A Machine Learning Approach for HyperParameter Tuning of Unconstrained Optimization Solvers

Final Project Report

Yuchen (Felix) Gao
Department of Mathematical and Computational Sciences
University of Toronto

Contents

Abstract

Optimization methods such as L-BFGS rely on hyperparameters that strongly influence convergence. In practice, these values are often chosen by heuristics or trial-and-error, which may lead to suboptimal performance on different problem types. We investigate a machine-learning approach to systematically select L-BFGS hyperparameters based on problem characteristics. We extract analytical features (e.g., problem size, memory usage) and empirical features (e.g., initial objective evaluation, initial gradient evaluation), and train a machine-learning model that predicts the optimal memory parameter `mem`. As `mem` increases, the memory used by the algorithm increases as well. This introduces a fundamental trade-off where increasing `mem` yields a more accurate curvature approximation that potentially reduces the total iteration count, but it proportionally increases memory consumption and per-iteration computational overhead. Experiments on both regular and scalable problems show that our approach improves runtime and reduces memory compared to standard default settings. This work highlights the potential of learning-based parameter selection to enhance classical optimization algorithms.

1 Background and Introduction

1.1 Unconstrained Optimization Algorithms

In the field of continuous numerical optimization, the aim is to develop solvers that can locate a local minimum of unconstrained optimization problems:

$$\min_{x \in \mathbb{R}^n} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and using first-order (gradient vector of size n) and eventually using second-order derivatives (Hessian matrix - symmetric matrix of size n) of the objective function f. There exists a variety of algorithms for such problem and you can find a list of implementations in the Julia package JSOSolver.jl. These algorithms are iterative algorithms that start from an initial guess $x_0 \in \mathbb{R}^n$ and compute a follow-up iterate until a stationary point is reached, i.e, $\nabla f(x) \approx 0$.

1.2 Line Search Methods

Line search methods form a broad class of iterative optimization algorithms in which, at each iteration, a search direction $\mathbf{p}_k$ is computed and a suitable step length $\alpha_k > 0$ is chosen to determine how far to move along that direction. The general update rule is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k,$$

where α_k is referred to as the *step length* or *learning rate*.
The performance of a line search method depends critically on two components: the choice of the search direction $\mathbf{p}_k$ and the strategy used to determine α_k. Typical search directions include the negative gradient direction used in the Steepest Descent method and the Newton direction in second-order methods.
An ideal step length achieves a sufficient reduction in the objective function while maintaining numerical stability. In practice, α_k is often determined through a line search procedure that satisfies certain conditions, such as the *Wolfe* or *Armijo* conditions, which ensure both sufficient decrease

and appropriate curvature. These conditions balance convergence speed with robustness, preventing steps that are excessively short or that overshoot the minimum.

1.3 Newton's Method

Newton's Method is a second-order iterative optimization algorithm that uses both the gradient and the Hessian of the objective function to determine the search direction. It can achieve significantly faster convergence compared to first-order methods such as Steepest Descent, particularly near the optimal point.
We consider a twice continuously differentiable scalar-valued function

$$f : \mathbb{R}^n \to \mathbb{R}.$$

At iteration k, Newton's Method computes the search direction $\mathbf{p}_k^N$ by solving the linear system

$$\nabla^2 f(\mathbf{x}_k)\, \mathbf{p}_k^N = -\nabla f(\mathbf{x}_k),$$

where $\nabla f(\mathbf{x}_k)$ is the gradient and $\nabla^2 f(\mathbf{x}_k)$ is the Hessian matrix of second derivatives. The parameter vector is then updated as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k^N,$$

where $\alpha_k > 0$ is the step size, often obtained by a line search procedure to ensure sufficient decrease in f.
In an ideal case with exact line search, $\alpha_k = 1$ is typically accepted for all sufficiently large k, and the method achieves *quadratic convergence.* Formally, if the Hessian $\nabla^2 f(\mathbf{x})$ is Lipschitz continuous in a neighborhood of the solution $\mathbf{x}^*$, and if $\nabla^2 f(\mathbf{x}^*)$ is positive definite, then:

1. For a starting point $\mathbf{x}_0$ sufficiently close to $\mathbf{x}^*$, the sequence $\{\mathbf{x}_k\}$ converges to $\mathbf{x}^*$.

