

Comparing different machine learning methods on NFL defensive player performance prediction

by

Jihyuk Gweon

B.S., Kansas State University, 2023

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Statistics
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2024

Approved by:

Major Professor
Haiyan Wang

Copyright

© Jihyuk Gweon 2024.

Abstract

This study examines the effectiveness of various machine learning techniques in predicting defensive player performance in the National Football League (NFL). The dataset from Kaggle's "NFL BIG DATA BOWL 2024" includes numerous variables related to games, players, plays, and tackle details for NFL players. Six datasets were created by pairing each of the three subsets of predictive variables with either of the two response variables, 'attack' or 'tackle'. These datasets were analyzed using multiple machine learning methods. The research aims to provide a comprehensive analysis of defensive performance by incorporating a broad array of algorithms, including Logistic Regression, Random Forest, K-Nearest Neighbors (KNN), AdaBoost, Support Vector Machine (SVM), Decision Tree, and XGBoost. Hyperparameter tuning was employed to optimize each model's performance.

Among the classifiers, Logistic Regression, XGBoost and AdaBoost consistently achieved the highest performance metrics, demonstrating their robustness and effectiveness in predicting NFL defensive performance. Decision Tree, SVM, and Random Forest have slightly lower AUC than the top three, but the difference is not statistically significant. KNN has lowest performance and the difference from the top three is significant even though the difference with the middle three classifiers is not significant. The analysis revealed that variables such as 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', and 'quarter' offer the most predictive power for both tackle and attack outcomes across all classifiers. This comprehensive evaluation underscores the importance of feature selection and hyperparameter tuning in optimizing machine learning models for sports analytics.

Table of Contents

List of Figures	v
List of Tables	vi
Acknowledgements	vii
1. Introduction.....	1
2. Analysis.....	5
3. Results.....	19
Conclusion and Discussion	48
Bibliography	53
Appendix A - Python code for Data Preparation	54
Appendix B - Python code for Logistic Regression	62
Appendix C - R code for Logistic Regression	66
Appendix D - Python code for Random Forest	68
Appendix E - Python code for K-Nearest Neighbor	71
Appendix F - Python code for AdaBoost.....	74
Appendix G - Python code for Support Vector Machine	76
Appendix H - Python code for Decision Tree	78
Appendix I - Python code for XGBoost	80
Appendix J - XGBoost result model over training assessment for the selected best hyperparameters.....	83

List of Figures

Figure 1. ROC Curve of Logistic Regression classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.52 to 0.92 on Dataset 1 to 6.	20
Figure 2. ROC Curve of Random Forest classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.52 to 0.91 on Dataset 1 to 6.	26
Figure 3. ROC Curve of KNN classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.50 to 0.90 on Dataset 1 to 6.....	30
Figure 4. ROC Curve of AdaBoost classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.92 on Dataset 1 to 6.....	33
Figure 5. ROC Curve of SVM classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.90 on Dataset 1 to 6.....	36
Figure 6. ROC Curve of Decision Tree classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.91 on Dataset 1 to 6.	39
Figure 7. ROC Curve of XGBoost classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.50 to 0.92 on Dataset 1 to 6.....	42
Figure 8. Negative Loglikelihood and Classification error Curve of XGBoost classifier on Datasets 1 to 3.	46
Figure 9. Negative Loglikelihood and Classification error Curve of XGBoost classifier on Datasets 4 to 6.	47

List of Tables

Table 1. Performance metrics and hyperparameters for logistic regression classifier across six datasets from Python LogisticRegressionCV.	22
Table 2. Performance metrics and hyperparameters for lasso logistic regression classifier across six datasets. These are the result trained with R function cv.glmnet using default l2 penalty. The results from Python are not better than those from R even though they were trained with other hyperparameter values.	23
Table 3. Performance metrics and hyperparameters for Random Forest classifier across the six datasets	27
Table 4. Performance metrics and hyperparameters for KNN classifier across the six datasets..	31
Table 5. Performance metrics and hyperparameters for AdaBoost classifier across the six datasets.	34
Table 6. Performance metrics and hyperparameters for SVM classifier across the six datasets..	37
Table 7. Performance metrics and hyperparameters for Decision Tree classifier across the six datasets.	40
Table 8. Performance metrics and hyperparameters for XGBoost classifier across the six datasets.	43

Acknowledgements

I would like to acknowledge and give my warmest thanks to my major professor Haiyan Wang who made this work possible. Her guidance and advice carried me through all the stages of writing my report.

