2025

Performance Evaluation of 16 Machine Learning Models Using the Dry Bean Dataset

Evaluating the Accuracy and Runtime of Traditional ML Models on a Supervised Classification Task

Ed Kaempf
August 2025

Performance Evaluation of 16 Machine Learning Models Using the Dry Bean Dataset

Executive Summary

This benchmarking project evaluated the accuracy and runtime of 16 machine learning models using the Dry Bean dataset. The dataset contains 13,611 samples across seven bean varieties. Each sample (also referred to as a dataset record) includes 16 numeric features representing physical characteristics of the bean. This was a supervised, multiclass classification benchmarking task, where each sample was labeled with one of the seven bean types, allowing models to learn patterns that distinguish among the seven beans and then predict the most likely bean type of unseen samples.

Before model training, all 16 dataset features were scaled using Scikit-learn's StandardScaler to normalize distributions and ensure consistent treatment across algorithms sensitive to feature magnitude. Bean names were encoded as integers 0 - 6 to enable compatibility with models that require numerical class values, keeping the bean types distinct while making them easier for the models to process.

Each model was fine-tuned using GridSearchCV across a small, tailored grid of hyperparameters, rather than relying solely on defaults. Parallel processing was enabled for the six models that support it via the n_jobs parameter to leverage multiple CPU cores and reduce runtime. While not all models benefit from multithreading, the tuning and measurement process was standardized to ensure fair comparisons. Runtime metrics exclude preprocessing and output-saving overhead. All runtimes were normalized to highlight relative efficiency rather than wall-clock times.

Key Findings

The most accurate models were Multi-layer Perceptron (MLP) Classifier (93.38 %), Support Vector Classifier (SVC) (93.14%), and XGBoost Classifier (93.02%). These three also ranked highest in both F1 Macro and F1 Weighted scores. The least accurate models are AdaBoost Classifier (84.41%), Gaussian Naive Bayes (GaussianNB) (89.67%), and Perceptron (90.21%). MLP accuracies for predicting each bean were Barbunya (92.81%), Bombay (100.00%), Cali (95.58%), Dermason (93.46%), Horoz (95.12%), Seker (95.36%), and Sira (88.09%).

Bombay was the most correctly identified bean type (94.89% overall accuracy). Notably, seven models predicted Bombay beans perfectly, and the 15 models other than AdaBoost had an overall 99.63% accuracy on Bombay beans. Among all misclassification of beans that occurred (i.e., labeling a bean incorrectly), 72.68% of them were those incorrectly labeling the Sira dry bean as a Dermason dry bean.

AdaBoost was a stark outlier in this study, correctly classifying the Bombay bean only 23.6% of the time, while the other 15 models averaged 99.6% accuracy, with seven achieving perfect

scores. Of the 399 Bombay samples AdaBoost misclassified, 398 were incorrectly labeled as Cali, making this the most consistent and lopsided misclassification by any model in the benchmark. This failure likely stems from AdaBoost's unique architecture: it uses very shallow trees without randomness, amplifies early misclassifications through aggressive sample reweighting, and lacks native support for multiclass classification. These traits made it especially vulnerable to confusion between visually similar, minority-class beans like Bombay, which comprised just 3.84% of the full dataset. In contrast, the other five tree-based models all handled Bombay with near perfect accuracy.

Runtimes were from just 20 ms to over 26,000 ms per test run (~68,055 records per feature-set test run). The slowest models, such as Gradient Boosting (26,160 ms) and MLP (12,355 ms), incurred substantial overhead due to iterative training and complex internal architectures. In contrast, models like Gaussian Naive Bayes, Ridge Classifier, Linear Discriminant Analysis, and Quadratic Discriminant Analysis ran consistently fast (20-40 ms). These fast models rely on closed-form mathematical formulas rather than trial-and-error iterative procedures, like gradient descent or tree boosting.

Hyperparameter tuning yielded negligible accuracy gains for most models, typically under 1 percentage point, despite each model being optimized solely for accuracy using GridSearchCV. However, tuning had substantial effects on runtime, with some models accelerating by over 70% through reduced depth or parallel execution. Others became significantly slower due to changes that increased optimization complexity or sequential workload. These tradeoffs show that tuning affected more than just accuracy. It influenced runtime and default behavior, depending on each model's design.

1. Introduction

Benchmarking machine learning models remains a useful tool in traditional ML workflows, offering information about how algorithms and models may perform under real-world conditions. Even as attention shifts toward deep learning and generative architectures, traditional algorithms continue to play a central role, particularly in business applications involving structured, tabular data, where interpretability, runtime efficiency, and deployment simplicity are often more important than model complexity.

This project evaluated the performance of 16 well-established machine learning models on a supervised classification task. The two objectives were:

- Measure how accurately each model classifies unseen samples from the Dry Bean dataset
- Determine each model's runtime during its classification training and testing process.

The benchmark used the Dry Bean dataset, a clean, fully numeric dataset with 13,611 records and seven types of beans. To reduce the impact of transient system activity (e.g., Windows background processes), each model's runtime was averaged over three full runs across all 137 possible feature subsets consisting of at least 14 of the 16 features, for every one of the 13,611 dry bean records.

While the dataset contains measurements of physical properties of dry bean types, the content was not the focal point of this evaluation. While this report examines feature importance, class

imbalance, and inter-feature correlations, this effort focused on measuring each model's behavior: predicting and runtime. The simplicity and cleanliness of the dataset made it a great candidate for isolating performance variables without adding on more preprocessing complexities. In this context, the beans were secondary. The primary interest was in evaluating how different algorithms and models handled a common classification task using standardized inputs. Where accuracy was impacted by feature correlations or distinctiveness, the impact is presented in this report.

The remainder of this report is organized as follows:

- Section 2 introduces the Python-based evaluation infrastructure and tools used to support repeatable benchmarking
- Section 3 provides an overview of the Dry Bean dataset
- Section 4 introduces the 16 machine learning models and categorizes them by algorithm family
- Section 5 describes initial data diagnostics and preparation
- Section 6 explains the benchmarking framework, including accuracy and runtime metrics, feature subset design, normalization techniques, and methods for measuring consistency
- Section 7 presents the results, including trade-offs between accuracy and speed, runtime variability, and practical considerations for selecting models based on task requirements
- The report concludes with a summary of model evaluation outcomes, feature-level observations, and opportunities to refine future benchmarking approaches
- Appendix A, Model Definitions, provides a brief description of how each machine learning model functions, key advantages or limitations, and typical use cases
- Appendix B, Stratified 5-Fold Cross-Validation Using StratifiedKFold, describes the cross-validation method and its relevance to evaluating the 16 machine learning models using the Dry Bean dataset.

2. Project Infrastructure

The software tools used to conduct the performance evaluation were developed in Python using Visual Studio Code, with Jupyter notebooks, on a local Windows PC. The workflow leveraged standard data science libraries, including pandas for data manipulation, and I/O (e.g., reading and writing Parquet and Excel files), and scikit-learn for model instantiations, training, hyperparameter tuning (via GridSearchCV), and cross-validation (StratifiedKFold). Additional libraries used included:

- **joblib** for saving model objects and results.
- **importlib** to dynamically import and instantiate machine learning model classes. This approach allowed the project to create model instances programmatically based on metadata, without having to hardcode each import and class name. It makes the code flexible and scalable for handling most other models. The metadata was stored as a

Python dictionary containing data for each model, such as class name, module path, default parameter values, and hyperparameter grid for tuning.

- **itertools** to create all 137 combinations of features (sets of 14, 15, and all 16 original features) for each of the 13,611 dry bean samples. These combinations ensured enough runtime measurements for each model to mitigate variability caused by background Windows activity.
- **IPython.display** for inline notebook outputs.
- **matplotlib**, **seaborn**, and **plotly.express** to create charts and results visualizations.

File and directory paths were centrally managed using a combination of a config.py module and pathlib, improving modularity and making the project extensible new datasets. JSON and Excel files stored model configurations and benchmarking results for downstream use.

The primary runtime metric in this evaluation is the relative difference between models, not their absolute runtimes. All runtimes were measured on a Windows 11 PC equipped with an Intel Core i7-14700F processor (20 cores, 28 logical processors, 2.10 GHz base speed), 32 GB of RAM, and a 1 TB SSD. The evaluation pipeline was implemented in Python 3.13.3 using scikit-learn 1.7.0, applying a cross-validation approach that captured both training and prediction times. Models capable of multithreaded execution were configured to utilize all available CPU cores. Runtime measurements were taken under controlled conditions, with background applications closed and power-saving features disabled to minimize interference.

3. Dry Bean Dataset Overview

The dataset contains 13,611 records of seven different registered dry beans. Each record has 16 numeric features (12 dimensions and 4 shape forms), and the bean type: Barbunya, Bombay, Cali, Dermason, Horoz, Seker, and Sira. **Figure 3.1** shows the distribution of beans. The dry bean dataset used for this project was obtained from the University of California, Irvine (UCI) Machine Learning Repository. [1]

Figure 3.1 Distribution of Dry Bean Samples

Bean	Count	Percentage
Barbunya	1,322	9.71%
Bombay	522	3.84%
Cali	1,630	11.98%
Dermason	3,546	26.05%
Horoz	1,928	14.17%
Seker	2,027	14.89%
Sira	2,636	19.37%
Total	**13,611**	**100.01%**

[1] Dry Bean [Dataset]. (2020). UCI Machine Learning Repository. https://doi.org/10.24432/C50S4B.

The Dry Bean dataset is a simple dataset as all 16 of its features are numeric, and there are no missing values, neither among any of the 16 features nor the class label (bean types). Fourteen features are "continuous" values and stored as floating-point numbers, and the remaining two are stored as integers: Area and Convex Area. The uniform structure eliminates the need to impute any feature values or transform feature values into numeric values. The 16 numeric features can be scaled to improve model performance.

To simplify the tables and charts developed for this project, bean abbreviations from the naming convention and definitions introduced by Koklu and Ozkan were adopted:[2]

1. Area (A): The area of a bean zone and the number of pixels[3] within its boundaries.
2. Perimeter (P): Bean circumference is defined as the length of its border.
3. Major axis length (L): The distance between the ends of the longest line that can be drawn from a bean.
4. Minor axis length (l): The longest line that can be drawn from the bean while standing perpendicular to the main axis.
5. Aspect ratio (K): Defines the relationship between L and l.
6. Eccentricity (Ec): Eccentricity of the ellipse having the same moments as the region.
7. Convex area (C): Number of pixels in the smallest convex polygon that can contain the area of a bean seed.
8. Equivalent diameter (Ed): The diameter of a circle having the same area as a bean seed area.
9. Extent (Ex): The ratio of the pixels in the bounding box to the bean area.
10. Solidity (S): Also known as convexity. The ratio of the pixels in the convex shell to those found in beans.
11. Roundness (R): Calculated with the following formula: $\frac{4\pi A}{P\text{^}2}$
12. Compactness (CO): Measures the roundness of an object: $\frac{Ed}{L}$
13. ShapeFactor1 (SF1) = $\frac{L}{A}$
14. ShapeFactor2 (SF2) = $\frac{l}{A}$
15. ShapeFactor3 (SF3) = $\frac{A}{\frac{L}{2} * \frac{L}{2} * \pi}$
16. ShapeFactor4 (SF4) = $\frac{A}{\frac{L}{2} * \frac{l}{2} * \pi}$

[2] KOKLU, M. and OZKAN, I.A., (2020), "Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques." Computers and Electronics in Agriculture, 174, 105507.
DOI: https://doi.org/10.1016/j.compag.2020.105507

[3] The Dry Bean features were derived from photographic images of the beans, with pixel-level data used to calculate features such as area and extent.

4. Benchmarked Models

This project evaluated 16 machine learning models, selected to represent a broad cross-section of classification algorithms. These models span a variety of learning strategies and computational approaches, illustrating the diversity of how algorithms detect patterns and make predictions from structured data. Each model included in this study is widely adopted in practice and well-supported by mainstream machine learning libraries, making them representative of the techniques most commonly applied in real-world classification tasks.

To support comparisons, the models are grouped into seven commonly used algorithmic families: tree-based, linear, kernel-based, instance-based, probabilistic, discriminant, and neural network models. These groupings, and the model accuracy and runtime results, can show how different types of algorithms come with their own strengths and weaknesses.

Each model is listed below by family. More detailed definitions, including how each model functions, typical use cases, and key advantages or limitations, are provided in Appendix A to this report.

A. Tree-based Models

1. Decision Tree Classifier
2. Random Forest Classifier
3. Extra Trees Classifier
4. Gradient Boosting Classifier
5. AdaBoost Classifier
6. XGBoost Classifier

B. Linear Models

7. Logistic Regression
8. Ridge Classifier
9. Stochastic Gradient Descent (SGD) Classifier
10. Perceptron

C. Kernel-based Models

11. Support Vector Classifier

D. Instance-based Models

12. KNeighbors Classifier

E. Probabilistic Models

13. Gaussian Naive Bayes (GaussianNB)

F. Discriminant Models

14. Linear Discriminant Analysis
15. Quadratic Discriminant Analysis

G. Neural Network Models

16. Multi-layer Perceptron (MLP) Classifier

To improve readability, model names are shortened throughout the report where appropriate. For example, “Classifier” is dropped once the context is clear, and widely recognized abbreviations (e.g., “XGBoost,” “SVC”) are used.

