Walk on Subdomains
Sharp Restart Point Evaluation using Machine Learning

by

Christoph Hagenauer

Bachelor of Science
University of South Carolina Beaufort 2024

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computational Science

Department of Computer Science and Mathematics

University of South Carolina Beaufort

2025

Accepted by:

W. John Thrasher, Director of Thesis

Davide Fusi, Reader

Xuwei Liang, Reader

Brian Canada, M.S. Computational Science Program Coordinator

Acknowledgments

I would like to thank my advisor, Dr. W. John Thrasher, for his guidance throughout my time in Computational Science at USCB and his support with my Master's thesis. He has been giving me great support throughout writing my thesis, be it setting up the environment, collecting and interpreting data, the model evaluation process, and most importantly reviewing my thesis paper.

I would like to thank my committee, Drs. Davide Fusi and Xuwei Liang for their support and feedback throughout the process of this thesis, but more importantly for their support through my time at USCB.

Furthermore, I would like to thank department chair Dr. Brian Canada for supporting me throughout my time at USCB as an undergraduate and graduate student. He has given me great guidance not only for my goals at USCB, but also for my future career.

Lastly, I want to thank my parents for their unwavering support! Without them I would not be who I am today, nor be where I am today.

Abstract

In this paper, I explore the feasibility of using machine learning to estimate the best sharp restart point for a Walk-on-Subdomains (WoSD) algorithm, which for this paper is a Monte Carlo algorithm that estimates certain partial differential equations. Specifically, I used an algorithm which estimates the electrostatic free energy of a biomolecule in a solution. I examine various sharp restart points with more than 500 biomolecules. Using these results, I adapted a machine learning algorithm to predict the optimal sharp restart point based on the geometry of the biomolecules. This enables the algorithm to run more efficiently as it prevents the walker from becoming entrapped indefinitely during the traversal process of the WoSD part of the algorithm. Finally, I discuss the results and give recommendations for future research directions concerning this algorithm and WoSD algorithms in general.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Walk-on-Subdomains (WoSD) is an algorithm used to estimate Monte Carlo differentials. WoSD is one of many methods to compute the electrostatic free energy of a biomolecule in a solution [14]. The design is based on a Monte Carlo estimator, which combines Walk-On-Spheres and Walk-On-Subdomains. A challenge with the Walk-On-Subdomains part of the algorithm is that it can get stuck, which causes the algorithm to take an abnormally long time to complete. This is due to the domain being divided into subdomains and the walker gets stuck in the caps of these subdomains as it traverses through them [14]. In Figure 1.1 a successful run is illustrated on the left, where the algorithm reaches the surface after 6 walks. On the right, the algorithm gets stuck at the complicated intersection of the subdomains. This is due to the fact that each WoSD step is biased to step towards points near the previous steps [14].

A potential solution was implemented and discussed in [14, 34], which implemented a sharp restart point. This implementation was tested on a small set of molecules, and the difference in runtime was discussed. They found immense variation in runtime depending on the molecule and the restart point used. A general speedup was observed with larger restart points [14, 34]. This shows enormous potential for the implementation of a hard restart point for entrapped runs. However, a "good" restart point has to be determined manually for each molecule by trial and error at the moment because every molecule has its own characteristics. A "good" restart point is considered to give the same underlying estimate while also providing

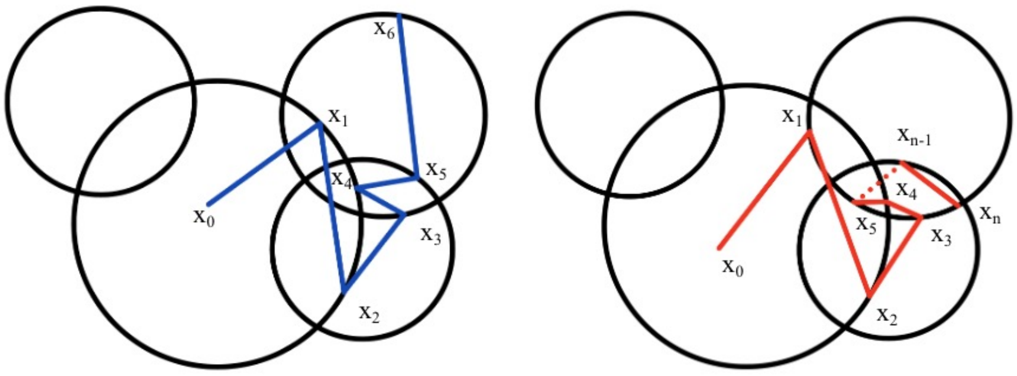


Figure 1.1 An illustration of a run of Walk-On-Subdomains, on the left successful run; on the right a run which gets stuck

overall speedup. Furthermore, a restart point gives more consistency when the algorithm is run in parallel. In addition, I explore the topic of determining a "good" sharp restart point by applying machine learning to the challenge at hand.

In this paper, I explore the feasibility of using machine learning to predict a "good" sharp restart point for the WoSD portion of the algorithm. First, let us define what a "good" sharp restart point is:

- It gives the same underlying estimate
- It provides some speedup compared to the non-restart version
- It stabilizes overall runtime

A sharp restart is specifically important for parallelizing the algorithm and receiving serious speedup. This is due to the nature of the algorithm, because if just one process in the parallel version is stuck, it affects the runtime of the entire program [14, 34]. In this paper, I focus primarily on the WoSD portion of the algorithm.

The paper is structured as follows. In Chapter 2, I explain the Background, in Chapter 3 I go over the data gathering process, in Chapter 4 I discuss the machine learning approach, and in Chapter 5 I give my conclusion and recommendation for future work.

Chapter 2

Background

The algorithm at hand estimates the electrostatic energy of a molecule in a solution. It has two Monte Carlo methods that work together to achieve this. On the exterior, Walk on Spheres (WoS) estimates the linear Poisson-Boltzmann equation; on the interior, Walk on Subdomains (WoSD) or Walk in Subdomains (WiSD) estimates the Poisson equation [21].

2.1 Walk on Spheres

WoS algorithms are used to solve Dirichlet problems for a variety of elliptic and parabolic partial differential equations. WoS was first introduced by Müller [24] to solve the N-dimensional Dirichlet problem for the Laplace equation. In this paper, Monte Carlo techniques are introduced, using stochastic models which are Markov processes. These techniques have been proven to converge with probability 1, and thus directly yield statistical estimates of the solution to the N-dimensional Dirichlet problem [24]. Müller defined the following:

D is a bounded finitely connected N-dimensional domain in a Euclidean space. $\Gamma[D]$ are the boundary points of domain D, and a point is defined as x, with x having coordinates $(x_1, x_2, x_3, ...x_n)$. There also exists a continuous function $f(x)$. The task is to find a function $u(x)$, which is continuous in $D + \Gamma[D]$ such that

$$\Delta^2 u(x) = \sum_{i=1}^{N} \frac{\partial^2 u(x)}{\partial x_i^2} = 0, \qquad x \in D, u(x) = f(x), \qquad x \in \Gamma[D]. \tag{2.1}$$

Müller first proves that the first-passage probabilities of a Brownian motion can be used to estimate the N-dimensional Dirichlet problem. Müller goes on to prove that the Spherical process can be used to simulate a Brownian motion process. The Brownian motion process is defined as a probability space (Ω, ϵ, Pr), with $\Omega = \{\omega\}$ is a set of elements $\omega, \epsilon = \{E\}$, which is a Borel field of subsets E of Ω. $Pr(E)$ is a probability measure defined on ϵ that is countably additive and satisfies the normalization condition $Pr(\Omega) = 1$. Additionally, Müller defines $X(t, \omega)$ as the well-known N-dimensional Brownian motion process starting from x, with

$$X(t,\omega) = \{(x^{(1)}(t,\omega), x^{(2)}(t,\omega), x^{(3)}(t,\omega), \ldots, x^{(N)}(t,\omega)) \mid 0 \leq t < \infty,\ \omega \in \Omega\} \quad (2.2)$$

He then uses these properties to prove that the Spherical process solves the Dirichlet problem with the following theorem:

Theorem 2.1.1. *Given any point x that belongs to a domain D with boundary $\Gamma[D]$, then with probability 1, the spherical process originating from x converges to a point of the boundary $\Gamma[D]$.*

Furthermore, Müller explains how his algorithm can be implemented for computers. For this, he points out the importance of being able to terminate computations in a reasonable time. To achieve this, Müller proposes the δ-truncation of first order. Therefore, any given walk that comes within δ of the boundary $\Gamma[D]$ is terminated [24]. δ-truncation is later referred to as ϵ-shell.

2.1.1 Alternative to WoS

With Green's function, there also exists an alternative to WoS. For a given geometry, the boundary Green's function is equivalent to the first-passage probability distribution for the Brownian motion. By precalculating Green's function, the exact first passage-probability distribution can be used to terminate walks. Therefore, Green's function is able to omit the ϵ-Shell entirely, giving an exact solution. However, Green's function is only known for few shapes and hence in many cases we depend on WoS. This is especially true for larger and more complicated geometries [15, 12].

2.1.2 Other Applications of WoS

Haji-Sheikh and Sparrow [13] expand on WoS for the Poisson equation to provide the solution to heat conduction problems, which was proved by Elepov et al. [7]. Later, Booth [3] used weighted WoS to solve homogeneous elliptic partial differential equations with constant coefficients. Another application is the calculation of capacitance. For example, Hwang, Mascagni, and Won [16] use Monte Carlo methods to compute the capacitance of the unit cube, as there does not exist an analytical expression.

2.2 Walk On Subdomains

When approaching the task, a given point is within a subdomain, and it is possible to find an exit point of said subdomain. Then, we are either in another subdomain, or the exterior. If we find ourselves in another subdomain, we repeat the process of finding an exit point and checking if we are at the exterior. This process repeats until we find the exterior. Once the exterior is found, the data is tallied on the particular equation being solved in the same manner as it would be in the WoS algorithm. The most efficient and fastest simulation for this problem is Green's function for the subdomains. Green's function is able to jump directly to the boundary of any given subdomain and this approach is also known as walk-on-subdomains (or sometimes walk-in-subdomains) [32]. However, Green's functions are only known for a few shapes as stated above; therefore, this algorithm relies on Poisson's formula as explained below.

Both WoS and WoSD are equivalent to simulating a first-passage location of a Brownian motion from a domain. WoSD is built on WoS and has the advantage of being able to simulate more complex shapes [30].

2.3 The Algorithm

On the outside (WoS) for this algorithm the charge distribution is described by the linearized Poisson-Boltzmann Equation [6, 10, 21, 22, 25]:

$$\Delta\psi(r) = \kappa^2\psi(r) \tag{2.3}$$

where κ is the inverse Debye length and associated with the Debye-Hückel theory:

$$\kappa^2 = \frac{8\pi c_b e^2}{\epsilon_e k_b \tau} \tag{2.4}$$

where c_b is the bulk salt concentration, e is the fundamental charge, ϵ_e is the dielectric constant outside of the molecule, k_b is the Boltzmann's constant, and τ is the absolute temperature. [6, 8, 11, 17, 18]

This WoS algorithm uses an ϵ-Shell to terminate walks, just like proposed by Müller [24]. Therefore, the boundary is thickened by said ϵ-Shell, which consequently introduces an error. However, this error is smaller than the statistical sampling error. The ϵ-shell induced error was investigated by Mascagni and Hwang. They found the error to be linear and that whenever accuracy beyond the preset error is required Green's function should be used [20].