2. The convergence rate is quadratic; that is,

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq C\|\mathbf{x}_k - \mathbf{x}^*\|^2,$$

   for some constant $C > 0$.

3. The gradient norms $\|\nabla f(\mathbf{x}_k)\|$ also converge quadratically to zero.

Despite its theoretical efficiency, Newton's Method can be computationally expensive for high-dimensional problems, since it requires forming and inverting the Hessian matrix at every iteration. Moreover, if the Hessian is not positive definite, the computed direction $\mathbf{p}_k^N$ may fail to be a descent direction, potentially leading to divergence. In practice, modified Newton methods or quasi-Newton algorithms (such as BFGS and L-BFGS) are employed to approximate curvature information efficiently while maintaining desirable convergence behavior.

1.4 Quasi-Newton Methods

Optimization methods such as steepest descent and Newton's Method are widely used in scientific computing to find local minima of differentiable functions. However, for large-scale problems, these methods can become computationally expensive and memory-intensive. Quasi-Newton methods, similar to the steepest descent approach, require only the gradient of the objective function at each iteration. By observing how the gradient changes between steps, these methods build an approximate model of the objective function that is accurate enough to achieve superlinear convergence. Compared to steepest descent, their improvement in speed and stability is substantial, particularly on challenging problems. Because they avoid computing second derivatives, quasi-Newton methods can even be more efficient than full Newton's Method. Modern optimization libraries now include many variants of quasi-Newton algorithms designed for unconstrained, constrained, and large-scale optimization problems.

1.5 Implementations of Quasi-Newton Methods

Algorithm 1 (BFGS Method).

```
1: Given starting point $x_0$, convergence tolerance $\epsilon > 0$,
       inverse Hessian approximation $H_0$;
2: $k \leftarrow 0$;
3: while $\|\nabla f_k\| > \epsilon$ do
4:     Compute search direction
       $p_k = -H_k \nabla f_k$;
5:     Set $x_{k+1} = x_k + \alpha_k p_k$ where $\alpha_k$ is computed from a line search
        procedure to satisfy the Wolfe conditions;
6:     Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;
7:     Compute $H_{k+1}$;
8:     $k \leftarrow k + 1$;
9: end while
```

Algorithm 2 (L-BFGS helper function using two-loop recursion).

```
1: $q \leftarrow \nabla f_k$;
2: for $i = k-1, k-2, \ldots, k-m$ do
3:     $\alpha_i \leftarrow \rho_i s_i^T q$;
4:     $q \leftarrow q - \alpha_i y_i$;
5: end for
6: $r \leftarrow H_k^0 q$;
7: for $i = k-m, k-m+1, \ldots, k-1$ do
8:     $\beta \leftarrow \rho_i y_i^T r$;
9:     $r \leftarrow r + s_i(\alpha_i - \beta)$;
10: end for
11: stop with result $H_k \nabla f_k = r$.
```

Algorithm 3 (L-BFGS).

```
1:  Choose starting point $x_0$, integer $m > 0$;
2:  $k \leftarrow 0$;
3:  **repeat**
4:      Choose $H_k^0$;
5:      Compute $p_k \leftarrow -H_k \nabla f_k$ from Algorithm 2;
6:      Compute $x_{k+1} \leftarrow x_k + \alpha_k p_k$, where $\alpha_k$ is chosen to
         satisfy the Wolfe conditions;
7:      **if** $k > m$ **then**
8:          Discard the vector pair $\{s_{k-m}, y_{k-m}\}$ from storage;
9:      **end if**
10:     Compute and save $s_k \leftarrow x_{k+1} - x_k$, $y_k = \nabla f_{k+1} - \nabla f_k$;
11:     $k \leftarrow k + 1$;
12: **until** convergence.
```

1.6 Expression Tree Features

An expression tree is a standard data structure in computer science for representing algebraic expressions in a form that makes their structure explicit. Each internal node encodes an operator (for example addition, multiplication, or a nonlinear function), and each leaf encodes an operand such as a variable or constant. This representation induces a natural notion of complexity: the tree length corresponds to the total number of nodes in the expression, while the tree depth corresponds to the maximum number of nested operations along any root to leaf path. Because these quantities reflect how many operations are composed and how deeply they are nested, they provide compact, problem specific descriptors of the objective that complement purely numeric timing features.