5. Data Understanding and Preprocessing

5.1. Feature Summary

Summary statistics for the Dry Bean dataset's 16 features are presented in **Figure 5.1**, on the following page . The coefficient of variation (CV), calculated as each feature's standard deviation divided by its mean and expressed as a percentage, supports comparisons across features with different measurement scales. While CV does not measure predictive value, it highlights which measurements are relatively stable (e.g., Solidity, CV ≈ 0.47 %) and which are highly dispersed (e.g., Area, CV ≈ 55 %). This structural snapshot was useful in spotting scale imbalances and potential outliers before any modeling began.

Figure 5.1 Summary Dry Bean Dataset Statistics

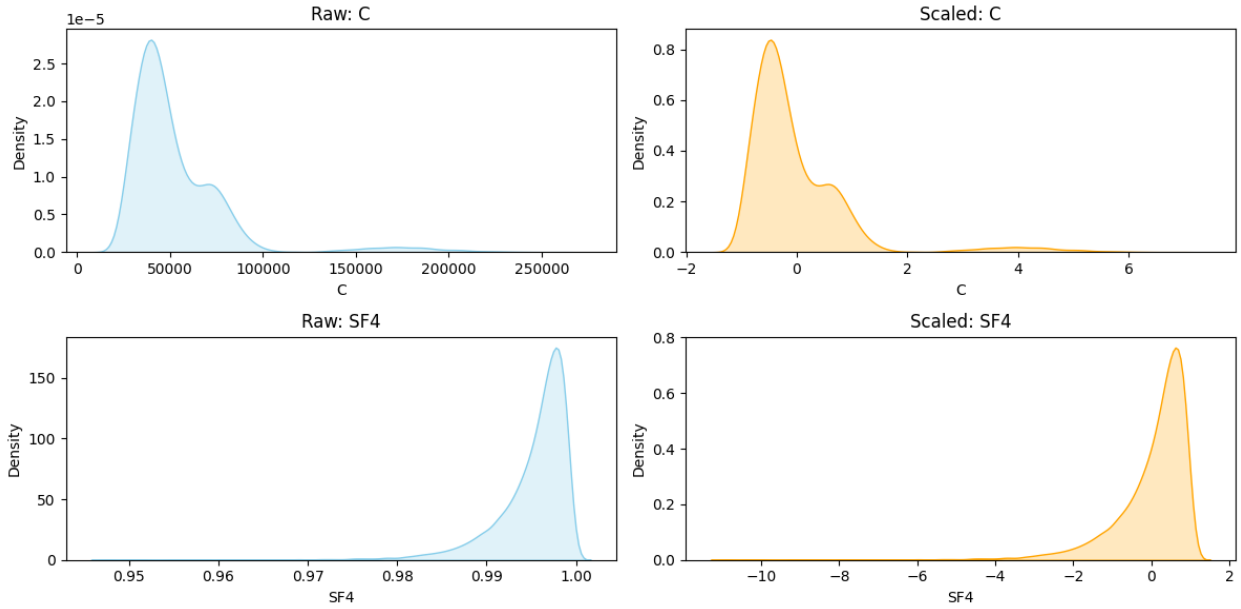
Feature	Minimum	Maximum	Mean	Std. Dev.	CV
Area	20,420	254,616	53,048.29	29,324.10	55.28%
Perimeter	524.7360	1,985.3700	855.2835	214.2897	25.05%
Major axis length	183.6012	738.8602	320.1419	85.6942	26.77%
Minor axis length	122.5127	460.1985	202.2707	44.9701	22.23%
Aspect ratio	1.0249	2.4303	1.5832	0.2467	15.58%
Eccentricity	0.2190	0.9114	0.7509	0.0920	12.25%
Convex area	20,684	263,261	53,768.20	29,774.92	55.38%
Equivalent diameter	161.2438	569.3744	253.0642	59.1771	23.38%
Extent	0.5553	0.8662	0.7497	0.0491	6.55%
Solidity	0.9192	0.9947	0.9871	0.0047	0.47%
Roundness	0.4896	0.9907	0.8733	0.0595	6.82%
Compactness	0.6406	0.9873	0.7999	0.0617	7.72%
Shape Factor1	0.0028	0.0105	0.0066	0.0011	17.18%
Shape Factor 2	0.0006	0.0037	0.0017	0.0006	34.73%
Shape Factor 3	0.4103	0.9748	0.6436	0.0990	15.38%
Shape Factor 4	0.9477	0.9997	0.9951	0.0044	0.44%

5.2. Feature Scaling

These statistics revealed a wide range of scales, from large-magnitude size metrics such as Area and Perimeter to compact, dimensionless shape descriptors such as Solidity and Shape Factor 4. Many machine learning models (particularly those using distance or gradient-based logic) can form distorted decision boundaries when features differ significantly in scale.

To eliminate this bias, all features were standardized using StandardScaler, resulting in a mean of 0 and a variance of 1. This preserved each feature's distribution shape while aligning them on a common scale. **Figure 5.2**, on the following page, illustrates with two example features, Convex Area (C) and Shape Factor 4 (SF4), showing identical distributions before and after scaling.

Figure 5.2 Pre- and Post-Scaling Feature Distributions (Convex Area and Shape Factor 4)



Standardization helped ensure fair treatment across models and reduced the likelihood that features with large raw values disproportionately influenced training. The following models in this benchmark are particularly sensitive to feature scale and directly benefited from this step due to their sensitivity to feature scale:

- Logistic Regression
- Ridge
- SGD
- Perceptron
- SVC
- K-Nearest Neighbors.
- LDA
- QDA
- MLP.

5.3. Feature Importance Analysis

To preview how features might influence predictions, a Random Forest classifier was trained on the raw Dry Bean dataset. Feature importance scores were extracted, which reflect how frequently and effectively each feature was used to split decision nodes across all trees in the ensemble.

These splits were evaluated based on how much they reduced impurity,[4] meaning how effectively they produced child nodes where most samples belonged to the same bean variety. Features that consistently led to purer splits received higher importance scores

Figure 5.3, on the following page, ranks the features by importance determined by Random Forest, normalized to sum to 1.00 (or 100%). For example, Perimeter contributed approximately 10.5% of the total impurity reduction across the forest. While higher scores indicate greater individual influence, lower-ranked features may still contribute value in interactions or under different model architectures.

[4] Impurity refers to how mixed the class labels are at a given node in a decision tree. A node is considered pure if all the samples it contains belong to the same class: higher impurity means the node contains a mix of different classes.

Figure 5.3 Feature Importance

Feature Name	Importance
Perimeter	0.104993
Shape Factor 3	0.104163
Shape Factor 1	0.102384
Compactness	0.088684
Major axis length	0.081993
Minor axis length	0.073249
Convex area	0.067083
Eccentricity	0.059967
Equivalent diameter	0.056802
Roundness	0.055443
Aspect ratio	0.053829
Area	0.049966
Shape Factor 2	0.042362
Shape Factor 4	0.029184
Solidity	0.018701
Extent	0.011197

5.4. Feature Class Separability Analysis

To complement the model-driven rankings from Random Forest above, this subsection presents a model-independent approach that assesses each feature's ability to separate bean classes based solely on statistical separation. To assess each feature's ability to differentiate the bean classes, the Confusion Susceptibility Score (CSS) was computed for each of the 16 features.[5] The CSS reflects how much a feature's values vary between bean types relative to how much they vary within each bean type. A higher CSS indicates that a feature produces greater distinction between classes, making it more valuable for classification tasks.

Figure 5.4, on the following page, presents the results. Features with higher scores indicate a stronger utility for distinguishing bean types. The top six highest scored features are all direct physical measurements of the bean, derived from the shape and size of the bean.

[5] A feature's CSS is calculated by summing the squared differences between its mean values across all 21 unique class pairs, then dividing by the sum of the variance of that feature for each bean type (i.e., seven variances).

Figure 5.4 How Well Each Feature Separates Bean Types

Feature	CSS
Equivalent Diameter	124.0
Minor Axis Length	121.9
Area	114.3
Convex Area	113.2
Perimeter	106.9
Major Axis Length	86.4
Shape Factor 1	70.6
Shape Factor 2	49.5
Aspect Ratio	29.1
Compactness	29.0
Shape Factor 3	28.5
Eccentricity	24.8
Roundness	17.9
Shape Factor 4	3.6
Solidity	1.8
Extent	1.5

The 10 lowest scored features are all derived metrics, calculated from each bean's physical measurements. These are designed to quantify shape, symmetry, or distinctiveness. Features such as Extent (1.5), Solidity (1.8), and Shape Factor 4 (3.6) show minimal separation capability. These low scores reflect not just similarity across bean types, but near-identical distributions, suggesting these features may add little value to classifying the beans, regardless of a machine learning model's algorithm.

5.5. Feature Correlation and Redundancy Analysis

To assess potential redundancy among the 16 features, Pearson correlation coefficients were calculated for all feature pairs.[6] Feature pairs with strong positive or negative correlation tend to introduce redundancy, which means multiple features convey essentially the same information. While this redundancy does not always degrade model performance, it can affect both accuracy and runtime, depending on the algorithm:

- Tree-based models like Decision Tree, Random Forest, Extra Trees, Gradient Boosting, AdaBoost, and XGBoost typically handle multicollinearity better, selecting the most useful feature for each split, but it can cause a model to favor one feature over another arbitrarily. This might reduce interpretability without significantly affecting accuracy.

[6] The Pearson correlation coefficient measures the linear relationship between two variables. It ranges from –1 (perfect negative correlation) to +1 (perfect positive correlation), with 0 indicating no linear correlation. It is calculated by dividing the covariance of the two variables by the product of their standard deviations, effectively standardizing the measure of their joint variability.

- Linear and distance-based models, including Logistic Regression, Ridge, SGD, Perceptron, K-Nearest Neighbors, LDA, and QDA, are more sensitive. For linear models, high multicollinearity makes it hard to determine the individual contribution of each feature. For distance-based (K-Nearest Neighbors), it means noisy or redundant dimensions can disproportionately influence distance calculations. For LDA and QDA, it can lead to computational issues or errors.
- Neural network models, such as the MLP, may take longer to train with redundant features, as the optimizer needs to adjust more weights without gaining new information.

Figure 5.5 lists feature pairs with Pearson correlation coefficients exceeding 0.90, suggesting near-linear dependence. These relationships often signal redundancy,[7] where two features convey similar structural information. These overlaps can make it harder for models to explain what matters most, because the two features provide largely the same predictive information.

Figure 5.5 Highly Correlated Feature Pairs

Feature 1	Feature 2	Pearson Correlation Coefficient
Aspect ratio	Compactness	0.99
Aspect ratio	Eccentricity	0.92
Convex area	Area	1.00
Eccentricity	Compactness	0.97
Eccentricity	Convex area	0.99
Equivalent diameter	Area	0.98
Major axis length	Area	0.93
Major axis length	Convex area	0.93
Major axis length	Equivalent diameter	0.96
Minor axis length	Area	0.95
Minor axis length	Convex area	0.95
Minor axis length	Equivalent diameter	0.95
Minor axis length	Perimeter	0.91
Minor axis length	Shape Factor 1	0.95
Perimeter	Area	0.97
Perimeter	Convex area	0.97
Perimeter	Equivalent diameter	0.99
Perimeter	Major axis length	0.98
Shape Factor 3	Compactness	1.00
Shape Factor 3	Eccentricity	0.98
Shape Factor 3	Aspect ratio	0.98

[7] Signal is useful information in the data that helps a model make accurate predictions. It is distinct from noise, which adds random or irrelevant variation.

Figure 5.6 lists feature pairs with Pearson correlation coefficients less than –0.90, indicating strong inverse relationships. Although these features move in opposite directions, they still encode overlapping signal, making their inclusion in models redundant. The inverse pattern does not reflect independence. It simply mirrors the same underlying information from a different angle.

Figure 5.6 Inversely Correlated Feature Pairs

Feature 1	Feature 2	Pearson Correlation Coefficient
Aspect ratio	Shape Factor 3	-0.98
Compactness	Aspect ratio	-0.99
Compactness	Eccentricity	-0.97
Eccentricity	Shape Factor 3	-0.98
Minor axis length	Shape Factor 1	-0.95

These feature pairs, whether strongly positively or strongly negatively correlated, are statistically dependent. They tend to capture the same underlying useful information, just in the same or opposite direction. Including both in a model may introduce:

- Redundant structure, meaning the model encounters the same underlying pattern multiple times, adding inputs without contributing new predictive insight
- Inflated feature sets, meaning the model receives more features than it needs, which can complicate optimization by increasing dimensionality and blurring the importance of truly unique features
- Slower training or convergence
- Reduced clarity in feature importance.

Figure 5.7, on the following page, presents feature pairs with Pearson correlation coefficients close to zero, indicating minimal statistical association. Unlike the strongly correlated pairs shown earlier, these combinations reflect true independence. Each feature varies without being constrained by the other. This lack of overlap makes them especially useful for machine learning because they offer distinct values that can help models learn cleaner patterns, build more robust decision boundaries, and improve accuracy.

