On the Design of Calculation Engines, Key Performance Indicators, and Metrics for Data Engineering-Based Projects

Final Report

Project ID: 2024-S2-68
Group ID: 11522-2S2-25

Authors
Vibhore Singh
Ishtiaq Chowdhury
Hasan Shariar Chowdhury
Rahul Kisan Dabilkar
Md Mayen Uddin Mozumder Tushar

Project Sponsor
Julio Romero

Project Mentor
Yasaman Baradaran

Contents

1 Introduction

In today's rapidly evolving technological landscape, data-driven decision-making is the core of the innovation and development. Machine learning (ML) plays an important role in extracting insights from vast datasets, enabling businesses and researchers to make informed decisions. However, the performance and efficiency of ML models can vary significantly depending on the programming language and the platform used for implementation. This presents a challenge for developers, data scientists, and engineers who need to select the best tools for their specific needs.

Since there are numerous programming languages, each having its unique strengths and limitations, complicates the decision-making process. For example, Python is renowned for its simplicity and extensive libraries, while C++ offers high performance but at the cost of complexity. Julia, a relatively newer language, promises high performance with ease of use, but its ecosystem is still developing. Similar pros and cons make it necessary to have a systematic approach to compare these languages across different metrics ensuring the most efficient implementation of the ML models.

1.1 Problem Statement

This project aims to address the problem of selecting the most efficient programming language for implementing machine learning algorithms on science and engineering datasets by evaluating multiple languages on a set of key performance indicators (KPIs). Our primary objective is to create a comprehensive catalogue of techniques and codes for several machine learning algorithms in several languages followed by a systematic comparison based on established as well as newly developed KPIs. The outcome will guide developers and researchers in choosing the right tools for their projects, optimizing performance, and enhancing the reliability of their ML models.

1.2 Key findings

Python was the easiest language to use while C++ and Rust were more complex and took extra effort to develop. When it came to available libraries for data analysis, Python and R had the best support, whereas C++ and Rust were not as well-suited for these tasks. Time wise, C++ and Julia stood , while R lagged behind. Python also had the shortest learning curve, making it the easiest to learn, while C++ and Rust were more challenging to learn. Julia struck a great balance between speed and ease of use, making it ideal for scientific computing. Lastly, Julia, Python, and R offered full support for generating visualisations, but C++ and Rust weren't suitable for this task.

2 Literature Review

In recent years, the diversity of programming languages for implementing machine learning (ML) algorithms has grown, each offering distinct advantages and challenges. This makes the evaluation of ML models across these languages a critical area of research. Python known for ML libraries like Scikit-learn[1] and TensorFlow, dominates the landscape due to its ease of use[2] and flexibility[3] . R is valued for its comprehensive ecosystem of package[4]s making it particularly suitable for in-depth statistical modeling.[5] Julia is gaining traction for its ability

to combine Python-like simplicity with near-C-level performance.[6] However, languages like C++[7] and Java,[8] known for performance and scalability.[9]

The comparative performance of classification algorithms, including Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Convolutional Neural Networks (CNN), Random Forests (RF), Decision Trees (DT), and Neural Networks has been focussed on by many researchers[3]and explored the application of RF and SVM[10] in high-dimensional data, emphasizing the importance of algorithm selection based on dataset characteristics.[10] also highlighted several algorithm's performance in handling structured data.

Performance evaluation using Key Performance Indicators (KPIs) like accuracy, precision, recall, and F1 score[11] is well-documented, but recent studies have introduced additional metrics such as AUC-ROC,[12] computational complexity, and memory usage for a more holistic assessment.

Despite these studies, a comprehensive evaluation of ML algorithms across different languages remains limited. This project aims to fill this gap by systematically comparing KPIs thereby guiding practitioners in selecting the most efficient tools for their specific needs.

3 Restatement of the scope and requirements

4 Project Outcomes

The project was structured into two primary stages, each with specific objectives and deliverables. In the first phase, we evaluated classification models across multiple programming languages, comparing their performance on essential KPIs such as accuracy, precision, recall, etc. The second stage involved the evaluation of statistical methods, where various inferential tests, linear models, and ANOVA were performed in the same languages, providing insights into their efficiency for advanced statistical tasks. Both stages were aimed at drawing comparisons across five programming environments—Python, R, Julia, Rust, and C++—to identify their strengths and limitations in real-world data analysis scenarios.

4.1 Deliverables

4.1.1 Catalogue of Techniques

We developed and implemented Logistic Regression (LR) and Support Vector Machine (SVM) algorithms in five different programming languages: Python, R, Julia, Rust, and C++. The deliverable aimed to provide a comprehensive comparison of the two machine learning techniques in several programming languages. Each technique was implemented according to the planned methodology and functionalities which were consistent across languages, ensuring a fair comparison. The implementations were tested using the diabetes.csv dataset, with appropriate modifications to ensure consistent inputs across all environments.
For Classification

- **Python**: Python was chosen for its rich ecosystem of machine learning libraries and ease of implementation. We used the Scikit-learn library for both LR and SVM.
- **R**: R is known for its statistical capabilities and fast processing and thus was selected to implement both LR and SVM, using the `glm`, `caret`, and `e1071` packages.

- **Julia**: Julia was selected for its high-performance capabilities in statistical computing, making it a strong contender for machine learning tasks. `GLM` and `LIBSVM` packages were used.
- **Rust**: Rust, known for its high-performance computation, provided excellent execution speeds. The `SmartCore` library was used for both LR and SVM.
- **C++**: C++, also known for high performance, allowed for manual implementations with optimizations for performance in handling both LR and SVM.

For Statistical Analysis

- **Python**: We used libraries like `statsmodels` and `scipy` for performing linear models and tests such as ANOVA and t-tests, whereas `matplotlib` and `seaborn` were used for visualization.
- **R**: With R's strong statistical background, we leveraged the `lm()`, `aov()`, and `MANOVA()` functions to fit linear models, perform ANOVA, and run multiple statistical tests. `ggplot2` was used for visualization.
- **Julia**: Julia's `GLM`, `SimpleAnova`, and `HypothesisTests` packages were used for statistical analysis, including linear regression and various inferential tests. However, manual calculations were also required to compute some statistical metric values. The `StatsPlots/Plots` library was used for graphs.
- **Rust**: The `smartcore` and `Nalgebra` libraries in Rust were used to handle statistical modeling and perform inference tests, though additional manual implementation was necessary for certain methods.
- **C++**: Manual implementation of statistical tests allowed us to find faster results with in-depth control over model fitting.

4.1.2 Codebase with Documentation

A complete, well-structured codebase has been submitted, featuring the implementation of classification and differentiation in all five languages. Each language's code is well documented, detailing the required dependencies, clear execution steps to run the models, and in-line comments explaining the logic behind the implementation. The codebase is designed to be modular and reusable, allowing it to be extended for other machine learning/ statistical models or adapted to include additional metrics.

4.1.3 KPI and Metrics Comparison

In the first part of our project, we evaluated the performance of the models using several key performance indicators (KPIs) and metrics while for the second part, we conducted various statistical tests to analyse the data and compare results across languages. **For Classification**

- **Accuracy**: The proportion of correctly predicted instances to the total instances, illustrating the overall correctness of the model.

- **Precision**: The proportion of true positive predictions out of all positive predictions, assessing how well the model predicts positive cases.
- **Recall**: Also known as sensitivity or the true positive rate, recall measures the proportion of actual positive instances that were correctly predicted.
- **F1 Score**: The harmonic mean of recall and precision, used to balance these metrics, particularly in cases of imbalanced datasets.
- **Specificity**: The proportion of true negatives out of all actual negative instances, evaluating the model's ability to avoid false positives.
- **Error Rate**: The complement of accuracy, representing the proportion of incorrect predictions.
- **Adjusted Rand Index (ARI)**: Measures the similarity between two clusterings, adjusted for chance, and used in data clustering tasks.
- **Jaccard Coefficient**: A measure of the similarity between sample sets, comparing genuine and predicted labels.
- **Entropy**: A metric that quantifies uncertainty or impurity in predictions, with higher values indicating greater unpredictability.
- **95% Confidence Interval (CI)**: Provides a range within which we can be 95% confident that the true precision lies.
- **Variance (Cross-Validation)**: The degree of variation in accuracy across different cross-validation folds, indicating model stability.
- **Purity**: Refers to the homogeneity within clusters, indicating how much of a single type a cluster contains.
- **Training Error**: The error rate observed during the model's training phase.
- **Generalization Error**: An estimate of how well the model will perform on new, unseen data, indicating the model's ability to generalize beyond the training set.

