Data Mining and Machine Learning Course Semester Project

Ahmet Furkan Ün
EM1SNO
https://www.kaggle.com/ahmetfurkanunn

December 14, 2024

Contents

1 Introduction

1.1 Background and Context

Infrastructure-as-Code (IaC) is a popular DevOps practice that allows teams to manage and set up infrastructure through code. Instead of manually configuring hardware or using interactive tools, you can use scripts that are easy to automate and share. This speeds up processes and ensures consistency. However, like any code, IaC scripts can have bugs, leading to significant issues.

To address this, I explore how machine learning can be used to predict defects in IaC scripts before they cause problems. This study focuses on the YAML-based Ansible language, analyzing a dataset of metrics collected from GitHub repositories to identify patterns and risks.

The goal of this project is to predict defects in Infrastructure-as-Code scripts. By analyzing structural, change, and process metrics, I aim to identify the best predictors of script quality and use them to build effective machine-learning models.

2 Dataset Analysis

2.1 Description of the Dataset

The dataset I used for our project groups metrics into three main types. First, there are IaC-oriented metrics, which are about the structural properties coming from the source code of infrastructure scripts such as the number of distinct modules and the number of times the script accesses a file through the file module. Second, Delta metrics measure how much a file changes between two releases, and they are collected for each IaC-oriented metric. Lastly, Process metrics focus on the development process instead of the code itself such as the total, average, and maximum number of lines added to a file and the average and maximum number of files committed together to the repository. Our target variable is a binary called "failure_prone," which shows whether a file is likely to lead to a failure.

2.2 Exploratory Data Analysis (EDA)

The training dataset (X_train) contains 109 columns, which include an identifier (id), IaC-oriented metrics, Delta metrics, and Process metrics. The dataset has a total of 159,090 rows, and the corresponding target labels (y_train) also have 159,090 rows. The target dataset (y_train) consists of two columns: id and failure_prone. The latter is a binary label indicating whether a file is prone to failure. The shapes of the datasets are as follows:

- X_train shape: (159,090, 109)
- y_train shape: (159,090, 2)

All features in the dataset are numerical, represented as either ‘int64‘ or ‘float64‘. Importantly, there are no missing values in the dataset.

The target variable (failure_prone) has a highly imbalanced distribution:

- Failure-prone = 0 (non-failure-prone): 149,452 instances (93.94%)
- Failure-prone = 1 (failure-prone): 9,638 instances (6.06%)

This imbalance indicates that most of the data represent non-failure-prone files, which might require handling during model training to avoid bias toward the majority class.

Figure 1 shows the correlation matrix of the features. Some features exhibit high correlations (above 0.95), suggesting redundancy within the dataset. Notable highly correlated feature pairs include:

- additions_avg and additions_max with (correlation: 0.956)
- additions_max and code_churn_max with (correlation: 0.953)
- code_churn_avg and code_churn_max with (correlation: 0.956)
- code_churn_count and code_churn_max with (correlation: 0.960)
- num_distinct_modules and num_tasks with (correlation: 0.989)
- delta_lines_code and delta_num_keys with (correlation: 0.964)
- delta_num_distinct_modules and delta_num_tasks with (correlation: 0.992)
- delta_num_keys and delta_num_tokens with (correlation: 0.952)

Such correlations could introduce multicollinearity, potentially impacting model performance. Dimensionality reduction techniques or feature selection methods will be explored to address this issue.

After excluding the id column, there are 93,440 duplicated rows in the dataset. These are not considered errors and are not an artifact of the data collection process from GitHub repositories. As they reflect real-world data, these duplicates are retained to preserve the integrity and representativeness of the dataset.

Several columns have values that deviate significantly from the interquartile range (1.5 IQR). These outliers are not errors but are assumed to carry valuable information, potentially indicative of unique cases or extreme scenarios in the data. Therefore, they are not removed during preprocessing.