To visually reinforce these impacts, three scatter plots were prepared, one each to illustrate one feature pair from each of the three figures (tables) above. The first two plots show feature pairs with strong linear relationships (positive and negative), confirming redundancy. The third is of a feature pair with statistical independence and greater class separation.

Figure 5.7 Uncorrelated Feature Pairs

Feature 1	Feature 2	Pearson Correlation Coefficient
Aspect ratio	Minor axis length	-0.01
Compactness	Shape Factor 1	-0.01
Minor axis length	Compactness	-0.02
Shape Factor 1	Shape Factor 3	-0.01
Shape Factor 3	Minor axis length	-0.02

Figure 5.8 shows a strong positive linear relationship, with feature values tightly clustered along an upward diagonal. This confirms that both features convey nearly identical information, adding redundancy without improving prediction.

Figure 5.8 Scatter Plot of Highly Correlated Features (Convex Area vs. Area)

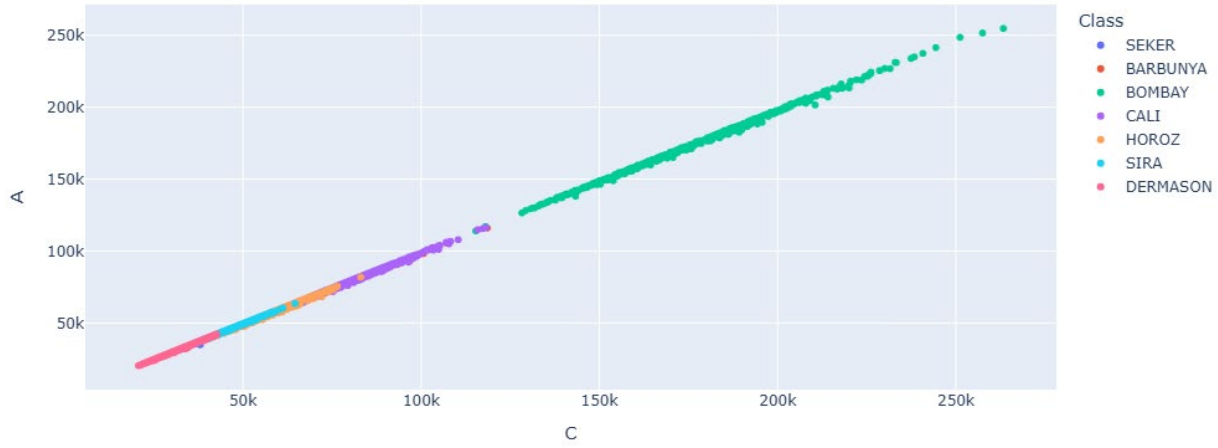


Figure 5.9, on the following page, shows a downward-sloping diagonal that reflects a strong inverse or negative relationship. Although the features move in opposite directions, their close linkage indicates overlapping signal (i.e., useful information), each offering limited added value over the other.

Figure 5.9 Scatter Plot of Inversely Correlated Features (Compactness vs. Aspect Ratio)

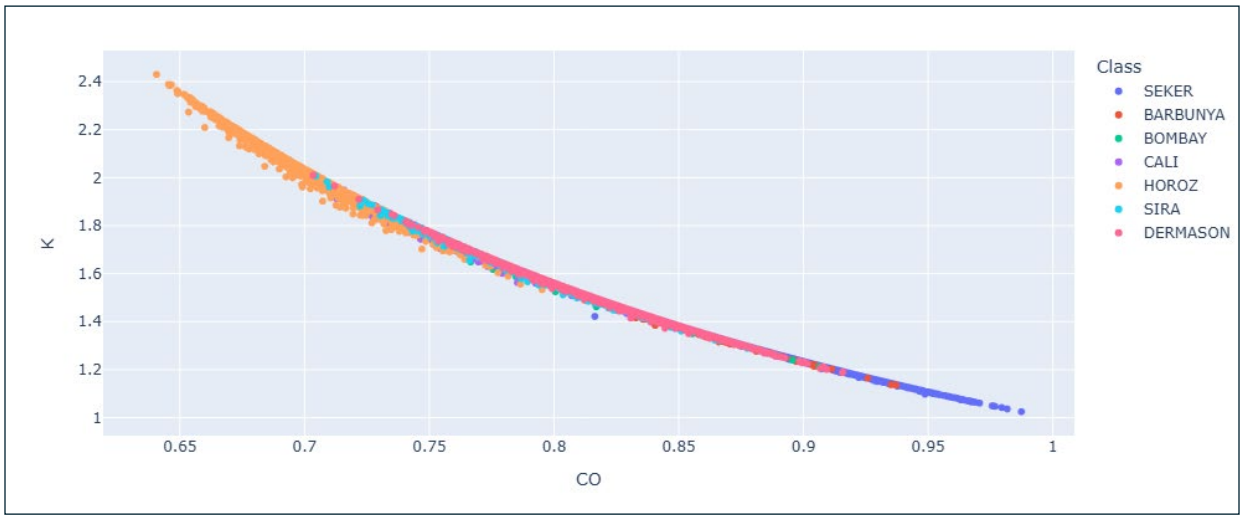
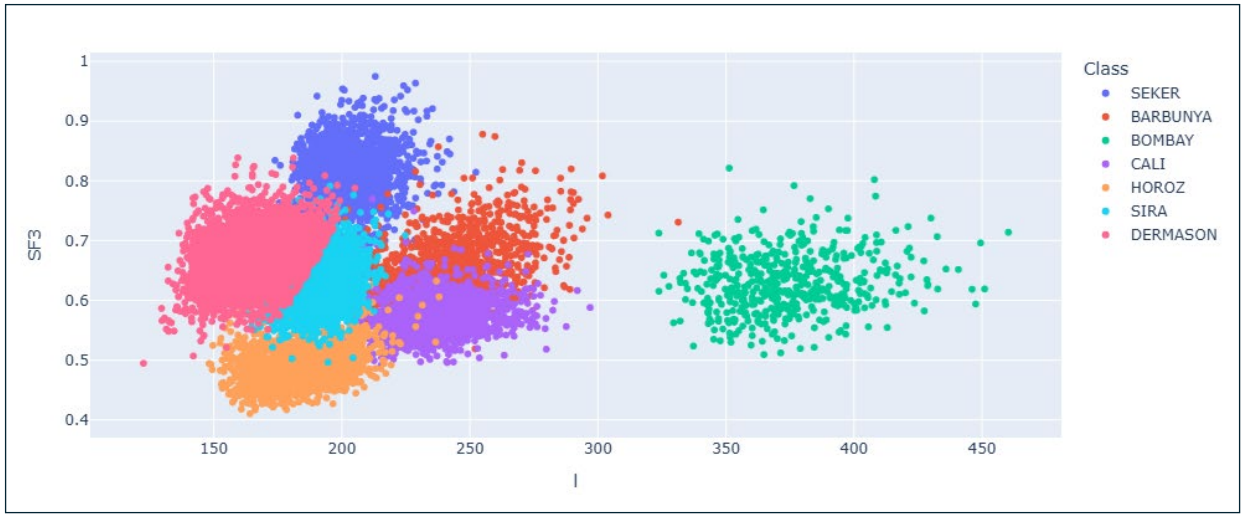


Figure 5.10 displays a clear diffusion of data points with no strong linear trend. The statistical independence between the two features allows each to contribute unique information, helping separate bean varieties and improving model performance.

Figure 5.10 Scatter Plot of Uncorrelated Features (Minor Axis Length vs. Shape Factor 3)



The review of the 16 features provided some insight into scaling needs, feature consistencies, and the relative capacity of features to distinguish bean types. The analysis helped clarify some structural properties of the dataset used for the project's main objective: evaluating and comparing model accuracy and runtime on a supervised classification task.

6. Benchmarking Framework

6.1. Evaluation Measures

Each of the 16 machine learning models was evaluated based on how accurately it predicted the bean variety of unseen dry bean records and how long it took to complete those predictions. The two primary performance metrics used in this benchmark were:

- **Accuracy**: The proportion of all test records that the trained model correctly classified, calculated both overall and for each of the seven bean varieties. F1 scores were also calculated to account for class-level precision and recall.
- **Runtime**: The total time required for each model's cross-validation, training, and prediction steps, excluding any preprocessing, data loading, or result-saving operations. This benchmarking effort is not focused on the absolute runtimes for individual models, but rather on relative performance: how well each model performed compared to the others, under identical conditions. To support a more accurate runtime evaluation, all runtime results were scaled to a common 0-1 scale before comparison.

6.2 Accuracy Measurement Methodology

Model accuracy was measured using standard, well-established practices in machine learning. By using a stratified 5-fold cross-validation and constructing confusion matrices to measure overall and per-class accuracy, the methodology provided a balanced, reproducible, and reliable assessment of each model's accuracy. Appendix B to this report provides a visual of this stratified 5-fold cross-validation.

The following are steps and calculations used to determine both overall and per-class accuracy for a single model using all 16 features.

1. **Stratified Cross-Validation Setup**

   The project used a stratified 5-fold cross-validation, which ensured that the class distribution in each fold mirrored that of the full dataset. This was important to reduce or avoid biased accuracy estimates, particularly for the imbalanced Dry Bean dataset. Steps included:

   - The dataset was split into five folds using StratifiedKFold(n_splits=5, shuffle=True, random_state=42) from scikit-learn.
   - In each fold, 80% of the data were used for training and 20% for testing.
   - The process was repeated five times so that every record served once as test data and four times as training data.
   - A fresh instance of the selected model was trained and evaluated in each fold.

2. **Aggregating Predictions Across Folds**

   During cross-validation, predictions were generated for all test folds and aggregated to calculate the model's overall classification performance:

   - True labels (y_test) and predicted labels (y_pred) from each fold were collected into two lists: y_true_all and y_pred_all.

- This ensured a one-to-one correspondence between all true and predicted labels across the entire dataset.

3. Confusion Matrix and Per-Class Accuracy Calculations

After completing cross-validation, a confusion matrix was built to summarize how well the model predicted each bean variety.

- The confusion matrix shows the counts of correct and incorrect predictions for every actual class (bean type) versus every predicted class.
- Using this matrix, per-class accuracy was computed for each bean type as:

$$Per\ Class\ Accuracy = \frac{Number\ of\ Correctly\ Predicted\ Samples}{Total\ Actual\ Samples\ of\ that\ Class}$$

4. Overall Accuracy Calculation

The overall model accuracy was calculated by comparing all predicted and true classes across the entire dataset:

$$Overall\ Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

This value represents the proportion of all samples that were correctly classified, regardless of class.

5. Identification of Outliers

The analysis identified clear accuracy outliers. These included models that underperformed compared to others, as well as specific bean classes that proved difficult to classify. The outliers helped identify some model limitations and determine beans that were easy to distinguish or more difficult to distinguish based on their features.

6.3 Runtime Measurement Methodology

To fairly compare model runtimes, a structured benchmarking routine was built to test each model under consistent and realistic conditions. The setup was designed to capture how long each model takes to run, not just once, but across a wide range of combinations of the dry bean dataset features (e.g., perimeter, area, aspect ratio), while smoothing out any timing noise from Windows background processes or system resource fluctuations.

The benchmarking module used a three-level loop to generate a sufficient number of dataset records for each model to train and test on, and to help stabilize runtime measurements:

1. **Feature Combinations**

   The outer loop iterated through all possible feature subsets containing:

   - **14 features**: 120 combinations (e.g., A, P, L, l, K, Ec, C, Ed, Ex, S, R, CO, SF1, SF2)
   - **15 features**: 16 combinations
   - **16 features**: the full feature set.

   This resulted in 137 unique feature sets generated per dataset record, per model.

2. **Repetition Runs**

   For each of the 137 feature sets, the middle loop executed three repeated trials (controlled by the variable num_runs, set to 3). This repetition helped smooth out runtime variability caused by background processes in Windows (e.g., system services, RAM management).

3. **Cross-Validation**

   Within each run above, the innermost loop performed stratified 5-fold cross-validation, using StratifiedKFold, ensuring that every record in the dataset is used for testing exactly once. Each run:

   - Split the full dataset of 13,611 samples into 5 folds
   - Used one fold for testing (~2,722 samples) and the remaining four for training (~10,889 samples)
   - Trained and tested the model five times per run, once for each fold.

Each timed run captured a model's total time required to:

- Train the model on four folds
- Generate predictions on the fifth fold
- Repeat this process across all five folds

This means a single runtime measurement reflects the time to process approximately 68,055 records (13,611 samples × 5 folds). Because this process was repeated three times for each of the 137 feature combinations, the benchmarking framework recorded 411 individual runtime measurements per model (each one for approximately 68,055 records).

Each of these 411 measurements represents the average time to train and test across five folds for a specific feature subset. The final runtime reported for each model is the average of these 411 measurements, providing a stable and representative estimate of performance.

Across all loops, the benchmarking process results in:

137 feature sets
× 13,611 records
× 5 cross-validation folds
× 3 repetitions.

