runtime when running AMG (Standard, Aggressive Standard, CLJP, HMIS, PMIS).*
- *Training set:* A total of 1174 large sparse *M-like* matrices A.
- *Features (measured for prediction): Properties of the matrix A (Diagonal dominance, row sums, symmetry check, Frobenius norm, percentage of rows with higher row sums than the average of their neighbors, etc.).*
- *Prediction models: For each problem, several ML models were trained and the one with the highest accuracy percentage was chosen as the learner (Multinomial Logistic Regression, Support Vector Classification, K-Nearest Neighbors, Random Forest Classification and Extreme Gradient Boosting).*

Each stage of the above workflow requires careful care in order to obtain meaningful, consistent, and accurate results on unseen data. We proceed to give an overview of the strategy employed, as well as a thorough description of each step of the workflow.

3.1 Strategy Overview

In order to successfully implement a ML model to predict the best coarsening strategy, several processes and sub-processes need to be taken care of. Our problem is a *classification problem*, in which, given a matrix A and a right-hand-side vector f as in Equation (2.1), we aim to classify A into one of the following coarsening strategies: Standard, Aggressive Standard, CLJP, HMIS or PMIS coarsening. The overall strategy is divided in the following steps:

1. **Data Preprocessing**

   (a) **Creating the training data**
   We use CSR formatting and parallelization techniques in order to efficiently create thousands of sparse M-like matrices with millions of degrees of freedom.

   (b) **Measuring matrix features**
   We measure all relevant features of the created matrices in order to have a full mathematical description of each of them.

   (c) **Balancing the data set**
   An unbalanced data set might lead to misleading results or biased models. With that being said, it is desirable to have approximately the same number of matrices that fall in each coarsening category -i.e., matrices whose optimal coarsening strategy is the same. However, producing a matrix that will fall under a certain category *a priori* is a difficult task, since that is precisely the problem that we are trying to solve. Because of this, statistical tests were employed with the expectation that matrices with similar features fall under the same coarsening category. Hundreds of matrices were created based on this statistical information, tested on all coarsening strategies, and classified according to the fastest resulting coarsening strategy for each processor count, successfully balancing the data set.

2. **ML model training and prediction**

   (a) **Data splitting**
   In order to train the models, data should be randomly split into training data and target data. This splitting is not the same for any two runs, making the model robust and independent of the data splitting.

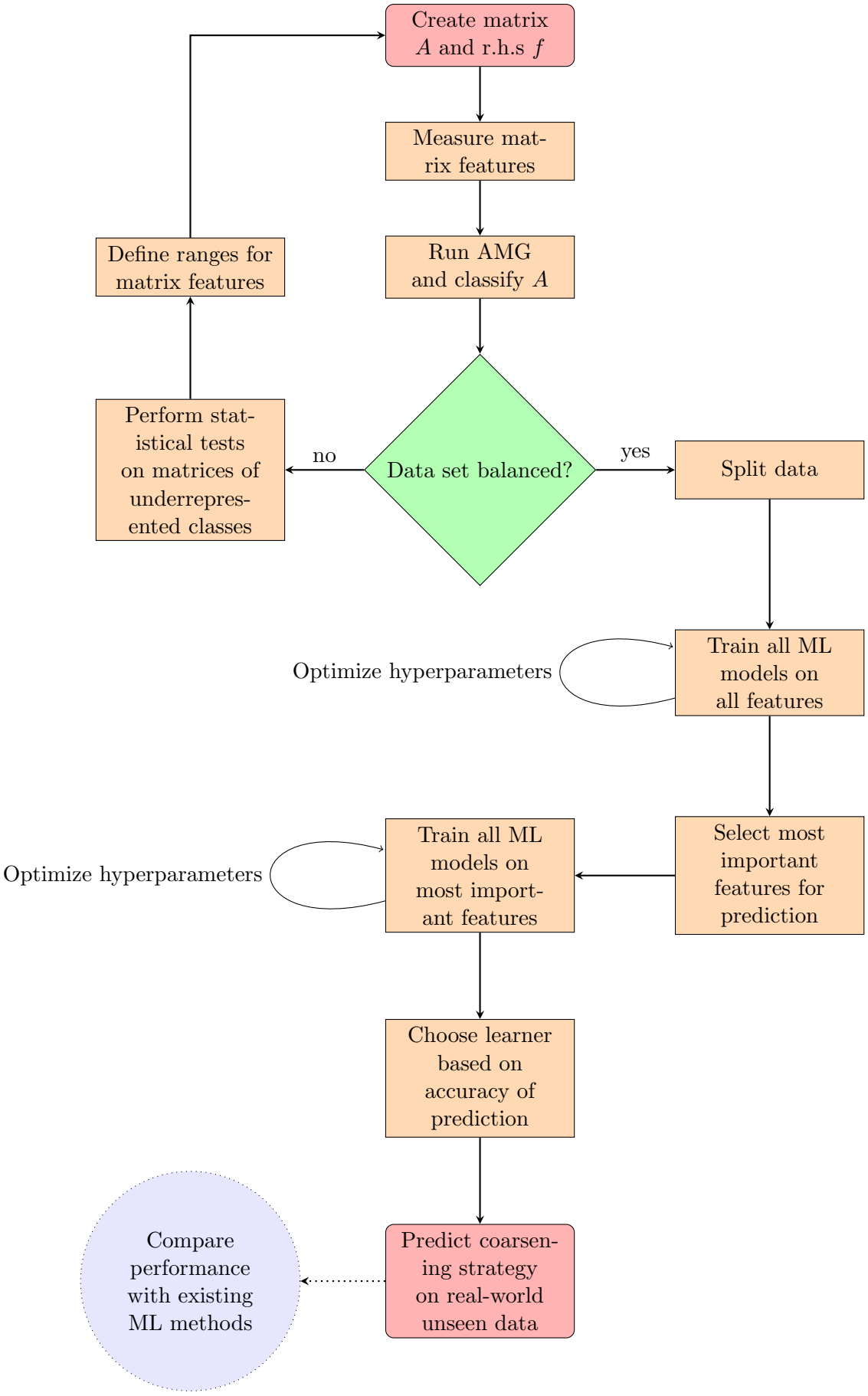
   (b) **Training all models on all features**
   We aim to measure only a small subset of features in order to predict a coarsening strategy, but such subset is unknown. We therefore need to run the models on all features and perform cross-validation, mutual information analysis, and other statistical tests to extract the most relevant features for prediction. Before each run, the hyperparameters of each model are optimized using Bayesian Optimization.

   (c) **Training all models on most relevant features**
   Once the most important features have been detected, we train all models using only those features and choose as learner the one with the highest prediction accuracy.

   (d) **Predicting coarsening strategy on real-world unseen data**
   The model is now fully trained and ready to predict a near-optimal coarsening strategy on real-world unseen data. Notice that the learner is run-dependant and that the predicted coarsening strategy might not be optimal for each run.

3. **Comparison with existing ML techniques**
   The previous workflow was designed and coded by me as part of the *Schnelle Löser* team at Fraunhofer SCAI, which already had an autonomous tool to accelerate the solution of systems of linear equations. Therefore, a comparison with the existing method is relevant and necessary, and we refer to the reader to Subsection 4.2 for the results of such comparison.

   For clarity, the above workflow is illustrated in the diagram on the next page.


Create matrix A and r.h.s f
Measure matrix features
Run AMG and classify A
Data set balanced?
no
yes
Define ranges for matrix features
Perform statistical tests on matrices of underrepresented classes
Split data
Optimize hyperparameters
Train all ML models on all features
Select most important features for prediction
Optimize hyperparameters
Train all ML models on most important features
Choose learner based on accuracy of prediction
Predict coarsening strategy on real-world unseen data
Compare performance with existing ML methods

3.2 Theoretical Background

As mentioned in the previous section, the problem of assigning a given matrix A which satisfies Equation (2.1) -for a given vector f- a corresponding coarsening strategy which optimizes the runtime of AMG is a *classification* problem, since it involves assigning each matrix to a category -namely Standard, Aggressive, CLJP, HMIS or PMIS- [14]. Because of this, all of the methods employed in the present work -namely Multinomial Logistic Regression, Support Vector Classification, K-Nearest Neighbors, Random Forest Classification and Extreme Gradient Boosting- are classification methods and thus adequate for our problem. They all belong to the class of *supervised learning models* since the algorithms learn to map input data to a specific output category based on example input-output pairs [14]. They are also called *probabilistic classifiers* since they compute the probability that a matrix A belongs to a certain class.

All ML classifiers used in this work require a training data set of m input-output pairs $(x^i, y^i)_{i=1,\dots,N}$, $x^i \in \mathbb{R}^d, y^i \in \mathbb{R}^K$ -in our case $x^i = (x^i_1, \dots, x^i_d)$ corresponds to the vector of d measured features of the i-th matrix A_i and y^i corresponds to vector such that $y^i_c = 1$ if and only if c is the correct class for x^i, $c \in \{1, \dots, K\}$- and have the following components [16]:

1. A feature representation of the input
   For each data point $x^i \in \mathbb{R}^d$ its feature representation is simply the vector of measured features $(x^i_1, \dots, x^i_d)$.

2. A classification function
   A function $\hat{y} \in [0, 1]$ that calculates the estimated class via the conditional probability $\hat{y}_i := p(y_i = 1|x^i)$.

3. An objective function
   A function that is optimized for learning, usually involving minimizing a loss function that measures the error on training examples.

4. An algorithm for optimizing the objective function
   Usual examples include gradient descent or Quasi-Newton optimizers.

Throughout this section we denote numbering indices as upper indices -such as $x^i \in \mathbb{R}^d$ denoting the i-th data input vector- while lower indices are used for entries of vectors -such as $x^i_j \in \mathbb{R}$ denoting the j-th entry of the i-th data input vector. Upper indices might be ignored when the vector is implied by context or irrelevant for the ongoing discussion.

3.2.1 Bayesian Optimization

All ML methods have parameters and hyperparameters which need to be tuned in order to achieve optimal performance. While parameters are estimated from the data automatically, hyperparameters need to be set by the user before the learning process begins, and thus require careful consideration. Until recently, such tuning was based on expert experience, inference, rules of thumb, or even brute-force search; but the growing interest and applicability that ML methods have had in the past decade has forced the scientific community to develop automatic approaches to such problems. In the present thesis, we have used the well known method of Bayesian Optimization in order to achieve optimal -or near-optimal- performance of the ML methods described in this section, and thus expose

its theory and implementation. It should be noted that Bayesian Optimization is not one of the ML methods used for predicting an optimal coarsening strategy, but rather a tuning algorithm which has been employed for optimizing the hyperparameters of all relevant ML methods used.

Much of the mathematical theory of optimization has been devoted to the problem of optimizing a nonlinear function $f(x)$ over a compact set $\mathbb{C} \subset \mathbb{R}^n$. However, in the context of ML, such assumptions generally do not hold and even evaluating $f(x)$ is extremely expensive or impossible, and thus other techniques have to be employed. Bayesian Optimization is a strategy used to address problems in which we do not have a closed form of the function $f(x)$ -nor of its derivatives- but where one can obtain certain evaluations -possibly noisy- of this function at sampled values [2]. Its strategy is to incorporate prior belief about the problem to help direct the sampling, and to balance exploration -where the objective function has high uncertainty- and exploitation -trying values of x where the objective function is expected to be high- of the search space.

Formally, given a "black-box" function $f : \mathcal{X} \to \mathbb{R}$ -meaning we do not have a mathematical expression of f nor of its derivatives-, we aim to find

$$x_* \in arg\max_{x \in \mathcal{X}} f(x) \quad \text{such that} \quad f(x_*) \geq f(x) \quad \text{for all} \quad x \in \mathcal{X}$$

where $\mathcal{X}$ is assumed to be a hyperrectangle of dimension d which denotes the space of hyperparameters. Let x_i be the i-th sample and $f(x_i)$ the observation of the objective function at x_i, and $\mathcal{D}_n := \{(x_i, f(x_i)) : i = 1, \ldots, n\}$ be the set of the first n observations.