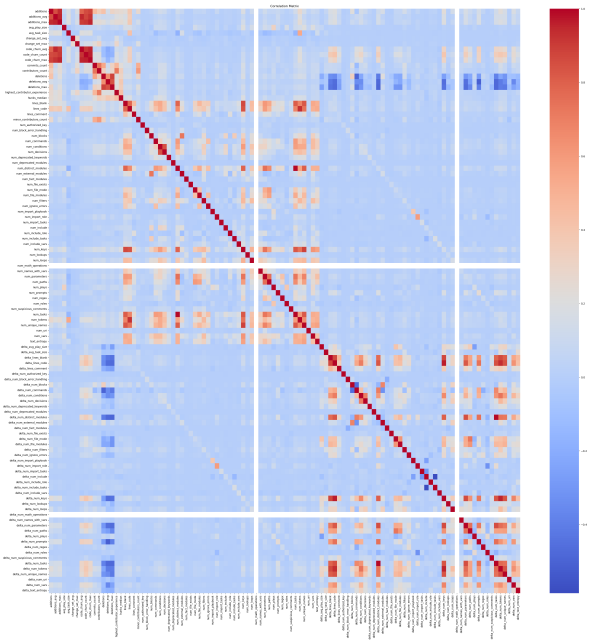

Figure 1: Correlation Matrix of the features

3 Data Preprocessing

3.1 Scaling

Feature scaling is an essential preprocessing step to ensure that all features contribute equally to the model training process, especially when using algorithms sensitive to feature magnitudes. I used StandardScaler for scaling the dataset. This scaler transforms each feature to have a mean of 0 and a standard deviation of 1.

Importantly, the mean and standard deviation values were calculated exclusively on the training dataset to avoid data leakage. These values were then applied to scale both the training and testing datasets. This ensures that information from the test data is not incorporated into the training phase, preserving the integrity of the evaluation process. Later in the report, this scaled version of the dataset that contains all features will be named as **'raw'**

3.2 Variance Inflation Factor (VIF)

Multicollinearity occurs when features are highly correlated with each other, which can negatively impact model performance by introducing instability and redundancy. To address this, I used the Variance Inflation Factor (VIF), a measure that quantifies the extent of multicollinearity.

The VIF value for each feature was calculated, and features with a VIF greater than 10 were considered highly correlated and excluded from further analysis. This threshold ensures that retained features have minimal overlap in the information they convey. The process resulted in a reduced set of features, which were then used to create filtered datasets (X_train_vif and X_test_vif). Later in the report, this filtered

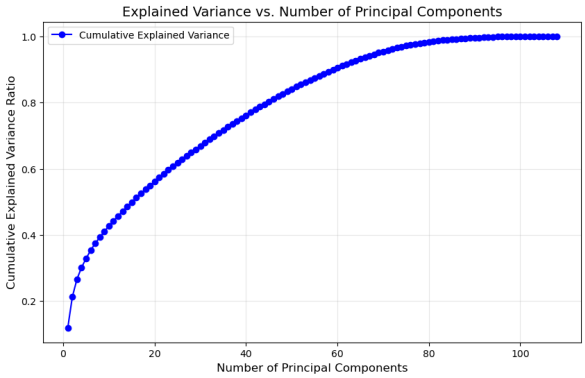

Figure 2: Effect of Number of Principal Components on the Explained Variance

version of the dataset will be named as **'vif'**

This step not only mitigates the risk of multicollinearity but also reduces the complexity of the dataset, which can improve the robustness of the models trained on it.

3.3 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) was applied to further reduce the dimensionality of the dataset while retaining as much variance as possible. PCA achieves this by creating new features (principal components) that are linear combinations of the original features. These components are orthogonal to each other, ensuring no redundancy. Before applying PCA, it is important to scale the features to eliminate the bias towards the features having large scales. Figure 2 shows how the explained variance changes with the number of principal components used.

Two configurations of PCA were tested:

- 60 Components: The first 60 principal components were retained, capturing approximately 90.49% of the total variance in the dataset.
- 85 Components: The first 85 principal components were retained, capturing approximately 99.04% of the total variance.

Later in the report, this first scaled and then PCA-applied version of the dataset will be named as **'pca60'** and **'pca85'** that contains the first 60 and 85 principal components respectively.