For Statistical Analysis

- **Summary Statistics**: Descriptive statistics that summarize data using measures like mean, median, and quartiles.
- **Mean and Standard Deviation**: Calculated to understand the data's spread and average values across groups.
- **Fitting a Linear Model**: Multiple Linear Regression models were fitted to explore relationships between variables, comparing coefficients and fit quality across languages.
- **Two-Sample t-Tests**:
  - **Equal variances**: Compares means of two groups with equal variances.
  - **Unequal variances**: Compares means of two groups with unequal variances.

- **Wilcoxon Rank Sum Test**: A non-parametric test to compare two independent groups when normality assumptions are not met in the data.
- **Wilcoxon Signed-Rank Test**: A non-parametric test for paired samples to assess differences in their population medians.
- **Pearson's Correlation Test**: Measures the strength and direction of the linear relationship between two continuous variables.
- **Chi-Squared Test for Independence**: Tests whether two categorical variables are independent of each other.
- **Binomial Test**: Evaluates whether the proportion of successes in a sample matches a hypothesized probability.
- **Two-Way ANOVA**: Analyze the interaction effects between two independent variables on a dependent variable.
- **Tukey's HSD Test**: Post-hoc analysis following ANOVA to identify significant differences between group means.
- **MANOVA (Multivariate Analysis of Variance)**: Tested for differences in means of multiple dependent variables.
  - **Pillai's Trace**: Analyzes the significance of group differences.
  - **Wilks' Lambda**: Measures the ratio of the within-group variance to the total variance.
  - **Hotelling-Lawley Trace**: Assesses group differences based on the largest eigenvalue.
  - **Roy's Largest Root**: Focuses on the maximum separation between groups.

4.1.4 Research Poster

In addition to the codebase and documentation, a research poster has also been prepared. The poster provides a visual summary of the project's goals, methodology, and key results. It serves as a concise and effective tool to showcase the project outcomes, with visual representations of the metric comparisons.

4.1.5 Final Report

A comprehensive final report has also been prepared, which documents the entire process, from the initial project plan to the implementation and final results. The report includes the goals and objectives of the project, the Methodology, key results, and how they align with the original goals and KPIs. A thorough analysis of the KPIs and metrics, with an emphasis on the advantages and challenges encountered in each language. The report addresses any outstanding issues, discusses the risks that were mitigated during the project, and concludes with lessons learned, including ethical considerations and suggestions for future improvements. The final report provides a complete overview of the project, serving as a standalone document that encapsulates all the work done during this phase.

4.2 Achievements

In this phase of the project, several key accomplishments were achieved. We successfully implemented classification techniques in the five programming languages, demonstrating the project's ability to work across diverse platforms. Each implementation was fully functional, achieving the project's objective of comparing machine learning models across programming environments with evaluation based on several key metrics mentioned above.

A critical success factor was the comparative analysis using KPIs such as accuracy, precision, recall, F1 etc and the execution time as well. These metrics provided concrete insights into how each language performed, both in terms of model accuracy and computational efficiency. Python, and R offered ease of implementation with extensive libraries like Scikit-learn and caret, while languages like C++ showcased higher performance in execution time, though requiring more manual coding / calculation efforts. Julia gave a balanced performance and ease of use showing that it is well-suited for data science tasks requiring both speed and flexibility.[13]

In addition to classification models, the second phase of the project included more advanced statistical analysis, including Multiple Linear Regression (MLR), simple inference, and ANOVA across the same platforms. The implementation of the aforementioned statistical methods across different languages allowed for a detailed comparison of how each environment handles statistical modelling tasks.

Furthermore, the development of a fully functional codebase was another achievement. The modular design of the code, accompanied by thorough documentation, ensures that the code can be easily adapted for other datasets and machine learning models in future projects.

Overall, the project successfully met its goals, providing a robust comparison of different languages, while delivering a highly adaptable codebase and comprehensive documentation. These outcomes strongly align with the original success criteria providing insights into which programming languages best suit different machine learning tasks.

4.3 Key results

The results from the implementation of classification models across the platforms provided several important insights regarding the strengths and limitations of each language in handling machine learning tasks. Overall, the results were very similar across all languages, with only a few minor deviations. These variations reflect the differences in how each language handles computational tasks, particularly in terms of computational efficiency, library support, and memory management.

For Classification

Tables 1 and 2 present the comprehensive performance results for all metrics of Logistic Regression (LR) and Support Vector Machine (SVM) models across the different programming languages. These tables provide a detailed comparison of key metrics highlighting the subtle variations in performance between the languages while reinforcing overall consistency.

- **Accuracy and Error Rate**: Most languages showed similar accuracy, with C++ and Julia slightly outperforming others while R was the exception, having a noticeably lower accuracy and higher error rate.

- **Precision and Recall**: Results were largely consistent across languages and around 75% for most, but R's SVM stood out with remarkably high precision, while C++ showed lower precision.
- **F1 Score and Specificity**: Julia and Rust delivered balanced F1 scores for both models, aligning closely with other languages. Python's SVM excelled in specificity, while others showed similar, slightly lower results.
- **Training Error**: Generalization and training errors were comparable across languages, with Rust performing slightly better in generalization. Julia and Python had the lowest training errors, indicating consistency in model performance.
- **Entropy, Confidence Interval, and Variance**: Python showed higher entropy, but confidence intervals and variance were largely similar across languages, demonstrating consistent predictive stability.
- **Jaccard Coefficient, Adjusted Rand Index, and Purity**: Results for Jaccard, Rand, and Purity were mostly consistent across languages. R and C++ slightly outperformed others in Jaccard and Rand, indicating better similarity between predicted and actual labels. Python and Julia also exhibited high purity, demonstrating consistent grouping performance.

Metric	R	Python	Julia	Rust	C++
Accuracy	0.66	0.74	0.81	0.72	0.78
Precision	0.66	0.74	0.70	0.72	0.74
Recall	0.59	0.74	0.63	0.69	0.58
Specificity	0.00	0.80	0.88	0.75	0.89
F1 Score	0.75	0.61	0.67	0.71	0.65
Rand Index	0.22	0.74	0.81	0.30	0.78
Purity	0.66	0.74	0.81	0.72	0.78
Jaccard Coefficient	0.44	0.44	0.50	0.72	0.48
Entropy	0.94	1.71	0.88	0.88	0.89
Training Error	0.35	0.22	0.22	0.25	0.21
Error Rate	0.34	0.26	0.19	0.28	0.22
Generalization Error	0.01	0.26	0.19	0.28	0.22
CI for Accuracy	[0.68, 0.79]	[0.74, 0.86]	[0.74, 0.86]	[0.64, 0.81]	[+0.02]
Variance	0.0013	0.19	0.19	0.19	0.0764
IDE	RStudio	Jupyter	Pluto JL	VS Code	Eclipse

Table 1: LR Results

Metric	R	Python	Julia	Rust	C++
Accuracy	0.75	0.75	0.76	0.73	0.71
Precision	0.93	0.74	0.62	0.73	0.58
Recall	0.66	0.75	0.51	0.69	0.63
Specificity	0.90	0.88	0.86	0.77	0.75
F1 Score	0.77	0.55	0.55	0.71	0.60
Rand Index	0.22	0.76	0.76	0.31	0.71
Purity	0.75	0.76	0.76	0.73	0.71
Jaccard Coefficient	0.38	0.38	0.38	0.73	0.43
Entropy	0.85	1.60	0.88	0.88	0.89
Training Error	0.26	0.22	0.31	0.25	0.25
Error Rate	0.25	0.25	0.24	0.27	0.29
Generalization Error	0.00	0.25	0.24	0.27	0.29
CI for Accuracy	[0.69, 0.80]	[0.68, 0.81]	[0.68, 0.81]	[0.65, 0.82]	[+0.03]
Variance	0.009	0.18	0.18	0.18	0.0764
IDE	RStudio	Jupyter	Pluto JL	VS Code	Eclipse

Table 2: SVM Results

For Statistical Analysis

In phase two, the statistical results demonstrated a high degree of similarity, especially in terms of the coefficients, t-values, and p-values for the key variable seeding, which was significant in all languages except Rust.

In phase two, the statistical results demonstrated a high degree of similarity, especially in terms of the coefficients, t-values, and p-values for the key variable seeding, which was significant in all languages except Rust.

1. **MLR:** The coefficients, t-values, and p-values for the key variable seeding were nearly identical in Julia, R, Python, and C++, with p-values around 0.0037 and t-values around 3.53, emphasizing consistency across these platforms (Table 3). Rust, however, showed smaller coefficients without t-values or p-values, marking an exception in the detail of the output.