This yields a total of 27,970,605 record appearances per model, ensuring that runtime estimates were based on extensive and uniform exposure to dataset records. Rather than relying on single runs or isolated feature sets, the benchmarking framework attempted to reduce runtime inconsistencies caused by:

- **System-level noise**, such as Windows background activity and RAM fluctuations
- **Feature-set volatility**, where different subsets may trigger varying computational loads
- **Train-test sensitivity**, where runtime can vary across individual fold splits
- **Class imbalance**, ensuring all bean varieties are consistently represented
- **Record sampling bias**, so no data point unfairly influences the timing.

To enable fair and interpretable comparisons of model efficiency, all runtime measurements were inverted and then normalized prior to analysis. The goal was to highlight relative performance rather than absolute wall-clock time, ensuring that faster models received higher (i.e., better) normalized scores on a consistent 0–1 scale. This transformation allowed runtime to be analyzed alongside accuracy in a balanced way.

An inverse-scaling and min–max normalization method was used, applied sequentially to each runtime value (in milliseconds) as follows:

1. **Invert Raw Runtime Values**
    - Each runtime (in milliseconds) was transformed by taking its reciprocal (1 / runtime) to ensure that lower runtimes correspond to higher (meaning better) scaled values.
2. **Apply Min-Max Normalization**
    - The inverted runtime values were rescaled to a common range between 0 and 1 using the min-max formula: Assuming x = inverted runtime, scaled x' was calculated as follows:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

    - This ensured the scale was consistent while preserving relative differences among the models.
3. **Record Normalized Scores**
    - The resulting normalized score for each of the 137 feature sets were recorded for each model. These are the basis for: (1) Comparing runtime performance across all models, (2) Computing the average normalized runtime score for each model, and (3) Visualizing model tradeoffs between speed and accuracy.

6.4 Model Hyperparameter Tuning

Each of the 16 machine learning models had its own set of configuration parameters, known as hyperparameters. These settings functioned as dials, controlling how the model learns, how complex it is, and how it makes predictions. To fine-tune performance, each model had a "grid" defined of up to four hyperparameters to tune, and a small choice of values for each to test.

To systematically tune these hyperparameters, a grid search with cross-validation was performed on 15 of the 16 models using scikit-learn's GridSearchCV. All models were tuned to maximize classification accuracy using cross-validation. Because runtime was not included in the optimization objective, changes in a model's runtime after tuning reflect incidental consequences of the selected hyperparameters.

Each model was set up with a small, tailored "param_grid" of important hyperparameters, exploring two to four reasonable values for each to find the best combination. GridSearchCV then trained and evaluated each model on the bean dataset, using every possible combination of the values given for every hyperparameter.

For example, a model with two hyperparameters to examine, with a choice of three values each, would have 9 possible combinations (3 x 3). For each combination, GridSearchCV ran a cross-validation (with 5-fold StratifiedKFold), collected accuracy scores, and selected the tuned values of the combination achieved the best result.

The one exception to this tuning was GaussianNB, a probabilistic model with no tunable hyperparameters in typical use. It uses simple statistical assumptions (e.g., features being normally distributed and independent), and its default settings typically produce stable results without much benefit from adjustment.

Below is the partial Python code of a dictionary for setting up RandomForestClassifier for hyperparameter tuning:

```
    "RandomForestClassifier": {
        "class": "RandomForestClassifier",
        "module": "sklearn.ensemble",
        "requires_numeric_labels": False,
        "search_type": "grid",
        "scoring": "accuracy",
        "default_params": {"random_state": 42, "n_jobs": -1},
        "param_grid": {
            "n_estimators": [50, 100],
            "max_depth": [None, 10, 20],
            "min_samples_split": [2, 5],
            "min_samples_leaf": [1, 3]
        }
    },
```

The Python dictionary entry above specifies both fixed and tunable hyperparameters setup for RandomForestClassifier. Two hyperparameters are preset before training to ensure consistent behavior: random_state for reproducibility and n_jobs=-1 to enable parallel execution across all CPU cores. The remaining four hyperparameters were: n_estimators, max_depth,

min_samples_split, and min_samples_leaf. Each was set up for tuning using grid search, each with a predefined set of candidate values to explore to optimize the model's accuracy. While these four hyperparameters were tuned primarily to improve accuracy, they also influenced runtime by controlling model complexity. For example, more trees ("n-estimators"), deeper trees ("max_depth"), and more splits ("min_samples_split) can each increase training and prediction time.

Exhibit 6.1, on the following page, shows the hyperparameters tested, the candidate values provided, and the resulting tuned value. The candidate values shown include the model's default value for each hyperparameter. Meaning that after the GridSearchCV routine, any one of a model's tuned hyperparameters could remain unchanged from its default value.

Figure 6.1 identifies which models support the reproducibility hyperparameter and/or the multi-threading hyperparameter. For each supported feature, the figure lists the specific value applied for benchmarking.

Figure 6.1 Additional Hyperparameters Set for Model Benchmarking

Support random_state (set to 42)	Support n_jobs (set to -1)
DecisionTreeClassifier	RandomForestClassifier
RandomForestClassifier	ExtraTreesClassifier
ExtraTreesClassifier	XGBClassifier
GradientBoostingClassifier	LogisticRegression
AdaBoostClassifier	SGDClassifier
XGBClassifier	Perceptron
LogisticRegression	
RidgeClassifier	
SGDClassifier	
Perceptron	
SVC	
MLPClassifier	

Exhibit 6.1 Hyperparameters Tuned for Each Model

Model	Hyperparameters and Choices Tested	Tuned Value
1. DecisionTreeClassifier	criterion (choices: gini, entropy) max_depth (choices: None, 5, 10, 20) min_samples_leaf (choices: 1, 3) min_samples_split (choices: 2, 5)	gini 10 1 5
2. RandomForestClassifier	max_depth (choices: None, 10, 20) min_samples_leaf (choices: 1, 3) min_samples_split (choices: 2, 5) n_estimators (choices: 50, 100)	20 1 2 50
3. ExtraTreesClassifier	max_depth (choices: None, 10, 20) min_samples_split (choices: 2, 5) n_estimators (choices: 50, 100)	None 5 100
4. GradientBoostingClassifier	learning_rate (choices: 0.05, 0.1) max_depth (choices: 3, 5) n_estimators (choices: 50, 100)	0.1 3 100
5. AdaBoostClassifier	learning_rate (choices: 0.5, 1.0) n_estimators (choices: 50, 100)	1 100
6. XGBClassifier	learning_rate (choices: 0.1, 0.2) max_depth (choices: 3, 6) n_estimators (choices: 50, 100) subsample (choices: 0.8, 1.0)	0.2 6 100 1
7. LogisticRegression	C (choices: 0.1, 1, 10) penalty (choices: l2) solver (choices: lbfgs, liblinear)	10 l2 lbfgs
8. RidgeClassifier	alpha (choices: 0.1, 1.0, 10.0) solver (choices: auto, sparse_cg)	1 Auto
9. SGDClassifier	alpha (choices: 0.0001, 0.001) loss (choices: hinge, log_loss) penalty (choices: l2, l1)	0.0001 hinge l1
10. Perceptron	alpha (choices: 0.0001, 0.001) penalty (choices: l2, elasticnet, None)	0.0001 None
11. SVC	C (choices: 0.1, 1, 10) gamma (choices: scale, auto) kernel (choices: linear, rbf)	10 scale rbf
12. KNeighborsClassifier	metric (choices: euclidean, manhattan) n_neighbors (choices: 3, 5, 7) weights (choices: uniform, distance)	euclidean 7 Distance
13. GaussianNB	no hyperparameters tuned	sklearn defaults
14. LinearDiscriminantAnalysis	shrinkage (choices: None, auto) solver (choices: svd, lsqr)	None Svd
15. QuadraticDiscriminantAnalysis	reg_param (choices: 0.0, 0.1, 0.5)	0.1
16. MLPClassifier	activation (choices: relu, tanh) alpha (choices: 0.0001, 0.001, 0.01) hidden_layer_sizes (choices: (50,), (100,), (50, 50)) solver (choices: adam, lbfgs)	relu 0.01 (50,) adam

7. Results and Analysis

7.1. Accuracy Results

Below is the average accuracy recorded for each model.[8]

MLP	93.38%	Quadratic Discriminant Analysis	91.94%
SVC	93.14%	SGD	91.90%
XGBoost	93.02%	Decision Tree	90.81%
Gradient Boosting	92.69%	Linear Discriminant Analysis	90.46%
Random Forest	92.56%	Ridge	90.26%
K-Nearest Neighbors	92.50%	Perceptron	90.21%
Logistic Regression	92.43%	Gaussian NB	89.67%
Extra Trees	92.29%	AdaBoost	84.41%

The top ten most accurate models range from 91.90% to 93.38% accuracy. The wide variety of machine learning algorithms produced fairly high accuracies labeling bean types when tuned and evaluated. Their close accuracies suggest that many models, including MLP, SVC, XGBoost, Gradient Boosting, and Extra Trees, can capture complex feature interactions through deep optimization or ensemble strategies. Meanwhile, models such as K-Nearest Neighbors, Logistic Regression, Quadratic Discriminant Analysis, and SGD, reach similarly strong results using more streamlined or interpretable approaches.

The close results show that for at least this dry bean classification task, the algorithm matters, but once baseline accuracy is secured, secondary concerns like runtime, scalability, and interpretability may carry more weight in determining the best model for deployment. For example, the neural network model MLP performs well thanks to its complexity. However, understanding how it makes decisions is not straightforward. Random Forest and Logistic Regression, while slightly less accurate, deliver insights like how feature values drive the overall predictions, and what features are more important.

AdaBoost, which had a very low accuracy, is traditionally built for binary classification and often applies a one-vs-all strategy in multiclass settings. For example, it creates a single binary classifier for bean Sira to distinguish Sira versus the other six beans, and a binary classifier for Dermason vs. all other six beans, and so on. It uses this “one-vs-all” strategy, which can dilute its effectiveness across several bean varieties. Compared to XGBoost or Gradient Boosting, AdaBoost tends to build fewer, simpler learners, which may make it less competitive for nuanced multiclass problems (like the seven bean types).

Most models showed a similar pattern in how well they predicted each bean type. **Figure 7.1**, on the following page, displays the ranking of prediction accuracy for each bean by each model, where a rank of 1 indicates the model’s best performance and 7 its worst. Bombay consistently ranked highest, with all but one model identifying it as the easiest bean to classify. In contrast, Sira was the most challenging, receiving the lowest accuracy ranking from the majority of

[8] The F1 macro weighted score for each model is identical to, or less than 0.2 percentage points different from, the accuracy score of 13 models, and less than 0.60% different for the other three.

models. This consistency across models highlights how feature separability strongly influences classification success.

Figure 7.1 Each Model's Ranking of Its Bean Prediction Accuracy (1=Best, 7=Worst)

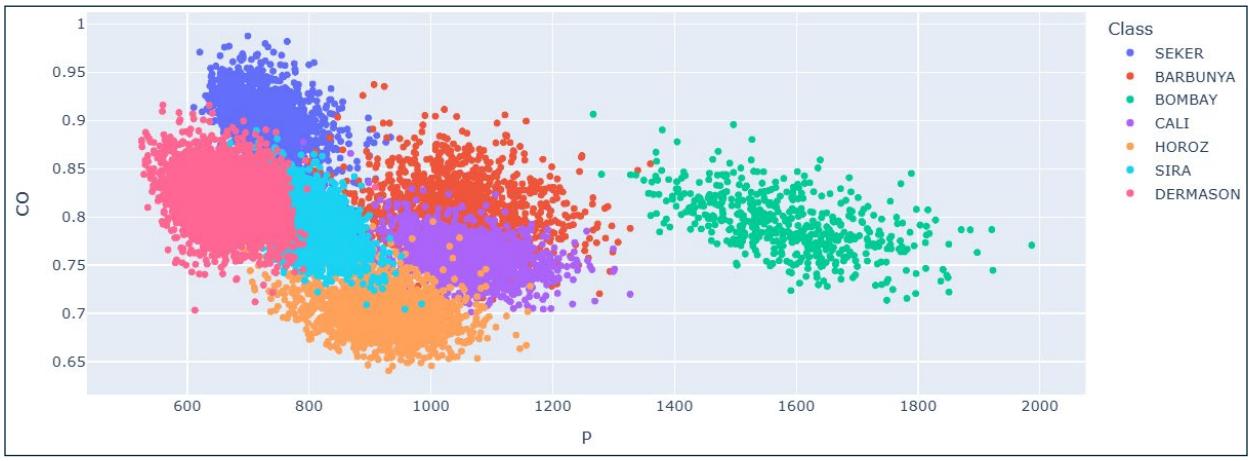
Model	Barbunya	Bombay	Cali	Dermason	Horoz	Seker	Sira
1. DecisionTree	6	1	4	5	3	2	7
2. Random Forest	6	1	4	5	2	3	7
3. Extra Trees	6	1	4	5	2	3	7
4. Gradient Boosting	6	1	2	5	3	4	7
5. AdaBoost	6	7	4	1	2	3	5
6. XGBoost	6	1	3	5	2	4	7
7. Logistic Regression	6	1	4	5	2	3	7
8. Ridge	7	1	2	6	4	3	5
9. SGD	6	1	4	5	2	3	7
10. Perceptron	6	1	3	4	5	2	7
11. K-Nearest Neighbors	6	1	3	5	2	4	7
12. SVC	6	1	4	5	3	2	7
13. GaussianNB	7	1	4	5	2	3	6
14. LDA	7	1	2	6	4	5	3
15. QDA	6	1	3	5	2	4	7
16. MLP	6	1	2	5	4	3	7

Bombay was the easiest bean for models to identify. Excluding AdaBoost, the remaining 15 models correctly predicted Bombay beans 99.64% of the time, a near-perfect result. AdaBoost was a clear outlier, achieving just 23.56% accuracy on this bean.