For this biomolecule algorithm, the molecule can be considered a domain G, with a boundary $\Gamma[G]$. One can imagine the domain G as the union of intersecting spheres B, where each sphere represents an atom.

$$G = \bigcup_{m=1}^{M} B(x_m, r_m) \tag{2.5}$$

where x_m is is the center of each sphere, and r_m is the radius for each sphere. On the interior (WoSD) the charge distribution is modeled by $\rho(x)$ [6, 9, 21, 22]:

$$\rho(x) = \sum_{m=1}^{M} q_m \delta(x - x_m) \tag{2.6}$$

where q_m is the electrical charge, δ is the Dirac delta function, and the electrostatic potential $u(x)$ is the solution to a boundary value problem of Poisson's equation:

$$\nabla u(x) = -\frac{1}{\epsilon_i}\rho(x), \qquad x \in G \\ \text{with } G \subset \mathbb{R}^3 \tag{2.7}$$

where ϵ_i is the interior dielectric permittivity and the domain G consisting of a union M overlapping spheres. Each sphere represents one atom and the boundary of the molecule is represented by $\Gamma[G]$.

With this we can represent the potential as the sum of two functions $u(x) = u^{(0)}(x) + g(x)$ [9, 21], with

$$g(x) = \sum_{m=1}^{M} \frac{q_m}{4\pi\epsilon_i} \frac{1}{|x - x_m|}. \tag{2.8}$$

For grounded molecule the boundary values of $u(x)$ are represented by

$$u(x) = 0 \text{ or } u^{(0)}(x) = -g(x), \qquad x \in \Gamma[G]. \tag{2.9}$$

The following have been found to be true:

- An unbiased estimator for each subdomain is unbiased on the entire domain [31].
- Using WoSD as a Monte Carlo method to sample to the exit point of G has the same properties as an estimate based on direct simulation of the exit point [32].
- For a sphere S_x, centered at x_c with radius of r, we have Poisson's formula for a function u_L, which satisfies the Laplace equation at every point $x \in S(x_c, r)$:

$$u_L(x) = \int_{S(x_c,r)} p_p(x \to y) u(y) d\sigma(y) \tag{2.10}$$

where the Poisson kernel is

$$p_p(x \to y) = \frac{1}{4\pi r} \frac{r^2 - |x - x_2|^2}{|x - y|^4} \tag{2.11}$$

WoSD operates following these steps [21, 22]:

1. Choose a point $x_m = x_m^{(0)}$ at the center of sphere $S(x_m, r_m)$.
2. Generate a series of points $\{x_m^{(0)}, x_m^{(1)}, x_m^{(2)}, ..., x_m^{(N_m)}\}$.
3. At any step i in the sequence, we have a point $x_m^{(i)}$ inside a sphere $S(x_j, r_j)$ for some index j and choose $x_m^{(i+1)}$ randomly on the surface of the sphere $S(x_j, r_j)$.
4. If $x_m^{(i)}$ is at the center of the sphere, then $x_m^{(i+1)}$ is chosen according to the isotropic distribution.
5. If $x_m^{(i)}$ is not at the center of the sphere, then $x_m^{(i+1)}$ is chosen according to the Poisson kernel density on the surface of the atom.
6. Following this procedure $x_m^{(i)}$ is either on $\Gamma[G]$, or inside at least one other sphere $S(x_k, r_k)$, where $k \neq m \neq j$.
7. If $x_m^{(i)}$ is on $\Gamma[G]$, the algorithm is complete. Otherwise, Steps 5 should be repeated until $x_m^{(i)}$ is on $\Gamma[G]$.

2.4 Sharp restart in WoSD

In one of the latest iterations of the code, Thrasher [34] implemented a sharp restart point for the WoSD part of the algorithm. This was necessary because some walks became entrapped within the geometry and oscillate between two atoms. The entrapment happens due to low connectivity at any given point in the molecule and the bias of the Poisson kernel to sample towards points near the current walker's position. This increases runtime by many orders of magnitude [14]. Thrasher [14] further iterates that sharp restart is essential to making a parallel version of the code run more efficiently, since one "stuck" walk would harm the overall speedup of the program. The specific implementation is based on research regarding the first passage under restart (FPUR) [26], which shows that the best runtime is achieved by implementing a sharp restart point after a non-random number of steps. In the case of WoSD sharp restart refers to restarting a portion of the algorithm after a non-random number of steps R. In the analysis of the results the author found that restarting the algorithm did not significantly change the result of the calculation, which is true for larger values of R. Furthermore, Thrasher notes that smaller values for R were more likely to bias the estimate and to reduce the overall runtime of the algorithm [14]. After proposing two different approaches to choose R, Larger Restart Point and Calculated Restart Point (based on linear model), Thrasher concludes to use a large restart point R of 10,000,000, since it performed better in almost all cases than a calculated restart point. Furthermore, he found that individual interior walk times are more stable after adding a sharp restart point [34].

Chapter 3

Methodology

3.1 Selecting Molecules

In a first step I collected 578 molecules which were selected with a similarity search starting with an original sample size of 21 molecules. These 21 molecules were hand-picked by Thrasher for testing his parallel version of the code which currently is in beta testing. He focused on having a diversity of shapes and sizes among the selected molecules. Similar molecules were identified using the rcsbapi version 1.1.2 [27, 29]. The structure similarity had to be above a threshold of 0.7 which was deemed appropriate for this work. After the request to the protein databank a total of 578 molecules were retrieved, see Appendix A for code details. As described in [8] any deterministic linearized Poisson Boltzmann equation solver requires an input file which provides the coordinates, charge, and radii for each atom of the biomolecule. The algorithm at hand was designed with this in mind, hence it accepts pqr files as its input. Therefore, pdb2pqr software is used to prepare molecules for the use with the LPBE algorithm used in this paper.

After converting the molecules I ended up with 562 unique molecules to use with the LPBE algorithm. This number reduced due to the underlying data available from the protein databank and the resulting failures in the conversion step of pdb2pqr. The collection of these molecules and conversion with pdb2pqr was performed using a MacBook Pro equipped with an M1 Pro chip and 32 GB of memory running MacOS 15.3.1. The proteins were retrieved on February 21st 2025; and the proteins were converted using pbd2pqr version 3.7.1 locally on the Macbook running python 3.12. The parameters for the conversion of pdb2pqr were kept similar to previous research with this algorithm [8]. Therefore, the following parameters were changed from their defaults; see Appendix B for code details:

- use CHARMM as the force-field
- use PROPKA to assign protonation states at pH level 7.0
- add whitespace
- log-level of INFO

3.2 LPBE results

To gather results of the LPBE algorithm I utilized USCB's partitions in USC Hyperion cluster. Each node is equipped with an Intel Xeon Platinum 8260 CPU @2.40 GHz and 192 GB of system memory. The cluster uses slurm to schedule jobs. Each job completed for this project was run on 1 Node using 1 core at a time. Results were gathered from one codebase compiled in three different ways; the different versions are a non-restart calculation, a restarted calculation, and a geometry calculation. The non-restart calculations runs until it finishes the calculation, without having any restart condition. The restarted calculations are performed with several restart points R, restarting the WoSD portion of the algorithm until it yields an estimate within R steps. The non-restart and restart calculations were performed with seeds from $1 - 20$, and the following algorithmic parameters were set:

- 15 initial walks per atom in the bias optimization stage
- A desired error of 0.001
- An ϵ-Shell of 0.05 units

- $\epsilon_{interior}$ of 1
- $\epsilon_{exterior}$ of 80
- A temperature of 298.15 K
- The radius of the auxiliary sphere for jumps on the boundary is 0.001

These parameters were chosen to keep this work consistent with previous work [34]. The properties for the geometry were calculated non-deterministically, hence it only needed to be run once and therefore is independent of the above described parameters.

3.3 Parallel version

Originally, I planned to use the parallel version of the code, which currently is in beta testing. However, while performing initial testing a bug was discovered which caused inconsistencies within the estimates. Therefore, I used the serial version of the code. This version already has lots of performance improvements, such as using the nano-Flann library.

Chapter 4

Machine Learning

All of the results and code used to acquire the results and to conduct machine learning is available in the Appendix.

4.1 Preparing the data

To get all the results I wrote a python script to read in the results from the individual output files and store them in a pandas dataframe [23, 33]. These results include the geometry data, runtime, estimates, errors, and walk lengths of the individual runs. This was done for the restarted runs and the no-restart ones. In addition, I calculated statistic metrics such as minimum, maximum, mean, median, and standard deviation for the runtime, estimates, and errors. To validate estimates of the restarted runs a simple ANOVA [5] test was used, as described in [14]. For these tests, each restart point for a molecule was considered an independent run and each run was validated against the corresponding no-restart estimates. The results were examined using a significance of $p < 0.05$, see Appendix C for details. These results are then saved as a comma-separated file (csv), for future investigation and the use with machine learning algorithms.

For the following sections, all the code I used to perform this analysis is available in Appendix D.

4.2 Preprocessing

4.2.1 Linear Correlation

I looked at the linear correlation in the hope of dropping some features in favor of less computational overhead. However, the performance of the models degraded; therefore, no features were removed. I can also see that none of the features have any linear correlation with valid result or restart point as seen in Table 4.2.1. However, some features show a linear correlation with the average runtime. Nevertheless, the average runtime and the restart point have no linear correlation, and hence I will not be using any linear models to predict the average runtime.

	R points	Valid result	Avg Run
Valid result	0.09	1.00	0.01
R points	1.00	0.09	0.01
Avg Run	0.01	0.01	1.00
num atoms	-0.00	-0.02	0.92
avg connectivity	0.00	-0.01	0.02
tot connectivity	-0.00	-0.02	0.92
avg radius	0.00	0.01	-0.18
avg size of intersection	0.00	0.01	-0.13
tot size of intersection	-0.00	-0.02	0.92
avg dist dist molecules	-0.00	-0.02	0.78
max dist between molecules	-0.00	-0.02	0.77
max x dist	-0.00	-0.01	0.67
max y dist	-0.00	-0.02	0.75
max z dist	-0.00	-0.02	0.70
furthest left	-0.00	0.00	-0.14
furthest right	-0.00	-0.01	0.22
furthest top	-0.00	-0.01	-0.35
furthest bottom	-0.00	-0.01	-0.35
Bounding box size	-0.00	-0.00	0.54