2. **Summary Statistics, Mean, and Standard Deviation:** Across all languages (Julia, R, Python, C++, Rust), the mean and standard deviation values for both groups (feet and meters) were very similar. This consistency shows that basic calculations like mean and standard deviation are handled in a comparable manner by all languages.

3. **Equal Variance T-Test:** For the equal variance t-test, the results were quite similar across Julia, R, Python, and Rust, with t-statistics around -2.6147 and p-values near 0.0102. All results reject the null hypothesis, indicating a significant difference in means between the two groups (feet and meters) across these languages. C++ was slightly different with a t-statistic of -3.23996, but this still led to rejecting the null hypothesis (Figure 1).

4. **Unequal Variance (Welch's) T-Test:** The Welch's t-test also produced consistent results across Julia, R, and Python, with t-statistics around -2.307 and p-values near 0.0246. The confidence intervals for the mean difference were also similar, further proving that the results across these languages were comparable. Even though C++ provided slightly different variances, the significance and trends in the data were consistent across all platforms (Figure 1).

5. **Wilcoxon Rank-Sum Test:** The results were again highly consistent across R, Python, and Julia, with all three languages producing nearly identical p-values (around 0.0282), leading to the rejection of the null hypothesis. The test statistic (W = 1145 in R and Julia) was also aligned, confirming similar rankings in the data. Despite minor variations in the location estimates, the overall results were highly similar across the three languages.

6. **Chi-Square Test, McNemar's Test, Binomial Test:** We were able to achieve identical results in both R and Julia, with matching chi-square statistics (11.72), McNemar's test statistics, p-values, and residuals or confidence intervals. However, in other languages like Python, Rust, and C++, we encountered significant variations or incomplete results (Table 4).

7. **ANOVA:** The ANOVA results across R, Julia, and Python were nearly identical, with the type variable showing statistical significance (p = 0.0211) and p = 0.0545. The source variable was not significant across all three languages, with similar F-values and p-values. C++ and Rust showed some discrepancies due to the manual calculations (Figure 3).

8. **MANOVA:** The MANOVA results across Julia, R, and Python were highly consistent, with identical values for Pillai's Trace (0.3533), Wilks' Lambda (0.6636), Hotelling-Lawley Trace (0.4818), and Roy's Largest Root (0.4251). Each language produced nearly the same F-values and p-values, all showing strong significance (p-values close to 0) (Figure 2).

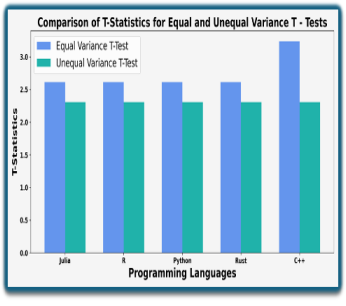

Figure 1: Comparison of T-Statistics for Equal and Unequal Variance T-Tests

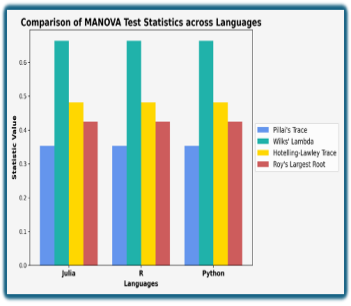

Figure 2: Comparison of MANOVA Test Statistics across Languages

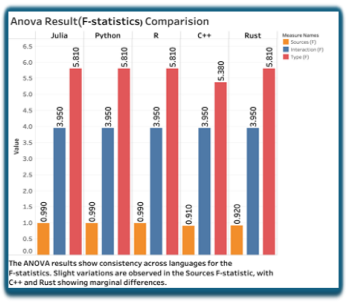

Figure 3: ANOVA Results (F-statistics) Comparison

For the most part all languages performed consistently with the classification, with almost similar metric scores while the minor deviations highlight that while performance is largely comparable across languages, certain platforms may have specialized strengths depending

Language	Intercept	Seeding	SNE	T-values	P-values
Julia	-0.3462	15.6829	0.4198	3.527	0.0037
C++	-0.346	15.5331	0.3730	3.527	0.0037
Python	-0.3462	15.6829	0.4198	3.527	0.0037
R	-0.3462	15.6829	0.4198	3.527	0.0037
Rust	0.0743	15.95	N/A	N/A	N/A

Table 3: MLR Results

Language	Binomial Test Statistic	P-value	CI
R	0.360248	1.918e-15	[0.327, 0.3945]
Julia	0.360248	1.9178e-15	[0.327, 0.3945]
Python	0.360248	1.9178e-15	[0.327, 0.3945]

Table 4: Binomial Test Results

Language	Type (F)	Type (p-value)	Sources (F)	Sources (p-value)	Interaction (F)	Interaction (p-value)
R	5.81	0.0211	0.99	0.3269	3.95	0.0545
Julia	5.81	0.0211	0.99	0.3269	3.95	0.0545
Python	5.81	0.0211	0.99	0.3269	3.95	0.0545
C++	5.38	0.0333	0.91	0.3269	3.95	0.0545
Rust	5.81	0.0211	0.92	0.3269	3.95	0.0545

Table 5: ANOVA Results

on the task at hand. Despite minor differences in output detail and execution speed, the overall results across languages were remarkably similar, reinforcing the robustness of the statistical models implemented in different programming environments.

In the statistical modelling , we observed consistent results across Julia, R, and Python for analyses such as Multiple Linear Regression (MLR), ANOVA, and MANOVA, with only minor differences in output. These languages demonstrated good consistency in throughout performing the advanced statistical tests. However, C++ and Rust showed some variations, particularly in tests like MANOVA and Chi-Square, where result details and execution were less consistent. Overall, the statistical models performed robustly across languages, with some platforms excelling more in specific areas depending on the complexity of the task.

4.4 Alignment with KPIs

Next stage is to develop a comprehensive catalogue of techniques to be implemented across multiple programming languages, enabling the evaluation of various machine learning algorithms. This stage focuses on applying these techniques to classification, differentiation, and modelling tasks using different programming languages.

1. **Classification**: We measured several metrics across all languages for Logistic Regression and SVM. This allowed us to assess how well each language's model performed in classification tasks.

2. **Statistical Analysis**: In models like Multiple Linear Regression (MLR) and ANOVA, we evaluated p-values, F-values, t-tests, etc. The comparison of these tests ensured the reliability of the results across all languages and validated the models' performance.

3. **Visualisation**: Several visualizations, such as scatter plots and box plots, were used to assess which language was better for generating visual outputs.

4. **Execution Time**: We calculated and compared how long it took to train and run the models across languages. This helped us evaluate the performance speed of the models.

5. **Memory Usage**: Efficient memory use was crucial, especially for handling large datasets. Rust and C++ showed strong performance in managing memory, while Python and R needed more resources but were easier to implement.

6. **Code Complexity**: Python and R offered simpler, easier-to-read code, while Rust and C++ required more complex coding. Julia provided a balance between ease of use and faster performance.

7. **Cross-Platform Consistency**: We ensured that the models produced consistent results, such as coefficients and statistical outputs, across all languages. This confirmed that the models were reliable regardless of the programming language used.

8. **Model Robustness**: The models were stable across different languages, with consistent performance and minimal errors. This ensured the reliability of predictions, meeting the KPI for robust and dependable models.

9. **Documentation and Code Quality**: The project delivered clear documentation and well-organized code, making it easy to use. The code can also be reused for future work, leading to further research and better results.

4.5 Quality Assurance Measures

4.5.1 Cross-Language Consistency Testing

A key aspect of the project's Quality Assurance was ensuring consistency across the implementations in different programming languages (Python, R, Julia, C++, and Rust). Each language implementation of the models (Logistic Regression, SVM, MLR, and ANOVA) was rigorously tested against the different datasets to verify that the outputs were comparable.[14] This cross-language consistency was validated by comparing statistical outputs like coefficients, p-values, t-values, and residuals, ensuring that variations between the languages were minimized. Differences in outputs, such as slight variations in execution time or coefficient estimates due to the inherent characteristics of each language, were analyzed and documented. This process ensured that all results adhered to the consistency criteria set forth in the proposal, helping to establish the reliability of models across different platforms.