I would also like to thank my committee members, Professor Cen Wu and Professor Mike Higgins for their encouragement and advice a lot.

I would also like to give special thanks to my family as a whole for their support and understanding when undertaking my report. Your prayer for me was what sustained me this far.

1. Introduction

The National Football League (NFL) represents the pinnacle of professional American football, a sport characterized by complex strategies and intense physical competition. Among the various aspects of gameplay, defensive performance is crucial, impacting game outcomes significantly. Traditionally, the evaluation of defensive players has relied on basic statistics such as tackles, interceptions, and sacks. However, these metrics do not fully capture the nuanced contributions of players to their teams' defensive efforts.

Recent advancements in data collection and analysis have spurred interest in applying predictive modeling to sports analytics. As such, there is an increasing interest in leveraging machine learning (ML) techniques to provide a more comprehensive analysis of defensive performance. In the context of the NFL, several studies have explored the use of machine learning to predict defensive performance, focusing primarily on the traditional models such as logistic regression and decision trees. Notable contributions include Kaggle report by Han (2023), Chen (2023), and DuBois (2023), which utilize various machine learning (ML) models to predict player performance based on historical data. These studies have laid a foundational understanding but often limit their scope to a small set of algorithms and basic parameter settings. Below is more detailed description of what was done in each of the three studies.

In Han (2023), an analysis was conducted where a new variable 'attack' was created as a combination of 'tackle,' 'assist,' and 'forcedFumble', serving as the dependent variable (Y). The independent variables included 'gameClockSeconds', 'playId', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber',

'nflId', 'collegeName', and 'weight'. The dataset was split into 80% train data and 20% test data before applying various machine learning algorithms. The methods used were Logistic Regression Classifier, Random Forest Classifier, Support Vector Machines (SVM) Classifier with linear, rbf, and poly kernels, and Adaboost Classifier. The model performance was evaluated based on accuracy. The reported accuracies were as follows: Logistic Regression Classifier: 0.8921, Random Forest Classifier: 0.8907, SVM Classifier with linear, rbf, and poly kernels: 0.8921, SVM Classifier with sigmoid kernel: 0.8161, and Adaboost Classifier: 0.8921.

In Chen (2023), an analysis was conducted using the dependent variable 'tackle' and independent variables 'downyards_interaction', 'player_bmi', 'quarter', 'assist', and 'forcedFumble'. The variable 'Downyards_interaction' was derived from the product of 'down' and 'yardsToGo', while 'player_bmi' was calculated as 703 multiplied by weight in pounds divided by height in inches squared. The dataset was split into three subsets: training, testing, and validation sets; 60% of the data for training, 20% for validation, and 20% for testing. Algorithms applied included Support Vector Machines (SVM), Logistic Regression, Random Forests, and Gradient Boosting. Model performance was assessed using the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve. The reported accuracies and AUC were: SVM model: accuracy = 0.8789, AUC = 0.86, Logistic Regression model: accuracy = 0.8789, AUC = 0.87, Random Forest model: accuracy = 0.8399, AUC = 0.86, and Decision Tree model: accuracy = 0.8058, AUC = 0.83.

In DuBois (2023), an analysis was conducted with the dependent variable 'tackle' and independent variables 'bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble',

'assist', and 'quarter'. The 'bmi' variable was calculated using the formula 703 multiplied by weight in pounds divided by height in inches squared. 'tackleSuccessRate' was derived from total Tackles divided by (total Tackles + missed Tackles), while 'assistSuccessRate' was calculated as total Assists divided by (total Assists + missed Tackles). The data was split into 80% for training, 10% for validation, and 10% for testing. Algorithms employed included Support Vector Machines (SVM), Naïve Bayes, K Nearest Neighbors, and Random Forest Classification. Model performance was assessed using the accuracy for each model. Reported accuracies and AUC were: SVM model: accuracy = 0.8801, AUC = 0.87, Naïve Bayes model: accuracy = 0.8795, Random Forest model: accuracy = 0.8508, AUC = 0.86, and K Nearest Neighbors model: accuracy = 0.8629.

Note that the three reports used different data splits for training and testing. The results from the different studies cannot be compared directly because the testing data are different. The portion for testing is 10% for one study and 20% for the other two studies. These proportions are relatively small which gives a lot of the data for training. Even with 80% or 90% data for training, the best result out of all these three studies only achieved AUC value of 0.87.