Table 4.1 Correlation of data

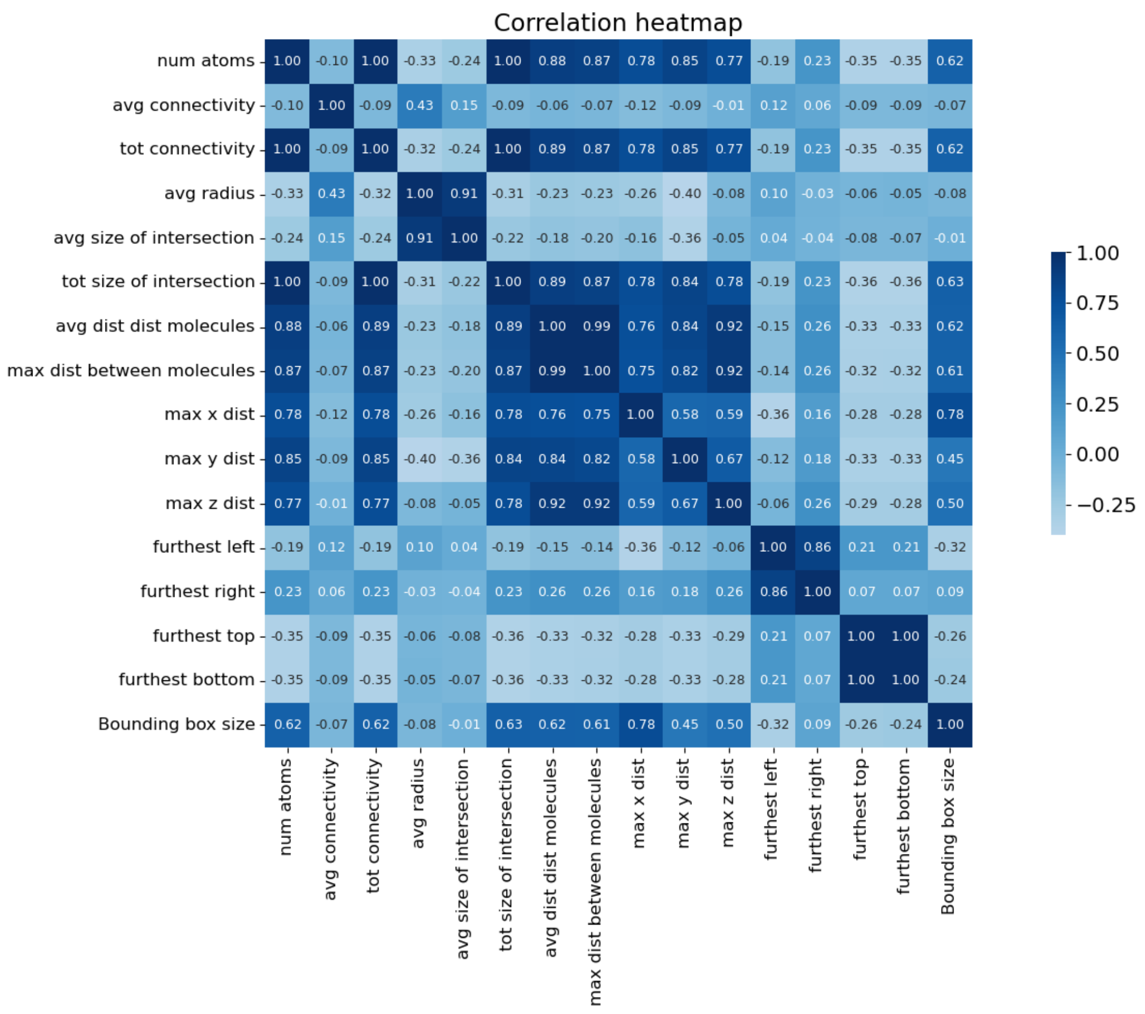


Figure 4.1 Correlation Heatmap

4.2.2 Scaling

I scaled all the continuous data present in the dataset to provide the model with uniform inputs, and it is considered a crucial step in preprocessing of the data as described in [2]. I used the Standard Scaler from scikit-learn [4]. The Standard Scaler scales an input x with $z = (x - u)/s$, where u is the sample mean and s is the sample standard deviation. In Appendix D one can observe what happens if the data is not scaled when performing a PCA analysis; the same effect is also true for a model's performance.

4.2.3 PCA Analysis

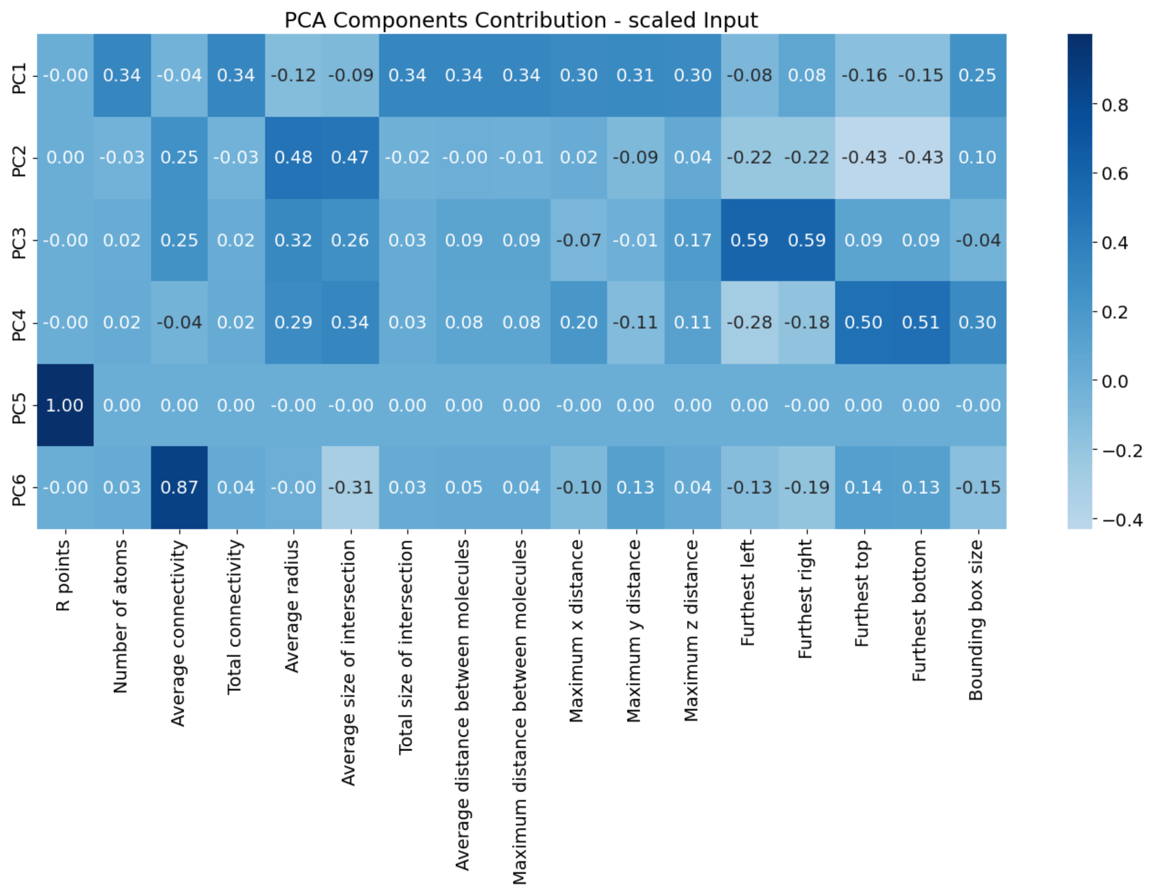
I performed Principal Component Analysis (PCA) to identify features that have a significant influence on the variability in the data. PCA works by computing orthogonal dimensions (principal components) that capture the maximum variance in the dataset, as explained in [1]. PCA can be used to perform dimensionality reduction. However, testing various PCA dimensionalities on the models which follow later in this chapter, the performance of the models significantly degraded. This is an indicator that all features are relevant, as they all add information to the first six dimensions; this can be observed in Figure 4.2. For this analysis I decided to use six principal components as they explain $> 90\%$ of variance in the underlying data. Figure 4.2 visualizes the importance of all the 17 features in each of these PCA components. This means the variance between the features is very balanced. Therefore, I also saw a significant performance drop when using only these six components for the models while testing, more on this in section 4.3.3.

4.3 Predicting if the result is going to be valid

4.3.1 Metrics

The following list explains the metrics I used to evaluate the classification models and choose the best performing one.

- True-Positive (tp) - Predicted Positive and it is an Actual Positive
- True-Negative (tn) - Predicted Negative and it is an Actual Negative
- False-Positive (fp) - Predicted Positive and it is an Actual Negative
- False-Negative (fn) - Predicted Negative and it is an Actual Positive



	R points	Number of atoms	Average connectivity	Total connectivity	Average radius	Average size of intersection	Total size of intersection	Average distance between molecules	Maximum distance between molecules	Maximum x distance	Maximum y distance	Maximum z distance	Furthest left	Furthest right	Furthest top	Furthest bottom	Bounding box size
PC1	-0.00	0.34	-0.04	0.34	-0.12	-0.09	0.34	0.34	0.34	0.30	0.31	0.30	-0.08	0.08	-0.16	-0.15	0.25
PC2	0.00	-0.03	0.25	-0.03	0.48	0.47	-0.02	-0.00	-0.01	0.02	-0.09	0.04	-0.22	-0.22	-0.43	-0.43	0.10
PC3	-0.00	0.02	0.25	0.02	0.32	0.26	0.03	0.09	0.09	-0.07	-0.01	0.17	0.59	0.59	0.09	0.09	-0.04
PC4	-0.00	0.02	-0.04	0.02	0.29	0.34	0.03	0.08	0.08	0.20	-0.11	0.11	-0.28	-0.18	0.50	0.51	0.30
PC5	1.00	0.00	0.00	0.00	-0.00	-0.00	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.00	0.00	0.00	-0.00
PC6	-0.00	0.03	0.87	0.04	-0.00	-0.31	0.03	0.05	0.04	-0.10	0.13	0.04	-0.13	-0.19	0.14	0.13	-0.15

Figure 4.2 PCA visualization

To evaluate the performance of the classifiers the following metrics were used:

- Accuracy $\frac{tp+tn}{tp+tn+fp+fn}$
- Precision $\frac{tp}{tp+fp}$
- Recall $\frac{tp}{tp+fn}$
- F-1 score $\frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$
- Receiver Operator Characteristic (ROC): measures True-Positive Rate and False-Positive Rate at a given threshold
  - True-Positive Rate $\frac{tp}{tp+fn}$
  - False-Positive Rate $\frac{fp}{fp+tn}$
- Area under ROC-curve (AUC): measures the area under the ROC-curve

4.3.2 Balance in the target variable

I provide an overview of the distribution of valid results within each restart point and the overall distribution, shown in Table 4.2. One can observe that beyond a restart point of 50 the positive percentage of results is above 90%, and beyond 200 is above 98%. This in some sense expected and confirms the findings of Thrasher and Mascagni [34]. With this distribution in mind, I explored training the models with predefined weights for each class as follows: for the positive class, the weight is 0.57, and for the negative class 4.24. The weight calculation was performed based on an example available in the Tensorflow API documentation [19].

4.3.3 Models

In this section, I perform the binary classification of the validity of the estimates which I described in Section 4.1. The goal is to separate the two classes present in

	Restart point	Total	Positive	Negative	Pos [%]	Neg [%]
0	20	562	0	562	0.00	100.00
1	25	560	12	548	2.14	97.86
2	50	562	522	40	92.88	7.12
3	75	562	533	29	94.84	5.16
4	100	562	536	26	95.37	4.63
5	125	562	546	16	97.15	2.85
6	150	562	551	11	98.04	1.96
7	175	562	545	17	96.98	3.02
8	200	562	555	7	98.75	1.25
9	225	562	554	8	98.58	1.42
10	250	562	552	10	98.22	1.78
11	500	562	556	6	98.93	1.07
12	1000	562	556	6	98.93	1.07
13	2500	562	554	8	98.58	1.42
14	5000	562	555	7	98.75	1.25
15	10000	562	556	6	98.93	1.07
16	100000	562	556	6	98.93	1.07
17	1000000	562	558	4	99.29	0.71
18	10000000	562	557	5	99.11	0.89
19	100000000	562	559	3	99.47	0.53
20	Total	11238	9913	1325	88.21	11.79

Table 4.2 Distribution of Target variable

the target while minimizing false-positives, also referred to as Type I error. This is the least desired error in this classification, as a Type I error means that the classifier predicts a valid estimate when the actual estimate is false. The other metric that is relevant for this error is the recall, as it measures the correctly classified results among all actual results within a given class. For model evaluation, the recall is reported on the negative class, which means the ratio of correctly classified negative predicted results over all actual negative results is reported. For these models, I split the data into training and testing data, with the testing data containing 40% of the total samples. Furthermore, the data was shuffled, the target variable was evenly distributed between the two sets, and the split was seeded to be able to reproduce the results. After the split, I fitted and transformed the training features using the Standard Scaler; afterwards, the testing data was transformed using the same scaler. Due to the analysis performed during the preprocessing stage, I focused my model testing on ensemble models from scikit-learn. This choice was made due to the ability of ensemble models to abstract information out of non-linearly related data. Ensemble models improve classification performance by combining multiple different base classifiers, allowing them to compensate for each other's errors and generalize better to unseen data [28]. I trained and evaluated the Gradient Boosting Classifier, Random Forest Classifier, Extra Trees Classifier, Bagging Classifier, and Ada Boost Classifier from the *ensemble* class. If the parameters are not specified for the following models, they are kept at their respective defaults. I performed some manual hyperparameter tuning for the models, and the following hyperparameters yielded the best result.