4 Modeling Approach

4.1 Naive Bayes

The Naive Bayes algorithm is a probabilistic classifier based on Bayes' theorem, assuming independence between features. Despite this simplistic assumption, it is com-

putationally efficient and often works surprisingly well for certain types of data. For this task, I used the `GaussianNB` implementation, which assumes that the continuous features follow a Gaussian distribution.

No hyperparameter tuning was applied for this model. It was tested on the raw dataset as well as on datasets processed with VIF-based feature selection, PCA (85 components), and PCA (60 components).

4.1.1 Results

- The highest balanced accuracy on the test set (0.5865) was achieved with the VIF-selected dataset, slightly outperforming the raw dataset.
- PCA-transformed datasets showed a significant drop in performance, particularly with 60 components, which had the lowest test balanced accuracy (0.5619).
- Training times were consistent and very low across all versions, reflecting the computational efficiency of Naive Bayes.
- The overall performance indicates that Naive Bayes may struggle to capture the complexities of this dataset, especially due to the strong correlations among features.

Table 1: Naive Bayes Results

Train Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset	Features	Params
0.1951	0.5855	0.5854	raw	None	None
0.1491	0.5877	0.5865	vif	None	None
0.1550	0.5707	0.5732	pca85	None	None
0.1130	0.5568	0.5619	pca60	None	None

4.2 Logistic Regression

Logistic Regression is a linear classification algorithm that models the relationship between features and the log odds of a binary outcome. It is computationally efficient and interpretable, making it suitable for this problem.

4.2.1 RFECV Feature Selection

I applied Recursive Feature Elimination with Cross-Validation (RFECV) to reduce dimensionality. RFECV selected 69 features, which are listed below, as the most predictive of the failure-prone label. This subset helped balance model complexity and performance.

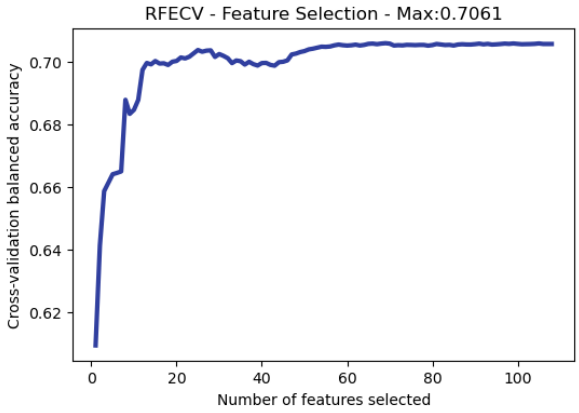

Figure 3: Effect of the number of features on the balanced accuracy score

- **Optimal Number of Features**: 9
- **Selected Features**:
  - additions, additions_max, avg_play_size, avg_task_size, change_set_avg, change_set_max, code_churn_avg, code_churn_count, code_churn_max, commits_count, contributors_count, deletions, deletions_avg, highest_contributor_exp lines_blank, lines_code, lines_comment, num_blocks, num_commands, num_conditions, num_decisions, num_deprecated_keywords, num_deprecated_modules, num_distinct_modules, num_external_modules, num_fact_modules, num_file_exists, num_file_mode, num_file_modules, num_filters, num_import_playbook, num_import_role, num_import_tasks, num_include, num_include_role, num_include_tasks, num_include_vars, num_keys, num_lookups, num_loops, num_names_with_vars, num_parameters, num_paths, num_plays, num_prompts, num_regex, num_suspicious_comments, num_tasks, num_tokens, num_unique_names, num_uri, num_vars, text_entropy, delta_lines_code, delta_num_blocks, delta_num_commands, delta_num_conditions, delta_num_decisions, delta_num_fact_modules, delta_num_filters, delta_num_import_role, delta_num_keys, delta_num_names_with_vars, delta_num_prompts, delta_num_tasks, delta_num_tokens, delta_num_unique_names, delta_num_vars, delta_text_entropy

4.2.2 Hyperparameter Tuning

A Randomized Grid Search over 30 iterations was conducted to optimize the following parameters:

- **Penalty**: Regularization type (`l2`, `none`).
- **Tolerance (`tol`)**: Convergence threshold (10^{-5} to 10^{-3}).
- **Inverse Regularization Strength (`C`)**: Values from 0 to 2.
- **Class Weight**: Handles class imbalance (`balanced`, `None`).
- **Solver**: Optimization algorithm (`lbfgs`, `sag`, `saga`).
- **Fit Intercept**: Bias term inclusion (`True`, `False`).