In this report I would like to allocate more data for testing and hope to find a model that has more than 90% for AUC. This report aims to expand on the previous studies by extensively train different machine learning models on the same data split and employing systematic hyperparameter tuning to optimize model performance. Unlike prior studies that often utilize a fixed parameter setting, this report intends to explore and compare a wide range of machine learning techniques with extensive parameter tuning. The models include the most basic K-

nearest neighbor, Decision Tree classifier, commonly used statistical models such as Regularized Logistic Regression and SVM, more complex models from ensemble methods such as Random Forest, AdaBoost, and XGBoost. The inclusion of extensive hyperparameter tuning aims to not only enhance model accuracy but also provide deeper insights into the factors influencing defensive performance in the NFL.

2. Analysis

2.1 Data Preparation

The datasets included information on games, players, plays, and tackles, each contributing essential data points for the subsequent analysis. To create a comprehensive dataset, several merging operations were performed. The tackles data was first merged with the players data based on the nflId field. This merged dataset was then combined with the plays data using the gameId and playId fields as keys. Finally, the resultant data was merged with the games data on the gameId field. This sequential merging ensured that all relevant information from the different datasets was consolidated into a single, cohesive dataset. Following the merging process, the combined data was sorted by gameId and playId. An additional variable named attack was created, which captures the maximum value across the tackle, assist, and forcedFumble columns for each play. This variable provides a new dimension for analysis, highlighting the most significant defensive actions in each play. Handling non-numeric columns was also a crucial step in the data preparation process. These columns were identified and processed to ensure they did not cause issues in subsequent analyses. For instance, the height column, initially in a feet-inches format, was converted to inches to standardize the measurements. The conversion was done by splitting the height string, extracting the feet and inches, and then calculating the total height in inches. Similarly, the gameClock column was transformed from a minute-second format to total seconds to facilitate time-based calculations. The birthDate column was converted to a datetime format, and the age of each player was calculated by subtracting the birth date from the current date and converting the result from days to years. These transformations ensured consistency and usability of the data for further analysis.

There are some of the variables that were excluded from analysis because their feature importance from prior study (Han 2023, Chen 2023, DuBois 2023) was low. These variables are 'position', 'displayName', 'ballCarrierDisplayName', 'playDescription', 'possessionTeam', 'defensiveTeam', 'yardlineSide', 'passResult', 'playNullifiedByPenalty', 'offenseFormation', 'foulName1', 'foulName2', 'gameDate', 'gameTimeEastern', 'homeTeamAbbr', 'visitorTeamAbbr'. Categorical variable 'collegeName' was converted to an integer variable using LabelEncoder, transforming text-based categories into 0 to 171 that could be processed by machine learning algorithms. Additionally, several new features were engineered:

- **down_yards_interaction:** This feature was calculated as the product of the down number and the yardsToGo, providing insight into the importance of the down situation relative to the yards needed.
- **player_bmi:** Body Mass Index (BMI) was calculated using the formula:
$$\text{BMI} = 703 \times \text{weight (lbs)} / \text{height (in)}^2$$
, giving a standardized measure of player physique.
- Success rates for tackles and assists were computed to provide insights into player performance. Tackle and assist successes and failures were counted, and the success rates were calculated as the ratio of successful actions to the total number of actions, multiplied by 100 to express the rates as percentages.

The data was then split into multiple feature sets and target variables for different models. These splits included various combinations of the engineered features and original columns, tailored to different predictive tasks. The datasets were divided into training and testing sets with a 75-25 split, ensuring sufficient data for model validation and testing. The train-test split was performed

by generating random binary variables with a specified probability of success, ensuring random yet controlled distribution of the data.

Numerical features were standardized to ensure that they had a mean of zero and a standard deviation of one, which is crucial for many machine learning algorithms. This standardization was done using `StandardScaler`. Additionally, missing values were imputed to the average using `SimpleImputer`, ensuring that no data points were lost due to missing values. This comprehensive data preparation ensured that the data was properly cleaned, transformed, and ready for analysis and modeling, providing a solid foundation for the subsequent stages.

2.2 Machine learning models

Several machine learning models will be considered in the analysis. For each model, a grid of parameter values will be defined to give a list of possible values for all or some of hyperparameters involved in the model. The model will be evaluated on the training data with cross validation for each combination of parameter settings in the grid to select the best hyperparameter values. This exhaustive search method ensures that the best combination is found but can be computationally expensive. What follows is a brief description of each model and their hyperparameters for cross validation.