The reason for the consistently strong performances classifying Bombay is straightforward: Bombay beans are noticeably larger than the others, making them stand out in the feature space. **Figure 7.2**, on the following page, illustrates this separation using two key features, Perimeter and Compactness (a measure of roundness). These features are both predictive and only weakly correlated, helping models make predictions, and helping this chart spread the beans apart visually.

In this scatter plot, Bombay beans form a distinct cluster, clearly separated from the other varieties. This visual and statistical separation makes it easy for most models to distinguish Bombay from the rest.

Figure 7.2 Separation of Bombay Beans from Other Classes Based on Perimeter and Compactness



The Bombay bean highlights a broader principle in machine learning: when a class (here, bean type) is well-separated in feature space, models are far more likely to classify it accurately. Differences in features such as size, shape, texture (images), temporal patterns, color, or other distinct traits can help with accuracy. In any dataset, when one class occupies a unique region in the feature space and key features are both informative and relatively uncorrelated with each other, models can learn boundaries with minimal confusion. The Bombay bean's consistent identification across nearly all models was not due to any algorithmic preference, but rather to its clear distinction from the other beans.

The sole exception to labeling Bombay correctly was AdaBoost, which achieved only 23.6% accuracy, far below the 99.6% average of all other 15 models. Of the 399 Bombay samples it misclassified, 398 were mislabeled specifically as Cali, making this the most pronounced and systematic misclassification of any bean variety by any model in the study.

AdaBoost's poor performance on the Bombay bean, which the other five tree-based models classified with near-perfect accuracy, can be traced to several unique aspects of AdaBoost's algorithm:

- **Very shallow base learners and no randomness**: AdaBoost typically uses decision stumps (depth-1 trees) and avoids randomness. In contrast, models like Random Forest, Extra Trees, and XGBoost benefit from using deeper trees and randomized subsets. Randomness helped uncover nuanced patterns in a minority class like Bombay that the deterministic AdaBoost may have missed.
- **Error amplification through iterative reweighting**: At each round, AdaBoost increases the weight of previously misclassified samples. For small or minority classes like Bombay (just 3.84% of the bean samples in the dataset), early and consistent misclassifications may be reinforced rather than corrected. This may have skewed the entire ensemble toward persistent misclassification of this minority bean.
- **Native multiclass support, but unreliable boundary handling:** AdaBoost includes native multiclass capabilities but failed to reliably identify Bombay beans. This suggests limitations in how AdaBoost detects clear-cut boundaries between classes,

potentially because its building blocks, such as weak learners, were not deep or well-aligned enough to capture the patterns specific to Bombay beans.

AdaBoost's sequential, error-driven approach appears to have reinforced early misclassifications of Bombay as Cali, compounding this confusion across iterations. In contrast, the other tree-based models, especially those using bagging (Random Forest, Extra Trees) or gradient-based correction (Gradient Boosting, XGBoost), achieved significantly better accuracy through randomized sampling, deeper trees, and native multiclass modeling.

While Bombay beans were consistently identified correctly by 15 of the 16 models, Sira beans presented a different challenge. Unlike Bombay's clear separation in feature space, Sira often overlapped with other varieties, particularly Dermason, making accurate classification more difficult for every model.

Sira was the most frequently misclassified bean variety among the 16 machine learning models evaluated in this project. On average, Sira was correctly identified 86.7% of the time, meaning 13.3% of Sira beans were mislabeled. Among these misclassifications, 72.7% were those that incorrectly labeled Sira as Dermason. Similarly, when models misclassified a Dermason bean, approximately 78% were those that incorrectly labeled Dermason as Sira. This indicates a consistent and model-agnostic confusion between these two beans.

To identify which feature pairs contributed most to the confusion between Sira and Dermason beans, a centroid-distance analysis was performed. This approach gave rise to a new diagnostic metric, the Confusion Susceptibility Index (CSI). The CSI ranks feature pairs by their tendency to induce misclassification based on proximity and spread. To the best of this report's author's knowledge, CSI is not a previously formalized measure in machine learning literature. It did prove effective in surfacing structural ambiguity between these two beans.

The analysis examined all possible feature pair combinations for Sira records (2,936) and Dermason records (3,546). The following steps were completed for each possible pair combination for all 16 features:

- Compute centroids: For each bean, calculate the average value of the two features being examined. This gave the 2D point for each bean (the centroid) in the space defined by the two features being examined.
- Calculate distance: Measure the Euclidean distance between the centroids. A smaller distance indicates greater overlap and a higher likelihood of misclassification.
- Calculate spread: Compute the standard deviation for each feature and each bean, resulting in four standard deviation measures (two beans x two features).
- Rank feature pairs: Sum the four standard deviations to determine a combined spread (or dispersion) across both beans and both features, match with the distance between the pair's two centroids, and identify the most confusion-prone feature pairs.

The feature pair Solidity and Shape Factor 2 was ranked highest in Sira–Dermason confusion due to their near-identical class centroids and their substantial combined dispersion introduced by Shape Factor 2. While Solidity values are nearly constant across all beans (CV < 0.5%), Shape Factor 2 has the second largest variability of all 16 features (CV ≈ 34.7%). Specifically, the sum

of the four standard deviations (the two features and for the two beans) was 12 times larger than the distance between their centroids.

This asymmetry placed the pair at the top of the confusion ranking.[9] This highlights how low centroid separation, when paired with even one highly dispersed feature, can produce significant class overlap and lead to misclassification.

Figure 7.3 is a scatter plot of Sira and Dermason beans based on Solidity and Shape Factor 2, the most confusion-prone feature pair identified. The plot visually confirms the statistical findings: the centroids of the two beans lie very close together, and the significant spread introduced by Shape Factor 2 contributed to significant feature-space overlap. Solidity values are tightly clustered across both classes, underscoring its minimal variability and reinforcing its limited role in class distinction.

Figure 7.3 Sira Classification Difficulty from Feature Overlap (Solidity and Shape Factor 2)

The scatter plot reveals another complication. A significant portion of Sira and Dermason beans occupy the same region, with many data points from one bean blocking seeing the other bean's dots. There are many Sira beans that cannot be seen on the scatter plat because they are "underneath" the Dermason beans. And vice versa. The overlap makes it impossible to visually distinguish between the two beans. As a result, the scatter plot not only illustrates the confusion but also conceals it.

7.2. Runtime Results

Average runtimes for each model were computed using 137 feature set variations created for each of the 13,611 dataset records. These variations include all combinations of 14 features out of 16 (120 sets), all combinations with 15 features (16 sets), and the original full feature set (1 set), totaling 137 unique feature configurations per model. The runtime for each configuration was captured independently, and the reported average runtime represents the mean time taken across all 137 runs for each model, as described in more detail earlier.

This project emphasizes the relative differences in training runtimes among classification models, rather than their absolute durations. The absolute runtimes can vary depending on hardware used and system load.

[9] Two other feature pairs ranked very high: Solidity – Shape Factor 1, and Shape Factor 1 – Shape Factor 2.

Below are the average runtimes calculated for each model. All runtimes are in milliseconds (ms) and reflect average training durations.

Gradient Boosting Classifier	26,160.03 ms	Decision Tree Classifier	559.55 ms
MLP Classifier	12,355.13	SGD Classifier	330.60
AdaBoost Classifier	7,932.67	K-Nearest Neighbors	266.42
SVC	7,728.77	Perceptron	95.56
XGBoost Classifier	2,546.49	Linear Discriminant Analysis	37.10
Logistic Regression	1,254.12	Quadratic Discriminant Analysis	34.05
Random Forest Classifier	820.77	Ridge Classifier	25.61
Extra Trees Classifier	594.10	GaussianNB	20.09

Gradient Boosting and MLP (the only neural network model examined) showed extremely long runtimes due to iterative training processes, whereas models like GaussianNB and Ridge completed evaluations quickly, often in just milliseconds.

Gradient Boosting builds trees sequentially, where each one learns from the mistakes of the previous stage, though some internal operations, such as split finding or data preparation, may be parallelized in optimized libraries. Because every tree depends on the outcome of the last, it cannot fully parallelize tree construction across CPU cores due to the sequential nature of boosting. It is inherently a step-by-step process which takes longer to complete its tasks.

In contrast, other decision tree models, like Random Forest, Extra Trees, and XGBoost can take advantage of multiple cores (n_jobs=-1 is the hyperparameter setting), allowing them to train multiple trees or process data across multiple threads simultaneously. This was a significant boost to efficiency, especially on machines with multiple cores (this project's Windows PC has 28 cores).

The four fastest models are GaussianNB, Ridge, QDA, and LDA. **Figure 7.4**, on the following page, provides some insight into why each of them may have got through training and testing so quickly.

They are fast because they rely on simple, closed-form calculations that avoid iterative optimization or tree construction (e.g., solutions that can be expressed as a fixed, finite formula). These models do not iterate or search for parameters using algorithms like gradient descent. Most of their work is done with simple linear algebra, which is very efficient on a PC's CPUs. And because they don't build deep model architectures or ensembles, they also are memory-efficient

Although comparing the four models is a bit misleading because there is a mix of model algorithms among the group, the performance gap clearly illustrates the impact of their analytical approach. All four models highlighted rely on analytical or closed-form solutions. There are no iterative solvers, gradient descent, or tree construction algorithms in the group.

Models with multithreading support completed tasks significantly faster. On average, the ten single-threaded models took nearly six times longer to run than the six models that can use multiple cores (Figure 6.1 identifies these six models).

Figure 7.4 Overview of the Models with the Fastest Runtimes

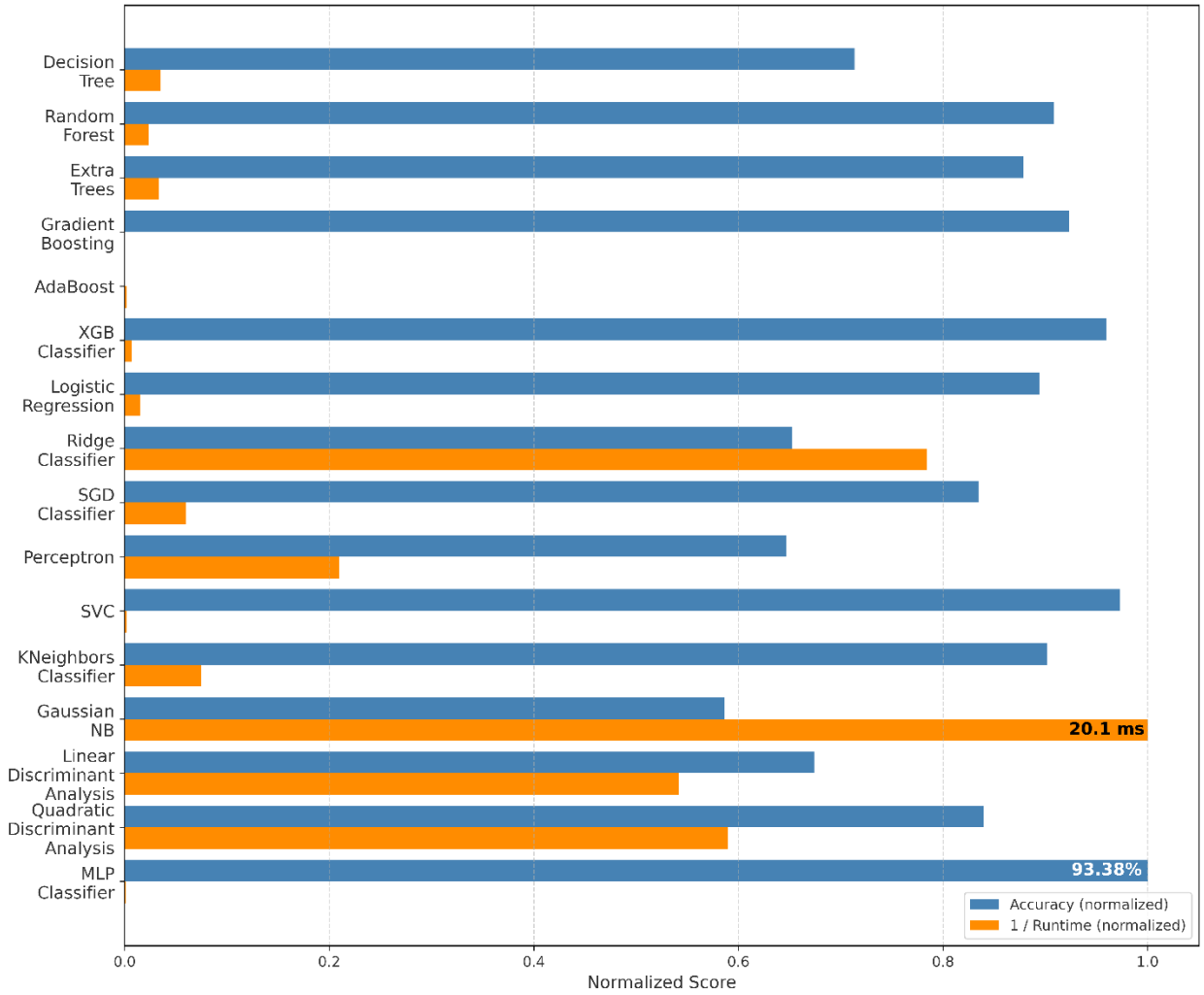
Model	Reason for a Fast Runtime
GaussianNB	Uses simple probabilistic formulas based on the assumption that each feature follows a normal (Gaussian) distribution and is conditionally independent of the others. Training consists only of calculating the mean and variance of each feature within each class. There is no iterative optimization, no distance computations, and no matrix inversion. This makes the training process extremely fast.
Ridge	Solves a regularized linear classification problem (L2-penalized least squares) using efficient matrix algebra. Many implementations use direct solvers or linear algebra libraries that exploit closed-form or highly optimized numerical methods. Because there is no iterative training (like gradient descent), the model converges almost instantly on modest-sized datasets.
Quadratic Discriminant Analysis	Extends LDA by allowing each class to have its own covariance matrix, increasing flexibility. Still avoids iterative training by relying on closed-form statistical computations: estimates of class priors, means, and covariances are computed directly from the training data. The extra work of estimating multiple covariance matrices adds minor overhead but keeps training fast overall.
Linear Discriminant Analysis	Computes class means and a shared covariance matrix to find linear boundaries that maximize between-class separation relative to within-class variation. The approach is entirely analytical, involving matrix inversion and projection, with no iterative fitting (“svd” was set as the solver in hyperparameter tuning). This simplicity leads to very fast training, especially on small (<10,000 samples) to medium-sized (~10,000–1 million samples) datasets, like the Dry Bean dataset.