Bayesian Optimization has two components:

1. The posterior distribution over the objective given by $p(f|\mathcal{D}_n) \propto p(\mathcal{D}_n|f)p(f)$ where $p(f)$ is a Gaussian process (GP). This posterior is implicitly used in Algorithm (7) when optimizing the acquisition function, i.e., in the step $x_t := \arg\max_x u(x|\mathcal{D}_n)$.

2. An acquisition function u which guides the search for the optimum of f.

The general algorithm for Bayesian Optimization can then be described as [2]:

Algorithm 7: Bayesian Optimization algorithm

Input: D_n, f.
Output: $x_* \in \arg\max_{x \in \mathcal{X}} f(x)$
for $t = 1, \ldots, T$ **do**
  Find $x_t := \mathbf{arg}\max_x u(x|D_n)$ by optimizing u over the Gaussian Process.
  Sample the objective function $f(x_t)$.
  Augment the data set $D_n = D_n \cup \{(x_t, f(x_t))\}$.

The number of iterations parameter T is based on practical considerations and pragmatic choices for balancing the trade off between search quality and runtime. Cheaper models or small spaces can use more trials -meaning a larger value of T-, while expensive trials use fewer -meaning a smaller value of T. In this thesis the value of T has been set to 30 for Random Forest Classification and Extreme Gradient Boosting, and to 100 for the remaining models, namely Multinomial Logistic Regression, Support Vector Classification, and K-Nearest Neighbors.

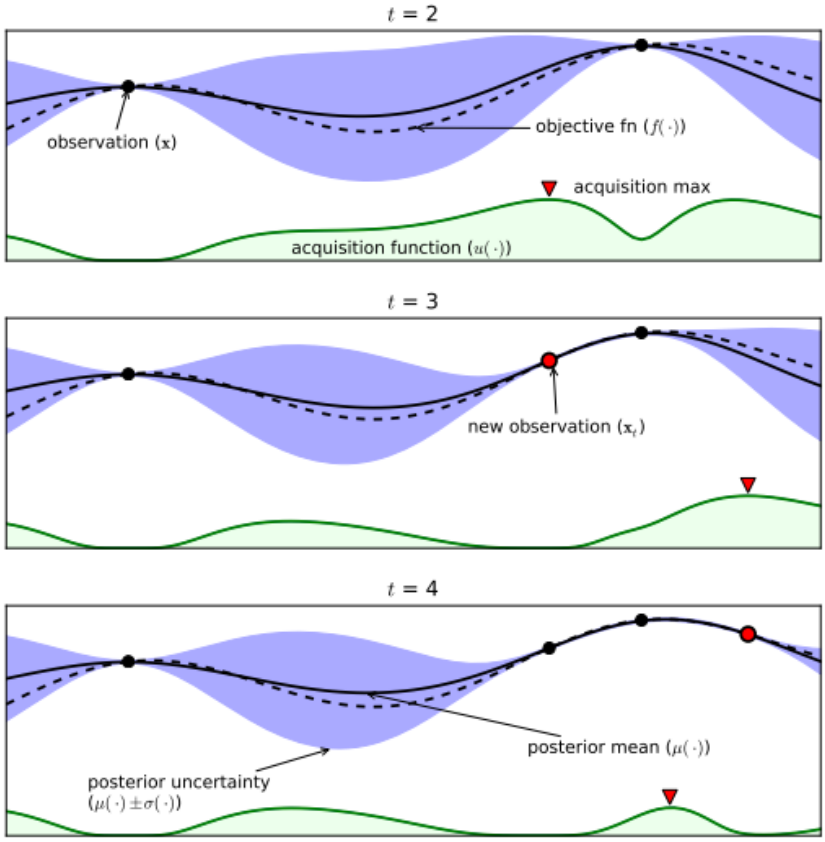


Figure 3.1: An example of using Bayesian optimization on a toy 1D design problem. The figures show a Gaussian process (GP) approximation of the objective function over four iterations of sampled values of the objective function. Note that the acquisition is high where the GP predicts a high objective (exploitation) and where the prediction uncertainty is high (exploration)—areas with both attributes are sampled first. Note that the area on the far left remains unsampled: although it has high uncertainty, it is (correctly) predicted to offer little improvement over the highest observation. Taken from [2]

How to obtain the prior distribution and which acquisition function to use lie at the core of the algorithm, and we thus turn our attention to these matters.

The prior distribution

It has proven by J. Mockus [2] that Gaussian Processes are well-suited as prior distributions for convergence of Bayesian optimization methods. A GP is an extension of a multivariate Gaussian distribution to an infinite dimensional stochastic process for which any finite combination of dimensions will be a Gaussian distribution [2], and thus is completely specified by its mean function $m(x)$ and covariance function $k(x_i, x_j)$:

$$f(x) \sim GP(m(x), k(x_i, x_j))$$

where common choices of mean and covariance functions are the zero function $m(x) = 0$ and the squared exponential function $k(x_i, x_j) := exp(-\frac{1}{2}||x_i - x_j||^2)$. Notice that $k(x_i, x_j) \sim 1$ if and only if $||x_i - x_j||^2 \sim 0$, meaning that points that are closer have a large influence on each other, while points that are distant have almost none -this is a necessary condition for convergence [2].

Suppose that we want to choose $x_{t+1} \in \mathcal{X}$ given D_n; let $f^t = (f(x_1), \ldots, f(x_t))$ and $f_{t+1} := f(x_{t+1})$. Then

$$\begin{bmatrix} f^t \\ f_{t+1} \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^T & k(x_{t+1,t+1}) \end{bmatrix}\right)$$

where

$$\mathbf{k} = [k(x_1, x_{t+1}), \ldots, k(x_t, x_{t+1})]$$
$$\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \ldots & k(x_1, x_t) \\ \vdots & \ldots & \vdots \\ k(x_t, x_1) & \ldots & k(x_t, x_t) \end{bmatrix}$$

and using the Sherman-Morrison-Woodbury formula [2] one can arrive at the following expression for the predictive distribution:

$$p(f_{t+1}|D_n, x_{t+1}) = \mathcal{N}(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1}))$$

where

$$\mu_t(x_{t+1}) = \mathbf{k}^T \mathbf{K}^{-1} f^t$$
$$\sigma_t^2(x_{t+1})) = k(x_{t+1}, x_{t+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k}.$$

The acquisition function

The role of the acquisition function u in Bayesian Optimization is to guide the selection of the next point x_t, where a high value of u typically corresponds to a potentially high value of f, thus choosing $x_t :=\text{argmax}_x\, u(x|D_n)$. While there are several choices of acquisition functions, we expose improvement-based acquisition functions since in the current thesis we have used a theoretical equivalent of maximizing these called TPE-samplers (see [23] for details).

The idea of improvement-based acquisition functions is to maximize the probability of improvement over the best current sampled point:

$$PI(x) := p(f(x) \geq f(x_m)) = \Psi\left(\frac{\mu(x) - f(x_m)}{\sigma(x)}\right)$$

where Ψ is the normal cumulative distribution function and $x_m = \arg max_{x \in D_n} f(x)$ is the best candidate so far. Notice, however, that $PI(x)$ favors only exploitation: we will always draw points that have a higher probability -potentially infinitesimal- of being better candidates, rather than drawing points that offer larger gains but less certainty [2]. In order to balance exploitation with exploration, the expected improvement is introduced:

$$\begin{aligned} EI(x) &= \mathbb{E}(max\{0, f(x_{t+1} - f(x_m))\}|D_n) \\ &= (\mu(x) - f(x_m))\Psi(Z) + \sigma(x)\phi(Z) \end{aligned}$$

where $Z := \frac{\mu(x) - f(x_m)}{\sigma(x)}$ and Ψ, ϕ denote the cumulative normal distribution and probability distribution functions respectively. Notice that $EI(x)$ strikes a balance between exploitation and exploration, since the first term depends on improving the mean μ -thus favoring exploitation-, while the second term depends on improving the variance σ -thus favoring

exploration-. The next point x_t is simply chosen as

$$x_t := \arg max_x EI(x)$$

Notice, also, that unlike the original objective function f, the acquisition function u can be cheaply sampled and optimized [2] making the choice of x_t in Algorithm (7) viable.

3.2.2 Multinomial Logistic Regression

Multinomial Logistic Regression (MLR) uses the so-called *softmax* function as classifier and typically the *negative log likelihood* as objective function [27]:

Definition 3.2.1 *Given $z \in \mathbb{R}^k$ we define the softmax function $\sigma : \mathbb{R}^k \to [0,1]^k$ as*

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for all $i \in \{1, \dots, k\}$.

MLR solves the classification task by learning, from a training set, a vector of weights $w^j \in \mathbb{R}^d$ and a bias vector $b^j \in \mathbb{R}^k$, one for each class $j \in \{1, \dots, k\}$. The i-th entry of any weight vector $w_i \in \mathbb{R}$ is associated with the i-th feature of any data point x_i and represents how important that input feature is to the classification decision [16]. Both the weights and the bias vectors are learned during the algorithm.

Definition 3.2.2 *Let $x \in \mathbb{R}^d$ be an input data vector with correct class $c \in \{1, \dots, k\}$, $w^c \in \mathbb{R}^d$ be the weight vector associated to c and $b^c \in \mathbb{R}^k$ be its bias vector. Let $\hat{y}, y :\in \mathbb{R}^k$ be vectors such that $y_c = 1$ and $y_j = 0$ for all $j \in \{1, \dots, k\} \setminus \{c\}$, and $\hat{y}$ is an estimate vector where $\hat{y}_k = p(y_k = 1|x)$. We define the negative log likelihood function $L : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ as*

$$L(\hat{y}, y) := -\log \sigma(w^c \cdot x + b^c) \tag{3.1}$$

Given a data point $x \in \mathbb{R}^d$, MLR calculates the probability of classifying x in the k-th category as

$$p(y_k = 1|x) := \sigma(w^k \cdot x + b^k) \tag{3.2}$$

for all $j \in \{1, \dots, k\}$. In this manner, the k-th entry of each vector $\hat{y}^i$ is precisely given by Equation (3.2) for all $i \in \{1, \dots, N\}, k \in \{1, \dots, K\}$. The weights $w^j \in \mathbb{R}^d$ and the bias vector $b^j \in \mathbb{R}^k$ are learned by minimizing the negative log likelihood function (3.1) -usually using gradient descent or Quasi-Newton methods- by using the training data vectors x^i and comparing them with their class assignment vector y^i [16]. In order to prevent over-fitting, we used an *L2 regularization term*

$$R(w) := \frac{1}{2C} \sum_{j=1}^{K} ||w_j||_2^2$$

of the weight vectors, where the parameter $C \in \mathbb{R}$ was optimized using Bayesian Optimization. Thus, the objective function to minimize becomes

$$J(w, b) := \frac{1}{N} \sum_{i=1}^{N} L(\hat{y}^i, y^i) + R(w)$$

which was achieved by the L-BFGS Quasi-Newton method. We refer the reader to [27] for further numerical and algorithmic details.

3.2.3 Support Vector Classification

Developed at AT&T Bell Laboratories by Vapnik and Chervonenkis in the 90's [28], Support Vector Classification (SVC) is a technique -one of the *Support Vector Machines* algorithms- which constructs a hyperplane or set of hyperplanes to classify data points into two distinct categories. However, our problem involves K classes -each one corresponding to a different coarsening strategy-, so we reduce the problem to many binary one-vs-one subproblems: after training all $\binom{K}{2}$ binary classifiers, we count the number of times that each vector x^i is assigned to each of the K classes; the final classification is performed by assigning it to the class to which it was most frequently assigned in the $\binom{K}{2}$ pairwise classifications [11].