2.2.1 Logistic Regression

Regularized logistic regression (Friedman et al. 2010) will be used as one of the methods to analyze the data. In regularized logistic regression, a penalty term is added to the logistic regression cost function to prevent overfitting and enhance the model's generalization

capabilities. This penalty term imposes a constraint on the magnitude of the coefficients. The two types of regularization used are L1 (Lasso) and L2 (Ridge).

- L1 Regularization (Lasso): Adds an absolute value of the coefficient ($|\beta|$) to the cost function. This can drive some coefficients to zero, effectively performing variable selection.
- L2 Regularization (Ridge): Adds the squared value of the coefficient (β^2) to the cost function. This tends to shrink the coefficients but usually does not set them to zero, allowing all features to be considered in the model.

The regularization term is controlled by the hyperparameter λ (lambda), where a higher λ increases the regularization strength, thus shrinking the coefficients more aggressively. This approach helps in reducing the model complexity and preventing overfitting, especially when dealing with high-dimensional data. By doing so, it ensures the model performs well on unseen data.

Cross-validation will be used to tune hyperparameters by splitting the dataset into multiple folds and evaluating the model's performance on each fold. The main hyperparameters to be tuned for logistic regression, particularly when regularization is applied, include:

1. Regularization Parameter (λ or C): This controls the strength of the regularization. In the context of the `cv.glmnet` function, λ is tuned across a range of values. In the `LogisticRegressionCV` from `scikit-learn`, C is the inverse of regularization strength, meaning smaller values specify stronger regularization.

2. **Penalty:** The type of regularization applied. Common penalties include 'l1' for Lasso, 'l2' for Ridge, and 'elasticnet' for a combination of both. The choice of penalty impacts the model's ability to perform feature selection (Lasso) or coefficient shrinkage (Ridge).
3. **Solver:** The algorithm used for optimization. Common solvers include 'liblinear' for small datasets, 'lbfgs' for large datasets, and 'saga' which supports both L1 and L2 penalties and works well for large datasets.
4. **Tolerance for Stopping Criteria (tol):** Determines the tolerance for stopping the optimization process. Lower values of tol make the convergence criteria stricter, which can lead to a more precise but computationally expensive model.
5. **Maximum Number of Iterations (max_iter):** Specifies the maximum number of iterations the solver should run. This is crucial to ensure that the solver does not run indefinitely and can converge to a solution.

Cross validation using R function `cv.glmnet` from packages `glmnet` was conducted with the default setting of choosing the best regularization parameter λ from 100 values using default lasso penalty and iteratively reweighted least square procedure. A separate training with Python `LogisticRegressionCV` from the `scikit_learn` package was also conducted for each dataset by choosing the best parameter from a grid of values for C in (0.001, 0.01, 0.1, 1, 10, 100, 1000), solver in ('liblinear', 'lbfgs', 'saga'), tol in (1e-4, 1e-3, 1e-2), max_iter in (100, 200, 300), and penalty in ('l1', 'l2', 'elasticnet').

2.2.2 Random Forest

Regularized Random Forest (Breiman 2001) also will be used to analyze the data. The regularization controls the model's complexity to prevent overfitting while maintaining its flexibility and predictive power. Unlike traditional logistic regression or linear models that use regularization to shrink coefficients, Random Forest regularization focuses on tree construction and ensemble behavior. The main idea is to manage the depth and size of the trees, the number of trees in the forest, and the randomness introduced during training to ensure the model generalizes well to unseen data. Below are some key parameters:

1. Tree Depth: Limiting the maximum depth of each tree prevents overfitting by stopping the tree from learning too fine details of the training data, which may not generalize well.
2. Number of Trees: Increasing the number of trees can enhance the stability and accuracy of predictions, but too many trees can lead to increased computational costs without significant performance gains.
3. Minimum Samples for Split and Leaf: Setting a minimum number of samples required to split a node or to be a leaf node controls the size of the tree, thus preventing overfitting by ensuring splits only occur when there is sufficient data.

When training a Random Forest model, these hyperparameters need to be optimized to achieve the best performance. Cross-validation is used to systematically search and evaluate the combination of hyperparameters that yield the best model performance. The hyperparameters and values include:

1. Number of Trees (`n_estimators`): This parameter specifies the number of trees in the forest. Increasing the number of trees generally improves performance, but with diminishing returns. The values to try range from 100 to 300.