1.7 Oracle Optimal Memory Parameter

We define the oracle-optimal memory parameter as the value that achieves the minimum observed wall-clock runtime for a given problem instance when evaluated exhaustively over the memory parameter range.

2 Methodology and Implementation

2.1 Data Dictionary

`Column`	**Type**	**Description**
`status`	Symbol	Termination status of the solver run (for example **first_order**, **max_time**, **unbounded**).
`problem`	String	Identifier of the optimization problem instance (problem name as stored in metadata).
`solver`	String	Name of the solver used for the run (here L-BFGS).
`mem`	Int	L-BFGS memory parameter controlling the number of stored curvature pairs.
`nvar`	Int	Number of decision variables n.
`time`	Float64	Wall-clock solve time in seconds.
`memory`	Float64	Memory usage associated with the solver run (MB).
`num_iter`	Int	Total number of iterations performed by the solver.
`nvmops`	Int	Number of vector–matrix style operations reported by the solver.
`neval_obj`	Int	Number of objective function evaluations.
`init_eval_obj_time`	Float64	Wall-clock time (seconds) for the initial objective evaluation at the starting point.
`init_eval_obj_mem`	Float64	Memory usage (MB) for the initial objective evaluation.
`init_eval_obj_alloc`	Float64	Number of heap allocations for the initial objective evaluation.
`neval_grad`	Int	Number of gradient evaluations.
`init_eval_grad_time`	Float64	Wall-clock time (seconds) for the initial gradient evaluation at the starting point.
`init_eval_grad_mem`	Float64	Memory usage (MB) for the initial gradient evaluation.
`init_eval_grad_alloc`	Float64	Number of heap allocations for the initial gradient evaluation.
`is_init_run`	Bool	Whether the row corresponds to the first solver call for a newly constructed instance (initialization run).
`is_scalable`	Bool	Indicator that the problem family supports programmatic variation of dimension.
`objtype`	String	Objective type label provided by the problem metadata (for example least squares versus other).
`variable_nvar`	Bool	Metadata flag indicating whether nvar can be specified at construction time.
`expr_leaves_count`	Int	Number of leaves in the objective expression tree (nodes with no children), computed by a recursive traversal.
`expr_max_depth`	Int	Maximum root-to-leaf path length in the objective expression tree, computed by a recursive traversal.

Table 1: Description of columns in the solver benchmark dataset.

2.2 Benchmark Suite Construction

Source and scope. Benchmark problems were obtained from *OptimizationProblems.jl*, a maintained library of smooth test problems with consistent metadata and evaluation routines. Since this project targets large scale unconstrained smooth optimization with L-BFGS, we restricted attention to problems that are unconstrained, free of bound constraints, and satisfy nvar >= 5. These criteria exclude trivial instances and ensure that curvature approximation and limited memory effects are practically relevant.

Regular and scalable problems. The benchmark contains two types of problem families. Regular problems have a fixed dimension determined by the problem definition. Scalable problems allow the dimension to be specified at construction time, as indicated by the metadata flag variable_nvar = true. For scalable families, we generate multiple instances by varying the dimension n in order to study how performance and the best choice of `mem` change with problem size.

Selection criteria. The first stage selection criteria are summarized in Listing 1. In addition, we require ncon = 0 as a consistency check to verify the absence of equality or inequality constraints.

Listing 1: First stage filtering criteria.

```
contype == :unconstrained
!has_bounds
nvar >= 5
```

Benchmark summary. After filtering, the benchmark contains 105 distinct problem families, including 84 scalable families. The experiments comprise 19,015 solver runs in total, partitioned into 10,605 runs on regular (fixed dimension) instances and 8,410 runs on scalable instances. For the regular subset, the dimension satisfies $n \in [5, 100]$, with median 100 and mean 77.94. For the scalable subset, the dimension satisfies $n \in [961, 1000]$, with median 1000 and mean 998.16. In both subsets, for all problems, we exhaustively evaluate the L-BFGS memory parameter over the integer range $\texttt{mem} \in \{1, 2, \dots, 100\}$.