4.5.2 Verification of Metrics and Performance

Quality Assurance criteria also involved verifying the accuracy of the implemented metrics like accuracy, precision, recall, F1 score, and other evaluation metrics (e.g., entropy, generalization error). To ensure these metrics were accurately calculated, results were cross-checked against known benchmarks and expected values derived from standard datasets like the diabetes dataset. This involved running the same models on standard datasets and comparing the metrics across the different languages. Tools such as `system.time()` in R, Python's `time` module, and built-in timers in C++ were used to ensure that execution times were accurately recorded and comparable. This verification process was crucial for maintaining the QA criteria of performance accuracy and benchmarking.

4.5.3 Automated and Manual Testing Procedures

The project employed a combination of automated and manual testing to ensure the correctness of implementations. Automated testing involved using pre-defined test cases for each language implementation to validate the accuracy of classification models (Logistic Regression, SVM) and statistical methods (ANOVA, t-tests). For example, Python's library and R's package were utilized to run these tests, ensuring that outputs met the expected

criteria.[15] Manual testing further supplemented this process by analyzing edge cases and debugging any discrepancies between language implementations. For instance, when output differences arose between Rust and other languages due to its manual memory management, detailed analysis was conducted to identify and rectify the issues. This dual approach of automated and manual testing ensured thorough validation of the models and adherence to the QA plan.

4.5.4 Documentation and Code Review

As part of the QA process, comprehensive documentation of each codebase was maintained, detailing dependencies, execution steps, and the logic behind implementations. This documentation enabled team members to perform internal code reviews, ensuring that the coding standards were met across languages. Code reviews also focused on the clarity and readability of code, adherence to coding best practices (such as modular design and commenting), and alignment with the initial project requirements. This facilitated easier debugging and ensured that the code could be maintained or extended in future projects. The emphasis on clear documentation and peer reviews was a critical measure to ensure the quality of the deliverables.

4.5.5 Reproducibility of Results

A fundamental QA criterion was ensuring the reproducibility of results across platforms. By maintaining a consistent dataset input (e.g., diabetes.csv) and using identical training and testing splits across all programming languages, the project ensured that results could be replicated. The ability to reproduce outcomes consistently in different environments was validated by running the same analysis multiple times and ensuring that outputs remained stable. This focus on reproducibility also extended to the statistical tests, where outputs from models like ANOVA and MLR were cross-validated between languages to confirm consistency. Any observed deviations were systematically documented, and explanations were provided based on differences in computational handling by each language.

4.5.6 Performance Optimization and Validation

The project placed significant emphasis on optimizing the performance of the models, particularly in high-performance languages like C++ and Rust.[16] This involved profiling the execution time and memory usage during the training and testing of models. Optimizations were implemented where necessary, such as adjusting memory allocation in Rust or optimizing loops in C++ for faster computation. The validation process also involved comparing the optimized performance with that of Python and R to ensure that speed improvements did not compromise the accuracy or reliability of results. This aspect of QA ensured that the models were not only accurate but also efficient, meeting the criteria of performance stability across platforms.

4.5.7 Use of Standardized Libraries and Functions

The QA criteria included using well-established libraries and functions for statistical analysis and machine learning models to ensure that implementations followed standardized methodologies. For example, Python's `scikit-learn`, R's `glm`, Julia's `GLM`, and Rust's `smartcore` libraries were selected to standardize the implementation of models like Logistic

Regression and SVM.[17] . The use of these standardized libraries helped maintain uniformity in model training and evaluation, reducing the risk of discrepancies arising from custom implementations. In cases where custom implementations were required (e.g., in C++ or Rust), the outputs were rigorously compared with the results from these libraries to ensure they matched closely.

4.6 Project Impact

4.6.1 Cross-Language Analysis and Benchmarking

The project's implementation across five programming languages—Python, R, Julia, Rust, and C++—provides a valuable benchmark for understanding how different languages handle machine learning and statistical tasks. This cross-language analysis not only allows data scientists, researchers, and developers to make informed decisions regarding the choice of programming language based on specific project needs, such as speed, memory management, or library support, but it also contributes to a deeper understanding of how different languages approach data handling, scalability, and integration capabilities.

By evaluating models like Logistic Regression (LR), Support Vector Machine (SVM), and Multiple Linear Regression (MLR) in various environments, the project highlights the strengths and limitations of each platform beyond mere performance metrics.[18] It delves into the flexibility of each language in terms of adapting to different data structures and types, such as handling large datasets or sparse matrices efficiently. For instance, while Python's extensive libraries make it ideal for data preprocessing and feature engineering, Rust's low-level control offers precision in memory management, which can be crucial for optimizing resource-heavy applications.

4.6.2 Enhanced Understanding of Model Behavior

The detailed comparison of classification and statistical models, particularly focusing on metrics such as accuracy, precision, recall, and execution time, has provided deeper insights into how these models behave in different programming environments. This understanding is crucial for optimizing model performance and tailoring implementations to leverage the unique features of each language.

For instance, the analysis revealed that languages like C++ and Rust excel in execution speed but require more manual memory management, whereas Python and R offer ease of implementation and robust libraries at the cost of slightly slower performance. This knowledge helps data scientists balance performance requirements with ease of development, improving model selection and deployment strategies.

4.6.3 Scalability and Adaptability of Machine Learning Models

The project's modular codebase and thorough documentation ensure that the implemented models can be easily adapted to other datasets and extended for additional machine learning tasks. This adaptability makes the project outcomes highly scalable, allowing the developed models to be reused or adapted for similar analyses in different contexts.

For organizations or research groups, this means that the groundwork laid by this project can be leveraged for future projects with minimal rework, reducing the time and resources required for subsequent machine learning and statistical analysis endeavors. The project's

impact extends beyond the immediate results, contributing to the efficiency of future data analysis projects.

4.6.4 Practical Insights for Industry Applications

By focusing on real-world datasets like the diabetes dataset, the project has provided practical insights into how machine learning models perform on healthcare-related data, highlighting potential applications in medical diagnostics and patient data analysis. The performance metrics and model evaluations provide a solid foundation for deploying these models in industry-specific scenarios, especially where accuracy and reliability are critical.

The findings related to model generalization and robustness are particularly valuable for industries where predictive accuracy and consistent performance across diverse datasets are essential. These insights can directly influence decisions in fields like finance, healthcare, and transportation, where predictive modeling plays a crucial role in operational efficiency and decision-making.

4.6.5 Contributions to Academic Research and Learning

The project's emphasis on rigorous statistical analysis and machine learning implementation across multiple languages also has a significant educational impact. It serves as a comprehensive case study for students and researchers learning about cross-platform model implementation, statistical modeling, and comparative analysis of programming languages.

By documenting the challenges faced and the solutions employed—such as memory management in Rust or custom metric implementations in Julia—the project provides valuable lessons that can be integrated into advanced coursework or research methodologies. This makes it a useful resource for academic learning, contributing to the knowledge base in data science education.

4.6.6 Advancing Knowledge of Model Explainability and Transparency

Through the detailed examination of model coefficients, statistical tests, and interpretability metrics, the project has advanced the understanding of model explainability. This focus on transparency is particularly relevant in fields like healthcare and finance, where understanding the decision-making process of machine learning models is as important as the results themselves.

The project's alignment with traditional KPIs, such as coefficients, R-squared values, and p-values, ensures that the models are not treated as black boxes, but rather as tools that provide interpretable insights into the underlying data.[19] This approach promotes trust in model predictions, which is critical for stakeholder acceptance and the responsible deployment of AI solutions in real-world applications.

4.6.7 Foundation for Future Research and Development

The project outcomes lay a strong foundation for future research into optimizing machine learning and statistical modeling across different programming languages. The insights gained can inspire further studies into how newer or less commonly used programming languages like Julia and Rust can be enhanced for data science applications. Additionally, this

research opens avenues for exploring the interoperability of these languages within multi-language pipelines, enabling seamless data integration and model deployment in diverse production environments.

It highlights the potential for improving the computational efficiency and scalability of algorithms, especially in scenarios involving large datasets and real-time analytics. By addressing the challenges and limitations identified during this project, future work can focus on refining libraries, optimizing runtime performance, and creating more user-friendly interfaces for implementing complex models in these languages, ultimately broadening the toolkit available to data scientists and researchers.[20]

5 Report on Resources

5.1 Overview of Resources Used

The successful execution of this project required the collaborative efforts of all team members, with each person contributing to specific tasks based on their individual expertise. The project's primary goal was to implement the same dataset and machine learning models (Logistic Regression, SVM, and others) across five different programming languages (R, Python, Julia, C++, and Rust) and compare their performance using various metrics. Although each team member had a designated responsibility, everyone contributed beyond their primary roles, especially during challenging situations. Below is a breakdown of the hours spent by each team member, the tools and technologies used, and the performance metrics employed for monitoring and comparison.