2. Maximum Depth of Trees (`max_depth`): This limits the depth of each individual tree. A deeper tree can capture more information about the data but risks overfitting. Cross-validation can help find the optimal depth, with values ranging from None (no limit) to 20.
3. Minimum Samples to Split a Node (`min_samples_split`): This parameter specifies the minimum number of samples required to split an internal node. Higher values prevent the model from learning overly specific patterns that might not generalize. Values range from 2 to 10.
4. Minimum Samples per Leaf (`min_samples_leaf`): This determines the minimum number of samples required to be at a leaf node. Like `min_samples_split`, higher values can help in reducing overfitting. Values range from 3, 5, 7.

2.2.3 K-Nearest Neighbors

The key point in regularized K-Nearest Neighbors (KNN) (Cover & Hart 1967) is the optimization of hyperparameters to balance the trade-off between bias and variance, thereby improving the model's generalization capability. Unlike models such as logistic regression, KNN does not have a built-in regularization mechanism. Instead, regularization in KNN is implicitly achieved by tuning key hyperparameters like the number of neighbors (k), the distance metric, and the weighting function. These hyperparameters control the complexity of the model and its sensitivity to noise in the data. By carefully selecting these parameters through cross-validation, we can prevent overfitting where the model is too sensitive to the training data and underfitting where the model is too simplistic to capture underlying patterns.

Several hyperparameters need to be optimized for the KNN algorithm to perform well. These include:

1. Number of Neighbors (k): The primary hyperparameter in KNN is the number of neighbors (k) to consider when making a prediction. A small k value (e.g., 1 or 3) can lead to high variance and overfitting, as the model may become too sensitive to individual data points. Conversely, a large k value can lead to high bias and underfitting, as the model may smooth over important patterns in the data. Cross-validation helps determine the optimal k value that balances this trade-off.
2. Distance Metric: KNN relies on a distance metric to determine the neighbors. Distance metrics include Euclidean, Manhattan distances. The choice of distance metric can significantly impact the model's performance. For example, Euclidean distance is sensitive to differences in scale between features, whereas Manhattan distance can be more robust to such differences. Cross-validation can help select the most appropriate distance metric for the given dataset.
3. Weighting Function: The weighting function determines how each neighbor contributes to the final prediction. In a standard KNN, each neighbor contributes equally (uniform weights). However, weighted KNN allows closer neighbors to have more influence, often using an inverse distance weighting scheme. Cross-validation can evaluate whether uniform or distance-based weighting yields better performance.

To perform cross-validation for KNN, the following steps are involved:

1. Grid Search: A range of k values (3, 5, 7, 9, 11, 13), different distance metrics (Euclidean, Manhattan), and weighting schemes (uniform, distance-based) are tested.

2. Evaluation Metrics: Performance metrics such as accuracy, AUC are used to evaluate the model's performance for each combination of hyperparameters. The set of hyperparameters that yields the highest performance metrics is selected as the optimal configuration.

2.2.4 AdaBoost

AdaBoost, or Adaptive Boosting (Freund & Schapire 1995), is an ensemble learning method that combines multiple weak learners to create a strong learner. Regularization in AdaBoost primarily involves managing the trade-off between bias and variance by tuning hyperparameters such as the learning rate and the number of estimators. The learning rate reduces the contribution of each weak learner, preventing overfitting by smoothing the model, while the number of estimators controls the complexity by determining how many weak learners are combined.

When optimizing AdaBoost, the main hyperparameters to be tuned through cross-validation include:

1. Number of Estimators (`n_estimators`): This parameter defines the number of weak learners (typically decision trees) to be combined in the model. Increasing the number of estimators usually enhances the model's performance, but after a certain point, it can lead to overfitting.
2. Learning Rate (`learning_rate`): This parameter controls the contribution of each weak learner by scaling the weights of newly added weak learners. A lower learning rate reduces the impact of each learner, leading to better generalization.

Grid Search: A range of values for `n_estimators` (50, 100, 200) and `learning_rate` (0.001, 0.01, 0.1, 1) are evaluated to identify the optimal combination.

2.2.5 Support Vector Machine

Regularization in SVM (Cortes & Vapnik 1995) aims to prevent overfitting by balancing the margin maximization with the penalty for misclassifications. This is achieved through the regularization parameter C , which determines the trade-off between maximizing the margin and minimizing classification errors. A lower C value increases the margin but allows more misclassifications, reducing overfitting, while a higher C value aims for perfect classification at the risk of overfitting. Additionally, kernel functions (linear, polynomial, RBF, etc.) can transform the input space to higher dimensions, allowing for complex decision boundaries. The use of appropriate kernels and tuning their parameters is also a crucial aspect of regularizing SVM.