4.2.3 Results

The RFECV-selected dataset achieved the best test balanced accuracy of 0.7023, highlighting the importance of feature selection. Hyperparameter tuning improved performance across all datasets, although training times increased. The best test balanced accuracy (0.7023) was achieved on the RFECV-selected dataset, demonstrating

the effectiveness of feature selection. Hyperparameter tuning consistently improved model performance compared to untuned versions, highlighting the importance of regularization and class balancing. The raw dataset and PCA (85 components) showed competitive results post-tuning, but the PCA (60 components) dataset consistently underperformed. Training times increased significantly with hyperparameter tuning and feature selection, reflecting the added computational complexity.

Table 2: Logistic Regression Results

Train Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
1.6304	0.7084	0.7012	raw	None	None
1.6404	0.6996	0.6938	vif	None	None
1.0533	0.7033	0.6989	pca85	None	None
0.8882	0.6887	0.6837	pca60	None	None
38.1905	0.7080	0.7008	raw	None	tol: 0.001, solver: lbfgs, penalty: l2, fit_intercept: False, class_weight: balanced, C: 2.0
27.9060	0.7000	0.6896	vif	None	tol: 0.001, solver: lbfgs, penalty: l2, fit_intercept: False, class_weight: balanced, C: 2.0
29.0381	0.7055	0.7017	pca85	None	tol: 0.001, solver: lbfgs, penalty: l2, fit_intercept: False, class_weight: balanced, C: 2.0
32.9627	0.6840	0.6807	pca60	None	tol: 1e-05, solver: saga, penalty: l2, fit_intercept: False, class_weight: None, C: 0.6667
24.5467	0.7082	0.7023	raw	RFECV	tol: 0.001, solver: lbfgs, penalty: l2, fit_intercept: False, class_weight: balanced, C: 2.0

4.3 K-Neigbours

K-Nearest Neighbors (KNN)is an instance-based learning algorithm that does not involve explicit training, as it memorizes the training data and uses it during inference to classify test instances based on the majority class of the ‘k‘ nearest neighbors.

4.3.1 Hyperparameter Tuning

- **n_neighbors**: Number of neighbors considered for classification.
- **weights**: Determines the weight function used in prediction (`uniform` assigns equal weight to all neighbors, while `distance` assigns higher weight to closer neighbors).
- **metric**: The distance metric to use (`minkowski`, `euclidean`, or `manhattan`).
- **p**: Parameter for the `minkowski` metric (`p=1` corresponds to Manhattan distance, and `p=2` corresponds to Euclidean distance).

- **distance**: The algorithm used to compute the nearest neighbors (`ball_tree`, `kd_tree`, or `brute`).

A 10-iteration random search was performed due to the computational cost of KNN's instance-based nature.

4.3.2 Results

Before doing the parameter tuning, for the 'raw', 'PCA85', and 'PCA60' datasets, performance is consistent, with the 'raw' dataset slightly outperforming others in terms of balanced accuracy on both train and test datasets. The 'vif' dataset shows comparatively lower performance, indicating that removing multicollinearity might negatively impact the KNN classifier in this context. After hyperparameter tuning, the best results are achieved using tuned parameters, especially with the raw dataset ('0.8609' test balanced accuracy). PCA and VIF datasets show reduced test accuracy compared to raw data, highlighting that dimensionality reduction may not always improve performance for instance-based methods.