We therefore present the SVM technique for the binary case, following the exposition from [28]:

Given input data vectors $\{x^i\}_{i=1,\dots,N} \subset \mathbb{R}^d$ and an associated class $y_i \in \{-1, 1\}$ for each, we wish to construct a separating hyperplane. However, in order to achieve robustness for individual vectors and better classification of most training vectors, we will allow for some input points to slightly violate the hyperplane constraints [11]. This gives rise to the so called *soft margin classifier* which is the solution to the optimization problem:

$$\max_{\beta_0,\dots,\beta_d,\epsilon_0,\dots,\epsilon_n} M \tag{3.3}$$

$$\text{s.t. } \sum_{j=1}^{p} \beta_j^2 = 1, \tag{3.4}$$

$$y_i(\beta_0 + \beta_1 x_1^i + \cdots + \beta_p x_d^i) \geq M(1 - \epsilon_i) \quad i = 1, \dots, n, \tag{3.5}$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C \quad i = 1, \dots, n, \tag{3.6}$$

where $C > 0$ is a tuning parameter which was optimized by Bayesian optimization on 100 trials. M is the width of the margin, which we seek to make as large as possible (over the mimunim distance from each vector to the hyperplane). Constraint (3.4) ensures that a hyperplane is searched and that $y_i(\beta_0 + \beta_1 x_1^i + \cdots + \beta_p x_p^i)$ is actually the perpendicular distance from the i-th vector to the hyperplane. Constraint (3.5) forces x^i to be on the correct side of the hyperplane, with some tolerance given by the size of ϵ_i, which in turn are bounded by the tuning parameter C by (3.6).

Although (3.3 - 3.6) has a clear interpretation, the standard form found in most textbooks -called *standard soft-margin SVC*- is attained by letting $w_i := \frac{\beta_i}{M}$ for all $i \in \{1, \dots, d\}$ and $b := \frac{b_0}{M}$. Then maximizing M is equivalent to minimizing $\frac{1}{2}||w||^2$, where w is the vector with entries w_i. Instead of the sharp constraint (3.6) we can penalize the size of ϵ_i in the objective, getting the following equivalent optimization problem:

$$\begin{aligned}
&\min_{w,b,\epsilon_1,\dots,\epsilon_n} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \epsilon_i \\
&\text{s.t.} \quad y_i(w \cdot x^i + b) \geq 1 - \epsilon_i \quad i = 1, \dots, n, \\
&\epsilon_i \geq 0 \quad i = 1, \dots, n
\end{aligned}$$

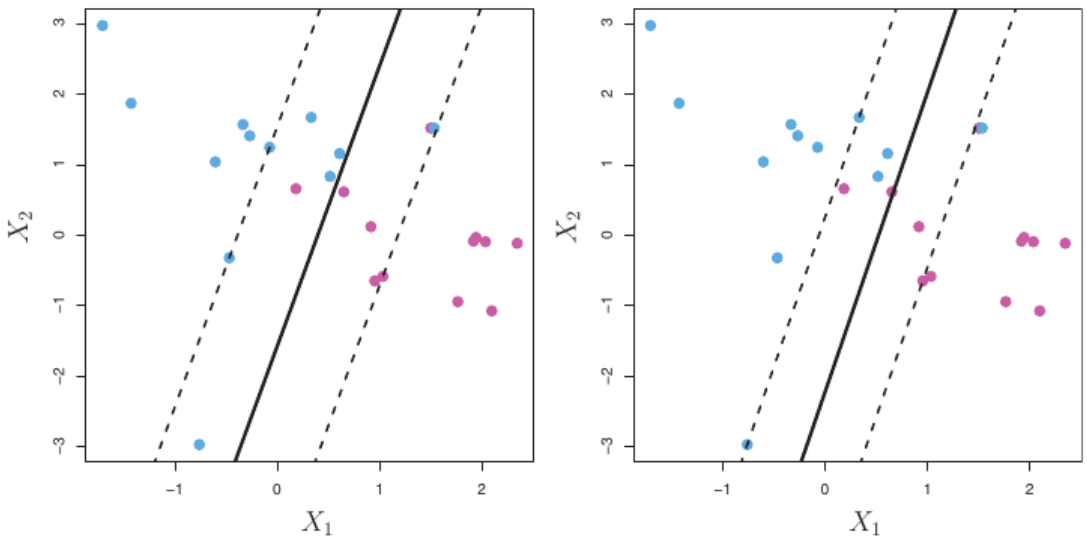


Figure 3.2: A support vector classifier was fitted using two different values of the tuning parameter C, depicted with a dashed line. The larger value of C is shown in the left panel, while the smaller value appears in the right panel. As stated in Equation (3.6), C controls the tolerance for data points to lie on the wrong side of the separating hyperplane. Taken from [14].

This formulation has the advantage that its dual via Lagrange multipliers can be solved efficiently [11]:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{n} \alpha_i - \tfrac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \left(x^i \cdot x^j\right) \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n, \\ & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned}$$

We refer the reader to [28] for computational and numerical details.

3.2.4 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is probably the most simple of the methods used, since it requires no model to be trained. Given input data $\{(x^i, y^i)\}_{i=1,\ldots,N}$, $x^i, y^i \in \mathbb{R}^d$ where $y_c = 1$ if and only if x^i belongs to the class $c \in \{1, \ldots, K\}$, and a real-world unseen data point x -which we wish to clasify-, we find the k training points x^i closest in Euclidean distance to x, and then classify using majority vote among these k neighbors [11]. Notice that parameter k is of utmost importance in the class assigned to x, and because of this we have optimized it via Bayesian Optimization before running KNN. Formally, the algorithm is as follows:

Algorithm 8: K-Nearest Neighbors algorithm

Input: $\{(x^i, y^i)\}_{i=1,\dots,N}$, $x^i, y^i \in \mathbb{R}^d$ where $y_c = 1$ if and only if x^i belongs to the class $c \in \{1, \dots, K\}$, x an unseen point to classify, $k \in \mathbb{Z}^+$ an optimized parameter

Output: $y \in \mathbb{R}^d$ where $y_c = 1$ if and only if x belongs to the class $c \in \{1, \dots, K\}$

Compute distances: $d_i := ||x - x^i||_2^2$ for all $i \in \{1, \dots, N\}$

Sort distances d_i so that d_{i_1} is the smallest distance and d_{i_N} is the largest distance.

Find indices of K nearest neighbors: $N_k := i_1, \dots, i_k$

Predict class: $c = arg\,max_i \sum_{i=1}^{d} \sum_{j \in N_k} y_i^j$

Arrange vector: $y_c = 1, y_i = 0$ for all $i \in \{1, \dots, K\} \setminus \{c\}$

return y

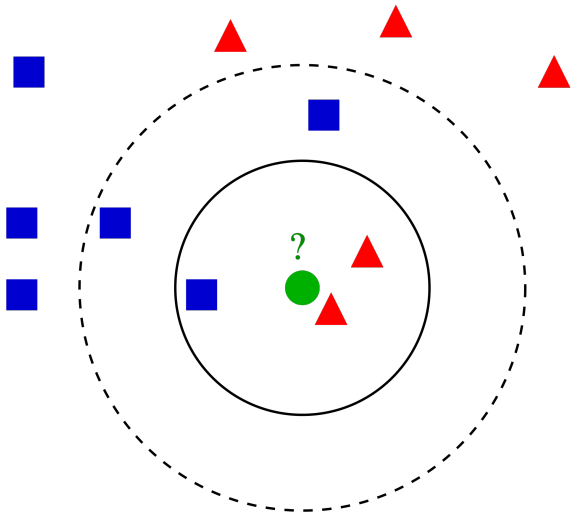


Figure 3.3: An example showing the importance of the parameter k. The unseen data point (green) should be classified either to blue squares or to red triangles. If $k = 3$ (solid line circle) it is assigned to the red triangles, whereas if $k = 5$ (dashed line circle) it is assigned to the blue squares. Taken from [32].

Despite its simplicity, k-Nearest Neighbors is often one of the most accurate classification algorithms of all the chosen in this thesis, as shown in Figure (4.1). It is ultimately based on the premise that objects with similar characteristics ought to fall under similar categories under classification.

3.2.5 Random Forest Classification

Random Forest Classification (RFC) algorithms were developed to correct for decision trees' habit of over-fitting to their training set by building several random trees with low correlation and applying a majority vote for the final classification prediction [11]. The idea behind random forests is quite simple: build several decision trees that divide the parameter space into non-overlapping regions, assign the majority category of each region to each unseen data point, and finally perform a majority vote over all trees for the final classification. Despite their simplicity, RFC methods are powerful tools for classification

tasks. The following exposition closely follows [26].

Given input data $\{(x^i, y^i)\}_{i=1,\dots,N}$, $x^i, y^i \in \mathbb{R}^d$ where $y_c = 1$ if and only if x^i belongs to the class $c \in \{1, \dots, K\}$, we proceed in accordance to the following heuristics [14]:

1. Divide the d-dimensional parameter space $\mathbb{R}^d$ into j non-overlapping regions $R_1, \dots, R_j$.
2. For every new data point that falls into the region R_i we predict that it belongs to the most commonly occurring class of training data points in R_i.

In order to achieve 1 we recursively divide the predictor space into binary regions of the form $R_1 := \{X|X_j < s\}, R_2 := \{X|X_j \geq s\}$ for a given index $j \in \{1, \dots, d\}$ and value $s \in \mathbb{R}$. We then choose $j' \in \{1, \dots, d\} \setminus \{j\}$ and $s' \in \mathbb{R}$ and proceed to divide R_1 and R_2 in the same fashion, so that $R_3 := \{X|X \in R_1, X_{j'} \leq s'\}, R_4 := \{X|X \in R_2, X_{j'} \geq s'\}$, and so on.

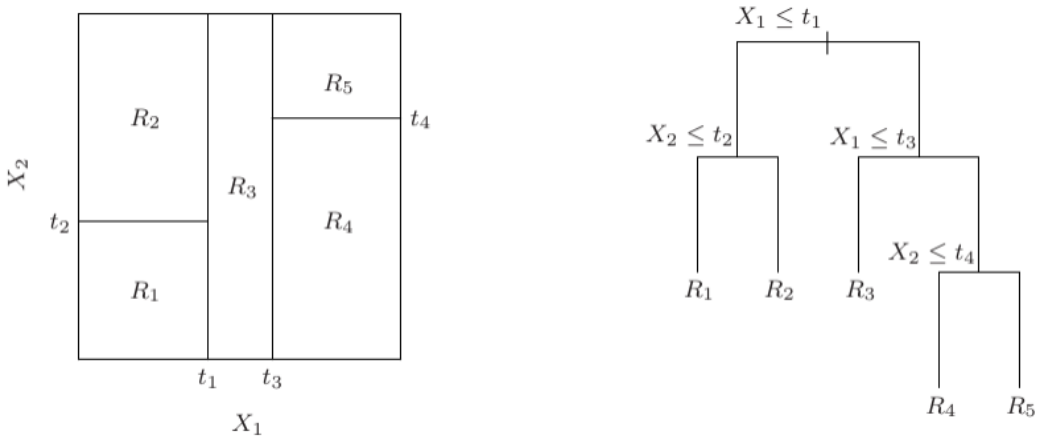


Figure 3.4: A partition of a two-dimensional feature space by recursive binary splitting (left) and its tree representation (right). Taken from [11].

An important question arises: how should we choose j and s in order to grow our tree? Intuitively, we want to choose j and s such that each region of the resulting partition $R_1, \dots, R_j$ has data points belonging to similar classes. We do so by greedily choosing the ones which minimize the *Geni index*.