Gradient Boosting Classifier

The Gradient Boosting Classifier was initialized with 100 estimators, a learning rate of 0.01, a maximum depth of 3, a no iteration change of 5, and a validation fraction of 0.2. This resulted in an accuracy of 98%, a precision of 98%, recall of 81%, and an f-1 score of 99%. In Figure 4.3 the performance of the Gradient Boosting Classifier is visualized. The classifier has a very low false-positive rate, which is desired as described earlier.

Random Forest Classifier

The Random Forest Classifier was initialized with 1000 estimators. This resulted in an accuracy of 97%, a precision of 98%, a recall of 82%, and an f-1 score of 98%. In Figure 4.4 the performance of the Random Forest Classifier is visualized. Therefore, the classifier performs slightly better than the Gradient Boosting Classifier, especially

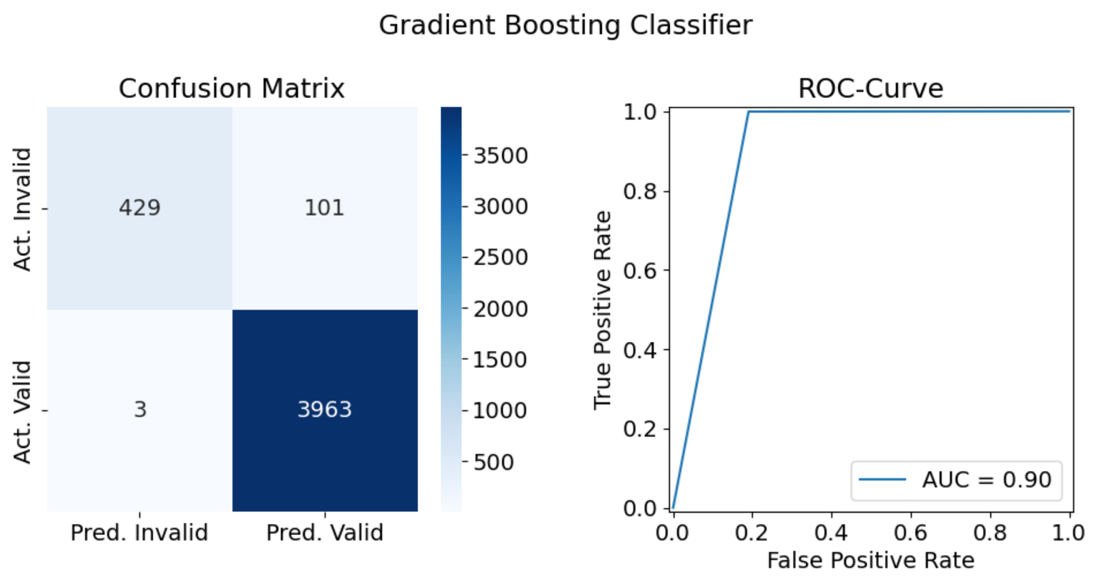


Figure 4.3 Gradient Boosting Classifier performance visualization

when it comes to false-positives, despite having slight drawbacks in false-negatives.

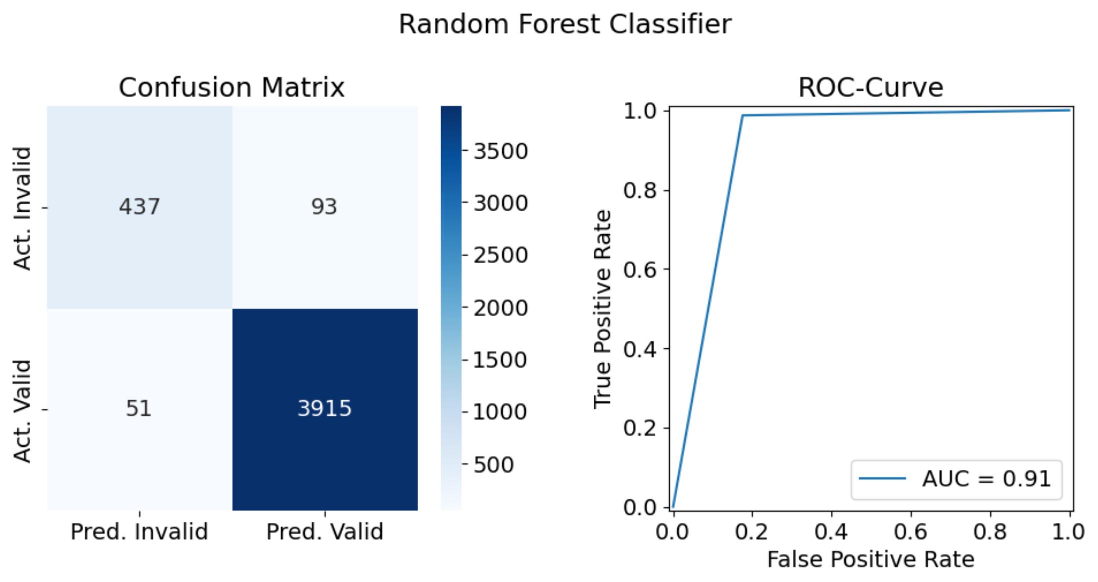


Figure 4.4 Random Forest Classifier performance visualization

Extra Trees Classifier

The Extra Trees Classifier was initialized with 1000 estimates, log-loss criterion, and class weights. This resulted in an accuracy of 91%, a precision of 94%, a recall of 52%, and an f-1 score of 95%. In Figure 4.5 the performance of the Extra Trees Classifier is visualized. Therefore, the classifier performs worse than the Gradient Boosting Classifier, especially when it comes to false-positive, only being able to classify more than 50% correctly in the negative class.

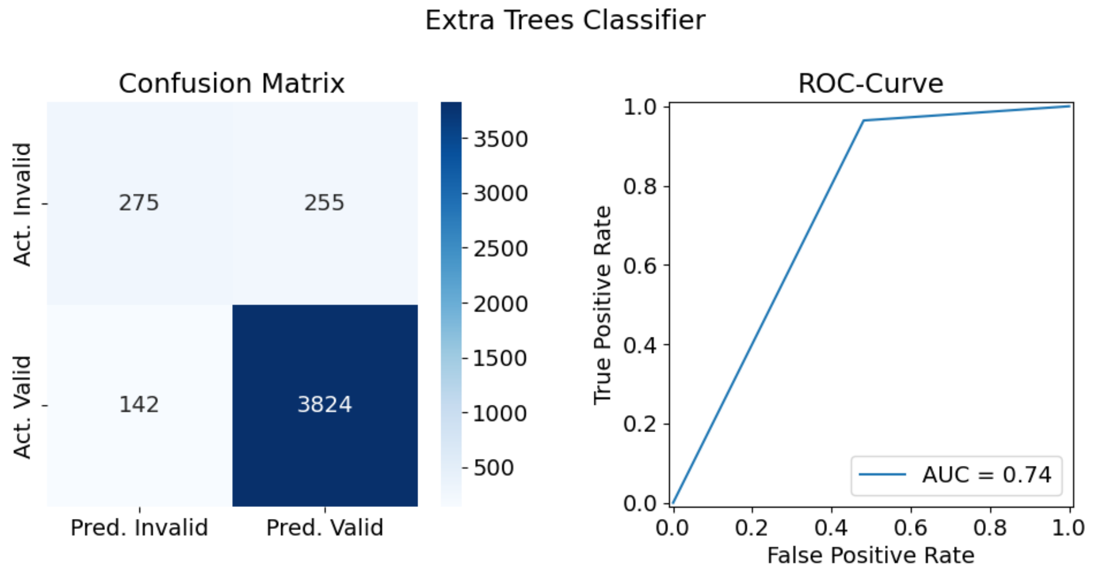


Figure 4.5 Extra Trees Classifier performance visualization

Bagging Classifier

The Bagging Classifier was initialized with 1000 estimates, criterion of log-loss, and class weights. This resulted in an accuracy of 97%, precision of 98%, recall of 82%, and f-1 score of 98%. In Figure 4.6 the performance of the Bagging Classifier is visualized. Therefore, the classifier performs slightly better than the Gradient Boosting Classifier, especially when it comes to false-positives.

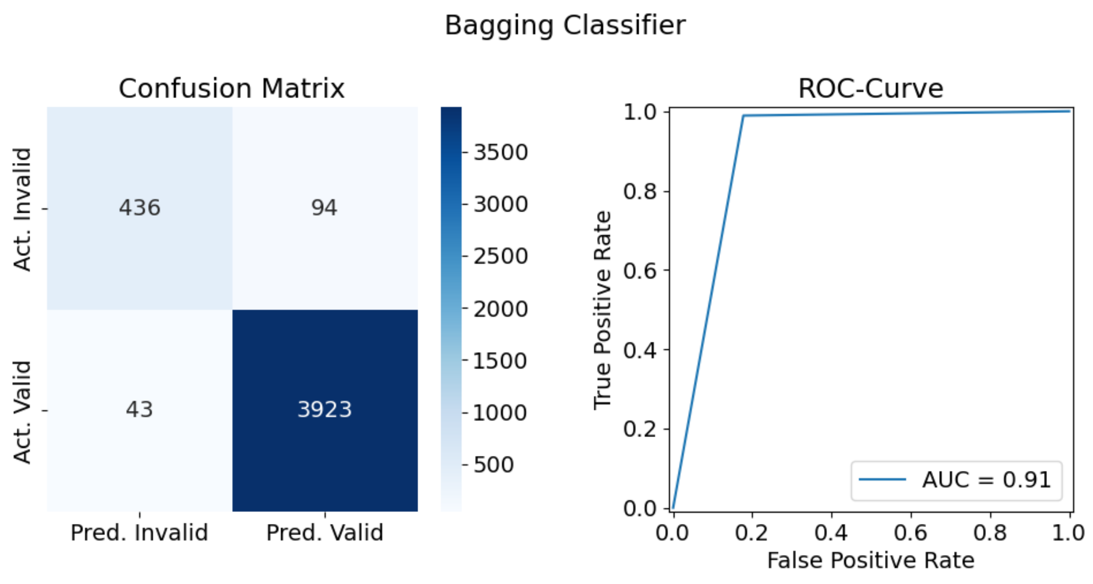


Figure 4.6 Bagging Classifier performance visualization

Ada Boosting Classifier

The Ada Boosting Classifier was initialized with 1000 estimates, criterion of log-loss, and class weights. This resulted in an accuracy of 98%, precision of 98%, recall of 81%, and f-1 score of 99%. In Figure 4.7 the performance of the Ada Boosting Classifier is visualized. Therefore, the classifier performs almost the same as the Gradient Boosting Classifier.

Considering the metrics of all of these classifiers, I would recommend using the Random Forest Classifier, as it has the lowest number of false-positives and the highest recall.

4.4 Predicting runtime

The runtime prediction is important in terms of minimizing the computational overhead potentially introduced by sharp restart.