Table 3: K-Nearest Neighbors Results

Train Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
0.0540	0.8530	0.8051	raw	None	None
0.0520	0.8380	0.7831	vif	None	None
0.0500	0.8500	0.8005	pca85	None	None
0.0390	0.8425	0.7885	pca60	None	None
2149.7272	0.9861	0.8609	raw	None	`weights: distance, p: 2, n_neighbors: 3, metric: manhattan, algorithm: kd_tree`
1583.2770	0.9838	0.8469	vif	None	`weights: distance, p: 1, n_neighbors: 3, metric: minkowski, algorithm: brute`
1432.7658	0.9876	0.8386	pca85	None	`weights: distance, p: 2, n_neighbors: 3, metric: manhattan, algorithm: kd_tree`
1022.8975	0.9861	0.8275	pca60	None	`weights: distance, p: 1, n_neighbors: 3, metric: minkowski, algorithm: brute`

4.4 Decision Tree

The Decision Tree Classifier is a versatile model that splits the dataset into subsets based on feature conditions, forming a tree-like structure. It recursively partitions the data to minimize impurity at each node, using criteria such as Gini impurity, entropy, or log loss. Decision Trees are intuitive and perform well with minimal preprocessing but are prone to overfitting without appropriate regularization.

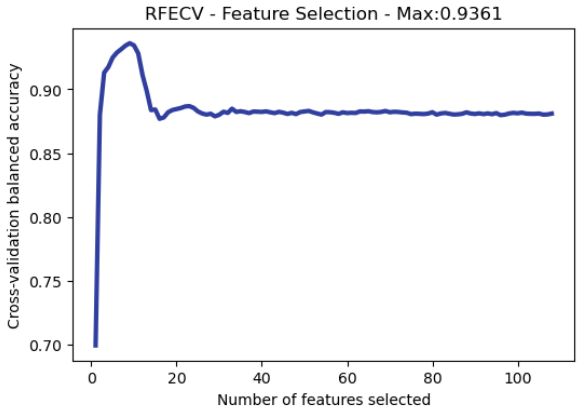

Figure 4: Effect of the number of features on the balanced accuracy score

4.4.1 RFECV Feature Selection

Recursive Feature Elimination with Cross-Validation (RFECV) was applied to identify the most significant features. RFECV eliminates features recursively and evaluates performance using cross-validation. For this model:

- **Optimal Number of Features**: 9
- **Selected Features**:
  - `avg_task_size`, `lines_blank`, `lines_code`, `lines_comment`, `num_conditions`, `num_keys`, `num_parameters`, `num_tokens`, `text_entropy`

4.4.2 Hyperparameter Tuning

A wide parameter grid was used for hyperparameter tuning, employing a random search with 100 iterations. The key hyperparameters and their significance are as follows:

- **max_depth**: The maximum depth of the tree. Limiting this prevents overfitting. A range of 1 to 50 was explored.
- **ccp_alpha**: Complexity parameter for Minimal Cost-Complexity Pruning, controlling tree size by penalizing complexity. Values ranged as 10^i, where i spans from -7 to 6.
- **min_samples_split**: The minimum number of samples required to split an internal node. A range of 1 to 100 was explored.
- **min_samples_leaf**: The minimum number of samples required to form a leaf node. A range of 1 to 100 was used.
- **criterion**: The function used to measure the quality of splits. Options included `gini`, `entropy`, and `log_loss`.

4.4.3 Results

The untuned Decision Tree on the raw dataset achieved a test balanced accuracy of **0.874** with the shortest training time, demonstrating the model's baseline performance. Hyperparameter tuning improved the test balanced accuracy to **0.899**, indicating that appropriate regularization (e.g., limiting depth and pruning) enhances generalization. Using RFECV-selected features further improved the test balanced accuracy slightly to **0.901**, highlighting the benefit of reducing feature dimensionality and focusing on the most relevant predictors. The training time increased slightly with hyperparameter tuning and RFECV, but the results justify the additional computational cost.

Table 4: Decision Tree Results

Training Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
2.300	0.997	0.874	raw	None	None
19.313	0.936	0.899	raw	None	`min_samples_split: 6, min_samples_leaf: 16, max_depth: 49, criterion: gini, ccp_alpha: 0.0001`
3.846	0.937	0.901	raw	RFECV	`min_samples_split: 6, min_samples_leaf: 16, max_depth: 49, criterion: gini, ccp_alpha: 0.0001`

4.5 Random Forest

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) or the mean prediction (regression) of the individual trees. This technique helps to reduce overfitting and improves generalization by combining the predictions of multiple trees.