Definition 3.2.3 *Given $\{x^i\}_{i=1,\dots,N}$, $x^i \in \mathbb{R}^d$, $R_1, \dots, R_m$ binary non-overlapping regions of parameter space $\mathbb{R}^m$, let p_{mk} represent the proportion of elements of $\{x^i\}$ in the m-th region that belong to the k-th class. We define the Gini index as*

$$G = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}) \tag{3.7}$$

Note that the Gini index takes small values if p_{mk} is close to either 0 or 1, meaning that a region contains predominantly data points from a single class [14]. Because of this, we greedily select the splitting parameters of the regions $R_1, \dots, R_j$ which minimize the Gini index; this task can be done efficiently [14]. Finally, a real-world unseen data point x will

be classified by each tree according to its splitting, and will be assigned to the majority category of the elements of the region R_i to which it belongs.

In order to decorrelate the trees, at each time split we randomly sample a subset of $m \approx \sqrt{d}$ of the d parameters of each x^i as possible split candidates. This has the effect of minimizing the influence of strong predictors, decorrelating the trees and thus resulting in a fair final majority vote [14]. Parameters such as number of trees in the forest, maximum depth of each tree, and tree pruning approaches have been optimized using Bayesian Optimization; we refer the reader to [26] for implementation and performance details.

3.2.6 Extreme Gradient Boosting

Extreme Gradient Boosting (XGB) is a scalable and highly effective machine learning classification method whose impact has been widely recognized in the data science community. As an example, in the challenges hosted by the machine learning competition site "Kaggle", 17 out of the 29 challenge winning solutions used XGB [3]. The power of XGB relies on taking several ML methods as base -in this case trees- and sequentially improve them in terms of minimizing the overall error to obtain a highly efficient classificator. The following exposition follows article [3] closely.

Given training data $\{(x^i, y^i)\}_{i=1,\dots,N}$, $x^i, y^i \in \mathbb{R}^d$ where $y_c = 1$ if and only if x^i belongs to the class $c \in \{1, \dots, K\}$, XGB builds k trees $f^1, \dots, f^k \in \mathcal{F}$ where $\mathcal{F}$ is the space of binary decision trees. In order to train these trees, XGB minimizes the regularized objective function

$$\mathcal{L}(\mathcal{F}) := -\sum_{i,k} y_k^i \log p_k^i + \frac{1}{2} \sum_{f^k \in \mathcal{F}} \sum_{\text{leaves} j} w_j^2, \tag{3.8}$$

where $p^i \in \mathbb{R}^k$ is the predicted probability vector

$$p_k^i = \sigma(s_k^i),$$

σ is the *softmax* function (3.2.1) and s_k^i is the score vector

$$s_k^i = \sum_{t=1}^{T} f_k^t(x_i)$$

which is continually updated throughout the execution of the algorithm, while w_j is the score on the j-th leaf of a given tree. Notice that the score of the i-th data point for class k is simply obtained by adding the scores that each tree assigned to it for such class. As with any other ML method, the first summand of L measures the distance from each probability vector p^i to the objective vector y^i and the second term is a regularizer to prevent over-fitting.

Notice that equation (3.8) involves functions as parameters and thus cannot be optimized with traditional optimization methods. Rather, XGB works in an additive manner, greedily minimizing L with each new tree. Let $p_k^{i,(t-1)}$ denote the i-th probability vector at the t-th iteration. Then, we add a tree f^t such that minimizes

$$\mathcal{L}^t := -\sum_{i=1}^{n} (y_k^i \log p_k^{i,(t-1)} + f^t(x_i)) + \frac{1}{2} \sum_{f^k \in \mathcal{F}} \sum_{\text{leaves} j} w_j^2$$

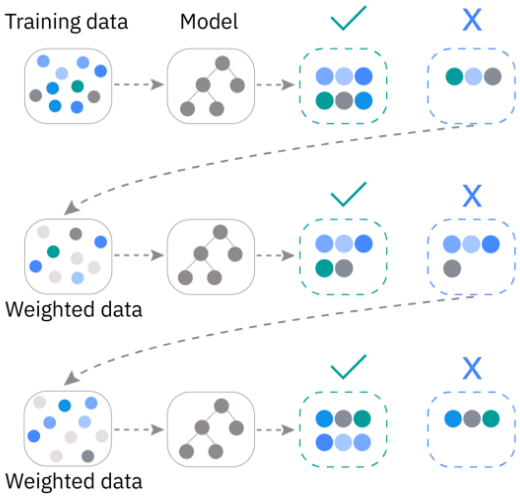


Figure 3.5: In XGB trees are sequentially added in order to minimize the error function at each iteration, thus refining the model. Taken from [17].

In practice, a second order Taylor approximation to $\mathcal{L}^t$ is used for its optimization. With such formulation at hand, it is possible to derive the optimal weight w_j of a leaf and to calculate the corresponding optimal value of $\mathcal{L}^t$ for all iterations, sequentially adding near-optimal trees [3]. We now present the XGB algorithm completely:

Algorithm 9: Extreme Gradient Boosting algorithm

Input: $\{(x^i, y^i)\}_{i=1,\dots,N}$, $x^i, y^i \in \mathbb{R}^d$ where $y_c = 1$ if and only if x^i belongs to the class $c \in \{1, \dots, K\}$.
Output: $\hat{y} \in \mathbb{R}^k$ classification vector on the set of classes $\{1, \dots, K\}$.
Initialize scores: $s^i \in \mathbb{R}^k$ score vector $s^i = 0$ for all $i \in \{1, \dots, N\}$;
Initialize probabilities: $p^i \in \mathbb{R}^k$ probability distribution vector $p^i = 0$ for all $i \in \{1, \dots, N\}$;
for $t = 1, \dots, T$ **do**
  for $j = 1, \dots, K$ **do**
    Compute: second-order Taylor approximation of $\mathcal{L}^t(\mathcal{F}) := -\sum_{i,k} y_k^i \log p_k^{i,(t-1)} + \frac{1}{2} \sum_{f^k \in \mathcal{F}} \sum_{\text{leaves} j} w_j^2$
    Fit a decision tree minimizing $L^t(\mathcal{F})$
  Update scores and probabilities: $s_k^i \leftarrow s_k^i + \nu f_t^k(x^i)$, $p_k^i = \sigma(s_k^i)$ (learning rate ν, default is 0.3 [3]).
Prediction: $c_i := arg\max_k p_k^i(x^i)$
Output: $\hat{y}$ where $\hat{y}_j = c_j$ for all $j \in \{1, \dots, K\}$

3.3 Summary of Machine Learning Methods

Proceeding as in Chapter 2, the table on the next page provides an overview of the machine learning methods presented in the current section. Based on this section's contents and closely following [3, 11, 14, 16], the main advantages and disadvantages of each method are summarized, providing the reader with a panorama of the machine learning methods involved in this work.

Table 3.1: Summary of machine learning methods.

ML method	**Advantages**	**Disadvantages**
Multinomial Logistic Regression	• Simple to understand and interpret. • Easy to implement.	• Might underperform on high-dimensional data. • Usually unable to handle non-linear relationships.
Support Vector Classification	• Handles high-dimensional data well. • Conceptually easy to understand.	• Can not handle overlapping classes. • Requires a somewhat long training period.
K-Nearest Neighbors	• Very fast training and ease of implementation. • Adapts easily to new data.	• Highly sensible to hyperparameter k. • Might struggle on high-dimensional data.
Random Forest Classification	• Low risk of over-fitting. • Provides information about the importance of each feature.	• More complex interpretation than a single decision tree. • Might be time-consuming (size of forest).
Extreme Gradient Boosting	• High performance and accuracy. • Scalable for large data sets.	• Sensible to hyperparameters. • Training can be expensive.

4 Implementation and Findings

4.1 Matrix Properties and Learner Selection

It is of interest to determine which of the ML methods presented in the previous chapter performs best in predicting the optimal coarsening strategy for an efficient AMG setup. In order to achieve this, we trained all ML models on 1173 artificially generated large matrices A -characterizing large linear systems of the form $Ax = b$ where b is a random vector. Each matrix was generated by a newly developed Python script in which matrix size, sparsity, symmetry, row sums, off-diagonal values, and diagonal dominance were varied -within certain thresholds- to produce a rich pool of matrices to work with. The following properties of each of them were then measured:

- **Matrix size and sparsity**
  - **Number of non-zero entries of matrix A:** nna
  - **Dimension of matrix A:** nnu
  - **Average non-zeros per row:** $\frac{\text{nna}}{\text{nnu}}$
- **Row sums and absolute row sums**
  - **Absolute row sums (min, avg, max):** $r_i = \sum_j |a_{ij}|$
  - **Relative absolute row sums (min, avg, max):** $r_i^{\text{rel}} = \frac{\sum_j |a_{ij}|}{a_{ii}}$
  - **Number of negative row sums:** $|\{i : \sum_j a_{ij} < 0\}|$
  - **Number of zero row sums:** $|\{i : \sum_j a_{ij} = 0\}|$
  - **Number of positive row sums:** $|\{i : \sum_j a_{ij} > 0\}|$
  - **Row sum quartiles:** 25th percentile, median, 75th percentile of $\sum_j a_{ij}$
  - **Percentage of rows with higher row sums than the average of their neighbors:** Percentage of rows whose row sum is greater than the average row sum of their (nonzero-column) neighbor rows.
  - **Number of negative abs-row sums:** $|\{i : r_i < 0\}|$
  - **Number of zero abs-row sums:** $|\{i : r_i = 0\}|$
  - **Number of positive abs-row sums:** $|\{i : r_i > 0\}|$
  - **Absolute row sum quartiles:** 25th percentile, median, 75th percentile of r_i
- **Symmetry**
  - **Symmetry check (relative):** Let $E \in \mathbb{R}^{nnu \times nnu}$ be the matrix such that $e_{ij} = \frac{a_{ij} - a_{ji}}{a_{ii} + a_{jj}}$ for all $i, j = 1, \ldots, nnu$, $i \neq j$. Then the symmetrcy check (relative) of A is the Frobenius norm of E: $||E||_F := \left(\sum_{i=1}^{nnu} \sum_{j=1}^{nnu} |e_{ij}|^2 \right)^{\frac{1}{2}}$.

- **Symmetry check (absolute):** $\|A - A^T\|_F$

- **Diagonal entries**
  - **Number of negative diagonals:** $|\{i : a_{ii} < 0\}|$
  - **Number of zero diagonals:** $|\{i : a_{ii} = 0\}|$
  - **Number of positive diagonals:** $|\{i : a_{ii} > 0\}|$
  - **Diagonal values (min, avg, max):** Statistics of a_{ii}
  - **ℓ_1-norm of diagonal vector:** $\|\text{diag}(A)\|_1 = \sum_i |a_{ii}|$
  - **ℓ_2-norm of diagonal vector:** $\|\text{diag}(A)\|_2 = \sqrt{\sum_i a_{ii}^2}$
  - **Scaled ℓ_1-norm:** $\frac{\|\text{diag}(A)\|_1}{\text{nnu}}$
  - **Scaled ℓ_2-norm:** $\frac{\|\text{diag}(A)\|_2}{\text{nnu}}$

- **Off-diagonal entries**
  - **Number of negative off-diagonals:** $|\{a_{ij} < 0,\ i \neq j\}|$
  - **Number of zero off-diagonals:** $|\{a_{ij} = 0,\ i \neq j\}|$
  - **Number of positive off-diagonals:** $|\{a_{ij} > 0,\ i \neq j\}|$
  - **Rows with negative off-diagonals:** Rows where $\exists j : a_{ij} < 0, i \neq j$.
  - **Rows with mixed off-diagonals:** Rows containing both $a_{ij} > 0$ and $a_{ij} < 0$ for $i \neq j$.
  - **Rows with positive off-diagonals:** Rows where all $a_{ij} > 0,\ i \neq j$.