2.3 Solver Configuration

All experiments use the L-BFGS solver from *JSOSolvers.jl*. Our objective is to isolate the effect of the limited-memory parameter `mem`. Accordingly, we vary only `mem` and keep all other solver settings at their default values. For each problem instance, we obtain the admissible integer domain of `mem` from `LBFGSParameterSet` and evaluate all values in that domain.

Each run starts from the same initial point provided by the test problem. To prevent pathological cases from dominating total compute time, we impose a wall-clock time limit of 60 seconds per run for regular instances and 300 seconds per run for scalable instances.

In addition to solver outcomes (for example runtime, iteration count, and termination status), we record the cost of evaluating the objective and gradient at the starting point. These measurements are used as empirical features and to characterize initialization overhead.

2.4 Data Collection and Target Definition

For each solver run, we record the termination status, runtime, iteration count, and solver statistics including the number of objective and gradient evaluations and memory or allocation measurements when available. Runs that exceed the wall-clock limit or terminate abnormally are treated as failures and are excluded when defining the optimal parameter.

For each problem instance, we define the optimal memory parameter `mem`* as the value of `mem` that minimizes runtime among successful runs within the tested domain. When multiple values achieve the same runtime within numerical tolerance, we break ties by selecting the smaller `mem` to favor lower memory usage.

Julia JIT compiler compiles code in memory only at the instance it encounters it for the first time unlike some other languages (C, fortran, C++) which compiles everything all at once. This feature results in an initial run taking substantially longer due to this initial load into memory. Therefore, initialization runs flagged by `is_init_run = true` are excluded from the primary analysis to avoid julia first-run compiling time, but are retained for secondary comparisons.

2.5 Feature Extraction

Gradients are computed via forward automatic differentiation through *ADNLPModels.jl*. We construct the learning feature set from three sources. First, we use analytical features from problem metadata, including `nvar`, `is_scalable`, `variable_nvar` and the objective type label. Second, we include empirical features measured at the starting point, namely the objective evaluation cost and gradient evaluation cost (recorded in time, memory, and allocation statistics). Third, we augment these features with symbolic structure descriptors by converting the objective into an expression tree using *ExpressionTreeForge.jl* and computing two recursion based quantities: the number of leaves in the tree (`expr_leaf_count`) and the maximum root to leaf path length (`expr_max_depth`).

2.6 Model Implementation Details

Data analysis is conducted in Python using the `pandas` library for data manipulation and data analysis and the `scikit-learn` library is used for model training. Problem instances are uniquely identified by (`problem`, `nvar`) and are split at the instance level into training (70%), validation (15%), and test (15%) sets by shuffling the unique instance list with an arbitrarily chosen random seed and merging back into the full table, ensuring no leakage across splits.

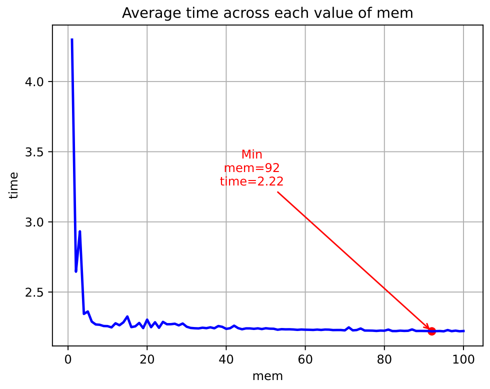
We trained three regression models - Decision Tree, Random Forest, and Gradient Boosting—to predict the log-transformed wall-clock runtime, log(time). The logarithmic transformation was applied to mitigate the impact of heavy-tailed runtime distributions and to stabilize the variance during training.

Model Hyperparameters were optimized via a grid search on the training data, targeting the minimization of the **Mean Squared Error (MSE)**. This loss function was selected to penalize large deviations in the logarithmic space, ensuring consistent predictive performance across varying problem scales. The final configurations were determined by evaluating performance on the validation set to mitigate overfitting prior to final evaluation on the test set.