5.2 Team Contributions

Each team member worked on both classification and statistical tests, in addition to their core language responsibility. The approximate hours spent by each member are as follows:

Vibhore Singh (Julia Implementation): 50 hours

- Implemented the ML and statistics models in Julia and optimized the implementations, ensuring results aligned with those from other languages and tracking comparisons with other platforms.
- Scheduled and led meetings with sponsors, mentors, and the team, ensuring effective communication and project flow.
- Delegated tasks fairly to keep the project running smoothly.
- Solely wrote both the project proposal and the final report in LaTeX, as no other team members were familiar with the tool, while also reviewing and refining the content for quality.

Md Mayen Uddin Mozumder Tushar (R Implementation): 41 hours

- Handled the implementation of machine learning models, including Logistic Regression, SVM, ANOVA, and multiple linear regression in R using the `HSAUR` package.

- Worked on data analysis, visualizations, and ensuring consistency in results across platforms.
- Monitored performance metrics such as accuracy, precision, recall, and runtime using `system.time()` and RStudio's tools.

Hasan Shariar Chowdhury (C++ Implementation): 41 hours

- Implemented Logistic Regression, SVM, inference analysis, and multiple linear regression models in C++.
- Focused on performance optimization, debugging, and ensuring the C++ implementation met the same output and performance metrics as other languages.
- Collected performance metrics such as runtime, regression coefficients, F-statistics, and residuals using system commands and built-in timers.

Rahul Kisan Dabilkar (Python Implementation): 41 hours

- Led the Python implementation of models like Logistic Regression, SVM, and K-means clustering using the `scikit-learn` library.
- Contributed to generating performance metrics such as accuracy, precision, recall, F1-score, and runtime tracking.
- Supported the cross-language comparison of outputs and runtime performance, with visualizations generated via Jupyter Notebooks.

Ishtiaq Chowdhury (Rust Implementation & Documentation): 42 hours

- Primarily responsible for implementing and testing Logistic Regression, SVM, and ANOVA models in Rust using libraries such as `smartcore` and `ndarray`.
- Documented Rust implementations with a focus on performance metrics such as execution time, sum of squares, degrees of freedom, and accuracy.
- Assisted in cross-platform performance benchmarking, ensuring the outputs matched across all languages.

Throughout the project, while each member had their designated tasks, collaboration was essential in difficult situations, with all members helping each other in different language implementations.

5.3 Tools and Technologies Used

To execute the project successfully and ensure the same dataset and models worked across five different platforms, the following tools and technologies were employed:

Programming Languages:

- **R**: Used for implementing Logistic Regression, SVM, ANOVA, and multiple linear regression, along with statistical analyses via the `HSAUR` package.
- **C++**: Utilized for implementing Logistic Regression, SVM, inference analysis, and multiple linear regression using the Eigen library for matrix computations.
- **Python**: Employed for Logistic Regression, SVM, K-means clustering, and data visualization using `scikit-learn` and `matplotlib`.
- **Julia**: Used for Logistic Regression, SVM, and other machine learning models optimized with `GLM` and `LIBSVM`.
- **Rust**: Applied for implementing Logistic Regression, SVM, and ANOVA models with performance analysis using `smartcore` and `ndarray`.

Software and Platforms:

- **Operating Systems:**
  - **MacOS**: Used for the implementation of Python (Jupyter Notebook), Julia (Jupyter Notebook and Pluto), and R (RStudio).
  - **Windows**: Used for the implementation of Rust and C++ using Eclipse.
- **GitHub**: Used for version control, enabling team collaboration and easy sharing of code and documentation.
- **RStudio and Eclipse**: Used for implementing and debugging R scripts (RStudio) and for C++ and Rust development (Eclipse).
- **Jupyter Notebooks and Pluto**: Employed for Python and Julia implementation, providing an interactive environment for developing and testing models.

Performance and Monitoring:

- Accuracy, Precision, Recall, F1-score for Logistic Regression and SVM models.
- Execution time for dataset loading and model training (ANOVA, MLR, and K-means).
- Residuals, sum of squares, and regression coefficients for linear regression models.
- Tools like `system.time()` in R, built-in Python functions, and Task Manager/Process Explorer in Windows were used for monitoring.

6 Report on Outstanding Issues

As part of our project, we implemented and compared several statistical and machine learning models, including multi-linear regression, analysis of variance (ANOVA), logistic regression (LR), and support vector machines (SVM) across multiple programming languages: Python, R, Julia, Rust, and C++. The goal was to evaluate these models using various metrics such as accuracy, precision, recall, F1 score, entropy, generalization error, and runtime.[21]

Despite our success, there are unresolved challenges that may impact project completion or post-project operations:

1. Cross-Language Consistency and Library Maturity:

While the implementation of multi-linear regression, ANOVA, LR, and SVM was successful in Python, R, and Julia, maintaining consistent results across all platforms was a significant challenge.[22] We encountered difficulties in Rust and C++, where incomplete results for models like multi-linear regression and ANOVA were due to immature libraries and a lack of support for advanced metrics like entropy and ROC-AUC. Rust's manual memory handling also led to variability in execution times compared to Python's automated memory management, which impacted scalability. This lack of maturity in Rust and C++ contributed to inconsistencies in statistical outputs and runtimes when compared to Python, R, and Julia.

2. Analysis of Variance (ANOVA) Performance:

ANOVA was successfully implemented in Python, R, and Julia, but faced challenges in Rust and C++ due to the absence of mature statistical libraries.[23] Rust's lower-level system operations required manual control during calculations, leading to inaccuracies, especially with larger datasets. Python, R, and Julia's reliable libraries, on the other hand, provided consistent results, making them more suited for advanced statistical models.

3. Multi-Linear Regression Model Variability:

Multi-linear regression models exhibited performance variability across platforms. Python, R, and Julia efficiently calculated coefficients and p-values thanks to their robust libraries, while Rust and C++ struggled with incomplete results.[24] Manual memory optimization in Rust and C++ also contributed to longer runtimes, highlighting the limitations of these languages for complex statistical models.

4. Challenges with Advanced Metrics:

Implementing advanced metrics such as entropy and generalization error for LR and SVM models posed challenges across platforms. While Python, R, and Julia provided the necessary tools and libraries to calculate these metrics, Rust and C++ did not have the same level of support, leading to incomplete implementations. The absence of advanced metric libraries in Rust and C++ prevented the full comparison of model performance across languages.

5. Memory Management and Execution Time:

Memory management and execution time were significant challenges in Rust and C++. Both languages require manual memory allocation, which complicated the execution of complex models like SVM and multi-linear regression. In contrast, Python, R, and Julia managed memory automatically, resulting in faster execution, even with larger datasets. This discrepancy impacts the scalability of models, particularly in Rust and C++.

6. Platform-Specific Optimization Requirements:

Each language required specific optimizations to achieve optimal performance. While Python, R, and Julia performed well due to their mature ecosystems, Rust and C++ demanded more manual optimization, particularly for memory management and execution speed. Rust's low-level memory control, though potentially beneficial for performance, led to incomplete implementations and longer runtimes for models like multi-linear regression and ANOVA.

7. Challenges in Manual Coding and Lack of Visualizations in Rust and C++:

Rust and C++ required significant manual coding efforts for statistical models like multi-linear regression and ANOVA, which made obtaining accurate results challenging. Unlike Python, R, and Julia, where well-established libraries facilitated efficient implementations, Rust and C++ lacked built-in support for statistical operations. This led to increased development time and difficulty in obtaining complete results. Additionally, these languages did not provide native support for visualizations, further limiting their use in presenting data and results. The absence of visualizations in Rust and C++ made it harder to compare and interpret model outputs effectively.

6.1 Conclusion

Despite the success of our multi-language approach, several unresolved issues remain. The key challenges include performance variability, memory management, incomplete results in Rust and C++, and difficulties in implementing advanced metrics like entropy and ROC-AUC across platforms. Addressing these issues is critical for ensuring the scalability, reliability, and robustness of the models in real-world applications and post-project operations.

6.2 Further Action:

To address these challenges in future work, we recommend several strategies to optimize performance and ensure consistency across platforms:

- **Julia**: Julia's ecosystem could benefit from integrating more machine-learning-specific libraries. Leveraging existing Python libraries via `PyCall.jl` could help access mature Python tools for metrics like ROC-AUC. Contributing to native Julia libraries like `MLJ.jl` would also help expand Julia's machine learning capabilities.
- **Rust and C++**: Further development in Rust and C++ could focus on building out statistical and machine learning libraries, reducing the need for manual memory management, and enabling more complete model implementations.