4.5.1 RFECV Feature Selection

Recursive Feature Elimination with Cross-Validation (RFECV) was applied to identify the most significant features. RFECV eliminates features recursively and evaluates performance using cross-validation. For this model:

- **Optimal Number of Features**: 10
- **Selected Features**:
  - `avg_task_size`, `lines_blank`, `lines_code`, `lines_comment`, `num_conditions`, `num_keys`, `num_parameters`, `num_tokens`, `num_unique_names`, `text_entropy`

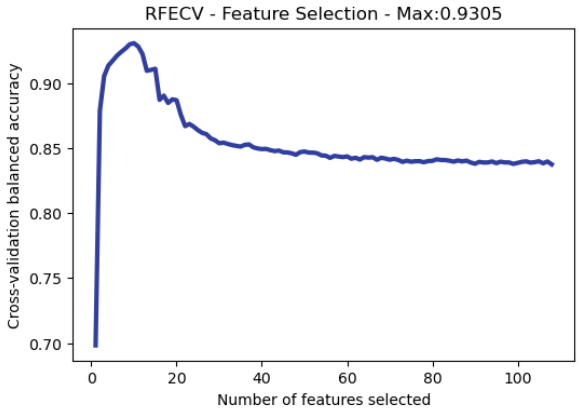

Figure 5: Effect of the number of features on the balanced accuracy score

4.5.2 Hyperparameter Tuning

The following hyperparameters were tuned using a random grid search with 100 iterations:

- **n_estimators**: The number of trees in the forest, ranging from 10 to 300 (step size: 10).
- **max_features**: The number of features to consider at each split, ranging from 1 to 50.
- **max_depth**: The maximum depth of each tree, ranging from 1 to 50.
- **min_samples_split**: The minimum number of samples required to split an internal node, ranging from 1 to 100.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node, ranging from 1 to 100.
- **bootstrap**: Whether bootstrap sampling is used when building trees (True or False).

4.5.3 Results

The Random Forest model demonstrates strong performance overall. When no feature selection or hyperparameter tuning is applied, the model achieves a test balanced accuracy of 0.835, which improves significantly with hyperparameter tuning (0.920). Incorporating RFECV-selected features further enhances the test balanced accuracy to 0.941, while also reducing the number of features to 10. However, the computational cost is notable, especially for the tuned models, as reflected in the training time.

Table 5: Random Forest Results

Training Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
18.555	0.997	0.835	raw	None	None
567.886	0.954	0.920	raw	None	n_estimators: 100, min_samples_split: 48, min_samples_leaf: 48, max_features: 46, max_depth: 35, bootstrap: False
432.358	0.975	0.941	raw	RFECV	n_estimators: 280, min_samples_split: 23, min_samples_leaf: 22, max_features: 5, max_depth: 31, bootstrap: False

4.6 Support Vector Classifier

The Support Vector Machine (SVM) is a supervised learning algorithm commonly used for classification tasks. It finds the hyperplane that best separates the data into different classes by maximizing the margin between the closest points of different classes (support vectors). Due to the computational intensity of SVMs, hyperparameter tuning and Recursive Feature Elimination with Cross-Validation (RFECV) were not performed in this case. This approach ensures that the training and prediction processes remain manageable within a reasonable timeframe.

4.6.1 Results

The training time for SVM is significantly higher compared to other models, reflecting the high computational cost of this algorithm. The balanced accuracy on the training set is consistent across dataset versions, but test accuracy is notably lower, suggesting potential overfitting. The PCA-reduced datasets (pca85 and pca60) show slightly lower training and test accuracies compared to the vif dataset, possibly due to the loss of critical information during dimensionality reduction.

Table 6: Suppert Vector Classifier Results

Training Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
1338.487	0.816	0.768	vif	None	None
1537.277	0.8159	0.767	pca85	None	None
1359.765	0.794	0.751	pca60	None	None

4.7 XGBoost Classifier

XGBoost is a powerful gradient-boosting framework optimized for speed and performance. It is known for handling missing data effectively and offering various tuning parameters to optimize model performance. The iterative boosting process builds models sequentially to correct the errors of the previous models.