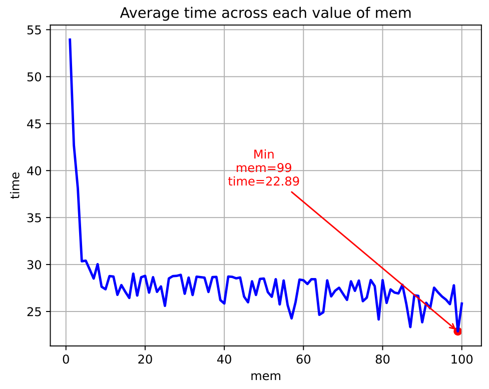
3 Results

3.1 Aggregate Best L-BFGS Memory Selection

Figure 1 shows the relationship between mem and time for both regular and scalable problems. The red arrows point at the best L-BFGS memory parameter.



(a) `Best aggregate mem for nvar` ≈ 100



(b) `Best aggregate mem for nvar` $\approx$ `1000`

Figure 1: Best Aggregate L-BFGS Memory Parameter

Based on the aggregate runtime profiles, a simple baseline heuristic is to select a fixed memory parameter of `mem` $= 92$ for regular problems and `mem` $= 99$ for scalable problems. Under this aggregate criterion, these choices minimize the average wall clock runtime across the evaluated benchmark instances. This heuristic therefore provides a strong global baseline for memory selection.

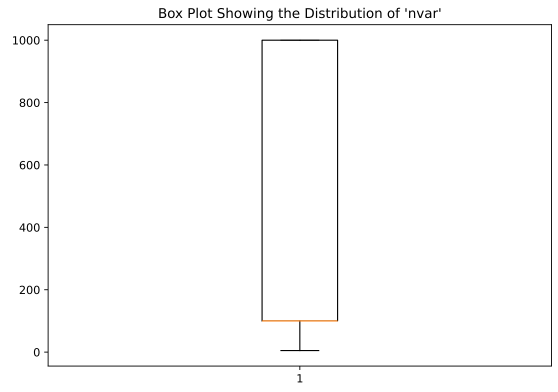
3.2 Model Feature Description

Table 2 summarizes the descriptive statistics for the features columns utilized in the regression models, providing an overview of the variable scale and distribution.

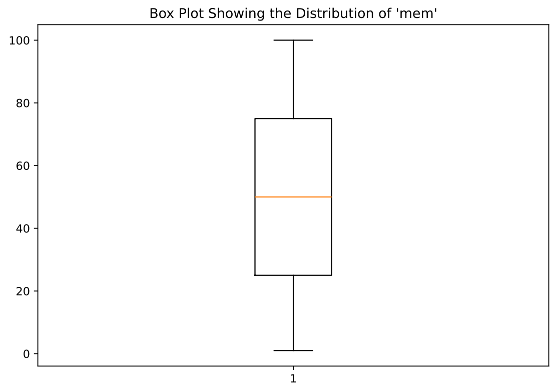
Feature Column	mean	std	min	25%	50%	75%	max
nvar	484.92	458.00	5.00	100.00	100.00	1000.00	1000.00
mem	50.45	28.88	1.00	25.00	50.00	75.00	100.00
expr_leaves_count	2858.91	7602.00	109.00	396.00	690.00	974.00	40200.00
expr_max_depth	7.35	1.88	4.00	6.00	7.00	8.00	13.00
init_eval_obj_time	0.27	3.29	0.00	0.00	0.01	0.02	47.23
init_eval_grad_time	33.05	406.78	0.00	0.05	0.18	4.49	5683.89

Table 2: Feature Columns used to train the regression models

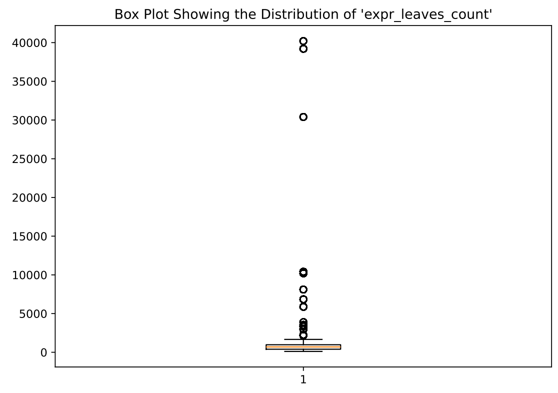
Figure 2 illustrates the results in Table 2 using boxplots.