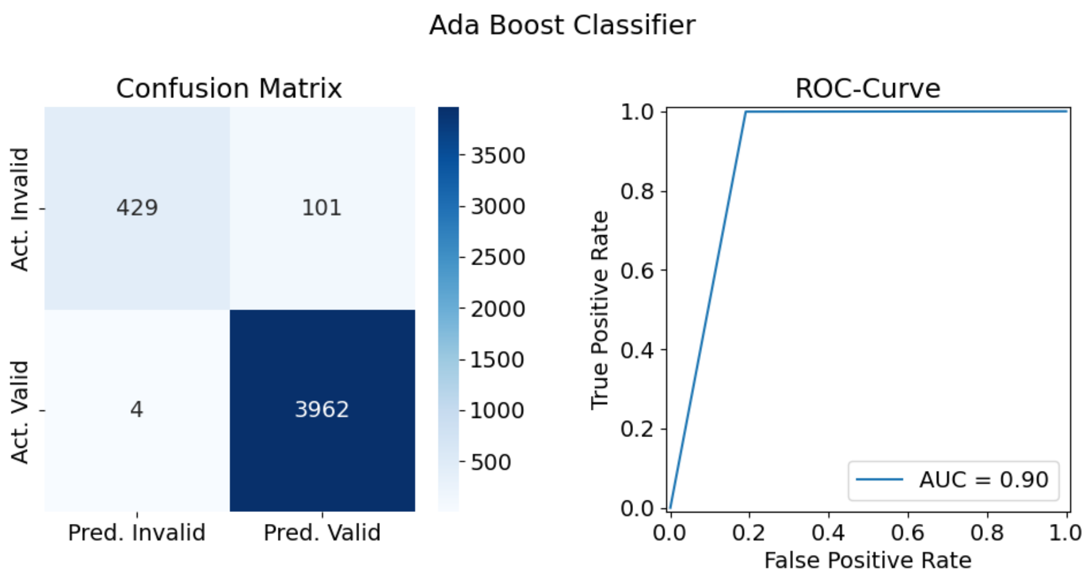


Figure 4.7 Ada Boosting Classifier performance visualization

4.4.1 Analysis of Runtime

Before I report the model used to predict the runtime, I want to focus on the effect of the restart point on the runtime. For this purpose, I grouped the runtimes by restart point and performed the five number summary. The results are presented in Table 4.4.1 and Figure 4.8. By looking at the table and figure, I can observe a minor runtime advantage for restart point 20, but for all other restart points the runtime is stable. However, when looking at individual runs the runtime versus restart point show some fluctuation. In Figure 4.9 six molecules' runtime versus restart point is visualized. In this visualization, I am able to observe a difference in runtime based on the restart point and molecule. This underlying behavior is what I wanted to model with the help of machine learning.

4.4.2 Metrics

For the regression models used in this section the following metrics are provided:

- Mean squared error (MSE): $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Restart Point	count	mean	std	min	25%	50%	75%	max
20	562.00	371.09	273.22	11.56	258.68	322.47	390.74	2788.41
25	560.00	382.85	270.95	12.32	272.41	336.15	413.36	2665.80
50	562.00	399.71	284.93	12.38	278.07	353.15	430.54	2933.30
75	562.00	402.60	287.92	12.01	282.22	353.40	432.65	2953.23
100	562.00	403.62	288.74	12.33	282.82	352.81	435.62	2952.50
125	562.00	408.90	299.81	12.18	285.01	355.30	432.97	3226.58
150	562.00	403.60	287.15	12.20	282.03	354.55	434.22	2817.62
175	562.00	408.42	298.40	12.55	282.33	354.76	435.22	3247.05
200	562.00	403.93	287.42	12.75	280.83	355.28	434.54	2819.58
225	562.00	408.46	295.85	12.18	280.85	356.88	433.66	2993.64
250	562.00	410.86	307.68	12.36	282.59	355.42	442.28	3005.72
500	562.00	410.28	308.07	12.48	283.50	356.20	442.93	3111.26
1000	562.00	410.31	306.11	12.35	281.52	356.11	443.37	2988.26
2500	562.00	411.44	307.15	12.49	283.26	354.95	444.04	2976.30
5000	562.00	409.29	304.00	13.54	281.57	357.47	444.07	3040.53
10000	562.00	411.36	307.88	12.87	281.82	358.19	443.45	3115.16
100000	562.00	410.68	305.89	13.60	282.98	356.38	443.32	3114.27
1000000	562.00	410.53	306.94	13.55	281.39	355.15	443.33	2993.97
10000000	562.00	412.25	309.65	12.81	284.84	357.59	444.11	3033.48
100000000	562.00	419.45	307.72	14.15	291.31	365.31	450.36	3047.29

Table 4.3 Five number statistic of Runtime grouped by Restart Point

- Root Mean Squared Error (RMSE): $RSME = \sqrt{MSE}$
- R^2-score (coefficient of determination):

  $1 - \frac{\sum_{i=1}^{n}(y-\hat{y})^2}{\sum_{i=1}^{n}(y-\bar{y})^2}$

4.4.3 Model

Based on the results achieved with the models performing classification I used ensemble models to predict the runtime, too. I tried the Random Forest Regressor first based on its satisfactory performance during the classification process. However, I also examined Gradient Boosting Regressor and Extra Trees Regressor. The parameters were left as their respective defaults for all the regressors. The metrics can be viewed in Table 4.4.

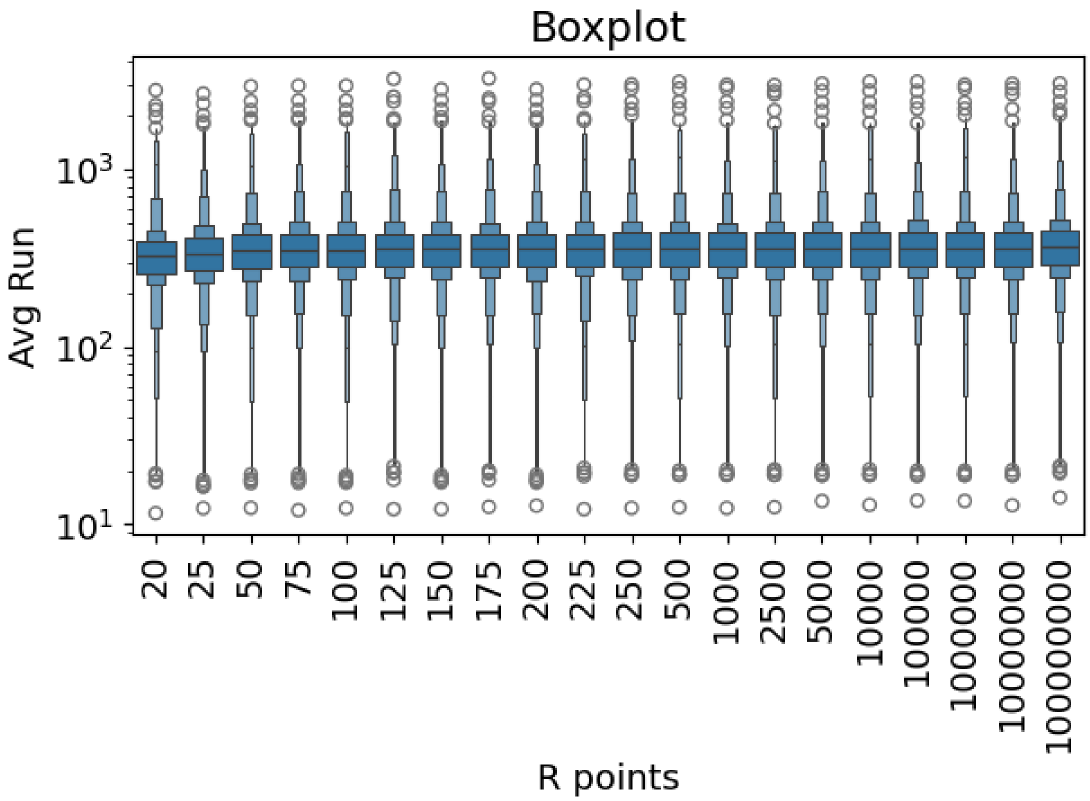


Figure 4.8 Boxplot Avg. Runtime grouped by Restart Point

Based on the results, the Random Forest Regressor performs the best again, achieving an almost perfect fit of the data. The high metrics show the ability of the regressor to model the data and make almost perfect predictions.

Regressor	MSE	RMSE	R2-score
Gradient Boosting Regressor	0.01	0.11	0.99
Random Forest Regressor	0.01	0.08	0.99
Extra Trees Regressor	0.01	0.09	0.99

Table 4.4 Regressor Metrics

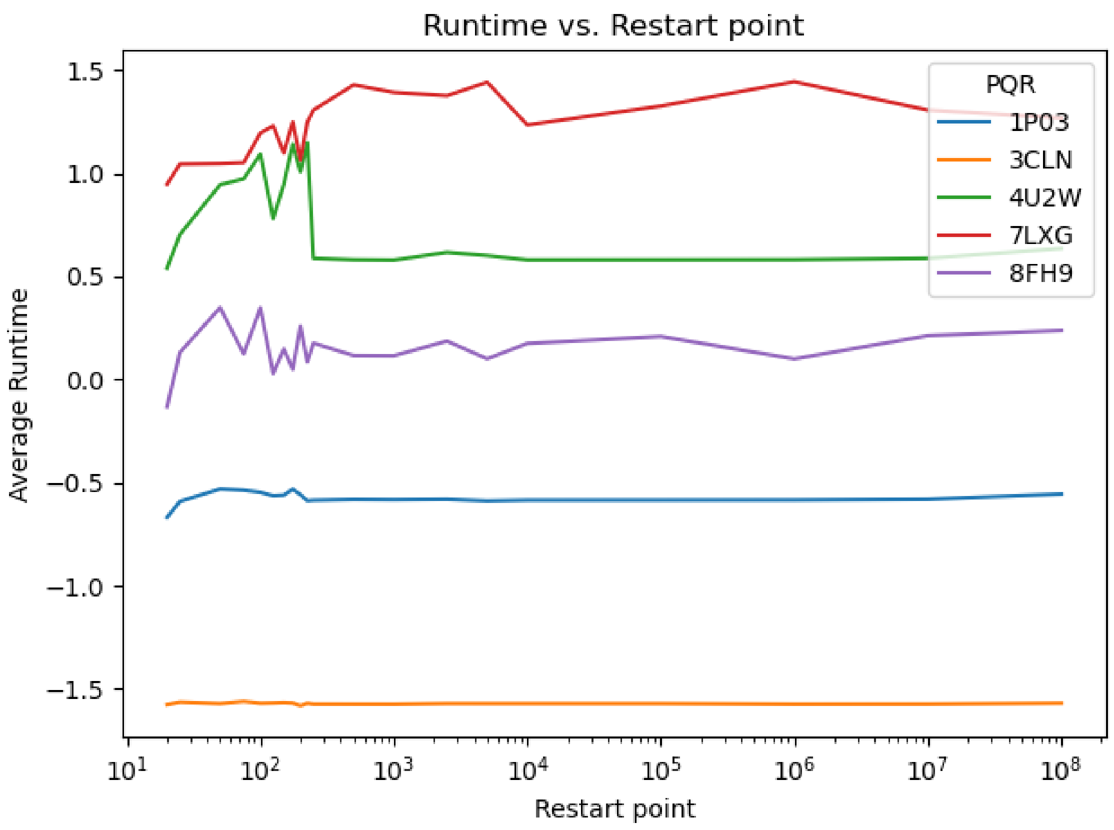


Figure 4.9 Average Runtime vs. Restart Point

Chapter 5

Conclusion

In this paper. I showed the feasibility of implementing machine learning to predict the "good" restart point. For this, I used a two-step process. First, I predicted whether the restart point will provide a valid estimate, and second, the runtime with all the valid restart points. Finally, I provide the fastest restart point out of the valid estimate predictions.

I conclude that the restart point has one major effect on the runtime, and this is stabilizing the runtime. As discussed in section 4.4 the restart point does not seem to have a major impact on the runtime when viewing the data in aggregate. However, looking more closely, I was able to see differences in runtime based on geometry and restart point. For each of the examined molecules, the runtime behaves slightly differently depending on the restart point. These findings align with the findings in [34].

Given that the true speedup comes from parallelizing the algorithm, I recommend first implementing the parallel version. Afterwards, perform the analysis I performed in this paper including more rigorous hyperparameter tuning. If that yields the same results as this paper, implement the two-step machine learning process laid out in this paper to find a "good" restart point in the algorithm. For the serial version used in this paper, choosing a "good" restart point based on this two-step process does not yield enough speedup to justify the time it takes to implement this into the algorithm. Therefore, I recommend using a larger restart point to get the advantage of a stabilized runtime, as observed in this paper and in [34].