- **Diagonal dominance**
  - **Diagonal dominance over rows (min, avg, max):** $\text{diagdom}_i = \left|\frac{\sum_{j=1, j\neq i}^{nnu} a_{ij}}{a_{ii}}\right|$
  - **Standard deviation of diagonal dominance:** $\sigma(\text{diagdom})$
  - **Diagonal dominance quartiles:** 25th percentile, median, 75th percentile of diagdom_i
  - **Percentage of rows with DiagDom in range 0–0.25:** $\frac{|\{i: 0 \leq \text{diagdom}_i < 0.25\}|}{\text{nnu}} \times 100$
  - **Percentage of rows with DiagDom in range 0.25–0.5:** $\frac{|\{i: 0.25 \leq \text{diagdom}_i < 0.5\}|}{\text{nnu}} \times 100$
  - **Percentage of rows with DiagDom in range 0.5–0.75:** $\frac{|\{i: 0.5 \leq \text{diagdom}_i < 0.75\}|}{\text{nnu}} \times 100$
  - **Percentage of rows with DiagDom in range 0.75–1:** $\frac{|\{i: 0.75 \leq \text{diagdom}_i < 1\}|}{\text{nnu}} \times 100$
  - **Percentage of connections of rows with DiagDom ≤ 0.35:** Percentage of nonzero connections (edges) from rows with low diagonal dominance that connect to other rows also having low diagonal dominance. Measures clusters of rows with low diagonal dominance.

- **Matrix norms**
  - **Frobenius norm:** $\|A\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$
  - **Energy norm of const $1/\text{nnu}$:** $\|A \cdot \mathbf{1}\frac{1}{/\text{nnu}}\|_2$

– **Scaled energy norm:** Energy norm divided by mean diagonal

The properties shown above were selected after having performed several computational experiments and benchmarks on the matrix properties which best characterize each linear system in terms of a ML learner. We do not present all steps towards the final selection of matrix properties in detail, but rather only the final set of properties which have been used by the ML methods.

After training all models on all of the above matrix properties and performing several statistical tests -such as mutual information analysis, global importance, permutation importance, and mean absolute coefficients- the following matrix properties were found to be the most useful in predicting AMG's optimal coarsening strategy:

Matrix Characteristic	**Count (Top-5 Selections)**
Relative absolute row sums (min)	60
Diagonal dominance over rows (max)	60
Absolute row sums (avg)	50
Scaled ℓ_2-norm of diagonal vector	50
Diagonal dominance third quartile	50
Relative absolute row sums (max)	10
Percentage of rows with DiagDom in value range $0.75 - 1.0$	10
Relative symmetry check	9
Percentage of rows with DiagDom in value range 0.5–0.75	1

Table 4.1: Number of times each matrix characteristic was chosen as one of the learner's five most meaningful properties in order to predict the coarsening strategy, when trained on the full set of matrix properties.

It would be impractical and computationally expensive to measure all properties of a matrix in order to try to predict AMG's optimal coarsening strategy. Therefore, it is desirable to focus on a small subset of matrix characteristics which will enable the learner to make good predictions. In accordance to the data shown in Table (4.1), we restrict ourselves to the following reduced list of matrix characteristics:

- **Relative absolute row sums (min, avg, max)**
- **Diagonal dominance over rows (min, avg, max)**
- **Absolute row sums (min, avg, max)**
- **Scaled ℓ_2-norm of diagonal vector**
- **Diagonal dominance quartiles**
- **Percentage of rows with DiagDom in range** 0.5–0.75
- **Percentage of rows with DiagDom in range** $0.75 - 1.0$

We now turn our attention to the question: which of the ML models that we have trained is *most frequently* selected as the learner? i.e. which of the ML models that we have trained achieves the highest accuracy percentage on the validation set *most frequently*?

In order to answer it, we executed the following workflow, which is presented as an algorithm for clarity:

Algorithm 10: Workflow for choosing Learner

Generate 1173 large sparse matrices.

foreach $i = 1, \ldots, 1500$ **do**

1. Split the matrices into training and validation data sets.
2. Train all ML models on the reduced features set outlined above.
3. Measure each ML model's predictive accuracy percentage.
4. Increase by 1 the counter corresponding to the model with highest accuracy percentage.

return Counters of each ML model

The counters of each ML model at the end of the workflow described above are the following:

1. Multinomial Logistic Regression: 0
2. Support Vector Machines: 303
3. K-Nearest Neighbors: 449
4. Random Forest Classification: 274
5. Extreme Gradient Boosting: 474

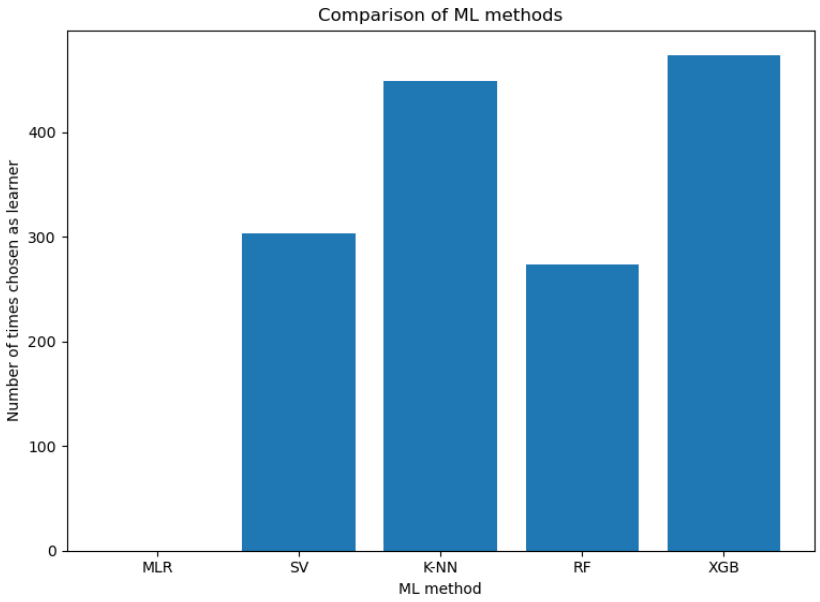


Figure 4.1: Bar plot showing the number of times each ML method was chosen as the learner over 1500 runs. Although Extreme Gradient Boosting and K-Nearest Neighbors are more frequently chosen as learners, all ML methods -except for Multinomial Logistic Regression- are commonly selected; the choice of learner depends on the training data set which varies across runs. Labels correspond to each exposed ML method: MLR = Multinomial Logistic Regression, SV = Support Vector Classification, K-NN: K-Nearest Neighbors, RF = Random Forest Classification, XGB = Extreme Gradient Boosting.

It is worth noting that the splitting of the full data set into a training set and a validation set is done each time before starting the training phase, making each run deal with a different subset of data and ensuring independence of the ML methods from the data.

4.2 Prediction of Optimal Coarsening Strategy

The ultimate goal of the present work is -once a handful of matrix properties have been identified in order to characterize each linear system- to be able to determine, with a trained ML model, the optimal coarsening strategy to be used in AMG for a given problem $Ax = b$. The workflow to achieve said goal is the following:

1. Train all ML models on the reduced matrix features.
2. Select the one with the highest accuracy on the validation data as the learner.
3. Run the learner on a given matrix A -corresponding to a problem $Ax = b$- and output the best 3 choices for coarsening strategy for each processor count.

We test our model against the latest and most novel optimization method used at SCAI -called Autonomous Solver Control (ASC)- which, based on genetic algorithms and accelerated by decision trees, is guaranteed to output an optimal coarsening strategy. For details on its functionality see [25]. Notice that steps 1 and 2 of the presented workflow simply correspond to one "for-loop" of Algorithm (10). This constitutes one of the main advantages of our proposed ML-based method over ASC: our method requires only one initial training, which is never repeated, whereas ASC must be trained during each simulation. In our method, once all models have been trained and a learner is selected, there is no need for retraining: the same learner will choose an optimal coarsening strategy for every matrix fed to it.

We present the results of the above workflow in the following table:

ML Method	**Learner (count)**	**Accuracy Top-1 (%)**	**Accuracy Top-3 (%)**
Logistic Regression	0	0.00	0.00
SVM	390	12.57	44.97
k-NN	780	14.09	63.49
Random Forest	330	11.86	43.29
XGBoost	795	17.97	48.38

Table 4.2: Comparison of machine learning methods by selection frequency and predictive accuracy. Learner (count) refers to the number of times each ML method was chosen as the learner, Top-1 (%) refers to the percentage of times that the learner chose the best coarsening strategy correctly, and Top-3 (%) refers to the percentage of times that the learner listed the best coarsening strategy as the top 3 choices. The number of processors considered in each case were $64, 128, 256, 512, 751, 1024$. The coarsening strategies considered were Standard, Aggressive Coarsening 51, Aggressive Coarsening 52, PMIS, CLJP and HMIS.

It is worth noting that the accuracy Top-1 (%) is not as high as expected due to the fact that the matrix properties considered do not include the number of processors as one of their characteristics. Thus, the learner outputs the same set of coarsening strategies independently of the number of processors. This, as shown in the discussion of chapter 1, is inaccurate, since massively parallel coarsening methods tend to outperform classical

coarsening methods for large problems and large processor counts (see Figure 2.1).

In order to correct this, we have repeated the experiment with the condition that if the processor count is larger that or equal to 256, the learner should consider only massively parallel coarsening strategies -namely PMIS, CLJP, and HMIS. The results are summarized in the following table:

ML Method	**Learner (count)**	**Accuracy Top-1 (%)**	**Accuracy Top-3 (%)**
Logistic Regression	0	0.00	0.00
SVM	430	7.84	45.00
k-NN	770	16.10	63.27
Random Forest	310	21.36	45.17
XGBoost	665	25.00	47.75

Table 4.3: Results of the adjusted experiment where, for large problems with processor counts greater than or equal to 256, only massively parallel coarsening strategies (PMIS, CLJP, HMIS) were considered. This correction addresses the initial limitation in Top-1 (%) accuracy, which was underestimated due to the exclusion of processor count as a feature.

The results are considerably better with respect to Top-1 (%), with XGBoost classification methods achieving the best Top-1 (%) accuracy, predicting the correct coarsening strategy 25% of the times, and considering it in the Top-3 choices almost 50% of the times. This is a remarkable result, due to the fact that the learner is only measuring a small subset of matrix features -those outlined in the previous section- and choosing an optimal coarsening strategy based on them.

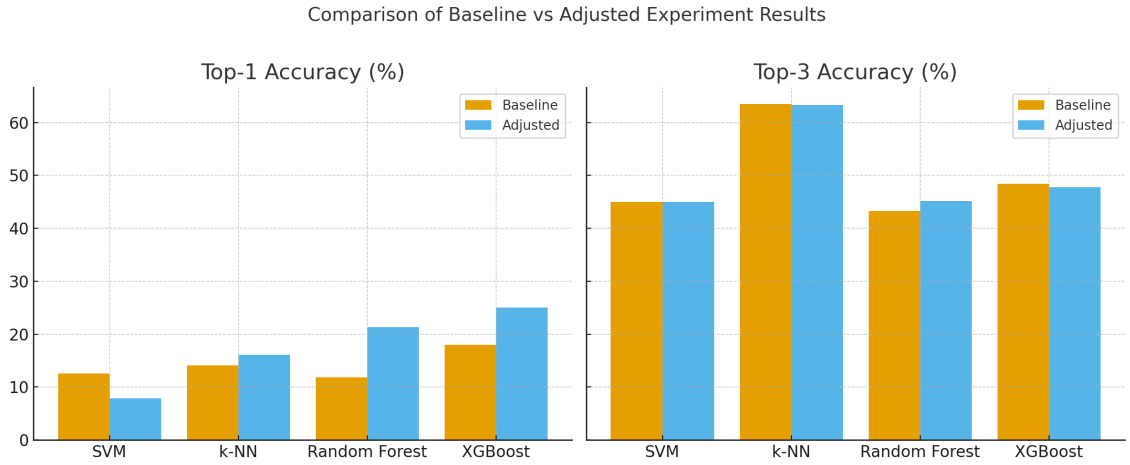


Figure 4.2: Comparison of ML methods under two conditions: (i) baseline experiment with all coarsening strategies, and (ii) adjusted experiment where massively parallel coarsening strategies (PMIS, CLJP, HMIS) were enforced for $p \geq 256$ on large problems. The adjustment improves Top-1 (%) accuracy by accounting for processor count, while Top-3 (%) accuracy remains relatively stable. Data taken from Tables (4.2) and (4.2).