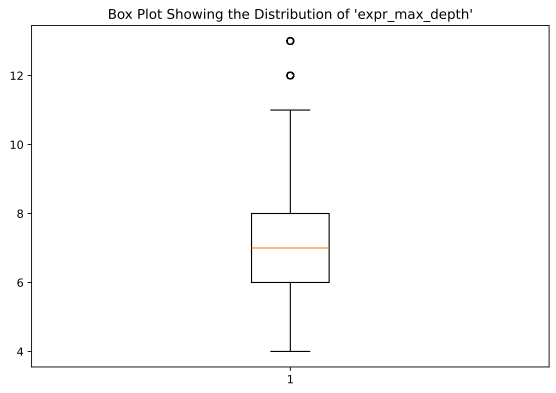
(a) `nvar`



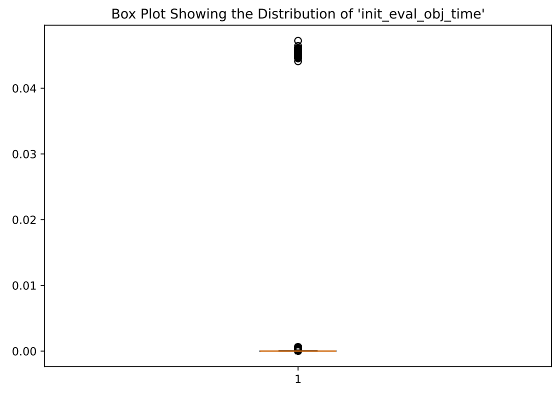
(b) `mem`



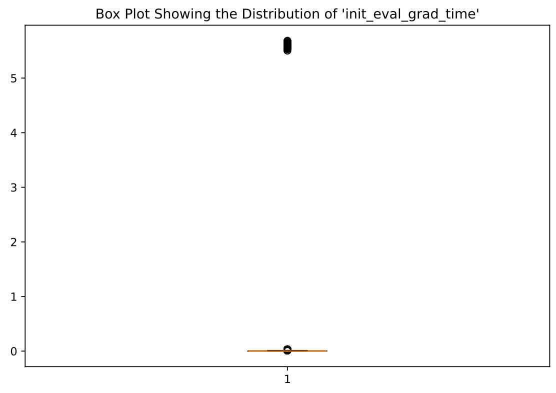
(c) `expr_leaves_count`



(d) `expr_max_depth`



(e) `init_eval_obj_time`



(f) `init_eval_grad_time`

Figure 2: Feature distributions for the regression models

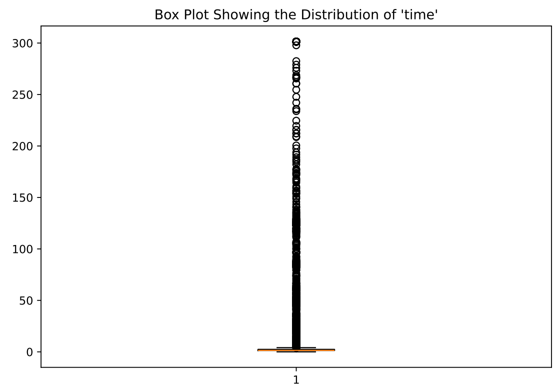
3.3 Model Target Description

Table 3 summarizes the descriptive statistics for the target columns utilized in the regression models, providing an overview of the variable scale and distribution.

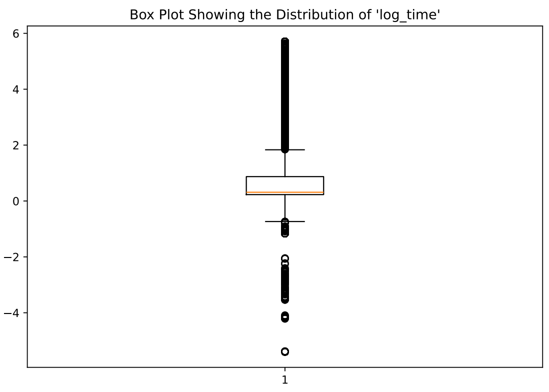
Target Column	mean	std	min	25%	50%	75%	max
time	13.590847	48.041225	0.004524	1.256812	1.366869	2.393711	301.532641
log_time	0.671700	1.651204	-5.398353	0.228578	0.312523	0.872845	5.708878

Table 3: Target Columns for the regression models to predict

Figure 3 illustrates the results in Table 3 using boxplots.