Bibliography

[1] Hervé Abdi and Lynne J. Williams. “Principal component analysis”. In: *WIREs Computational Statistics* 2.4 (2010), pp. 433–459. DOI: https://doi.org/10.1002/wics.101. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.101. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.101.

[2] Md Manjurul Ahsan et al. “Effect of Data Scaling Methods on Machine Learning Algorithms and Model Performance”. In: *Technologies* 9.3 (2021). ISSN: 2227-7080. DOI: 10.3390/technologies9030052. URL: https://www.mdpi.com/2227-7080/9/3/52.

[3] Thomas E Booth. “Exact Monte Carlo solution of elliptic partial differential equations”. In: *Journal of Computational Physics* 39.2 (1981), pp. 396–404. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(81)90159-5. URL: https://www.sciencedirect.com/science/article/pii/0021999181901595.

[4] Lars Buitinck et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[5] Rudolf N Cardinal and Michael RF Aitken. *ANOVA for the behavioral sciences researcher*. Psychology Press, 2013.

[6] Malcolm E Davis and J Andrew McCammon. “Electrostatics in biomolecular structure and dynamics”. In: *Chemical Reviews* 90.3 (1990), pp. 509–521.

[7] BS Elepov et al. “Solution of boundary value problems by the Monte Carlo method”. In: *Science: Novosibirsk, Russia* (1980).

[8] Marcia O. Fenley et al. “Using correlated Monte Carlo sampling for efficiently solving the linearized Poisson-Boltzmann equation over a broad range of salt concentration”. In: *Journal of Chemical Theory and Computation* 6 (1 2010). ISSN: 15499618. DOI: 10.1021/ct9003806.

[9] Charles Fleming, Michael Mascagni, and Nikolai Simonov. "An Efficient Monte Carlo Approach for Solving Linear Problems in Biomolecular Electrostatics". In: *Computational Science – ICCS 2005*. Ed. by Vaidy S. Sunderam et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 760–765. ISBN: 978-3-540-32118-7.

[10] F. Fogolari, A. Brigo, and H. Molinari. "The Poisson–Boltzmann equation for biomolecular electrostatics: a tool for structural biology". In: *Journal of Molecular Recognition* 15.6 (2002), pp. 377–392. DOI: `https://doi.org/10.1002/jmr.577`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/jmr.577`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/jmr.577`.

[11] Federico Fogolari et al. "Biomolecular electrostatics with the linearized Poisson-Boltzmann equation". In: *Biophysical Journal* 76 (1 I 1999). ISSN: 00063495. DOI: `10.1016/S0006-3495(99)77173-0`.

[12] James A. Given, Michael Mascagni, and Chi Ok Hwang. "Continuous path brownian trajectories for diffusion monte carlo via first- and last-passage distributions". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2179. 2001. DOI: `10.1007/3-540-45346-6_4`.

[13] A. Haji-Sheikh and E. M. Sparrow. "The Solution of Heat Conduction Problems by Probability Methods". In: *Journal of Heat Transfer* 89.2 (May 1967), pp. 121–130. ISSN: 0022-1481. DOI: `10.1115/1.3614330`. eprint: `https://asmedigitalcollection.asme.org/heattransfer/article-pdf/89/2/121/5718589/121\_1.pdf`. URL: `https://doi.org/10.1115/1.3614330`.

[14] Preston Hamlin et al. "Geometry entrapment in Walk-on-Subdomains". In: *Monte Carlo Methods and Applications* 25 (4 2019). ISSN: 15693961. DOI: `10.1515/mcma-2019-2052`.

[15] Chi-Ok Hwang, Michael Mascagni, and James A. Given. In: *Monte Carlo Methods and Applications* 7.3-4 (2001), pp. 213–222. DOI: `doi:10.1515/mcma.2001.7.3-4.213`. URL: `https://doi.org/10.1515/mcma.2001.7.3-4.213`.

[16] Chi-Ok Hwang, Michael Mascagni, and Taeyoung Won. "Monte Carlo methods for computing the capacitance of the unit cube". In: *Mathematics and Computers in Simulation* 80.6 (2010), pp. 1089–1095.

[17] B. Z. Lu et al. *Recent progress in numerical methods for the Poisson-Boltzmann equation in biophysical applications*. 2008.

[18] Travis MacKoy et al. “Numerical optimization of a walk-on-spheres solver for the linear Poisson-Boltzmann equation”. In: *Communications in Computational Physics*. Vol. 13. 2013. DOI: 10.4208/cicp.220711.041011s.

[19] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[20] Michael Mascagni and Chi Ok Hwang. “ϵ-shell error analysis for "Walk On Spheres" algorithms”. In: *Mathematics and Computers in Simulation* 63 (2 2003). ISSN: 03784754. DOI: 10.1016/S0378-4754(03)00038-7.

[21] Michael Mascagni and Nikolai A. Simonov. “Monte Carlo method for calculating the electrostatic energy of a molecule”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2657 (2003). ISSN: 16113349. DOI: 10.1007/3-540-44860-8_7.

[22] Michael Mascagni and Nikolai A. Simonov. “Monte Carlo methods for calculating some physical properties of large molecules”. In: *SIAM Journal on Scientific Computing* 26 (1 2005). ISSN: 10648275. DOI: 10.1137/S1064827503422221.

[23] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[24] Mervin E. Müller. “Some Continuous Monte Carlo Methods for the Dirichlet Problem”. In: *The Annals of Mathematical Statistics* 27 (3 1956). ISSN: 0003-4851. DOI: 10.1214/aoms/1177728169.

[25] Gábor Náray-Szabó and Arieh Warshel. *Computational approaches to biochemical reactivity*. Vol. 19. Springer Science & Business Media, 2002.

[26] Arnab Pal and Shlomi Reuveni. “First Passage under Restart”. In: *Phys. Rev. Lett.* 118 (3 Jan. 2017), p. 030603. DOI: 10.1103/PhysRevLett.118.030603. URL: https://link.aps.org/doi/10.1103/PhysRevLett.118.030603.

[27] Dennis W. Piehl et al. “rcsb-api: Python Toolkit for Streamlining Access to RCSB Protein Data Bank APIs”. In: *Journal of Molecular Biology* (2025), p. 168970. ISSN: 0022-2836. DOI: https://doi.org/10.1016/j.jmb.2025.168970. URL: https://www.sciencedirect.com/science/article/pii/S0022283625000361.

[28] Akhlaqur Rahman and Sumaira Tasnim. “Ensemble Classifiers and Their Applications: A Review”. In: *International Journal of Computer Trends and Technology* 10.1 (Apr. 2014), pp. 31–35. ISSN: 2231-2803. DOI: 10.14445/22312803/

ijctt-v10p107. URL: http://dx.doi.org/10.14445/22312803/IJCTT-V10P107.

[29] Yana Rose et al. “RCSB Protein Data Bank: Architectural Advances Towards Integrated Searching and Efficient Access to Macromolecular Structure Data from the PDB Archive”. In: *Journal of Molecular Biology* 433.11 (2021). Computation Resources for Molecular Biology, p. 166704. ISSN: 0022-2836. DOI: https://doi.org/10.1016/j.jmb.2020.11.003. URL: https://www.sciencedirect.com/science/article/pii/S0022283620306227.

[30] Karl K Sabelfeld and Nikolai A Simonov. *Stochastic methods for boundary value problems: numerics for high-dimensional PDEs and applications.* Walter de Gruyter GmbH & Co KG, 2016.

[31] Karl Karlovich Sabelfeld. “Monte Carlo methods in boundary value problems”. In: *(No Title)* (1991).

[32] Nikolai A Simonov. “A random walk algorithm for the solution of boundary value problems with partition into subdomains”. In: *Methods and algorithms for statistical modelling. Akad. Nauk SSSR Sibirsk. Otdel., Vychisl. Tsentr, Novosibirsk* (1983), pp. 48–58.

[33] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[34] W. John Thrasher and Michael Mascagni. “Examining sharp restart in a Monte Carlo method for the linearized Poisson–Boltzmann equation”. In: *Monte Carlo Methods and Applications* 26 (3 2020). ISSN: 15693961. DOI: 10.1515/mcma-2020-2069.

Appendix A

Molecule Similarity Search

```
import requests
import os
from rcsbapi.search import GroupBy, StructSimilarityQuery

existing_molecules = [
    '4u2w', '3cln', '1tqg', '1a6m',
    '1u1r', '1nwz', '1ah0', '4s2l',
    '1iua', '5bmx',  '1hje', '2fwh',
    '1edm', '1tg0', '1f94', '6hq2',
    '1etl', '1u61', '1ssx', '2bf9',
    '1cex'
    ]
new_molecules = set()
# Get new molecules with Similarity Search
for molecule in existing_molecules:
    q2 = StructSimilarityQuery(structure_search_type="entry_id",
    entry_id=molecule)
    results = list(q2(results_verbosity="minimal"))
    for result in results:
        if result['score'] > 0.7:
            new_molecules.append(result['identifier'])

print(f"We have {len(new_molecules)} molecules now.")

def download_pdb(pdb_id, save_dir=os.getcwd()+'/new_pbds'):
    """Download a PDB file from RCSB API and save it locally."""
    url = f"https://files.rcsb.org/download/{pdb_id}.pdb"
    response = requests.get(url)

    if response.status_code == 200:
        file_path = f"{save_dir}/{pdb_id}.pdb"xw
        with open(file_path, "wb") as f:
            f.write(response.content)
        #print(f"Downloaded: {file_path}")
    else:
        print(f"Failed to download {pdb_id}: {response.status_code}")

for molecule in molecules:
   download_pdb(molecule)
```