We now address the question of quantifying the error between the selected coarsening strategy and the optimal coarsening strategy. Even though our learner does not choose the optimal coarsening strategy each time, if the error between the optimal coarsening strategy's run time and the selected run time is not too large, its results are still

promising as candidates to feed AMG with. Moreover, despite our learner not taking optimal decisions for every problem, it outperforms AMG's default coarsening strategy in amost all cases (see Figures 4.4 and 4.5). We quantify these through the quotients $R := \frac{\text{Predicted runtime}}{Optimalruntime}$ and $R_3 := \frac{\text{Best-of-3 Predicted runtime}}{Optimalruntime}$ in the Top-1 (%) and Top-3 (%) cases, as summarized in the following table:

ML Method	**Learner (count)**	**R**	**R_3**
Logistic Regression	0	0.00	0.00
SVM	390	4.17	1.09
k-NN	780	5.36	1.14
Random Forest	330	4.09	1.65
XGBoost	795	3.85	2.00

Table 4.4: Quantification of the average error between the selected and the optimal coarsening strategies. While the learner does not always predict the optimal strategy, the ratio of predicted runtime to best runtime remains within reasonable bounds, suggesting that the selected strategies are still promising candidates for AMG. R and R_3 are taken as averages over all times the method was selected as learner.

Once again, we repeat the experiment with the introduced condition that for processor counts larger than or equal to 256 on large problems, the learner should consider only massively parallel coarsening strategies, obtaining the following results:[1]

ML Method	**Learner (count)**	**R**	**R_3**
Logistic Regression	0	0.00	0.00
SVM	430	3.83	1.09
k-NN	770	4.40	1.16
Random Forest	310	3.02	1.76
XGBoost	665	4.15	3.32

Table 4.5: Quantification of the error between the selected and the optimal coarsening strategies under the condition that, for processor counts larger than or equal to 256 on large problems, the learner considers only massively parallel coarsening strategies (PMIS, CLJP, HMIS). The runtime ratios remain within reasonable bounds, supporting the suitability of the chosen strategies for AMG.

[1]When quantifying the error between different coarsening strategies, we observed that HMIS coarsening behaved differently when executed as part of the ASC routine versus when run independently. Therefore, we discarded any runs affected by this discrepancy.

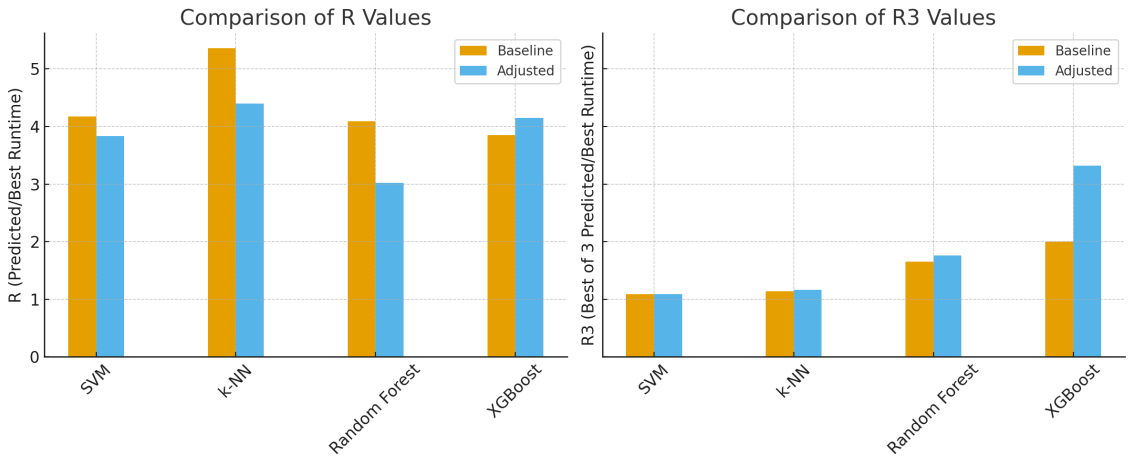


Figure 4.3: Comparison of runtime ratios between the baseline and adjusted experiments. The left panel shows the average ratio of predicted runtime to best runtime (R), while the right panel shows the average ratio when considering the best of three predicted strategies (R_3). The adjusted experiment enforces massively parallel coarsening strategies (PMIS, CLJP, HMIS) for large problems with processor counts larger than or equal to 256, leading to improved runtime ratios in almost all cases. A smaller ratio reflects a better result.

Notice that, for almost all learners, $R_3 \leq 2$, meaning that running the learner on the few matrix characteristics will, on average, result in a 2-approximation for the optimal strategy of the corresponding problem.

We have seen that the performance of our ML-based methods lies within reasonable bounds of ASC's performance. We now turn our attention to the question of how it compares against AMG's default coarsening strategy -namely, Standard/Ruge-Stüben Coarsening. For this, we use the benchmark problems from Subsction 2.4 with different process counts. The results, which demonstrate favorable performance of our ML-based methods' coarsening choice in almost all cases, are presented in the following plots:

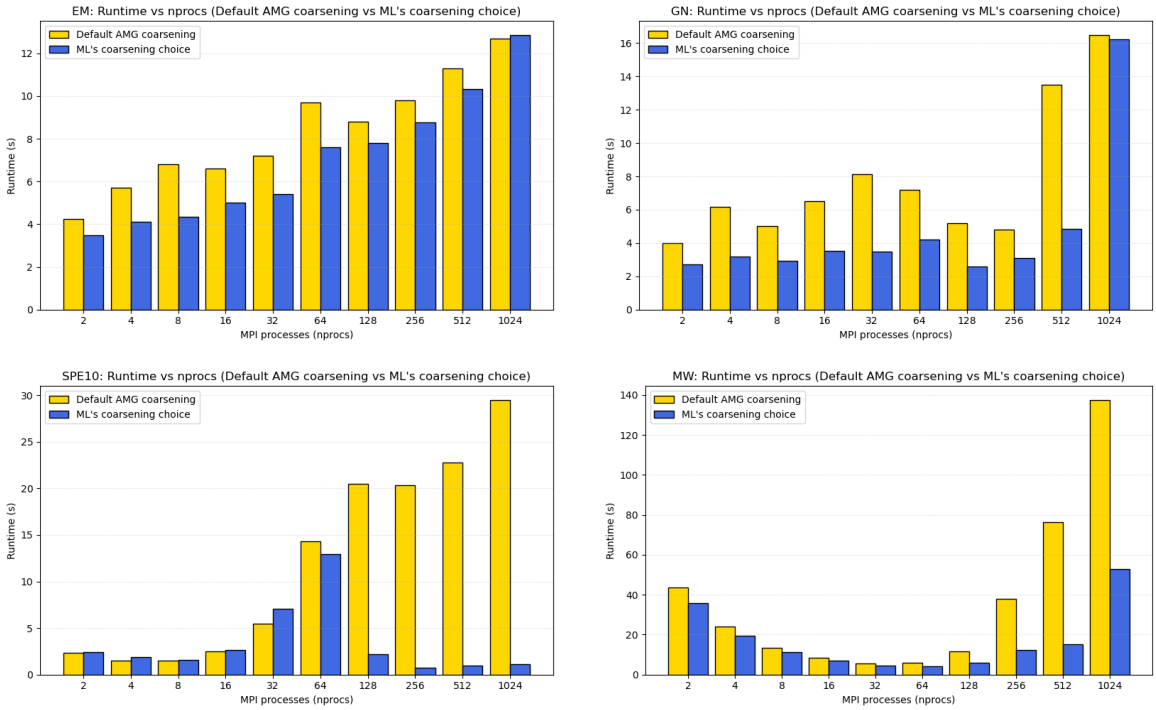


Figure 4.4: Comparison of runtime using AMG's default coarsening strategy (yellow) versus the coarsening strategy recommended by our ML-based tool (blue) as grid size grows. The plots show that our ML-based tool, despite not always making optimal decisions, delivers performance that consistently surpasses AMG's default settings in almost all scenarios. The problems considered are, in lexicographical order (top-to-bottom and left-to-right): Electro Magnetic, Groundwater North, Reservoir SPE10, and Meshfree Watercrossing. Processor count takes values 2^n for $n = 1, \ldots, 10$ and runtime is measured in seconds.

The following table shows the percentage of improvement of our ML-based methods over AMG's default coarsening strategy, where the values are taken from the above plot. Positive values indicate faster runtimes (improvement), while negative values indicate slower performance (slowdown). The final result -the average improvement across all processor counts and all problems- shows an overall speedup of 32.47%.

Table 4.6: Percentage of improvement of ML-based runtime over default AMG coarsening.

nprocs	**EM**	**GN**	**SPE10**	**MW**
2	17.88%	32.17%	-5.63%	17.76%
4	27.72%	47.97%	-25.33%	18.69%
8	36.18%	42.03%	-2.63%	17.07%
16	24.24%	46.00%	-5.18%	14.83%
32	24.86%	57.26%	-30.09%	22.86%
64	21.55%	41.25%	9.52%	32.27%
128	11.25%	50.38%	89.27%	48.11%
256	10.41%	35.00%	96.26%	67.95%
512	8.68%	64.15%	95.74%	79.92%
1024	-1.34%	1.70%	96.27%	61.65%
Average	**18.14%**	**41.79%**	**31.82%**	**38.11%**
Overall Average	**32.47%**			

Notice that in the two plots shown at the bottom (Reservoir SPE10 and Meshfree Watercrossing), our ML-based tool largely outperforms AMG's default coarsening's runtime as processor count grows, while for the two plots shown at the top (Electro Magnetic and Groundwater North) the improvement is less pronounced. The reason behind this is that in the cases of Reservoir SPE10 and Meshfree Watercrossing, our ML-based tool chose one of the three natively parallel coarsening algorithms -namely CLJP, PMIS or HMIS- as the number of processors grew, while in the first two cases it did not. Based on this observation and on our previous analysis of coarsening strategies, we will add the condition that if the processor count is larger than or equal to 256, the learner should consider only natively parallel coarsening strategies. The results are shown in the following plots:

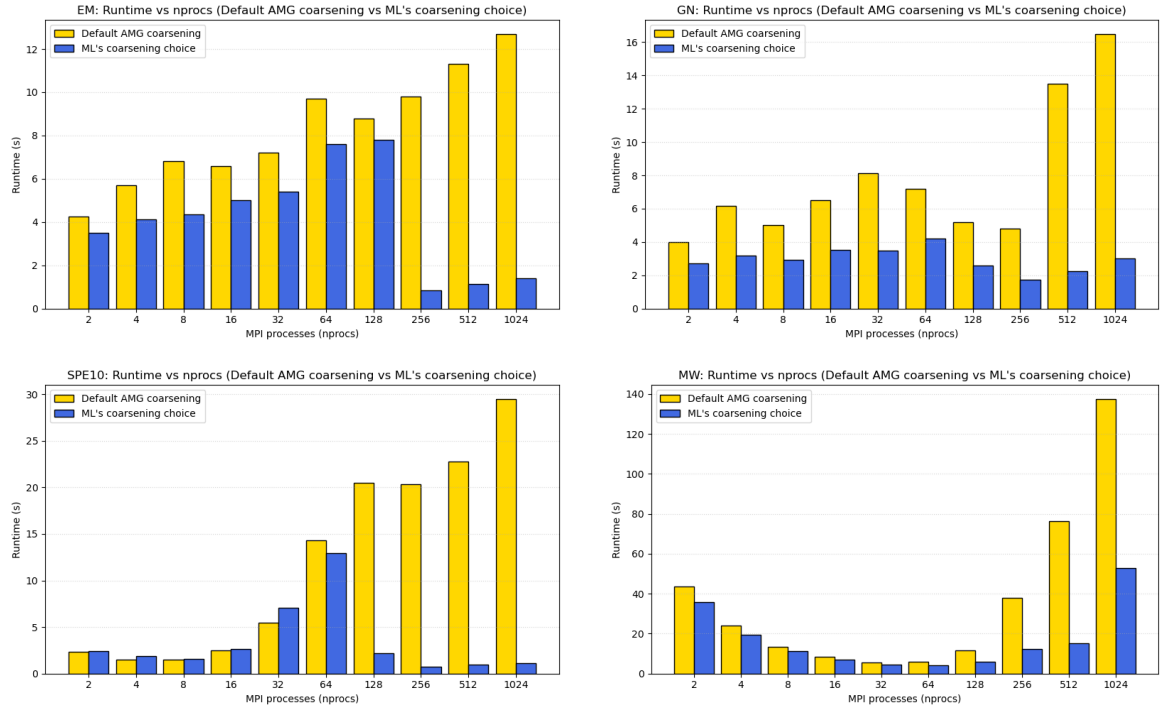


Figure 4.5: Comparison of runtime using AMG's default coarsening strategy (yellow) versus the coarsening strategy recommended by our ML-based tool (blue) as grid size grows, with the condition that for processor counts larger than or equal to 256, our learner will only consider natively parallel coarsening strategies. The plots show that our ML-based tool with this additional condition consistently outperforms AMG's default settings for large processor counts. The problems considered are, in lexicographical order (top-to-bottom and left-to-right): Electro Magnetic, Groundwater North, Reservoir SPE10, and Meshfree Watercrossing. Processor count takes values 2^n for $n = 1, \ldots, 10$ and runtime is measured in seconds.

Analogously as in Table 4.6, the following table shows the percentage of improvement of our ML-based methods over AMG's default coarsening strategy with the added condition for large processor counts. Positive values indicate faster runtimes (improvement), while negative values indicate slower performance (slowdown). The final result -the average improvement across all processor counts and all problems- shows an overall speedup of almost 42%, a considerable improvement on our previous result of 32.47%.

Table 4.7: Percentage of improvement of ML-based runtime over default AMG coarsening with added condition for large processor counts.

nprocs	EM	GN	SPE10	MW
2	17.88%	32.17%	-5.63%	17.76%
4	27.72%	47.97%	-25.33%	18.66%
8	36.18%	42.03%	-2.63%	17.07%
16	24.24%	46.00%	-5.18%	14.83%
32	24.86%	57.26%	-30.09%	22.87%
64	21.55%	41.25%	9.53%	32.27%
128	11.25%	50.38%	89.27%	48.09%
256	91.22%	63.96%	96.26%	67.95%
512	89.82%	83.26%	95.74%	79.92%
1024	88.90%	81.64%	96.27%	61.66%
Average	**43.36%**	**54.59%**	**31.82%**	**38.11%**
Overall Average	**41.97%**			

4.3 Efficiency of Implementation

One of the main advantages of the ML approach exposed here -and its main advantage against the existing ASC method- is that it is extremely fast, due to the fact that it relies on only a handful of matrix properties which are relatively cheap to compute and it only needs to be trained once. Once the training is done and the learner is chosen, deciding which coarsening strategy to use is done in a fraction of a second -without the need to retrain the model.

To show this, we will measure how much time it takes for our implementation at Fraunhofer's SCAI *Schnelle Löser* team's Fortran code to measure the reduced matrix characteristics previously listed, namely:

- **Relative absolute row sums (min, avg, max)**
- **Diagonal dominance over rows (min, avg, max)**
- **Absolute row sums (min, avg, max)**
- **Scaled ℓ_2-norm of diagonal vector**
- **Diagonal dominance quartiles**
- **Percentage of rows with DiagDom in range** 0.5–0.75
- **Percentage of rows with DiagDom in range** $0.75 - 1.0$

on the benchmark problems from Subsection 2.4 on different process counts, namely, Electro Magnetic (EM), Groundwater North (GN), Reservior SPE10 (SPE10), and Meshfree Watercrossing (MW), which have the following properties:

Table 4.8: Matrix properties of benchmark problems

Property	**EM**	**GN**	**SPE10**	**MW**
Dimension (nnu)	$143,495$	$737,191$	$1,122,004$	$5,012,292$
Number of nonzero entries (nna)	$3,825,843$	$5,854,193$	$5,878,243$	$200,488,366$
Absolute average rowsum (relative)	0.27	0.7	0.3	0.11
Diagonal dominance over all rows (max)	7.3	2.6×10^4	1	2
L2 norm of diagonal vector (scaled)	5×10^{-14}	4.7	1.8×10^3	4.4×10^{-4}
Symmetry check (relative)	0	1	0.89	0.19

The results are shown in the following plot:

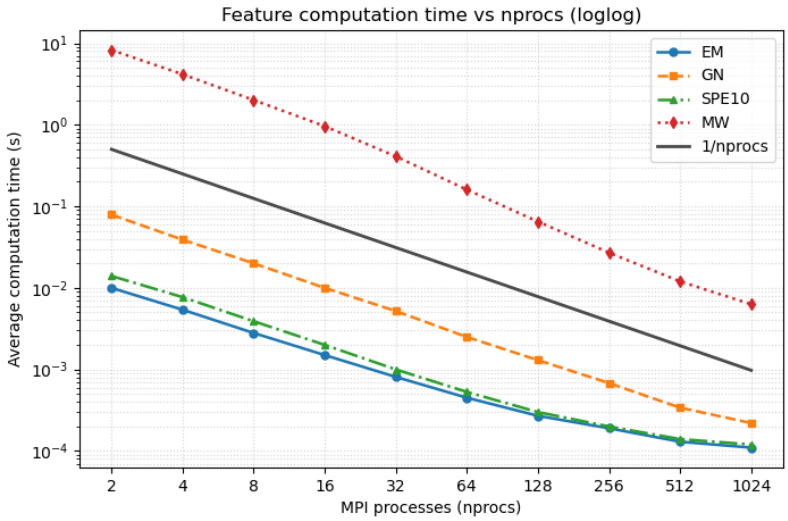


Figure 4.6: Loglog plot of the time it takes (seconds) to compute the relevant matrix features as the number of processors grow. The reference line $\frac{1}{nprocs}$ suggests that the behavior in all cases is that of $\frac{C}{nprocs}$ for a given constant $C > 0$. The plot also shows the great speed of the proposed ML method: for medium-sized problems, it gathers all the information it needs to make a decision in a fraction of a second.

Furthermore, the time needed to compute said matrix properties is minimal with respect to AMG's total runtime, as shown in the following table and its corresponding plot:

Table 4.9: Percentage of total runtime spent on feature computation

nprocs	EM	GN	SPE10	MW
2	0.310%	3.074%	0.651%	1.331%
4	0.138%	1.230%	0.550%	1.297%
8	0.064%	0.694%	0.300%	1.031%
16	0.033%	0.292%	0.083%	0.850%
32	0.017%	0.159%	0.022%	0.587%
64	0.047%	0.057%	0.004%	0.491%
128	0.034%	0.049%	0.002%	0.347%
256	0.022%	0.023%	0.002%	0.157%
512	0.011%	0.007%	0.001%	0.082%
1024	0.008%	0.010%	0.001%	0.015%

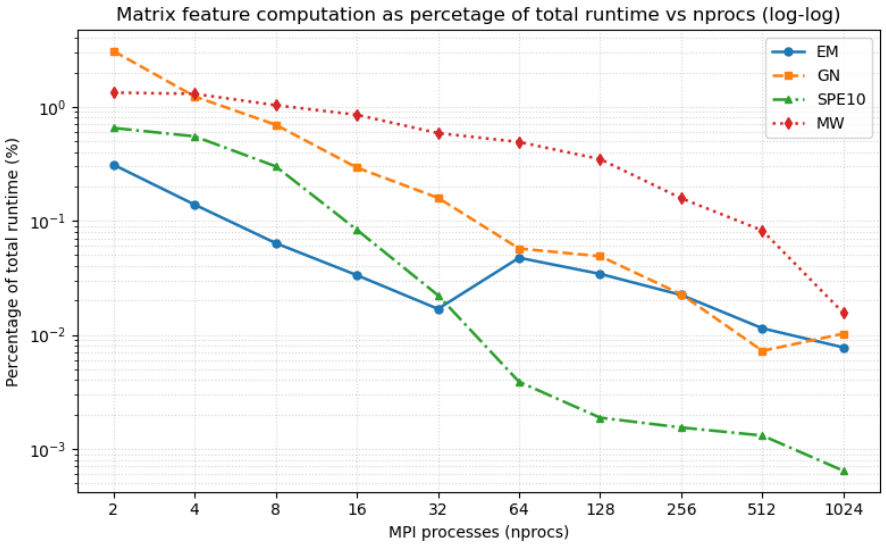


Figure 4.7: Loglog plot of the percentage that computing the relevant matrix features represents with respect to the total running time as the number of processors grow. In all cases it is extremely low, highlighting the fact that evaluating the matrix characteristics -on which our ML-based method bases its decisions- is negligible in terms of runtime.

The complete prototypical workflow of AMG's software package -with our ML-based setup optimizator method- as implemented at Fraunhofer's SCAI *Schnelle Löser* team is shown on the next page.

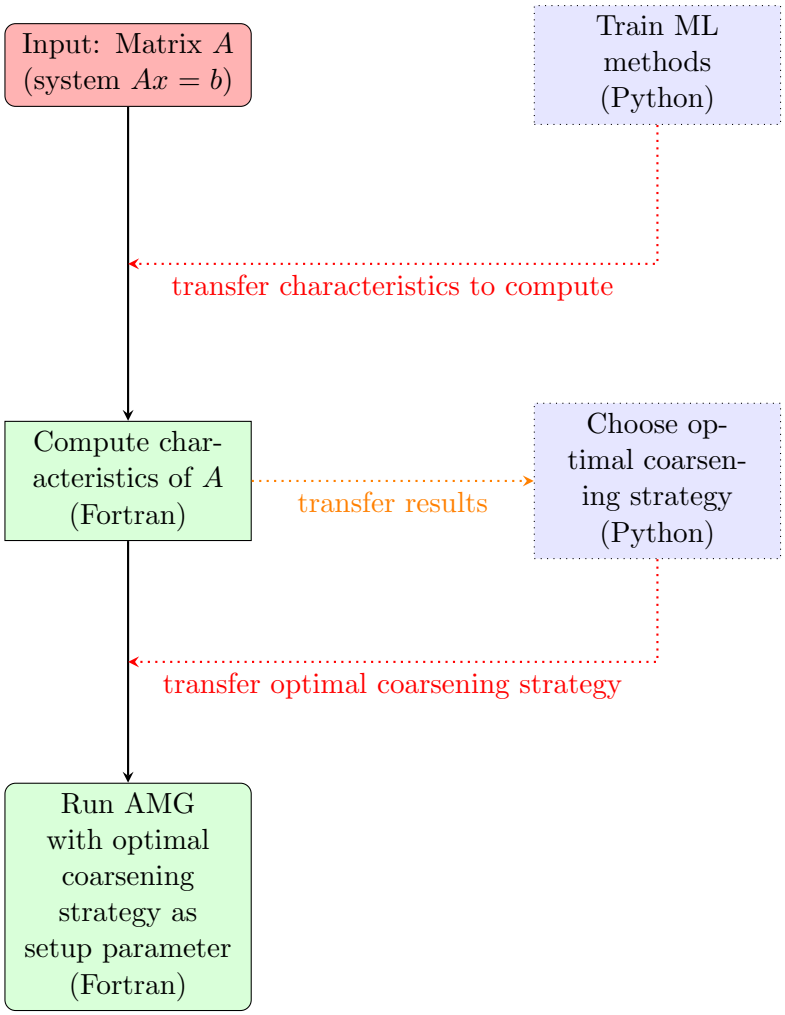


Figure 4.8: Diagram of AMG's complete prototypical workflow with our ML-based optimizator included. Dashed red and orange arrows indicate transfer of information as shown by their labels: red arrows indicate unavoidable processes while orange arrows indicate processes that might be ignored in favor of efficiency. Green nodes correspond to code executed in Fortran, while blue nodes indicate code executed in Python. The workflow is designed to optimize time and computational resources by exploiting the advantages of both programming languages.