Several hyperparameters need to be optimized for the SVM model to achieve optimal performance. These include:

1. **Regularization Parameter (C):** This parameter controls the trade-off between achieving a low error on the training data and minimizing the model complexity. A lower value of C makes the decision surface smooth, while a higher value aims to classify all training examples correctly by giving the model freedom to select more complex surfaces. Values for C range from 0.1 to 10.

2. Kernel: The kernel function defines the type of hyperplane used to separate the data.

Common kernels include:

- Linear: Suitable for linearly separable data.
 - Polynomial (poly): Useful for non-linear data by adding polynomial features.
 - Radial Basis Function (RBF): Maps data to a higher dimension, effective for non-linear data.
3. Gamma (γ): For RBF, polynomial, and sigmoid kernels, γ defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. It can significantly impact the performance of the model. Values to evaluate include 'scale' ($1 / (n_features * X.var())$) and 'auto' ($1 / n_features$).

Grid Search: The grid of hyperparameters in cross validation is defined by varying C values (0.1, 1, 10), kernel types ('linear', 'poly', 'rbf'), gamma values ('scale', 'auto').

2.2.6 Decision Tree

Decision trees (Fürnkranz 2011) are prone to overfitting as they can grow very deep and capture noise in the training data. Regularization techniques for decision trees include setting constraints on the tree growth such as maximum depth of the tree (`max_depth`), the minimum number of samples required to split a node (`min_samples_split`), and the minimum number of samples required at a leaf node (`min_samples_leaf`). These parameters help in simplifying the model, ensuring it generalizes better to unseen data, and reducing the risk of overfitting by preventing the tree from becoming overly complex.

When optimizing a Decision Tree model, the following hyperparameters are tuned to achieve optimal performance:

1. Maximum Depth (`max_depth`): This parameter sets the maximum depth of the tree.

Limiting the depth of the tree helps prevent it from becoming too complex and overfitting the training data.

2. Minimum Samples Split (`min_samples_split`): This parameter specifies the minimum number of samples required to split an internal node. Higher values prevent the model from learning overly specific patterns that do not generalize well.

3. Minimum Samples Leaf (`min_samples_leaf`): This parameter determines the minimum number of samples that a leaf node must have. This helps in smoothing the model, especially in regression trees.

Grid Search: A grid of hyperparameters is evaluated using cross validation. The search compares all possible combinations of `max_depth` (5, 10, 15, None), `min_samples_split` (2, 5, 10), and `min_samples_leaf` (1, 2, 4) to find the best hyperparameter values.

2.2.7 XGBoost

XGBoost (Extreme Gradient Boosting) (Chen & Carlos 2016) is an advanced implementation of gradient boosting that incorporates several regularization techniques to enhance generalization and prevent overfitting. The primary regularization methods in XGBoost include L1 and L2 regularization (controlled by `reg_alpha` and `reg_lambda`, respectively), which penalize large coefficients, and the tree-specific parameters that control the structure of the boosting trees.

These parameters help to constrain the model complexity, ensuring that the model does not overfit to the training data and maintains good performance on unseen data.

Optimizing XGBoost involves tuning several hyperparameters that significantly affect the model's performance. The key hyperparameters include:

1. **Number of Estimators (`n_estimators`):** This parameter controls the number of boosting rounds or trees. More estimators can improve performance but may also lead to overfitting.
2. **Learning Rate (`learning_rate`):** This parameter determines the step size at each iteration while moving towards a minimum of the loss function. A lower learning rate generally requires more trees to achieve good performance.
3. **Maximum Depth (`max_depth`):** This parameter sets the maximum depth of each tree. Deeper trees can model more complex patterns but may overfit the training data.
4. **Minimum Child Weight (`min_child_weight`):** This parameter specifies the minimum sum of weights of all observations required in a child. It helps in controlling overfitting. Higher values lead to more conservative models.
5. **Regularization Parameters (`reg_lambda`):** These control the L1 and L2 regularization terms, respectively. Higher values lead to more regularization.
6. **Minimum Split Loss (`min_split_loss`):** This parameter controls the minimum reduction in the loss function required to make a further partition on a leaf node of the tree. Higher values make the algorithm more conservative by preventing splits that do not significantly reduce the loss, which can help reduce overfitting. Lower values allow more splits, potentially increasing model complexity.

To perform cross-validation for XGBoost, the following is done:

Grid Search: Define a grid of hyperparameters and evaluate all possible combinations using cross-validation on varying values for `n_estimators` (2, 4, 6, 8, 10, 12, 14, 16, 18), `learning_rate` (0.001, 0.01, 0.1, 1.0), `max_depth` (1, 2, 3, 4, 5, 6, 7), `min_child_weight` (2), `min_split_loss` (0), and `reg_lambda` (0.1, 0.5, 1, 10).