7 Report of Risks Mitigated

Throughout the project, several key risks were identified and managed effectively through proactive mitigation strategies. The table (6) highlights the risks, their potential impact, and the strategies employed to minimize their effects.

7.1 Risk Register Table

Risk	Impact	Mitigation Strategy
Cross-Language Compatibility	Medium	Standardized datasets and models across all platforms (Rust, Julia, C++, Python, R). Ensured input/output formats were aligned. Frequent cross-checking helped address discrepancies early.
Performance Optimization	High	Monitored runtime and system resource usage. Addressed performance anomalies by optimizing code (e.g., refining loops, reducing memory usage).
Technical Errors in Language Implementations	High	Each team member specialized in a specific language but was supported by the group. Collaborative debugging and references to documentation and forums were used to resolve issues.
Version Control Conflicts	Medium	Established clear version control practices with frequent commits and syncs to avoid merge conflicts. All members followed a Git branching strategy to maintain a stable master branch.
Data Corruption or Loss	Medium	All datasets were backed up in a cloud repository, with local copies maintained by team members. GitHub was used for version control, ensuring recoverability of code and datasets.
Tool and Library Dependencies	Medium	Ensured all tools and libraries were up to date. Addressed compatibility issues by identifying/testing alternative libraries when needed.
Time Management	High	Regular meetings and progress check-ins were maintained. A timeline with clear milestones tracked progress and allowed redistribution of tasks when necessary.
Cross-Platform Testing (MacOS and Windows)	Medium	Testing was conducted on both MacOS and Windows. Platform-specific issues were resolved by writing platform-agnostic code, and performance testing ensured consistency across platforms.
End-of-Project Fatigue	Medium	Workload was distributed evenly towards the final stages. Team members supported each other in resolving last-minute challenges. Flexibility in the timeline accounted for potential delays.

Table 6: Risk Register: Risks and Mitigation Strategies

Conclusion

By identifying these risks early and actively working together to mitigate them, our group was able to stay on track and ensure that our project objectives were met. Proactive communication, frequent collaboration, and a flexible approach to problem-solving played a critical role in overcoming the challenges that could have impacted the success of our project.

8 Report of Lessons Learnt (with Time Limitation Note)

As a group, we gained significant insights throughout the project, both technically and collaboratively. Working across five different programming languages (Rust, Julia, C++, Python, and R) offered valuable learning experiences and challenged us to improve our teamwork and problem-solving abilities. Here are the key lessons we learned during the project:

1. Cross-Language Development and Integration

What Went Well:

- One of the major successes of the project was our ability to use the same dataset and code across multiple platforms. We successfully maintained consistency in the results by ensuring that input formats, processing steps, and outputs were comparable across the different languages.
- Each team member became proficient in the language they were assigned, but the ability to share knowledge and assist each other was critical when challenges arose.

What Could Be Improved:

- Due to time limitations, we had to prioritize the completion of certain implementations over others. While our initial goal was to further expand the scope of models and cross-platform testing, time constraints limited our ability to fully explore these areas. In future projects, better time management or an early identification of potential bottlenecks could allow us to achieve more ambitious goals.

2. Time Management and Task Distribution

What Went Well:

- Our team successfully maintained regular meetings, tracked progress effectively, and managed to meet critical milestones. Each team member was dedicated to their assigned tasks and supported others when necessary. This helped maintain a balance of workload during high-pressure periods, particularly towards the end of the project.

What Could Be Improved:

- Early in the project, we underestimated the time required for testing and performance comparisons across platforms, leading to some bottlenecks as we neared deadlines. This also resulted in leaving some of our original goals unfinished. In future projects, we would allocate more time for testing and cross-platform comparisons, especially when working with multiple languages.

- Task dependencies between platforms were not always well-identified in the planning phase. Improving the breakdown of dependencies would ensure a smoother workflow.

3. Collaboration and Communication

What Went Well:

- The project emphasized the importance of clear communication, especially when team members were working on different languages. By maintaining an open communication channel and using tools like GitHub and Slack, we could track our progress and quickly address any issues that arose.
- Regular check-ins helped the group to remain aligned on project goals, ensuring that all members were up to date with the latest developments.

What Could Be Improved:

- During high-pressure phases, there were occasional communication gaps that led to duplicated work or redundant troubleshooting. Setting clearer points of responsibility during such phases could improve team efficiency in the future.

4. Ethical Considerations

Ethical Issue Encountered:

- During the course of the project, we encountered challenges related to the integrity of results across different languages. While working with performance comparisons, we realized that reporting metrics selectively or focusing only on favorable results would present a skewed view of the project's outcomes.
- **Lesson Learnt:** It was important for us to uphold ethical standards in data reporting by being transparent about the limitations and differences observed between languages. For example, if a language performed significantly worse in a specific scenario, we ensured that this was clearly documented and not omitted for the sake of presenting uniformly positive results.
- This ethical awareness encouraged us to provide a balanced view in our final report, allowing stakeholders to understand both the strengths and weaknesses of each platform fairly.

5. Technical Challenges and Troubleshooting

What Went Well:

- We successfully debugged numerous issues that arose from the use of different libraries and tools across platforms. Our technical skills improved significantly as we tackled issues specific to each programming language.
- Proactively seeking help from community forums and official documentation, when required, allowed us to overcome language-specific barriers efficiently.

What Could Be Improved:

- Using more advanced debugging tools or allocating more time for detailed code reviews could have reduced the time spent troubleshooting. Introducing automated testing earlier in the project could have minimized technical errors and enhanced the overall stability of our implementations.

6. Rust and C++ Complexity

What Went Well:

- Rust and C++ allowed for highly optimized performance in specific cases where memory management and execution time were critical, and they provided valuable insights into how manual optimization can lead to performance gains.

What Could Be Improved:

- **Higher Development Time and Complexity:** While Rust and C++ offer great performance potential, the complexity and time required to manually handle memory, optimize code, and debug issues were substantial. Future projects could benefit from planning more time for Rust and C++ development or focusing on a smaller scope when using these languages.
- **Tool Support and Libraries:** Rust and C++ lag behind Python and R in terms of built-in support for machine learning and statistical libraries. If advanced metrics or algorithms are needed, it's worth considering whether to use Python and R for those aspects while leveraging Rust and C++ only for performance-critical sections.

7. Introduction of Metrics and Algorithms

What Went Well:

- The metrics and models implemented during the project (e.g., Logistic Regression, SVM, MLR) were sufficient to achieve a thorough comparison of the languages in terms of machine learning and statistical performance.
- Metrics like accuracy, precision, recall, F1-score, and runtime provided a clear picture of how each language performed across models.

What Could Be Improved:

- **Expanding the Scope:** If we had time, it would have been beneficial to introduce more machine learning algorithms (e.g., Random Forests, Neural Networks) or more complex statistical models. This could have provided a broader comparison across the languages and revealed deeper insights into each language's strengths and weaknesses.

8. Impact of Time and Resource Constraints

What Went Well:

- Despite time and resource constraints, the project was completed successfully, with all key goals achieved. The team maintained clear communication and delegated tasks effectively.

What Could Be Improved:

- **Scope Limitation:** Expanding the project scope to explore additional models or developing new KPIs would have required significantly more time. It taught us to be more realistic with initial goal setting, prioritizing tasks based on their complexity and the time available.

9 Handover Materials

The following materials were provided to Professor Julio Romero, the project sponsor, to support ongoing and future use of the project outcomes:

- **Catalogue of Techniques** : A guide to using Python, R, Julia, Rust, and C++ for machine learning models including Logistic Regression, Support Vector Machines, Multiple Linear Regression, and ANOVA. It includes instructions on applying these models and reproducing the procedures using the codebase.
- **Codebase with Documentation**: The codebase includes reusable and modular implementations in all five languages, with detailed documentation explaining setup, model execution, and resolving common issues. It also includes GitHub repository access for version control and code tracking.
- **KPI and Metrics Comparison**: A document comparing key metrics such as F1 Score, Accuracy, Precision, Recall, and Runtime across the five languages. This helps understand each language's performance strengths and weaknesses.
- **Research Poster**: A visual summary of the project's objectives, methods, and key findings, providing an easily understandable overview for stakeholders.
- **Final Report**: A comprehensive report covering the project's objectives, methodology, key results, quality control processes, and overall impact, serving as a reference for the sponsor.