The orange arrow in the above diagram indicates that the process of computing the matrix characteristics of A, while much faster done by Fortran than by Python, since there are so few matrix characteristics to be computed, might also be done in Python -in case the cost of transferring information outweighs the cost of computation in Python. The original workflow does include this transfer of information between programming languages.

5 Conclusion and Outlook

In this thesis we successfully applied machine learning methods to choose an optimal coarsening strategy for the setup phase of AMG of different M-like matrices corresponding to large, sparse systems of linear equations.

While AMG is, to this day, one of the most efficient numerical algorithms for solving large, sparse linear systems of equations, in practice it needs to be tailored to the problem at hand in order to achieve optimal functionality [10]. One of the key steps in AMG's setup phase is the coarsening phase, which drastically affects AMG's running time and performance [31]. For this reason, being able to automatically choose a coarsening strategy given a handful of matrix characteristics is of utmost importance if peak performance of AMG is desired.

Numerical experiments presented in this thesis show that the ML methods here considered are able to decide, in a fraction of a second and based on a handful of the properties of matrix A, which is the optimal coarsening strategy in around one fourth of cases with the best performing learner. The best learner is also able to list the optimal coarsening strategy as one of its three top choices almost two thirds of the times, and the average among all learners lies at around half of the times (see Table 4.3), making our method an efficient classification method. Moreover, in most cases, the best of runtimes achieved by our method's three top choices of coarsening strategy tends to be around 1.8 times that of the optimal runtime (see Table 4.5), making it a 2-approximation of the optimal running time achieved by all coarsening strategies considered. Moreover, our numerical results show that this approach yields average improvement factors of up to 41% over default coarsening settings (see Subsection 4.2), demonstrating its effectiveness in significantly reducing AMG's runtimes.

We firmly believe that our method is not to be considered a competitor to other automatic setup algorithms -such as Fraunhofer's ASC- but rather as a complementary tool or an alternative to them. While ASC might have the guarantee of optimality, our ML based methods are faster -since they don't require retraining at every run- and computationally cheaper -since they rely only on a handful of easily computable matrix properties-, and thus preferable when few computational resources are at hand. It could, for example, serve as a source of input for ASC, helping to reduce its search space and select an optimal set of parameters more efficiently. In any case, due to its 41% performance improvement over default settings, it is a highly effective and practical tool for accelerating AMG's setup and providing reliable guidance on coarsening strategy selection.

Future work could focus on extending our ML models in order to handle a multivariate objective function. Say, adapting our ML based approach to output not only the coarsening strategy which results in the fastest AMG runtime, but also the one that maximizes a multivariate objective function which takes into account multiple aspects of AMG: CPU

capacity, computational costs, memory efficiency, energy consumption, among others. We are confident that, as ML methods advance, these techniques will be able to automatically handle complex problems like the one stated, and our proposed method provides a solid foundation that can be naturally extended to these broader domains.

Bibliography

[1] Briggs William L., Henson Van Emden and McCormick Steve F. *A Multigrid Tutorial: second edition*. USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0898714621.

[2] Brochu Eric, Cora Vlad M. and Freitas Nando de. *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. arXiv preprint arXiv:1012.2599. Dec. 2010.

[3] Chen Tianqi and Guestrin Carlos. 'XGBoost: A Scalable Tree Boosting System'. Version v3. In: *arXiv preprint arXiv:1603.02754* (June 2016). arXiv preprint, revised version submitted June 10, 2016. URL: https://arxiv.org/pdf/1603.02754.

[4] Christie Michael Andrew and Blunt M. J. 'Tenth SPE Comparative Solution Project: a comparison of upscaling techniques'. In: *Proceedings of the SPE Reservoir Simulation Symposium*. Paper SPE 66599-MS. Houston, Texas, United States, Feb. 2001. DOI: 10.2118/66599-MS.

[5] Cleary A. J. et al. 'Coarse grid selection for parallel algebraic multigrid'. In: *Proceedings of the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*. Ed. by Ferreira A. et al. Vol. 1457. Lecture Notes in Computer Science. New York: Springer, 1998.

[6] De Sterck H., Yang U. M. and Heys J. 'Reducing Complexity in Parallel Algebraic Multigrid Preconditioners'. In: *SIAM Journal on Matrix Analysis and Applications* (Sept. 2004). Lawrence Livermore National Laboratory Report UCRL-JRNL-206780. URL: https://www.osti.gov/servlets/purl/883821.

[7] Feurer M. and Hutter F. 'Hyperparameter Optimization'. In: *Automated Machine Learning*. Springer, 2019.

[8] Ghojogh B., Sharifian S. and Mohammadzade H. 'Tree-based Optimization: A Meta-Algorithm for Metaheuristic Optimization'. In: (2018).

[9] Griebel Michael et al. 'Coarse Grid Classification: A Parallel Coarsening Scheme For Algebraic Multigrid Methods'. In: *Numerical Linear Algebra with Applications* (2005). URL: https://ins.uni-bonn.de/media/public/publication-media/copper2005.preprint_bBPxAOP.pdf?name=copper2005.preprint.pdf.

[10] Gries Silvia. 'Algebraic Multigrid for Eigenproblems in Industrial Applications of Big Data and Engineering'. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät. PhD thesis. Bonn, Germany: Rheinische Friedrich-Wilhelms-Universität Bonn, 2023.

[11] Hastie Trevor, Tibshirani Robert and Friedman Jerome. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd. New York, NY: Springer, 2009. ISBN: 978-0-387-84858-7. URL: https://web.stanford.edu/~hastie/ElemStatLearn/.

[12] Henson V. E. and Yang U. M. 'BoomerAMG: a parallel algebraic multigrid solver and preconditioner'. In: *Applied Numerical Mathematics* 41.1 (2002), pp. 155–177. DOI: 10.1016/S0168-9274(01)00115-5.

[13] Henson Van Emden and Yang Ulrike Meier. *BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner*. Tech. rep. UCRL-JC-145828. Lawrence Livermore National Laboratory, 2002.

[14] James Gareth et al. *An Introduction to Statistical Learning: with Applications in R*. Corrected 7th Printing. Springer, 2013. ISBN: 978-1-4614-7137-0. URL: https://www.statlearning.com/.

[15] Jankowski D. and Jackowski K. 'Evolutionary Algorithm for Decision Tree Induction'. In: *International Conference on Computer Information Systems and Industrial Management*. 2014.

[16] Jurafsky Daniel and Martin James H. 'Logistic Regression'. In: *Speech and Language Processing*. Draft. Stanford University, 12th Jan. 2025. Chap. 5. URL: https://web.stanford.edu/~jurafsky/slp3/5.pdf.

[17] Kavlakoglu Eda and Russi Erika. *Was ist XGBoost?* IBM Think. Published on 9 May 2024. May 2024. URL: https://www.ibm.com/de-de/think/topics/xgboost.

[18] Langevin Christian D. et al. *Documentation for the MODFLOW 6 Groundwater Flow Model*. Tech. rep. 6–A55. Prepared in cooperation with the U.S. Geological Survey Water Availability and Use Science Program. Reston, VA: U.S. Geological Survey, Aug. 2017, p. 197. DOI: 10.3133/tm6A55. URL: https://pubs.usgs.gov/tm/06/a55/tm6a55.pdf.

[19] Luby Michael. 'A Simple Parallel Algorithm for the Maximal Independent Set Problem'. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1036–1053. DOI: 10.1137/0215074.

[20] Luz Ilay et al. 'Learning Algebraic Multigrid Using Graph Neural Networks'. In: *Proceedings of Machine Learning Research* 119 (2020), pp. 5483–5493. URL: https://proceedings.mlr.press/v119/luz20a/luz20a.pdf.

[21] Mishev I., Fedorova B. and Terekhov S. 'Linear Solver Performance Optimization in Reservoir Simulation Studies'. In: (2009).

[22] Nick Fabian Pascal. 'Algebraic Multigrid for Meshfree Methods'. Dissertation zur Erlangung des Doktorgrades Dr. rer. nat. Ph.D. dissertation. Bonn, Germany: Rheinische Friedrich-Wilhelms-Universität Bonn, 2020.

[23] Optuna Team. *optuna.samplers.TPESampler (API) — Optuna Documentation*. https://optuna.readthedocs.io/en/stable/reference/samplers/generated/optuna.samplers.TPESampler.html. Accessed: 2025-09-02. 2025.

[24] Ruge J. W. and Stüben K. 'Algebraic Multigrid'. In: *Multigrid Methods*. Ed. by McCormick S. F. Vol. 5. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, 1987, pp. 73–130.

[25] Schulz Daniel and Bauckhage Christian, eds. *Informed Machine Learning*. Cognitive Technologies. Springer, Cham, 2025. ISBN: 978-3-031-83096-9. DOI: 10.1007/978-3-031-83097-6.

[26] scikit-learn developers. *sklearn.ensemble.RandomForestClassifier — scikit-learn: Machine Learning in Python*. Accessed: 2025-08-22. scikit-learn. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[27] scikit-learn developers. *sklearn.linear$_m$odel.LogisticRegression*. API documentation, version 1.7.1. scikit-learn. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

[28] scikit-learn developers. *Support Vector Machines*. User Guide — Supervised Learning section 1.4. scikit-learn. 2025. URL: https://scikit-learn.org/stable/modules/svm.html.

[29] Stüben Klaus. *Algebraic Multigrid (AMG): An Introduction with Applications*. GMD Report 70. Guest contribution to the book "Multigrid" by U. Trottenberg, C. W. Oosterlee, and A. Schüller, Academic Press, 2001. Sankt Augustin, Germany: German National Research Center for Information Technology (GMD), Institute for Algorithms and Scientific Computing (SCAI), 1999.

[30] Taghibakhshi Ali et al. 'Optimization-Based Algebraic Multigrid Coarsening Using Reinforcement Learning'. In: *Advances in Neural Information Processing Systems* 34 (2021). URL: https://lukeo.cs.illinois.edu/files/2021_TaMaOlWe_optmg.pdf.

[31] Trottenberg Ulrich, Oosterlee Cornelis W. and Schüller Anton. *Multigrid*. San Diego: Academic Press, 2000. ISBN: 9780127010700.

[32] Wikipedia contributors. *k-Nearest Neighbors Algorithm*. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm. Accessed: 2025-04-16 23:48 UTC. 2025.