7.3. Accuracy and Runtime Tradeoffs

Figure 7.5, on the following page, summarizes and compares the normalized accuracy and runtime performance of all 16 models evaluated. Blue bars represent accuracy, while orange bars show inverse runtime, so longer bars indicate better performance for both measures. This chart reinforces earlier findings, highlighting models like MLP for top accuracy and GaussianNB for exceptional speed.

The tradeoff between accuracy and runtime becomes especially apparent when comparing runtimes of the most precise models and least precise models. On average, the top five most accurate models took 67 times longer to execute than the five least accurate models (excluding AdaBoost).

Figure 7.5 Accuracy and Runtime Performance Across 16 ML Models (Longer is Better)

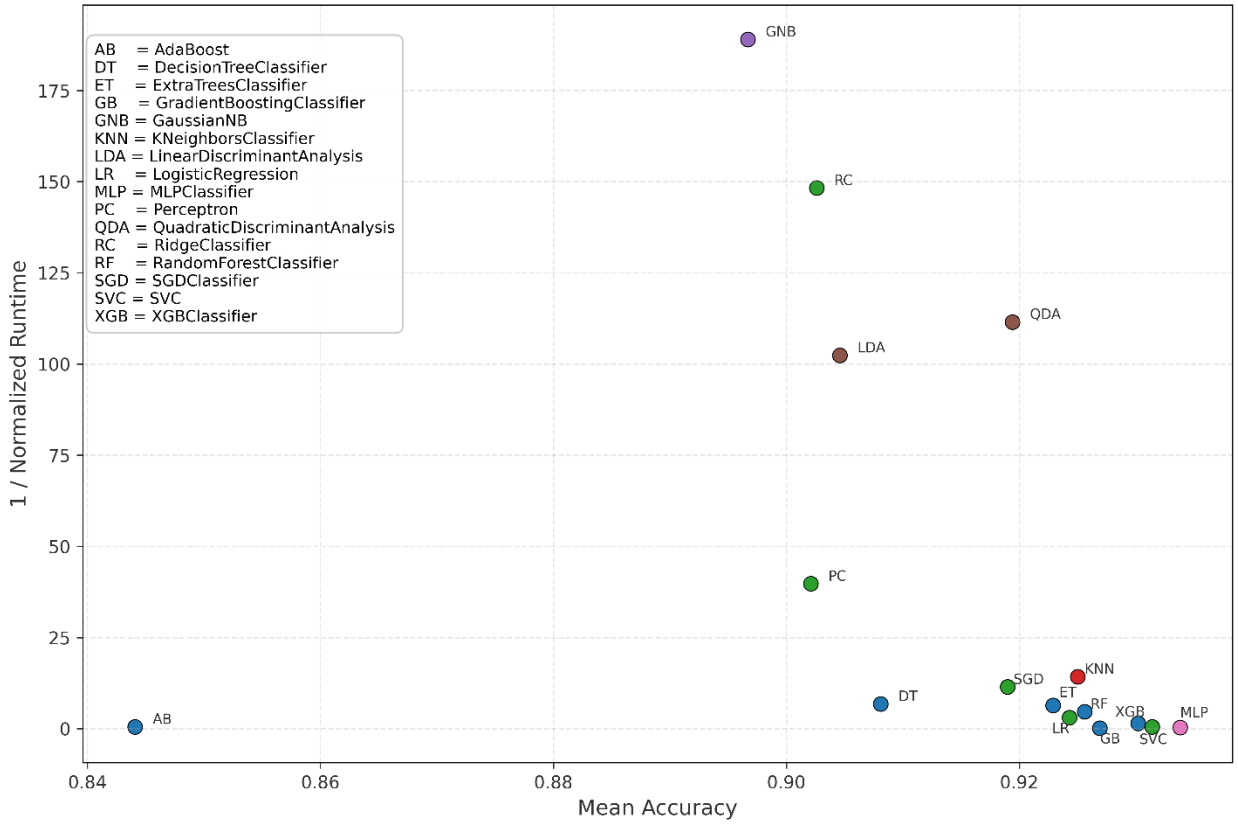


An alternative view of model performance is the accuracy vs. runtime tradeoff chart provided in **Figure 7.6**, on the following page. The chart plots each model by its average accuracy (x-axis) and inverse normalized runtime (y-axis), allowing for direct comparison of predictive strength versus computational cost.

There were trade-offs between accuracy and runtime. Models appearing in the upper right portion of the plot excel both in accuracy and in speed, representing the most favorable options for practical deployment. Conversely, models in the lower left require more computational resources and are less accurate.

Seven of the eight most accurate models, MLP, SVC, XGBoost, Gradient Boosting, Random Forest, Logistic Regression, and Extra Trees, are in the group of eight models with the longest runtimes (AdaBoost being the eighth). This suggests a natural tradeoff: higher accuracy among these models comes at the cost of increased computational time. Among all models, AdaBoost stands out for underperforming in both dimensions, scoring the lowest in accuracy (84.41%) and near the bottom in speed (0.4787 normalized, 7,932.67 ms actual).

Figure 7.6 Accuracy vs. Inverted Runtime – Models in the Upper Right Excel in Both

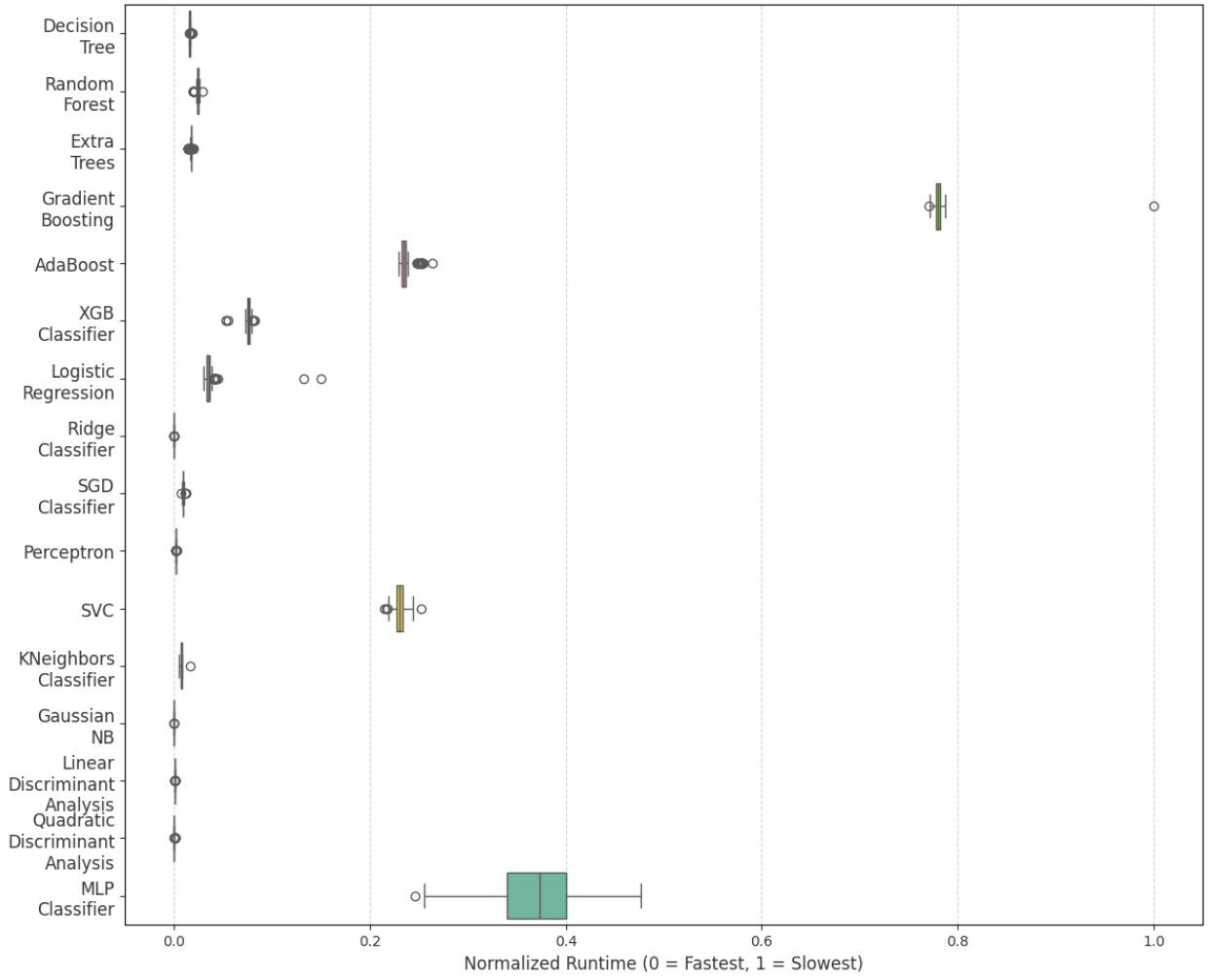


Just four models, QDA, LDA, Ridge, and GaussianNB, delivered a more favorable balance between classification accuracy and runtime efficiency. These models operate significantly faster than their more complex counterparts, while still achieving respectable accuracy scores (≥89.67%). These could be candidates for scenarios where speed is critical and top-tier precision is less essential.

7.4. Runtime Variability

Runtime consistency provides some insight into a model's reliability. To illustrate this, box plots were generated for each model's measured runtimes across 137 feature combinations. The runtime box plots are one means to display the distribution of each model's runtimes, by showing its minimum, first quartile (Q1), median (Q2), third quartile (Q3), and maximum values. It helps visually to spot variability, skewness, and outliers in the runtimes. **Figure 7.7**, on the following page, provides the normalized runtime box charts. While most models appear to show narrow distributions, indicating stable runtimes, a few exhibited noticeably broader spreads and some outliers.

Figure 7.7 Dispersion of Each Model's Normalized Runtimes Across 137 Feature Sets



The clearest outlier appears to be MLP, whose box plot span appears to be at least an order of magnitude wider than all the other models. This confirms earlier observations of its sensitivity to training conditions and convergence behavior.

However, box plots can be visually misleading. Because MLP's absolute runtimes are significantly longer than most other models, even a small variation is large on the normalized x-axis scale compared to the other models. If runtime dispersion is instead reframed proportionally to each model's actual runtimes, there is a somewhat different story.

To account for this distortion, runtime dispersion was reframed proportionally, using interquartile range (IQR) normalized to each model's typical runtime. The interquartile range (IQR) is measured as Q3 (the 75th percentile runtime) minus Q1 (the 25th percentile runtime). Dividing IQR by Q1 gives a measure of relative variability for a model. **Figure 7.8**, on the following page, presents these calculations for each model. The quartile column times are normalized, as are the box charts, because of the large differences in absolute runtimes.