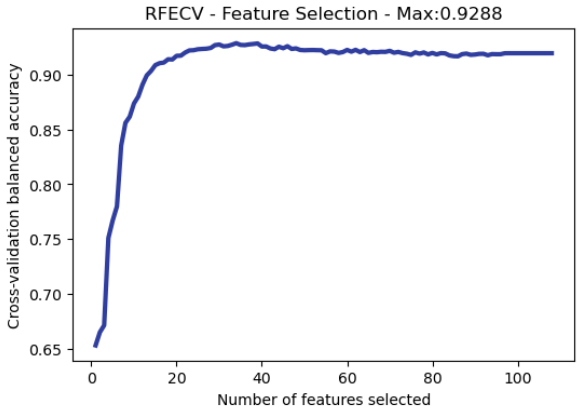

Figure 6: Effect of the number of features on the balanced accuracy score

4.7.1 RFECV Feature Selection

Recursive Feature Elimination with Cross-Validation (RFECV) was applied to identify the most significant features. RFECV eliminates features recursively and evaluates performance using cross-validation. For this model:

- **Optimal Number of Features**: 34
- **Selected Features**:
  - avg_play_size, avg_task_size, highest_contributor_experience, lines_blank, lines_code, lines_comment, num_blocks, num_commands, num_conditions, num_decisions, num_deprecated_keywords, num_distinct_modules, num_file_mode, num_file_modules, num_filters, num_ignore_errors, num_import_tasks, num_include_role, num_include_tasks, num_include_vars, num_keys, num_lookups, num_loops, num_names_with_vars, num_parameters, num_paths, num_plays, num_prompts, num_regex, num_tasks, num_tokens, num_unique_names, num_vars, text_entropy

4.7.2 Hyperparameter Tuning

The tuning process for XGBoost involved multiple steps:

- First, the optimal number of estimators was determined using XGBoost's internal tools, resulting in 588 estimators.
- The learning rate was set to 0.3 initially, as reducing it would have significantly increased the number of estimators.
- With the number of estimators fixed, the following parameter grid was used for tuning other hyperparameters with 150 iterations:

```
param_grid = {'max_depth':range(3,20),
              'min_child_weight':range(1,10),
              'gamma': np.arange(0, 0.5, 0.05),
              'subsample':np.arange(0.5, 1.05, 0.05),
              'colsample_bytree':np.arange(0.5, 1.05, 0.05),
              'reg_alpha':[10**i for i in range(-7, 7)]
             }
```

- After tuning these parameters, the best number of estimators was re-evaluated and updated to 1796.
- Finally, the learning rate and number of estimators were tuned using the following grid:

```
param_grid = {'learning_rate': np.arange(0, 0.51, 0.01),
              'n_estimators' : [588, 1796]
             }
```

4.7.3 Results

XGBoost demonstrates strong performance across the experiments, especially when optimized with feature selection and hyperparameter tuning. The models exhibit high balanced accuracy on both training and test datasets, indicating effective generalization. The RFECV-selected features further enhance the model's ability to focus on relevant inputs, improving computational efficiency while maintaining performance. Hyperparameter tuning shows significant improvements, particularly when the learning rate and number of estimators are fine-tuned. The best results are achieved with tuned parameters and RFECV-selected features, demonstrating the importance of careful feature and parameter selection.