(a) `time`



(b) `log_time`

Figure 3: Target Distribution for the regression models

3.4 Model Performance Evaluation Metrics

The Mean Squared Error (MSE) measures the average of the squares of the errors, penalizing larger deviations:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}{e_i}^2 \tag{1}$$

The Mean Absolute Error (MAE) provides the average magnitude of the errors in a set of predictions:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| = \frac{1}{n}\sum_{i=1}^{n}|e_i| \tag{2}$$

The Coefficient of Determination (R^2) represents the proportion of variance for the dependent variable that's explained by the independent variables:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{SSE}{SST} = \frac{SSR}{SST} \tag{3}$$

where:

n The total number of observations in the dataset ($N = 18,826$).

y_i The actual observed value for the i-th instance.

$\hat{y}_i$ The predicted value generated by the model.

$\bar{y}$ The mean of all actual observed values.

e_i The residual (error) for the i-th instance, defined as $(y_i - \hat{y}_i)$.

SSE The Error Sum of Squares, representing the unexplained residuals.

SST The Total Sum of Squares, representing the total variance in the $18,826$ problem instances.

SSR The Regression Sum of Squares, representing the variance explained by the model.

3.5 Model Performance on Test Set

Table 4 summarizes the out-of-sample performance on the test set, evaluating each model's predictive accuracy and generalization to unseen instances.

Model	Mean Squared Error	Mean Absolute Error	R^2	Status
Decision Tree	7018.06	29.86	-25.80	Unstable
Random Forest	89.91	1.84	0.97	Robust
Gradient Boosting	39.82	1.48	0.99	Optimal

Table 4: Test statistics for all three regression models

Ideally, the objective is to minimize both MSE and MAE to reduce prediction error. The R^2 score measures the proportion of variance explained by the model; a negative value indicates that the model performs worse than a simple horizontal line representing the mean runtime. Such a result suggests the model is unable to account for the high variance and outliers inherent in the dataset.

3.6 L-BFGS Memory Selection Evaluation Metrics

Let x denote a problem instance identified by ($\texttt{problem}, \texttt{nvar}$), and let $m \in \{1, \ldots, 100\}$ denote the L-BFGS memory parameter. For each pair (x, m), $T(x, m)$ denotes the observed wall-clock runtime recorded in our dataset, while $\widehat{T}(x, m)$ denotes the runtime predicted by a trained regressor.
Using the predicted runtimes, we select

$$\hat{m}(x) = \arg \min_{m \in \{1,\ldots,100\}} \widehat{T}(x, m), \qquad m^\star(x) = \arg \min_{m \in \{1,\ldots,100\}} T(x, m),$$

where $\hat{m}(x)$ is the predicted memory parameter and $m^\star(x)$ is the oracle-best memory parameter under the observed runtime for problem x. We evaluate the selector via the achieved-time ratio

$$r = \rho(x) = \frac{T(x, \hat{m}(x))}{T(x, m^\star(x))}.$$

where $r = 1$ represents optimal selection.

3.7 L-BFGS Memory Selection Performance

Table 5 reports L-BFGS memory selection performance using summary statistics of the achieved-time ratio r, including its median value and the fraction of instances satisfying $r \leq 1.05$ and $r \leq 1.10$, together with exact match accuracy.

Metric	Decision Tree	Random Forest	Gradient Boosting
Exact match accuracy	0.00	0.00	0.03
Median ratio r	1.22	1.00	1.00
Fraction with $r \leq 1.05$	0.44	0.76	0.62
Fraction with $r \leq 1.10$	0.44	0.79	0.72

Table 5: L-BFGS Memory selection performance on the test set.