Figure 7.8 Model Runtime Variability (Quartile Values are Normalized Runtimes)

Model	Mean Runtime (ms)	Q1	Q3	IQR	IQR As a Pct. of Q1
Gaussian NB	20.09	0.000012	0.000028	0.000015	125.00%
MLP	12,355.13	0.339888	0.400573	0.060685	17.85%
K-Nearest Neighbors	266.42	0.006817	0.007929	0.001112	16.31%
Ridge Classifier	25.61	0.000177	0.000196	0.000019	10.73%
SGD Classifier	330.60	0.008832	0.009728	0.000896	10.14%
Logistic Regression	1,254.12	0.033839	0.036431	0.002593	7.66%
QDA	34.05	0.000420	0.000437	0.000018	4.29%
Random Forest	820.77	0.023728	0.024727	0.000999	4.21%
LDA	37.10	0.000515	0.000533	0.000018	3.50%
SVC	7,728.77	0.227448	0.233968	0.006520	2.87%
XGBoost	2,546.49	0.074744	0.076766	0.002023	2.71%
AdaBoost	7,932.67	0.232972	0.237159	0.004188	1.80%
Decision Tree	559.55	0.015890	0.016154	0.000264	1.66%
Extra Trees	594.10	0.017291	0.017575	0.000284	1.64%
Perceptron	95.56	0.002263	0.002281	0.000018	0.80%
Gradient Boosting	26,160.03	0.778005	0.782080	0.004075	0.52%

GaussianNB, whose variability is barely perceptible in the box plots, had the highest relative variability in runtimes. Despite very short runtimes overall, its IQR exceeded 125% of Q1, meaning even minor runtime fluctuations appear large in percentage terms.

While MLP shows a wide runtime dispersion in absolute terms, a closer look at the relative variability shows a bit more balanced picture. Measuring IQR as a percentage of Q1 shows models like K-Nearest Neighbors (16.31%), Ridge (10.73%), and SGD (10.14%) have similar degrees of variability relative to their own runtime baselines. Also, MLP's dispersion is not as dramatically out of line with other optimization-sensitive models. It simply operates on a larger timescale (i.e., it is slower). The relative runtime variabilities are small for several models, but the IQR table makes it a bit easier to see the range of variances among the models, especially because actual runtimes span orders of magnitudes.

Exhibit 7.1, on the following page, presents an overview of each model's relative runtime, runtime variance, and key traits that may have influenced performance. The exhibit briefly describes structural and algorithmic factors that can influence a model's behavior: whether it is greedy tree splitting, parallel ensemble training, or use of direct mathematical formulas that avoid loops and parameter tuning, (i.e., closed-form statistical computations).

Exhibit 7.1 Model Runtime and Stability Traits

Model	Relative Runtime	Variance	Reason
1. Decision Tree	Moderate	Low	Greedy splitting requires no iterative fitting. Only one tree to grow and single threaded, leads to stable runtimes.
2. Random Forest	Moderate	Moderate	Trains multiple trees in parallel. Aggregation adds cost. Runtime varies based on tree depth/size.
3. Extra Trees	Moderate	Low	Similar to Random Forest but uses randomized tree split points. Randomized splits and parallel processing may help keep runtimes stable.
4. Gradient Boosting	Slowest	Smallest	Sequentially builds hundreds of weak learners, each correcting prior errors. Slow due to additive stages and lack of parallelism.
5. AdaBoost	Very Slow	Low	Trains weak learners sequentially. Fewer trees than Gradient Boosting but still incurs iterative cost.
6. XGBoost	Slow	Moderate	Optimizes tree building and regularization but still trains sequentially. Parallel splits improve speed.
7. Logistic Regression	Moderate	Moderate	Uses iterative optimization to adjust weights. Trains one binary model per class in multiclass tasks (e.g., 7 bean types), increasing runtime.
8. Ridge Classifier	Fast	High	Solves regularized least squares with fast solvers. Minor fluctuations in the fast runtimes can produce large proportional differences
9. SGD Classifier	Moderate	High	Iteratively updates model weights over multiple passes through data, leading to variable runtime depending on convergence.
10. Perceptron	Fast	Low	Uses a simple linear model with updates made over a fixed number of epochs. Its low complexity and fast convergence result in a consistent, fast runtime.
11. SVC	Very Slow	Moderate	Applies kernel methods with iterative optimization. Runtime grows with data size and feature count.
12. K-Nearest Neighbors	Moderate	Moderate	Skips training phase, but runtime increases linearly during prediction as dataset grows.
13. Gaussian NB	Fastest	Highest	Computes closed form probabilities quickly (i.e., no repeated updates, no convergence checks, no loops). Small fluctuations in fast runtimes amplifies variance.
14. Linear Discriminant Analysis	Extremely Fast	Moderate	Computes closed form class means and shared covariances. Relies on matrix operations with no iteration.
15. Quadratic Discriminant Analysis	Extremely Fast	Moderate	Computes closed form, class-specific covariances. Slightly more complex than LDA, but still highly efficient
16. MLP Classifier	Very Slow	High	Uses gradient-based training across multiple layers. Requires variable convergence times. Training not explicitly parallelized across models or hyperparameters.

Models such as Gradient Boosting, AdaBoost, Logistic Regression, SGD, SVC, and MLP rely on iterative optimization processes involving repeated parameter updates across many steps. In contrast, GaussianNB, LDA, and QDA are closed-form models that compute solutions directly in a single pass. This makes closed-form models extremely fast compared to iterative models.

7.5. Hyperparameter Tuning Effects on Model Performance

Figure 7.9 summarizes how tuning hyperparameters impacted each model's accuracy and runtime. For most models, accuracy gains were minor, typically well under one percentage point. This outcome reflects the tuning objective: each model was tuned using GridSearchCV, with "scoring":"accuracy" as its sole metric. Models that showed no change in accuracy may: (a) have had default configurations that already were close to optimal for the dry bean dataset, or (b) were not given the hyperparameters or value choices that may have improved accuracy. In contrast, runtime changes were incidental and not directly optimized.

Figure 7.9 Accuracy and Runtime Changes After Hyperparameter Tuning

Model	Accuracy (Percentage Point Change)	Runtime (Percent Change)
1. Decision Tree	1.42	-22.9%
2. Random Forest	0.13	-91.9%
3. Extra Trees	0.28	-69.9%
4. Gradient Boosting	0.10	-81.5%
5. AdaBoost	1.65	95.0%
6. XGBoost	0.19	-13.0%
7. Logistic Regression	0.05	174.5%
8. Ridge	0.00	-9.3%
9. SGD Classifier	0.35	82.0%
10. Perceptron	0.05	-4.5%
11. SVC	0.27	246.3%
12. K-Nearest Neighbors	0.09	-28.8%
13. Gaussian NB	0.00	-9.7%
14. Linear Discriminant Analysis	0.00	-7.8%
15. Quadratic Discriminant Analysis	0.80	-10.1%
16. MLP	0.01	-22.0%

Among all models, only two showed accuracy increases above one percentage point:

- Decision Tree (+1.42 percentage points) capped max_depth (from none to 10) and increased min_samples_split (from 2 to 5). These changes prevented the model from overfitting to the training data, which may have improved its ability to generalize unseen data.

- AdaBoost (+1.65 percentage points) doubled n_estimators from 50 to 100. This gave AdaBoost more opportunities to correct errors, possibly improving accuracy by refining its decision boundaries.

Several models became substantially faster after tuning, particularly those where tree depth was reduced or parallel execution was enabled. Three models had runtime reductions of 70% or more:

- Random Forest (-91.9%) reduced tree count (changed n_estimators from 100 to 50), and capped max_depth (down from none to 20), which put a ceiling on how deep each tree can grow, lowered complexity, and made it faster to build trees. Multithreading (via n_jobs=-1) accelerated runtime.
- Extra Trees (-69.9%) made a single hyperparameter change, increasing min_samples_split (from 2 to 5). This produced shallower, less complex, and faster-to-build individual trees. Because Extra Trees builds many trees, making each individual tree faster to construct, even slightly, led to a substantial time reduction. Parallel processing (via n_jobs=-1) also boosted speed.
- Gradient Boosting (-81.5%) made a single change, lowering maximum tree depth (from 5 to 3). Shallower trees trained faster and required fewer splits and node evaluations. Gradient boosting builds trees sequentially, so each learner must be trained in order, so reducing tree depth compounds the runtime benefit. Shallower trees required fewer splits and evaluated with fewer nodes, which cut per-stage compute time. The model does not support parallel processing.

Four models had significantly longer runtimes after tuning, often due to more complex optimization demands or lack of parallelism. Four models had runtimes that increased 82% or more:

- AdaBoost (+95.0%) changed a single hyperparameter, doubling the number of estimators (from 50 to 100). This parameter dictated the number of individual weak learners (typically decision stumps) that the AdaBoost algorithm will train sequentially. By doubling it, the algorithm was forced to train twice as many base models. The model is single-threaded.
- Logistic Regression (+174.5%) also changed a single hyperparameter, decreasing the regularization strength by increasing C from 1 to 10, while keeping the solver as 'lbfgs'. A higher C value is less tolerant of classification errors, and so the model tried to fit the training data more closely, reducing regularization. But this also required more optimization effort to converge on a solution. Multithreading (via n_jobs=-1) was not enough to offset the change in C.
- SGD (+82.0%) changed a single hyperparameter, switching from penalty='l2' to penalty='l1'. The 'l1' regularization often requires more complex optimization steps or iterations, which can make convergence slower and less stable. Multithreading (via n_jobs=-1) was not enough to offset the change to l1.

- SVC (246.3%) also changed a single hyperparameter, increasing C from 1 to 10, reducing regularization and causing the model to fit the training data more tightly. This led to more support vectors, which are computationally expensive to identify and process. Keeping the rbf kernel added to this cost by projecting the data into a higher-dimensional space for its calculations. The model is inherently single-threaded.

The six models that support multi-threading (n_jobs=-1) used it after tuning. However, their shifts in runtime were significantly different. They shared few hyperparameters in common that were tuned, and where there was some overlap, the hyperparameters were tuned to different values:

- Random Forest (-91.9%) reduced n_estimators and capped tree depth, directly reducing its fit-time burden while leveraging multi-threading.
- Extra Trees (-69.9%) held tree count and depth unchanged, but increased min_samples_split, limiting splits and reduced the number of nodes per tree, resulting in a steep runtime drop.
- XGBoost (-13.0%) tuned four hyperparameters (n_estimators, max_depth, learning_rate, subsample) from “None” to explicit values. This resulted in a modest improvement in runtime. This suggests that XGBoost's "None" default settings were already highly optimized for the dry bean classification task.
- Logistic Regression (+174.5%) decreased its regularization strength by increasing C from 1 to 10. The optimizer had to perform more steps to reach a less-regularized solution.
- SGD (+82.0%) switched from L2 to L1 penalty, adding optimization complexity and shifting sparsity dynamics, both possibly increasing runtime.
- Perceptron (-4.5%) did not change its default settings from the tuning process. The model is inherently fast, and its lightweight architecture meant multi-threading had little additional effect in this benchmark.

7.6. Model Performance Evaluation Summary

The benchmarking results show machine learning models remain relevant for structured data for supervised classification applications because they offer a reasonable combination of accuracy, efficiency, interpretability, and robustness. Key findings from this study include the following:

Close Accuracies, Ignoring AdaBoost

The average accuracy of the top 5 models was 92.96%. The bottom 5 models, excluding AdaBoost. was 90.28%. Most models fell within a narrow performance band, demonstrating that classical approaches can deliver dependable results. The top-performing models came from tree-based, kernel-based, and neural network families, reflecting diverse algorithmic strengths.

Efficiency Leaders

GaussianNB, Ridge, and QDA delivered the fastest runtimes, under 40 ms, but generally scored lower in overall accuracy. This highlights the classic trade-off between speed and predictive power.

Model Architecture Matters

AdaBoost emerged as a clear outlier, with the lowest overall accuracy (84.4%) and a striking misclassification pattern for Bombay beans (23.6% accuracy), mislabeling 398 of the 399 misclassifications as Cali. This result sharply contrasted with the five other tree-based models, all of which classified Bombay beans with near-perfect accuracy.

Runtime Patterns

Model runtimes varied significantly across families. Iterative approaches, such as boosting and neural networks, took the longest to run. Linear and probabilistic models were quicker, thanks to their use of algebraic or closed-form solutions. Tree-based ensembles showed mid-range runtimes by comparison. The ability of a model to use multithreading had a significant impact. On average, single-core models took nearly six times longer than those with multithreading support.

Runtime Trade-Offs

Models with the highest accuracy tended to be very slow. For example, MLP was the most accurate but also had the second longest runtime, at over 12,000 ms per run. On average, the top five most accurate models required 67 times more time than the five least accurate models (again excluding AdaBoost).

No Simple Correlation of Accuracy and Runtimes

While higher accuracy generally required longer runtimes, the relationship was not strictly linear. Some ensemble models achieved balanced performance, while others skewed toward either lower accuracy or higher runtimes.

Hyperparameter Tuning Outcomes

Hyperparameter tuning had minimal effect on accuracy for most models. Nearly all models showed less than a one percentage point improvement. This suggests that many default configurations were already near-optimal for the dry bean dataset, or that the choice of hyperparameters and value ranges to tune were not the best choices to improve accuracy more. The effect on runtimes was more pronounced, in both directions. Three models reduced runtimes by over 70%, largely due to simplified tree structures or multithreading. Four others had substantial increases in runtimes, more than doubling in some cases, primarily due to increased complexity or lack of parallel processing.