3. Results

Each model is applied to the six datasets. For each dataset, each of the models was trained on the training data with grid search for the best hyperparameters, which are then used in the final model to predict the response variable on the test data. This section lists the results from each model one by one. Specifically, the ROC curves from each model on the six datasets will be presented along with the tables of selected hyperparameters. Result from Logistic Regression will be presented first.

Figure 1 shows the ROC (Receiver Operating Characteristic) curve for the Logistic Regression classifier applied to six different test datasets after training and parameter tuning with the training data. The ROC curve is essential for evaluating the model's diagnostic ability by plotting the True Positive Rate against the False Positive Rate at various threshold settings. The AUC (Area Under the Curve) values for these ROC curves range from 0.517 to 0.919, indicating varying levels of classification performance. An AUC of 0.5 suggests no discriminative power, equivalent to random guessing, whereas an AUC close to 1.0 signifies excellent discrimination. The increasing trend in AUC values from Dataset 1 to Dataset 6, progressing from 0.517 to 0.919, reflects significant improvement in model performance. The ROC curves for later datasets show a steep rise towards the top-left corner, indicating a higher True Positive Rate and a lower False Positive Rate, thus showcasing better performance.

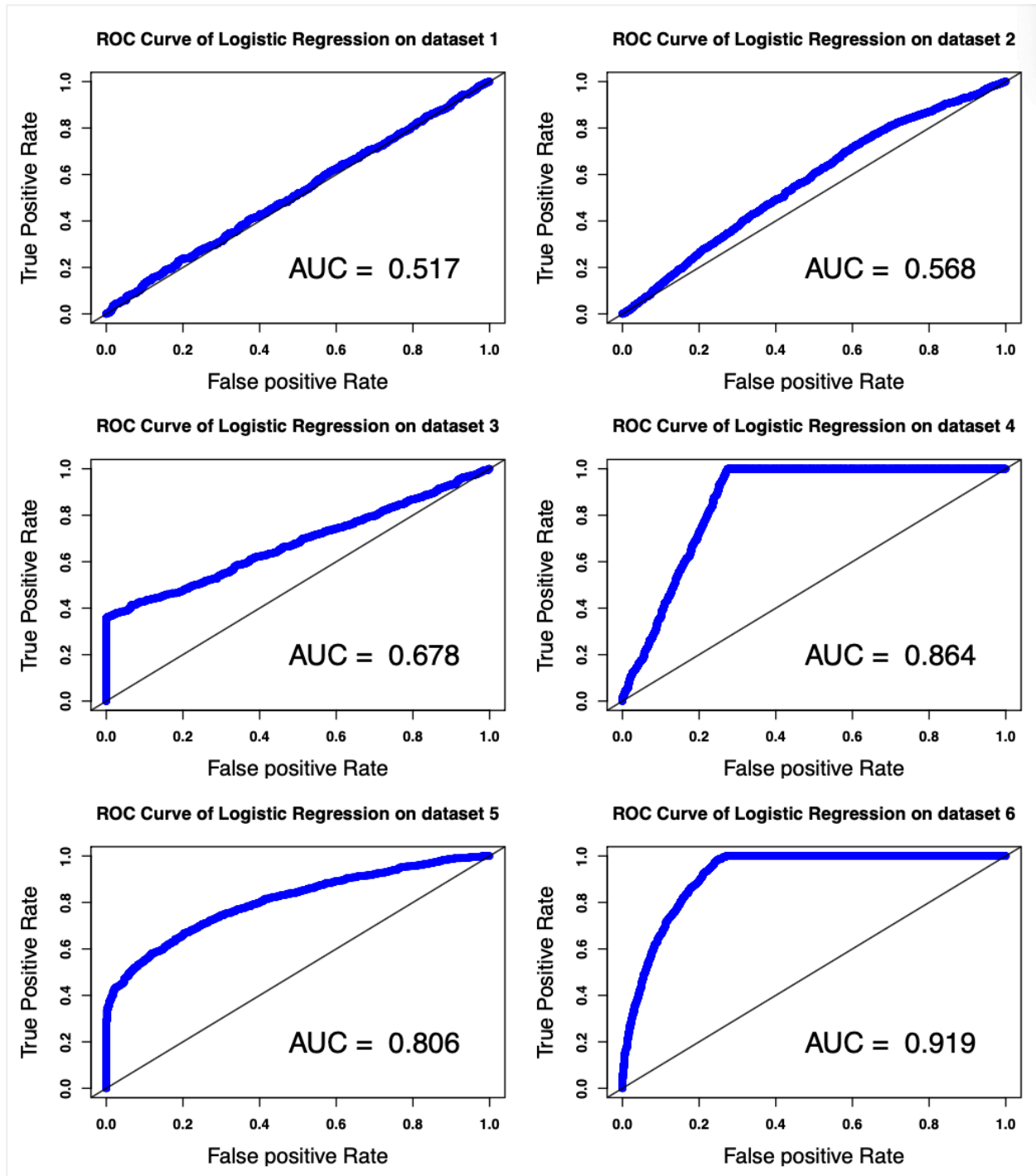


Figure 1. ROC Curve of Logistic Regression classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.52 to 0.92 on Dataset 1 to 6.

Recall what variables are in the six datasets, we can get an idea of what features are helpful to predict which response variable. Below is a list of the X's and Y's in the six datasets:

```
feature_data1 =
```

```
data[['gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', 'weight']]
```

```
Dataset1: Y = attack, X = feature_data1
```

```
Dataset2: Y = tackle, X = feature_data1
```

```
feature_data2 = data[['down_yards_interaction', 'player_bmi', 'quarter', 'assist', 'forcedFumble']]
```

```
Dataset3: Y = attack, X = feature_data2
```

```
Dataset4: Y = tackle, X = feature_data2
```

```
feature_data3 =
```

```
data[['player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', 'quarter']]
```

```
Dataset5: Y = attack, X = feature_data3
```

```
Dataset6: Y = tackle, X = feature_data3
```