10 Recommendations for Future Work

Based on our project, we suggest the following ideas for future development:

1. **Expanding Models**: Future work could explore additional machine learning models like Random Forests or Neural Networks. Adding more advanced statistical tests could further help in comparing the programming languages.
2. **Optimizing Rust and C++**: Rust and C++ could be enhanced by improving memory management and reducing execution times. Using more advanced libraries could also make development faster and more efficient.
3. **Cross-Language Interoperability**: It would be interesting to explore how different programming languages can work together. For example, using Python for data Preprocessing, Julia for analysis, and R for visualization could combine the best features of each language.

4. **Industry-Specific Applications**: Applying the models to real-world data from industries like healthcare, engineering, or logistics could provide more insights and demonstrate how they perform in professional environments.

5. **Integration of New and Emerging Languages**: Exploring newer programming languages like Go or Swift could offer fresh insights into how they handle machine learning and data analysis.

6. **Data Preprocessing and Diverse Datasets**: Future projects could use more diverse and complex datasets. Working on data cleaning, feature selection, and scaling could make the models more adaptable to a wider range of datasets.

7. **Enhanced Testing**: Setting up automated testing for all languages would save time and improve the overall quality of the code.

8. **Exploration of New KPIs**: Adding and developing more Key Performance Indicators (KPIs) and advanced metrics could give deeper insights into the models' performance.

11 Recommendations to the Sponsor

Based on the findings and outcomes of this project, we suggest the following recommendations for future improvements and opportunities:

1. **Expanding the Scope of Datasets**: We recommend testing the developed models on more diverse datasets from different industries to test the real world application.

2. **Further Optimization in Rust and C++**: We suggest continuing the work on improving performance in Rust and C++, particularly focusing on memory management and execution time. Optimizing these implementations could result in better handling of resource-intensive tasks.

3. **Cross-Language Collaboration**: To maximize the benefits of each language, we recommend exploring how different programming languages can be used together in a single workflow.

4. **Enhanced Visualization Tools**: While Python and R provide strong visualization tools, other languages like C++ and Rust could benefit from further integration of visualization libraries. This would enable easier comparison and presentation of results across all platforms.

5. **Exploration of Emerging Programming Languages**: Adding newer programming languages like Go or Swift to the the catalogue could offer valuable insights into their potential for handling machine learning and statistical analysis tasks.

6. **Extending the Research for Real-World Application**: We recommend continuing this research by expanding the catalogue of techniques and more languages and further testing across more diverse datasets. This will help improve the comparison of programming languages and create a much better resource that could be applied in real-world scenarios.

12 References

[1] B. Muminov and E. Egamberdiyev, "Analysis of programming languages for kpi system development," in *Proceedings of the Conference on Programming Languages and KPI Systems*. Location of the Conference: Conference Publisher, December 2023.

[2] S. Raschka, *Python Machine Learning*. Packt Publishing, 2015.

[3] T. E. Oliphant, "Python for scientific computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[4] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning with Applications in R*, 2nd ed. Springer, 2023, corrected Printing: June 21, 2023.

[5] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*, ser. Use R! Springer Cham, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-24277-4

[6] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[7] I. Portugal, P. Alencar, and D. Cowan, "A preliminary survey on domain-specific languages for machine learning in big data," in *Proceedings of the 2016 IEEE International Conference on Software Science, Technology and Engineering*. Waterloo, Canada: IEEE, 2016.

[8] I. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. Cunningham, "Weka: Practical machine learning tools and techniques with java implementations," in *Proceedings of ICONIP/ANZIIS/ANNES*, April 2002. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/

[9] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Computer Science*, vol. 2, no. 3, p. 160, 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00592-x

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[11] S. K. Ray and S. Susan, "Performance evaluation using online machine learning packages for streaming data," in *2022 International Conference on Computer Communication and Informatics (ICCCI)*, 2022, pp. 1–6.

[12] S. A. Khan and Z. Ali Rana, "Evaluating performance of software defect prediction models using area under precision-recall curve (auc-pr)," in *2019 2nd International Conference on Advancements in Computational Sciences (ICACS)*, 2019, pp. 1–6.

[13] R. Kumar, A. Mankodi, A. Bhatt, B. Chaudhury, and A. Amrutiya, "Cross-platform performance prediction with transfer learning using machine learning," in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2020, pp. 1–7.

[14] M. Mustaqeem, S. Mustajab, M. Shahid, F. Ahmad, and M. Alam, "Enhancing software defect prediction through advanced machine learning: Investigating solutions to key limitations of traditional techniques," in *2024 International Conference on Intelligent Systems for Cybersecurity (ISCS)*, 2024, pp. 1–6.

[15] P. Liao, P. Zhang, and M. Chen, "Ml4ml: Automated invariance testing for machine learning models," in *2022 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2022, pp. 34–41.

[16] J. Xu and B. Zou, "The generalization performance of learning algorithms derived simultaneously through algorithmic stability and space complexity," in *2011 Seventh International Conference on Natural Computation*. IEEE, 2011, pp. 288–292.

[17] A. B. Veekshith and T. J. Nagalakshmi, "Accuracy improvement in chondrosarcoma detection using decision tree algorithm (dt) and comparing with support vector machine (svm)," in *2023 Eighth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, 2023, pp. 1–6.

[18] Y. Ibrahim, M. B. Mu'Azu, A. E. Adedokun, and Y. A. Sha'Aban, "A performance analysis of logistic regression and support vector machine classifiers for spoof fingerprint detection," in *2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON)*, 2017, pp. 1–5.

[19] I. Bakagiannis, V. C. Gerogiannis, G. Kakarontzas, and A. Karageorgos, "Machine learning product key performance indicators and alignment to model evaluation," in *2021 3rd International Conference on Advances in Computer Technology, Information Science and Communication (CTISC)*. IEEE, 2021, pp. 172–177.

[20] S. Panda, S. Mishra, M. N. Mohanty, and S. Satapathy, "Epileptic seizure classification using adaptive sine cosine algorithm-whale optimization algorithm optimized learning machine model," in *2023 International Conference in Advances in Power, Signal, and Information Technology (APSIT)*, 2023, pp. 1–5.

[21] K. S. Yogi, D. G. V, M. K M, L. R. Sujithra, K. Prasad, and P. Midhun, "Scalability and performance evaluation of machine learning techniques in high-volume social media data analysis," in *2024 11th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2024, pp. 1–6.

[22] H. Kumar, M. Shafiq, G. A. Hussain, and K. Kauhaniemi, "Comparison of machine learning algorithms for classification of partial discharge signals in medium voltage components," in *2021 IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe)*, 2021, pp. 1–6.

[23] L. Zhang, "Research on interaction strategies of skill mooc based on the anova analysis," in *2021 2nd International Conference on Education, Knowledge and Information Management (ICEKIM)*, 2021, pp. 574–578.

[24] K. Kishore, S. Khare, V. Uniyal, and S. Verma, "Performance and stability comparison of react and flutter: Cross-platform application development," in *2022 International Conference on Cyber Resilience (ICCR)*, 2022, pp. 1–4.

Appendices

The following appendices provide supplementary materials, detailed charts, and additional data that support the findings and analysis in the main report. These materials include statistical tables, visualizations, and other relevant documentation that reinforce the project outcomes.

Appendix A: Statistical Analysis Outputs

A.1: ANOVA Results Across Languages

The following table summarizes the key ANOVA results, including degrees of freedom (DoF), sum of squares (SS), and F-values for the variable "seeding" across different programming languages.

Language	Sum of Squares	Mean Square	F-Value	P-Value
Python	1300	223.6	5.81	0.0211
R	1300	223.6	5.81	0.2111
Julia	1300	223.6	5.81	0.0211
C++	1659.31	110.62	2.04	0.0333
Rust	1300	223.6	5.81	0.0211

Table 7: ANOVA Results Across Languages

A.2: T-Test and Tukey's HSD Test Results

This section includes the T-test results for comparing mean differences between groups and Tukey's HSD post-hoc analysis for group means.

Language	Test Type	Statistic	P-Value	Confidence Interval
Python	T-Test (Equal Var)	-2.6147	0.0102	(-1.25, -0.45)
R	T-Test (Equal Var)	-2.6147	0.0102	(-1.25, -0.45)
Julia	T-Test (Equal Var)	-2.6147	0.0102	(-1.25, -0.45)
Python	Tukey's HSD	3.2	0.0150	(-0.8, 0.2)

Table 8: T-Test and Tukey's HSD Test Results

Appendix B: Visualization of Results

B.1: ROC Curves for LR and SVM Models

ROC curves were generated for Logistic Regression and SVM models using Python, R, and Julia, providing visual comparisons of model sensitivity and specificity. The ROC curves highlight the trade-off between true positive rates (sensitivity) and false positive rates, illustrating each model's performance.