Table 7: XGBoost Results

Training Time (s)	Train Bal. Acc.	Test Bal. Acc.	Dataset Version	Features	Params
1.2488	0.9588	0.9176	raw	None	None
253.8824	0.9921	0.9358	raw	None	n_estimators: 260, max_depth: 13, learning_rate: 0.04
117.1456	0.9891	0.9503	raw	RFECV	n_estimators: 260, max_depth: 13, learning_rate: 0.04
256.5854	0.9899	0.9377	raw	None	subsample: 0.95, reg_alpha: 0.01, min_child_weight: 3, max_depth: 4, gamma: 0.05, colsample_bytree: 0.65, n_estimators: 588, learning_rate: 0.3
70.6013	0.9883	0.9515	raw	RFECV	subsample: 0.75, reg_alpha: 1e-06, min_child_weight: 7, max_depth: 6, gamma: 0.4, colsample_bytree: 0.7, n_estimators: 588, learning_rate: 0.3
95.9848	0.9889	0.9477	raw	RFECV	subsample: 0.86, reg_alpha: 10, min_child_weight: 9, max_depth: 8, gamma: 0.37, colsample_bytree: 0.81, n_estimators: 588, learning_rate: 0.3
94.5798	0.9902	0.9453	raw	RFECV	subsample: 0.86, reg_alpha: 10, min_child_weight: 11, max_depth: 14, gamma: 0.18, colsample_bytree: 0.85, n_estimators: 588, learning_rate: 0.3
215.0480	0.9830	0.9492	raw	RFECV	subsample: 0.75, reg_alpha: 1e-06, min_child_weight: 7, max_depth: 6, gamma: 0.4, colsample_bytree: 0.7, n_estimators: 588, learning_rate: 0.3
116.7467	0.9857	0.9516	raw	RFECV	subsample: 0.75, reg_alpha: 1e-06, min_child_weight: 7, max_depth: 6, gamma: 0.4, colsample_bytree: 0.7, n_estimators: 588, learning_rate: 0.2
112.3774	0.9857	0.9516	raw	RFECV	subsample: 0.75, reg_alpha: 1e-06, min_child_weight: 7, max_depth: 6, gamma: 0.4, colsample_bytree: 0.7, n_estimators: 588, learning_rate: 0.2

5 Results and Discussion

In this study, I conducted an extensive evaluation of various machine learning models, including Naive Bayes, Logistic Regression, Decision Trees, Random Forests, Support Vector Machines (SVM), Gradient Boosting, and XGBoost, to determine their performance in predicting the target variable based on the provided feature set. Each model was rigorously trained, tuned, and assessed with a focus on optimizing accuracy, precision, and recall while balancing computational efficiency.

Naive Bayes and logistic regression were the baseline models, demonstrating simplicity and reasonable performance. With minimal tuning and feature engineering, it showed its limitations in capturing non-linear relationships, resulting in relatively lower scores compared to more advanced models. K-Neighbors Classifier was also evaluated during this study as a baseline non-parametric model. While simple and intuitive, its performance was heavily dependent on the choice of k and the distance metric. Using hyperparameter tuning, I identified the optimal configuration, but the model struggled with scalability and sensitivity to noisy data. Although it performed reasonably well on smaller datasets, it was outperformed by ensemble methods like Random Forests and XGBoost, especially in terms of handling class imbalances and higher-dimensional feature spaces. Decision Trees provided better interpretability but faced challenges with overfitting, especially at higher depths. By tuning the maximum depth and minimum samples per split, I improved the model's generalizability but found its standalone performance insufficient compared to ensemble methods.

Random Forests brought significant improvements, leveraging ensembling to mitigate overfitting. The model exhibited robust performance across accuracy, precision, and recall metrics. Feature selection through RFECV further improved its computational efficiency and interpretability without compromising performance.

Gradient Boosting built upon Random Forests, achieving comparable or slightly better scores in handling class imbalances. Its iterative boosting nature allowed the model to refine predictions effectively, though the training time increased substantially.

Support Vector Machines (SVMs) were evaluated despite their computational expense, particularly on larger datasets. With various preprocessing techniques (e.g., PCA for dimensionality reduction), the model showed competitive results but lagged behind XGBoost and Random Forests. However, due to the computational inefficiency of training and predicting with SVMs, hyperparameter tuning and RFECV were not applied. The results may improve when these two techniques are applied.

XGBoost emerged as the top-performing model in this study. Its iterative optimization process, ability to handle high-dimensional data, and inherent robustness to class imbalance set it apart. With a comprehensive and iterative tuning approach, XGBoost achieved the highest balanced accuracy score. Notably, RFECV reduced the feature set to 34 essential features, improving both efficiency and interpretability.