Feature-Level Insights

Dataset Simplicity and Integrity

The Dry Bean dataset required only two minimal preprocessing steps: feature scaling and class label encoding. All features were complete, numeric, and free from missing values, eliminating the need for imputation, type conversion, or one-hot encoding. Unlike many multi-class classification datasets, it presented no issues with duplicate records, outliers, high-cardinality categorical variables,[10] inconsistent labels, or embedded null logic. This rare level of cleanliness enabled a straightforward benchmarking framework where model behavior could be evaluated without interference from upstream data preparation artifacts.

Feature Redundancy and Benchmarking Implications

Statistical analysis revealed that many features exhibited strong positive or negative correlations, indicating redundancy within the dataset. These relationships suggest that multiple features convey overlapping structural information, which can complicate interpretation across some modeling approaches. Conversely, uncorrelated feature pairs demonstrated independence, offering distinct inputs that helped models clarify feature space, and supported a bit more comparative insight across the benchmarked models.

Bombay Bean Distinctiveness

Bombay beans stood out because they are larger than the other six beans. Fifteen of sixteen models rarely misclassified them, including seven of which achieved perfect accuracy. These results underscore how unique, easily separable features can improve classification.

Sira Bean Misclassification Pattern

Sira was the most frequently misclassified bean variety, with most errors involving it being mislabeled as Dermason. The confusion stemmed from feature pairs with nearly identical average values (centroids) and a wide dispersion in one or both features. For example, Shape Factor 2 showed substantial variability in values for each bean, while Solidity remained tightly clustered for each bean. This imbalance, one feature highly dispersed and the other narrowly concentrated, created overlap in feature space making it difficult for models to define clear class boundaries. The combined standard deviation across Sira and Dermason classes was 12× larger than the centroid distance between them. This made it challenging for models to distinguish the two beans.

Ambiguous Feature Pairs

Certain feature pairs with nearly identical class centroids and wide dispersion contributed to classification errors. These overlaps blurred boundaries between bean types and reduced accuracy across several models.

[10] High-cardinality categorical fields refer to features that contain a large number of distinct category labels (e.g., hundreds or thousands). These often complicate encoding strategies and can increase model overfitting risks by introducing confusing patterns with little predictive value.

Appendix A. Model Definitions

This appendix provides a brief overview of the 16 machine learning models benchmarked in this project, categorized by their underlying algorithm family. Each entry includes a brief description of how the model functions, key advantages or limitations, and typical use cases.

A. Tree-Based

1. Decision Tree Classifier

Builds interpretable decision rules by recursively splitting features based on thresholds (e.g., binary yes/no questions), making it easy to follow logic paths. It performs well when data have clear, hierarchical structure but is prone to overfitting, especially with noisy or small datasets. The model often is used as a baseline for classification tasks with simple relationships.

2. Random Forest Classifier

Aggregates predictions from multiple randomized decision trees (a "forest") to improve accuracy and reduce overfitting. Its robustness and ability to generalize across data types make it well-suited for tasks requiring reliability and interpretability through feature importance.

3. Extra Trees Classifier

Constructs an ensemble of trees using randomized splits for both feature selection and thresholding, which increases diversity and speeds up training. While computationally efficient and stable, its randomness can introduce variance unless regularization or ensemble size are well-tuned. This model is useful for rapid experimentation on structured datasets.

4. Gradient Boosting Classifier

Sequentially builds an ensemble of weak learners, where each new model corrects errors made by previous ones. It is capable of modeling complex patterns but is prone to overfitting without careful regularization or feature selection. This model is common in competition-grade pipelines.

5. AdaBoost Classifier

Emphasizes misclassified samples across boosting rounds, directing subsequent weak learners to focus more on hard-to-predict instances. It is effective on clean data but highly sensitive to noise or mislabeled points. This model is ideal for binary tasks with well-defined decision boundaries.

6. XGBoost Classifier

An optimized gradient boosting framework that incorporates regularization, native missing value handling, and parallel computation. Known for strong accuracy and versatility across structured problems, XGBoost is widely used in production and research contexts.

B. Linear

7. Logistic Regression

A discriminative, parametric model that estimates the probability of class membership using a linear boundary to separate categories. Fast and interpretable, it performs well on clean, scaled data but struggles when relationships are highly nonlinear. For multiclass tasks, it trains multiple binary classifiers using a one-vs-rest (OvR) approach. It is a staple baseline for binary classification.

8. Ridge Classifier

Adds an L2 regularization penalty to the loss function of a linear model to discourage the model from assigning overly large weights to any single feature. This helps reduce overfitting and improve generalization in the presence of multicollinearity. It is best used when features are moderately correlated but still linearly related to target labels.

9. Stochastic Gradient Descent (SGD) Classifier

Fits linear models using incremental updates from randomly sampled training instances, making it highly scalable to large or sparse datasets. It requires careful data preprocessing and hyperparameter tuning due to sensitivity to outliers and potential convergence issues. This model is useful in streaming or online learning setups.

10. Perceptron

A basic linear classifier that updates weights in response to misclassified examples until convergence or a maximum number of epochs. While fast, It is limited in capacity (only learns linearly separable patterns) and easily destabilized by noisy or overlapping data. Historically foundational, the Perceptron is now used primarily as a baseline or educational tool.

C. Kernel-Based

11. Support Vector Classifier (SVC)

A discriminative classifier that separates classes by finding an optimal hyperplane, often by implicitly mapping data into a higher-dimensional feature space through the use of kernel functions. SVC is a strong performer on structured problems but can be computationally expensive and is sensitive to feature scaling. It is particularly effective for binary classification with complex boundaries.

D. Instance-Based

12. K-Nearest Neighbors (KNN) Classifier

Classifies samples by assigning them the majority class of their 'K' nearest training points, based on a distance metric. Simple and flexible, KNN is computationally expensive at prediction time (especially with large datasets) and highly sensitive to irrelevant features, scaling, and outliers. It often is effective on small, clean datasets with few dimensions.

E. Probabilistic

13. Gaussian Naive Bayes (GaussianNB)

A generative probabilistic model that assumes features are statistically independent and normally distributed within each class. It is extremely fast and effective on high-dimensional or small datasets but relies heavily on the data fitting its assumptions. Gaussian Naive Bayes often is used for baseline comparisons and text classification.

F. Discriminant

14. Linear Discriminant Analysis

A generative classification model that projects data onto axes that maximize class separation based on class means and common variance. LDA works well when feature distributions are similar across classes and covariance assumptions hold. It is common in dimensionality reduction and balanced classification tasks.

15. Quadratic Discriminant Analysis

Also a generative classification model, QDA extends LDA by modeling each class with its own covariance matrix, allowing for nonlinear decision boundaries. QDA offers more flexibility but requires careful handling of small sample sizes and noisy data. It performs best when class distributions are well-separated and Gaussian-distributed.

G. Neural Network

16. Multi-layer Perceptron (MLP) Classifier

A multi-layer feedforward neural network with one or more hidden layers, trained via backpropagation. It is capable of capturing complex, nonlinear relationships, but typically requires clean, scaled data and careful hyperparameter tuning. It is effective for structured data when other models might underfit or when complex interactions are suspected.

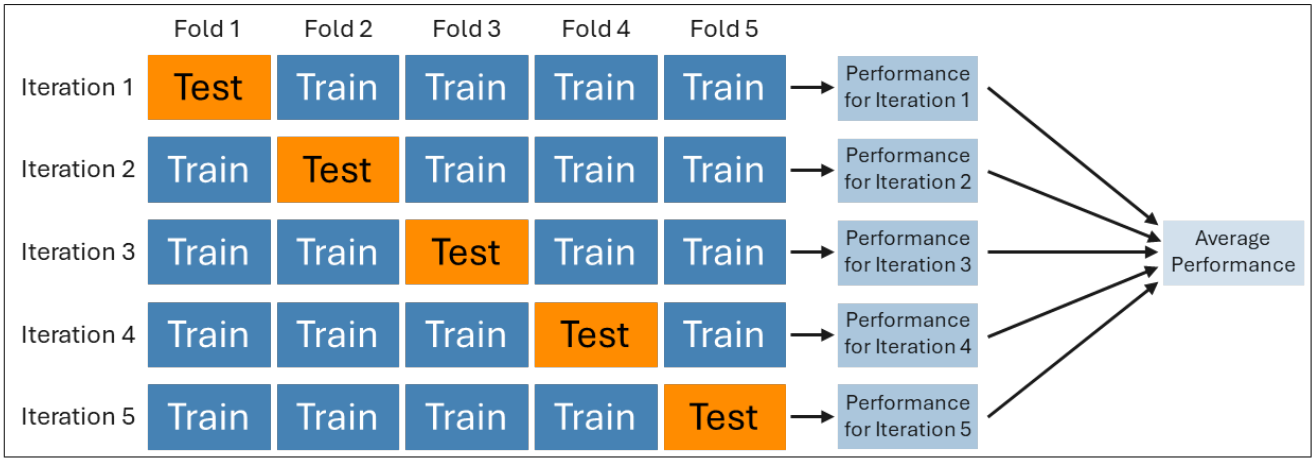
Appendix B. Stratified 5-Fold Cross-Validation Using StratifiedKFold

This appendix explains the use of stratified 5-fold cross-validation in this benchmarking study, why it was chosen, and how it was implemented using the StratifiedKFold class from the Scikit-learn library. The approach described here is a standard, widely accepted approach for evaluating supervised classification models.

Stratified 5-fold cross-validation is a resampling strategy used in machine learning to assess model performance in a consistent and reliable manner. The standard k-fold cross-validation may create folds with imbalanced class distributions. The stratified variant ensures that the distribution of target classes, such as the seven bean varieties in the Dry Bean dataset, is proportionally maintained in each fold. This is especially important when working with class imbalance because it prevents overrepresentation or underrepresentation of any class in training or test sets.

Figure B.1 provides an overview of the process and was adapted directly from Medium's *Python in Plain English*. [1]

Figure B.1Training-Test Split Workflow Across 5 Stratified Folds



In this benchmarking study, stratified 5-fold cross-validation was implemented using the StratifiedKFold class from Scikit-learn. The entire process was configured with a single line of Python code:

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

[1] "Cross-Validation in Machine Learning: Enhancing Model Performance with Confidence", https://python.plainenglish.io/cross-validation-in-machine-learning-enhancing-model-performance-with-confidence-39402f6ca60f

where:

- n_splits=5 specifies the creation of five distinct folds for cross-validation
- shuffle=True ensures the data are randomly shuffled before splitting, which can reduce bias in fold composition
- random_state=42 sets a fixed seed for reproducibility, ensuring the same splits are used across different runs.

The Python cv object (an instance of StratifiedKFold) then was used to generate training and test set indices (i.e., the specific records in the Dry Bean dataset), ensuring that each model was evaluated under consistent and class-balanced conditions.

Cross-Validation Workflow

1. **Dataset Partitioning**

   The full dataset of 13,611 records was divided into five equal-sized folds. Each fold contained approximately 20% of the total records (~2,722 samples per fold), while maintaining the original class proportions within each subset.

2. **Iterative Training and Testing**

   The cross-validation loop performed five iterations. In each iteration:

   - One unique fold was selected as the test set (~20% of records)
   - The remaining four folds (~80%) were combined to form the training set
   - The model was trained on the training set and evaluated on the test set.

3. **Rotation Across Folds**

   This rotation proceeded systematically:

   - Fold 1 was used as test, Folds 2–5 as train (Iteration 1)
   - Fold 2 was used as test, Folds 1, 3–5 as train (Iteration 2)
   - …and so on, until each fold had served as the test set exactly once.

4. **Coverage Guarantee**

   By the end of the process:

   - Every record had been used for testing exactly once
   - Each record had contributed to model training four times
   - The class distribution was preserved in all training and testing subsets.

Relevance to This Benchmarking Project

This validation strategy has several benefits that made it well-suited for this project:

1. **Reliable Model Evaluation**

   Performance metrics, such as accuracy and F1 score, were averaged across folds, reducing the likelihood of misleading results caused by an unusually easy or difficult train–test split.

2. **Class Balance Maintenance**

   Each model saw a representative distribution of all bean types in every training and test fold, ensuring a fair and unbiased comparison. This was particularly important for underrepresented classes. The distribution of beans is imbalanced, as shown below:

Bean	Count	Percentage
Barbunya	1,322	9.71%
Bombay	522	3.84%
Cali	1,630	11.98%
Dermason	3,546	26.05%
Horoz	1,928	14.17%
Seker	2,027	14.89%
Sira	2,636	19.37%
Total	**13,611**	**100.01%**

3. **Consistent Runtime Conditions**

   Because all models were trained and tested across the same five-fold configurations, runtime comparisons are consistent and more meaningful.

4. **Efficient Use of Data**

   All records were used in both training and testing, maximizing data utility. This was especially beneficial for modestly sized datasets like Dry Bean (13,611 records), which is at the lower end of what is typically considered medium-sized in machine learning.

5. **Improved Reliability of Bean Type Metrics**

   Because all bean varieties were proportionately represented in every test fold, this project could more reliably compute per-class accuracy metrics (including the discussion in the report of Bombay and Sira beans). Without stratification, some classes might be underrepresented or absent in certain folds, reducing the reliability of per-class insights.