Appendix B

PDB to PQR

```python
import argparse
import os
from concurrent.futures import ThreadPoolExecutor
import subprocess

def convert_file(source_path, dest_dir):
    """
    Convert a single .pdb file to .pqr and save it to dest_dir.
    Returns True if successful, False otherwise.
    """
    filename = os.path.basename(source_path)
    pqr_path = os.path.join(dest_dir, f"{os.path.splitext(filename)[0]}.pqr")

    command = [
    "pdb2pqr",
    "--ff=CHARM",
    "--whitespace",
    "--titration-state-method=propka",
    "--with-ph=7",
    "--log-level=INFO",
    source_path,
    pqr_path
    ]

    # Run the command
    try:
        result = subprocess.run(command, check=True, text=True, capture_output=
    True)
        print("Output:", result.stdout)
        print("Errors:", result.stderr)
    except subprocess.CalledProcessError as e:
        print(f"Command failed with error: {e.stderr}")

def convert_pdb_to_pqr(source_dir, dest_dir, max_workers=1):
    """
    Convert all .pdb files in source_dir to .pqr files using pdb2pqr,
    saving them to dest_dir with optional parallel processing.
    """
    # Create destination directory if it doesn't exist
    if not os.path.exists(dest_dir):
        os.makedirs(dest_dir)
    # Get list of .pdb files
    pdb_files = [os.path.join(source_dir, f) for f in os.listdir(source_dir)
                 if f.endswith(".pdb")]
    # Process files in parallel
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = []
        for file_path in pdb_files:
            future = executor.submit(convert_file, file_path, dest_dir)
            futures.append((file_path, future))

        # Track progress and results
        successes = 0
        failures = 0

        for file_path, future in futures:
            try:
                result = future.result()
                if result:
                    successes += 1
            except Exception as e:
                print(f"Error processing {file_path}: {str(e)}")
                failures += 1

    print(f"\nConversion complete: {successes} successful, {failures} failed")

def create_inp_large_file(base_name, output_dir):
    """
    Creates an .inp.large file for a given molecule.

    Args:
        base_name (str): The base name of the corresponding .pqr file without
    extension.
        output_dir (str): Directory where the .inp.large file will be saved.
    """
    # Define the content template
    content = (
        f"{base_name}.pqr\n"
        "15\n"
        ".001\n"
        ".05\n"
        "1\n"
        "80\n"
        ".01\n"
        "10\n"
        "298.15\n"
        ".1\n"
        "1\n"
    )

    # Construct the full file path
    file_path = os.path.join(output_dir, f"{base_name}.inp.large")

    try:
        # Write the content to the file
        with open(file_path, 'w') as f:
            f.write(content)
        print(f"Successfully created .inp.large file: {file_path}")
    except Exception as e:
        print(f"Error creating .inp.large file: {e}")
```

```
# Run main
def main()
    parser = argparse.ArgumentParser(description='Convert .pdb files to .pqr using
     pdb2pqr')
    parser.add_argument('--source', type=str, default=os.getcwd()+"/new_pbds",
                        help='Source directory containing .pdb files (default:
    current directory)')
    parser.add_argument('--dest', type=str, default=os.getcwd()+"/new_pqrs",
                        help='Destination directory for .pqr files (default: current
     directory)')
    parser.add_argument('--jobs', type=int, default=15,
                        help='Number of parallel processes to use (default: 1)')
    args = parser.parse_args()
    print(f"Processing .pdb files in {args.source}")
    print(f".pqr files will be saved to {args.dest}")
    print(f"Using {args.jobs} parallel process(es)")
    convert_pdb_to_pqr(args.source, args.dest, max_workers=args.jobs)
    molecules = [
        '4U2W', '3CLN', '1CLL', '1CLM', '5E1N', '2V01', '5E1K', '1TQG', '1A6M',
        '1A6K', '5YCE', '2Z6T', '7VDN', '1JPB', '1A6N', '1A6G', '1JP9', '1L2K',
        '8FDJ', '2Z6S', '5ILP', '5XL0', '2ZT0', '2ZSY', '2ZSO', '1JP8', '3E55',
        '3ECX', '2ZSN', '2ZT1', '1JP6', '5IKS', '5ILR', '3EDA', '2ZSZ', '3E5O',
        '5UTB', '2MYD', '1VXH', '2ZSX', '2MYB', '2ZSS', '1MBC', '2MYE', '2ZST',
        '3ECZ', '3ED9', '2ZT2', '4NS2', '2ZT4', '2ZSQ', '3EDB', '2ZSR', '1AJH',
        '1AJG', '2EKU', '2ZT3', '3E4N', '7DDS', '1VXC', '3ECL', '1VXE', '2SPO',
        '6JP1', '1VXF', '2ZSP', '1SWM', '7DDU', '3RJN', '1MNO', '1YOG', '6E02',
        '1VXA', '1V9Q', '5WJK', '1DTM', '2MYA', '5VZP', '1YOH', '2MGM', '4TWU',
        '3E5I', '1IOP', '112M', '1J3F', '1CH2', '2MYC', '1VXG', '1MBI', '1UFJ',
        '6CF0', '3U3E', '5KKK', '108M', '1MBO', '7XCF', '1IRC', '1SPE', '2MGL',
        '1DUK', '4OF9', '8J4L', '1YCB', '2IN4', '7SLH', '6KRC', '1PMB', '5VZO',
        '3HC9', '8J4K', '4FWX', '7XCQ', '7CEZ', '5B85', '1BZP', '6E03', '1BZ6',
        '104M', '2MGH', '2OHA', '1MTI', '1M6M', '5XKV', '5ZZF', '1BZR', '2MGE',
        '3HEP', '4PNJ', '1MYG', '5YCG', '2E2Y', '5UTC', '107M', '5M3S', '3ASE',
        '2W6W', '1FCS', '1MOB', '1DUO', '1ABS', '3H58', '1CIO', '3H57', '1MWD',
        '6Z4R', '3A2G', '1MOD', '7SPF', '1VXD', '4IT8', '6E04', '1CH7', '6F17',
        '2MBW', '1OFK', '5HLX', '4HOB', '1MWC', '1JW8', '4MBN', '2MGD', '4QAU',
        '4PQ6', '1UFP', '1MOA', '111M', '1CH1', '1MLS', '1MYJ', '5UTA', '3K9Z',
        '1CPW', '3BA2', '5UT8', '1MLO', '6M8F', '1CQ2', '5UT9', '1VXB', '2MGJ',
        '6G5A', '6F1A', '1BVD', '3NML', '7XC9', '3V2Z', '2CMM', '1OBM', '2MB5',
        '2MGK', '2MGG', '1MLG', '7EHX', '4LPI', '4PQC', '1TES', '2EVK', '5VRT',
        '1CO9', '110M', '1MLR', '1YOI', '106M', '3HEO', '2MGA', '5UT7', '6D45',
        '1MYI', '1MTK', '1MLL', '1MGN', '1MOC', '6Z4T', '1DO3', '1MDN', '7VUC',
        '1LUE', '1CH3', '2OH8', '1O16', '1U7R', '1MTJ', '1H1X', '2MGI', '5YCH',
        '1MBD', '1MLQ', '1MYH', '1YCA', '2SPM',
        '6G5B', '1MLN', '6F18', '8F9N', '4NXC', '109M', '1MLK', '1MYM', '1LTW',
        '1CP0', '3O89', '1CIK', '6BMG', '1CH9', '2D6C', '5HLQ', '8FB0', '4H07',
        '5OJ9', '1J52', '2SPN', '1MNI', '1CP5', '1MLH', '1OFJ', '2MGC', '2G0S',
        '2G10', '2G11', '2G12', '2G14', '1MYZ', '1MZ0', '5YCI', '5VNU', '1U1R',
        '1U10', '1PO6', '1U1P', '1U1Q', '1U1L', '1U1M', '9GPJ', '1NWZ', '3PYP',
        '6MHN', '1F9I', '6MKT', '1OT9', '1OT6', '6UN2', '1UWN', '1UWP', '1S4S',
        '6UMY', '3UME', '6MHI', '1AH0', '1AH3', '1AH4', '1EK0', '5Y7N', '2FZB',
        '3Q65', '5HA7', '6Y03', '2F2K', '6T5G', '4WEV', '8FH8', '2NVC', '4JIR',
        '8BJL', '2NVD', '4NKC', '4RPQ', '2PDJ', '3P2V', '2PDK', '4PR4', '4XZI',
        '4PRR', '8FH7', '2PDL', '2IKH', '2PD9', '2PDG', '3LEN', '3V35', '2AGT',
        '8FH6', '2IKG', '6SYW', '3U2C', '3Q67', '2PDI', '4ICC', '3M64', '2PDH',
        '4Q7B', '2PEV', '2PDC', '5OU0', '8AE9', '2HV5', '6TD8', '2IKJ', '4GA8',
        '5OUK', '4YU1', '2PZN', '3MB9', '3GHR', '2PDQ', '4PRT', '3LBO', '1Z89',
        '4GCA', '8FH9', '1PWL', '3DN5', '2PDB', '1Z8A', '2HVO', '8FH5', '2FZ9',
        '5LIK', '1X97', '3GHS', '4YS1', '4XZH', '2ACQ', '3MC5', '3T42', '1MAR',
        '2DUZ', '5LIU', '3GHU', '2FZ8', '5LIX', '2PDF', '2PDN', '2DVO', '4IGS',
        '2PDU', '3BCJ', '3M4H', '3MOI', '4XZL', '1X96', '2PDW', '2PF8', '1ZUA',
        '1AZ2', '5OUJ', '4XZM', '2DUX', '2PD5', '2PDM', '1X98', '4PUW', '4GAB',
        '6Y1P', '1T41', '8B3N', '4PUU', '4XZN', '2PDP', '2R24', '6XUM', '3GHT',
        '4LAU', '5M2F', '4LBS', '4QR6', '2PDY', '8A4N', '2ACU', '8CNP', '1AZ1',
        '6TUF', '4JIH', '1PWM', '4QX4', '5LIW', '2HVN', '4S2L', '5HAR', '7KHY',
        '7KHZ', '5ODZ', '5HAQ', '4S2M', '7KHQ', '5OE0', '5FDH', '7AUX', '7ASS',
        '4S2J', '6NLW', '5FAQ', '5DVA', '5FAS', '3HBR', '6HB8', '6P99', '7AW5',
        '4WMC', '5OE2', '7PEH', '6P9C', '7K5V', '6UVK', '5HAP', '6V1O', '6P96',
        '7L8O', '5FAT', '6I5D', '7LXG', '7O5N', '6XQR', '6ZRJ', '7PSF', '5DTS',
        '7PEP', '5OFT', '7R6Z', '1IUA', '2FLA', '3A38', '2AMS', '5WQQ', '5WQR',
        '3A39', '6AIQ', '5D8V', '7VOS', '6AIR', '1EYT', '5BMX', '4ZOS', '3PSV',
        '4ZOJ', '3PSW', '4ZZ9', '2VFG', '2VFE', '4YWI', '2VFF', '3PY2', '5BRB',
        '1VGA', '1WOB', '2VFD', '2VFH', '1M7P', '1LZO', '3PWA', '1HJE', '2FWH',
        '1EDM', '1TGO', '1F94', '6HQ2', '1ETL', '1U61', '1SSX', '1QRX', '1TAL',
        '2H5C', '1QRW', '2ULL', '3URD', '3M7T', '1GBA', '3URC', '1QQ4', '1GBL',
        '2ALP', '1P01', '1GBD', '1BOQ', '1P05', '1GBB', '1GBC', '1GBJ', '1GBE',
        '3URE', '1P09', '1GBK', '3QGJ', '1P03', '1GBM', '2H5D', '9LPR', '1P04',
        '7LPR', '8LPR', '2LPR', '6LPR', '1P06', '1P02', '3LPR', '5LPR', '1GBF',
        '1GBH', '1P10', '2BF9', '1CEX', '1XZL', '1CUX', '1FFE', '1FFA', '1CUS',
        '1XZF', '1XZJ', '1XZG', '1CUF', '1XZA', '1FFD', '1XZC', '1XZM', '1CUU',
        '1FFC', '1XZB', '1CUY', '1AGY', '1XZH', '1CUG', '1CUA', '1XZI', '1XZK',
        '1CUC', '1CUH', '1CUD', '1OXM', '1CUE', '3ESA', '1CUI', '1FFB', '3ESC',
        '1XZE', '3ESD'
        ]
    for molecule in molecules:
        create_inp_large_file(molecule, os.getcwd()+"/new_pqrs")

if __name__ == "__main__":
```