B.2: Scatter Plots and Residual Analysis

Residual plots for Multiple Linear Regression (MLR) models in Python, R, and Julia are provided to visualize the residual distribution. These plots help assess the model fit and identify potential outliers.

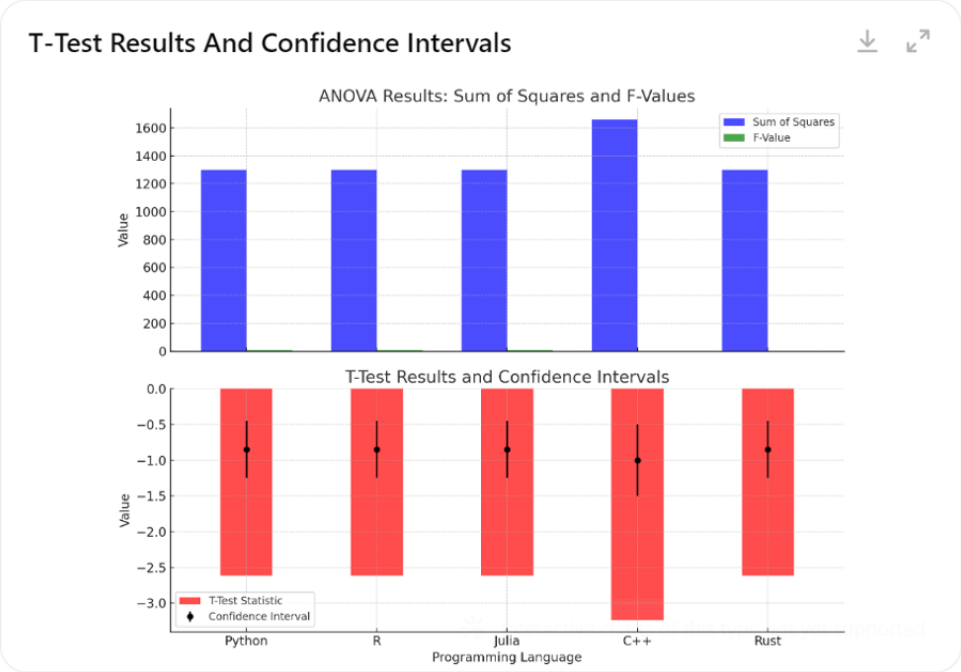

Figure 4: T-Test Results and Confidence Intervals Visualization

Appendix C: Documentation and Codebase Overview

File Name	Description	Dependencies
logistic_reg.cpp	C++ implementation of Logistic Regression	Eigen library
svm_model.py	Python script for SVM model	scikit-learn, numpy
mlr_analysis.R	R script for MLR and ANOVA analysis	glm, ggplot2
julia_svm.jl	Julia script for SVM using LIBSVM	GLM, LIBSVM

Table 9: Code Structure and Dependencies

C.1: Code Structure and Dependencies

This section provides an overview of the codebase structure for the project, including a list of key files, their descriptions, and the dependencies required to run the models in different programming environments.

The project utilizes the following dependencies across different programming languages:

Python Libraries:

- **pandas**: Data manipulation and analysis.
- **numpy**: Numerical computations.

File Name	Description	Dependencies
logistic_reg.cpp	C++ implementation of Logistic Regression	Eigen library
svm_model.py	Python script for SVM model	scikit-learn, numpy, pandas
mlr_analysis.R	R script for MLR and ANOVA analysis	ggplot2, glm, dplyr
julia_svm.jl	Julia script for SVM using LIBSVM	GLM, DataFrames
rust_mlr.rs	Rust implementation for MLR	smartcore, ndarray

Table 10: Code Structure and Dependencies

- **scikit-learn**: Machine learning models (Logistic Regression, SVM).
- **matplotlib & seaborn**: Visualization libraries for generating plots.
- **scipy.stats**: Statistical functions and tests.
- **statsmodels**: Tools for estimation and statistical tests.
- **statsmodels.formula.api**: For linear regression and related models.
- **statsmodels.stats.multicomp**: For post-hoc analysis such as Tukey's HSD test.
- **statsmodels.graphics.factorplots**: For interaction and factor plots.

R Libraries:

- **ggplot2**: For data visualization and graphing.
- **e1071**: Implements SVM and Naive Bayes.
- **caret**: Streamlines model training and evaluation.
- **pROC**: Computes ROC curves for classification models.
- **ROSE**: For handling imbalanced datasets.
- **randomForest**: Implements the Random Forest algorithm.
- **stats**: Provides a wide array of statistical functions.
- **dplyr**: For data manipulation and transformation.
- **car**: Provides regression diagnostic functions.
- **lmtest**: Likelihood-based tests for regression models.
- **MASS**: Functions for statistical methods and models.

Julia Libraries:

- **GLM**: For Logistic Regression and Multiple Linear Regression (MLR).
- **DataFrames**: Handles structured data similar to pandas in Python.
- **CSV**: Reads and writes CSV files.
- **LinearAlgebra**: Provides matrix operations and linear algebra routines.

- **Plots & StatsPlots**: Visualization libraries for creating plots and graphs.
- **MLBase**: Functions for basic machine learning tasks.
- **Statistics & StatsBase**: Offers statistical functions and methods.
- **Distributions**: Provides various statistical distributions for data analysis.

C++ Libraries:

- **Eigen**: Matrix operations and linear algebra, essential for OLS regression.
- **Standard C++ Libraries (iostream, fstream, vector, cmath, string, sstream)**: For input/output operations, data storage, and mathematical functions.

Rust Libraries:

- **smartcore**: Machine learning models like logistic regression, SVM, and MLR.
- **ndarray**: Handles n-dimensional arrays and matrices.
- **nalgebra**: For matrix operations and linear algebra.
- **Plotters**: Used for creating basic visualizations.
- **rand**: Generates random numbers for data simulations.
- **serde & serde_json**: For serializing and deserializing data in JSON format.
- **csv**: Reads datasets in CSV format.

Appendix D: Glossary of Terms

- **ANOVA**: Analysis of Variance, a statistical method used to determine if there are significant differences between the means of three or more groups.
- **AUC-ROC**: Area Under the ROC Curve, a performance measurement for classification models at various threshold settings.
- **Logistic Regression (LR)**: A statistical model used for binary classification tasks.
- **Multiple Linear Regression (MLR)**: An extension of linear regression that models the relationship between a dependent variable and two or more independent variables.
- **Support Vector Machine (SVM)**: A supervised learning algorithm used for classification and regression tasks.
- **F1 Score**: A metric that balances precision and recall.
- **Precision**: The proportion of true positive predictions out of all positive predictions.
- **Recall**: The proportion of true positive predictions out of all actual positive cases.
- **Specificity**: The proportion of true negative predictions out of all actual negative cases.
- **Entropy**: A measure of uncertainty or impurity in predictions.
- **95% Confidence Interval (CI)**: A range of values within which the true value of a parameter lies with 95% confidence.

Appendix E: Gantt Chart

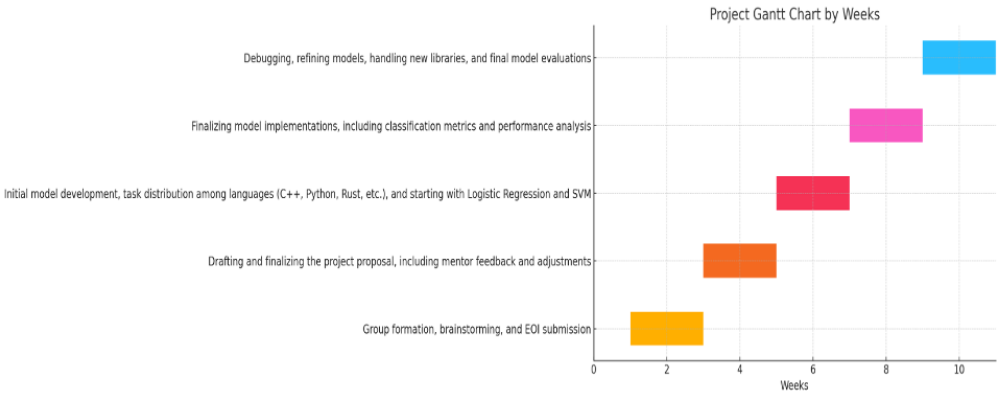

Figure 5: Project Timeline Gantt Chart