3.8 Comparative Performance Evaluation Metric Against Baseline

To quantify practical efficacy, we define an improvement metric based on the **Relative Gap** (g) to oracle-best performance. This metric represents the scale-invariant computational overhead incurred by a specific memory choice relative to the optimal runtime:

$$g = \frac{T_{\text{selection}} - T_{\text{best}}}{T_{\text{best}}} \tag{4}$$

By comparing the mean relative gap of the model's selections (μ_m) against the `mem=5` baseline (μ_b), we calculate a **Reliability Improvement** percentage:

$$\text{Improvement} = \left(1 - \frac{\mu_m}{\mu_b}\right) \times 100\% \tag{5}$$

This metric is uniquely suitable as it directly measures the proportion of latent performance reclaimed by the predictive selector. By normalizing errors across varying problem scales, it provides a robust assessment of how effectively the models reduce the computational overhead inherent in the default heuristic.

3.9 Comparative Performance Against Baseline

The results in Table 6 indicate that both the Random Forest and Gradient Boosting models outperform the `mem=5` baseline. Conversely, the Decision Tree shows a significant performance degradation, suggesting it is unable to generalize effectively for this selection task.

Model	Improvement (%)	Status
Decision Tree	−282.09	Unstable
Random Forest	25.36	**Best Performance**
Gradient Boosting	5.04	Optimal

Table 6: Overall Performance Improvement Relative to Baseline (`mem=5`).

4 Conclusions

This research addressed the challenge of L-BFGS memory parameter selection through a data-driven runtime prediction framework. While Gradient Boosting achieved the highest predictive accuracy as measured by R^2, the **Random Forest** model demonstrated the greatest practical utility. By achieving a **25.36% Reliability Improvement** over the standard $mem = 5$ baseline, the Random Forest proved most effective at reclaiming latent performance. These findings suggest that ensemble-based models are highly suitable for automating parameter selection in numerical optimization, offering significant efficiency gains over static heuristics.

5 Future Work

Building upon the successful validation of the Random Forest selector, several avenues for further research are proposed:

Deployment and Interface: The learned memory selector could be deployed as a lightweight web-based service. Given a problem identifier and dimension, the interface would return a recommended memory parameter together with an estimated runtime profile. As the system evolves, the interface could be extended to incorporate additional problem features or multiple feature sets, enabling more expressive and flexible memory recommendations across a broader range of problem instances.

Extension to Other JSOSolvers Algorithms: While this study focuses on memory selection for L-BFGS, the proposed framework could be extended to other optimization algorithms implemented in *JSOSolvers.jl* that expose tunable internal parameters. In particular, truncated Newton and trust-region based solvers involve algorithm-specific trade-offs between memory usage, curvature approximation, and computational cost. Evaluating whether learned parameter selection strategies transfer across these solver families is a natural direction for future work.

Autonomous Benchmarking Agent: A longer-term direction is to develop an autonomous benchmarking agent that continuously runs solver evaluations and expands the benchmark corpus. The agent would schedule new experiments, validate results, and log standardized performance data and metadata, enabling the dataset to grow over time as new problems and solver variants are introduced. This would create a reusable benchmarking resource for future researchers and support ongoing improvement of data-driven parameter selection methods.

Automatic Differentiation–Driven Features: Future work could further exploit automatic differentiation to extract richer problem characteristics beyond basic structural descriptors. For example, curvature-related information derived from Hessian–vector products or gradient variability could be used to construct more informative features for memory selection. Such features would directly leverage solver internals and optimization theory, potentially improving both predictive accuracy and interpretability.

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[2] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[4] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

[5] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] JuliaSmoothOptimizers. JSOSolvers.jl: Julia smooth optimization solvers. https://github.com/JuliaSmoothOptimizers/JSOSolvers.jl, 2023.

[7] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

[8] Wes McKinney. Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[9] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Science+Business Media, LLC, New York, NY, USA, second edition, 2006.

[10] Dominique Orban et al. ADNLPModels.jl: Automatic differentiation for nonlinear programming models. https://github.com/JuliaSmoothOptimizers/ADNLPModels.jl, 2023.

[11] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] Christopher Rackauckas et al. OptimizationProblems.jl: A collection of optimization test problems. https://github.com/SciML/OptimizationProblems.jl, 2023.