Appendix C

ANOVA validation

```
import pandas as pd
import os

no_restart_results = pd.read_csv("NanoFlann_no_restart/restart_NO_results.csv")
restart_results = pd.read_csv('NanoFlann_restart/restart_results.csv')
geom_Data = pd.read_csv('geomData/geomData.csv', index_col=0)
for pqr in no_restart_results.PQR.values:
    if not (pqr in geom_Data.PQR.values):
        if debug: print(pqr)
        no_restart_results.drop(no_restart_results.loc[no_restart_results.PQR ==
    pqr].index, inplace=True)
        restart_results.drop(restart_results.loc[restart_results.PQR == pqr].index
    , inplace=True)
# Setup new column for result validation; default to false
restart_results.insert(loc=0, column='Valid_result', value=0)
# Adding column with valid / not valid result
from scipy.stats import f_oneway
import numpy as np
valid_results = 0
loops = 0
if debug: print(restart_results)
for pqr in no_restart_results.PQR.unique():
    for restart_point in restart_results['R points'].unique():
        restart_estimates = restart_results.loc[(restart_results.PQR == pqr) & (
    restart_results['R points'] == restart_point), 'estimate 1':'estimate 20'].
    values
        no_restart_estimate_mean = no_restart_results.loc[(no_restart_results.PQR
    == pqr), 'estimate 1':'estimate 20'].values
        index = restart_results.loc[(restart_results.PQR == pqr) & (
    restart_results['R points'] == restart_point)].index
        restart_estimates = restart_estimates.reshape(-1,1)
        no_restart_estimate_mean = no_restart_estimate_mean.reshape(-1,1)
        _, p_value = f_oneway(restart_estimates, no_restart_estimate_mean)
        if p_value > 0.05:
            valid_results += 1
            if debug:
                print(f'p value: {p_value}')
                print('True')
                print(index, pqr, restart_point, type(index))
            restart_results.loc[index, 'Valid_result'] = 1
        else:
            if debug:
                print("False")
                print(index, type(index))
            restart_results.loc[index, 'Valid_result'] = 0
print('Number of valid results:',valid_results)
print(restart_results)
print(restart_results.loc[restart_results['Valid_result'] == 0, ['R points']].
    value_counts())
# dropping all estimates, errors, and individual runtimes
restart_results.drop(columns=restart_results.loc[:,'estimate 1':].columns, inplace
    =True)
restart_results.drop(columns=restart_results.loc[:,'run 0':'run 19'].columns,
    inplace=True)
restart_results.drop(columns=['Node count', 'Core count', 'Max Run','Min Run', '
    Std Run'], inplace=True)
# Dropping molecules where the no restart version did not finish
ml_data.drop(ml_data.loc[ml_data['PQR']=='2SPO'].index, inplace=True)
ml_data.drop(ml_data.loc[ml_data['PQR']=='1AZ1'].index, inplace=True)
ml_data.drop(ml_data.loc[ml_data['PQR']=='1CUD'].index, inplace=True)
ml_data.to_csv('ml_data.csv')
```

Appendix D

Machine Learning with Scikit-learn

```
import pandas as pd
pd.set_option('display.max_columns', None)
pd.set_option('display.float_format', '{:.3f}'.format)
import numpy as np
import random
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier,
    ExtraTreesClassifier, BaggingClassifier,
from sklearn.ensemble import GradientBoostingRegressor, ExtraTreesRegressor,
    RandomForestRegressor
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score,
    RocCurveDisplay, classification_report, mean_squared_error, r2_score,
    root_mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

np.random.seed(42)
random.seed(42)

ml_data = pd.read_csv('ml_data.csv', index_col=0)
print(f"Number of unique molecule: {len(ml_data.PQR.unique())}")
print(ml_data.sort_values(['num atoms', 'Avg Run']))
print(ml_data.describe())
# Correlation
plt.figure(figsize=(16, 10))
sns.heatmap(ml_data.drop(columns=['PQR', 'R points', 'Valid_result', 'Avg Run']).
    corr(), cmap='Blues', center=0, annot=True, fmt='.2f',
         annot_kws={"size": 9}, square=True, cbar_kws={"shrink": 0.4})
plt.xticks(rotation=270, fontsize=12)
plt.yticks(fontsize=12)
plt.title("Correlation heatmap")
plt.tight_layout()
#PCA-scaled
scaler = StandardScaler()
X = scaler.fit_transform(ml_data.drop(columns=['Valid_result','PQR', 'Avg Run']))
pca = PCA(n_components=6)
X_train_pca = pca.fit_transform(X)
# Print explained variance ratios
print("Explained variance ratios:")
for i, ratio in enumerate(pca.explained_variance_ratio_):
    print(f"PC{i+1}: {ratio:.4f} ({ratio*100:.2f}%)")
print(f"\nTotal variance explained: {sum(pca.explained_variance_ratio_)*100:.2f}%"
    )
# 2. PCA Components Visualization
plt.figure(figsize=(14, 10))
components_df = pd.DataFrame(
    pca.components_,
    columns=[col for col in ml_data.columns if col not in ['Valid_result', 'PQR',
    'Avg Run']],
    index=[f'PC{i+1}' for i in range(pca.n_components_)])
sns.heatmap(components_df, cmap='Blues', center=0, annot=True, fmt='.2f')
plt.title('PCA Components Contribution - scaled Input')
plt.tight_layout()
plt.show()
# PCA-no scaling
X = ml_data.drop(columns=['Valid_result','PQR', 'Avg Run'])
pca = PCA(n_components=6)
X_train_pca = pca.fit_transform(X)
# Print explained variance ratios
print("Explained variance ratios:")
for i, ratio in enumerate(pca.explained_variance_ratio_):
    print(f"PC{i+1}: {ratio:.4f} ({ratio*100:.2f}%)")
print(f"\nTotal variance explained: {sum(pca.explained_variance_ratio_)*100:.2f}%"
    )
# 2. PCA Components Visualization
plt.figure(figsize=(14, 9))
components_df = pd.DataFrame(
    pca.components_,
    columns=list(X.columns),
    index=[f'PC{i+1}' for i in range(pca.n_components_)])
sns.heatmap(components_df, cmap='Blues', center=0, annot=True, fmt='.2f')
plt.title('PCA Components Contribution')
plt.tight_layout()
plt.show()
# Target variable distribution
total_list, pos_list, neg_list, r_point_list = [], [], [], []
for r_point in ml_data['R points'].unique():
    result = ml_data.loc[ml_data['R points'] == r_point, 'Valid_result']
    if len(result.unique()) > 1:
        neg, pos = np.bincount(result)
        total = neg + pos
        total_list.append(total)
        neg_list.append(neg)
        pos_list.append(pos)
        r_point_list.append(r_point)
    elif result.all() == 1:
        total_list.append(len(result))
        neg_list.append(0)
        pos_list.append(len(result))
        r_point_list.append(r_point)
    else:
        total_list.append(len(result))
        neg_list.append(len(result))
        pos_list.append(0)
        r_point_list.append(r_point)
r_point_list.append("Total")
neg, pos = np.bincount(ml_data['Valid_result'])
total = neg + pos
```

```
pos_list.append(pos)
neg_list.append(neg)
total_list.append(total)
valid_result_distro = pd.DataFrame(zip(r_point_list, total_list, pos_list,
    neg_list), columns=["Restart point", "Total", "Positive", "Negative"])
valid_result_distro["Pos %"] = valid_result_distro.Positive / valid_result_distro.
    Total * 100
valid_result_distro["Neg %"] = valid_result_distro.Negative / valid_result_distro.
    Total * 100
print(f"{valid_result_distro.to_latex(float_format="%.2f" , caption="Distribution
    of Target variable")}")
sns.barplot(valid_result_distro, x="Restart point", y="Pos %")
plt.xticks(rotation=90)
# Weight calculation
neg, pos = np.bincount(ml_data['Valid_result'])
total = neg + pos
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
# Adding class weights
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.
weight_for_0 = (1 / neg) * (total / 2.0)
weight_for_1 = (1 / pos) * (total / 2.0)
class_weight = {0: weight_for_0, 1: weight_for_1}
print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
# Splitting into targets and features
ml_data.drop(columns=['PQR'], inplace=True)
X = ml_data.drop(columns=['Avg Run', 'Valid_result']).values
y_runtime = ml_data.loc[:,'Avg Run'].values
y_valid_res = ml_data.loc[:, 'Valid_result'].values
print(f"Shape of X {X.shape}")
print('X:\n', X)
print('y_runtime:\n', y_runtime)
print('y_valid\n', y_valid_res)
# Classification
X_train, X_test, y_train, y_test = train_test_split(X, y_valid_res, test_size=0.4,
     shuffle=True, random_state=42, stratify=y_valid_res)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
def classification_analysis(y_true, y_pred, title):
    """
        Calculate Metrics for classification
        and plot them.
    """
    print(classification_report(y_true, y_pred))
    cm = confusion_matrix(y_true, y_pred)
    print(cm)
    fig, ax = plt.subplots(1, 2, figsize=(10,5))
    sns.heatmap(cm, annot=True, cmap="Blues", fmt="d",
            xticklabels=["Pred. Invalid", "Pred. Valid"],
            yticklabels=["Act. Invalid", "Act. Valid"], ax=ax[0])
    ax[0].title.set_text("Confusion Matrix")
    fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    roc_auc = roc_auc_score(y_test, y_pred)
    plot = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc)
    plot.plot(ax=ax[1])
    ax[1].title.set_text("ROC-Curve")
    plt.suptitle(title)
    plt.tight_layout()
    plt.savefig(f"{title}.png")
    plt.show()
# Gradient Boosting
# print(X_test)
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.01, max_depth=3,
     n_iter_no_change=5, validation_fraction=0.2, verbose=0)
gb.fit(X_train, y_train)
y_pred = gb.predict(X_test)
classification_analysis(y_test, y_pred, "Gradient Boosting Classifier")
# Random Forest Classifier
rf = RandomForestClassifier(n_estimators=1000, n_jobs=-1)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
classification_analysis(y_test, y_pred, "Random Forest Classifier")
# Extra Trees Classifier
et = ExtraTreesClassifier(n_estimators= 1000, criterion='log_loss', n_jobs=-1,
    class_weight=class_weight)
et.fit(X_train, y_train)
y_pred = et.predict(X_test)
classification_analysis(y_test, y_pred, "Extra Trees Classifier")
# Bagging Classifier
bg = BaggingClassifier(n_estimators=1000, n_jobs=-1)
bg.fit(X_train, y_train)
y_pred = bg.predict(X_test)
classification_analysis(y_test, y_pred, "Bagging Classifier")
# Ada Boost Classifier
ad = AdaBoostClassifier(n_estimators=1000, learning_rate=0.1)
ad.fit(X_train, y_train)
y_pred = ad.predict(X_test)
classification_analysis(y_test, y_pred, "Ada Boost Classifier")
# Regression
regressor = []
mse = []
rmse = []
r2 = []
def regression_metrics(y_true, y_pred, title):
    """
        Calculate Metrics for regression and
        store them to display them in one table.
    """
    print(f"Report for {title}")
    regressor.append(title)
    print(f"Mean squared error: {mean_squared_error(y_true, y_pred)}")
    mse.append(mean_squared_error(y_true, y_pred))
    print(f"root mean squared error: {root_mean_squared_error(y_true, y_pred)}")
    rmse.append(root_mean_squared_error(y_true, y_pred))
    print(f"R^2 score: {r2_score(y_true, y_pred)}")
    r2.append(r2_score(y_true, y_pred))
X_train, X_test, y_train, y_test = train_test_split(X, y_runtime, test_size=0.4,
    shuffle=True, random_state=42)
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train = scaler_X.fit_transform(X_train)
```

```
X_test = scaler_X.transform(X_test)
y_train = y_train.reshape(-1,1)
y_test = y_test.reshape(-1,1)
y_train = scaler_y.fit_transform(y_train)
y_test = scaler_y.transform(y_test)
# Gradient Boosting Regressor
gbr = GradientBoostingRegressor()
gbr.fit(X_train, y_train)
y_pred = gbr.predict(X_test)
regression_metrics(y_test, y_pred, "Gradient Boosting Classifier")
# Random Forest Regressor
rfr = RandomForestRegressor(n_jobs=-1)
rfr.fit(X_train, y_train)
y_pred = rfr.predict(X_test)
regression_metrics(y_test, y_pred, "Random Forest Regressor")
# Extra Trees Regressor
etr = ExtraTreesRegressor(n_jobs=-1)
etr.fit(X_train, y_train)
y_pred = etr.predict(X_test)
regression_metrics(y_test, y_pred, "Extra Trees Regressor")
# Displaying all the metrics in table
regressor_metrics = pd.DataFrame(zip(regressor, mse, rmse, r2),
    columns=["Regressor", "MSE", "RMSE", "R2-score"])
# Boxplot for runtime vs. restart point
sns.boxenplot(ml_data, x="R points", y="Avg Run")
plt.xticks(rotation=90)
plt.title("Boxplot")
plt.yscale("log")
plt.tight_layout()
plt.savefig("graphics/Boxplot_Avg_Runtime.png")
```