Between Dataset 1 and 2, we can see that the logistic regression model makes better prediction for tackle than for attack using the same set of feature variables. The same pattern is observed between Dataset 3 and 4, and between Dataset 5 and 6. In Dataset 1, 3, and 5, the same response variable attack was used but with different X features. Note the difference between feature_data2 and feature_data3 is in 'age', 'tackleSuccessRate', 'assistSuccessRate' versus 'down_yards_interaction' while the rest of the variables are common. It tells that 'age', 'tackleSuccessRate', 'assistSuccessRate' carry more information about both tackle and attack than 'down_yards_interaction'. Those in feature_data1 on the other hand, don't really have a lot of

predictive power for either tackle or attack when logistic regression was used with different cut off threshold.

Dataset	Y	X	Test accuracy	AUC	Best hyperparameters from training
1	attack	feature_data1	0.8827	0.5000	C=0.001, penalty=l1, solver=liblinear, tol=0.0001, max_iter=100
2	tackle	feature_data1	0.5888	0.5695	C=1.000, penalty=l2, solver=saga, tol=0.0100, max_iter=100
3	attack	feature_data2	0.8827	0.6755	C=0.001, penalty=l1, solver=liblinear, tol=0.0001, max_iter=100
4	tackle	feature_data2	0.8827	0.8627	C=0.001, penalty=l1, solver=liblinear, tol=0.0001, max_iter=100
5	attack	feature_data3	0.8847	0.8031	C=0.010, penalty=l1, solver=liblinear, tol=0.0001, max_iter=100
6	tackle	feature_data3	0.8845	0.9078	C=0.001, penalty=l2, solver=liblinear, tol=0.0001, max_iter=100

Table 1. Performance metrics and hyperparameters for logistic regression classifier across six datasets from Python LogisticRegressionCV.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training in R cv.glmnet
1	attack	feature_data1	0.8826	0.517 (0.500, 0.500)	$\lambda=0.00589$
2	tackle	feature_data1	0.5865	0.568 (0.551, 0.585)	$\lambda=0.00625$
3	attack	feature_data2	0.8826	0.678 (0.659, 0.697)	$\lambda=0.00022$
4	tackle	feature_data2	0.8826	0.864 (0.851, 0.876)	$\lambda=0.00110$
5	attack	feature_data3	0.8838	0.806 (0.789, 0.822)	$\lambda=0.00052$
6	tackle	feature_data3	0.8838	0.919 (0.910, 0.928)	$\lambda=0.00100$

Table 2. Performance metrics and hyperparameters for lasso logistic regression classifier across six datasets. These are the result trained with R function `cv.glmnet` using default l2 penalty. The results from Python are not better than those from R even though they were trained with other hyperparameter values.

Table 1 presents the test accuracy, AUC values, and best hyperparameters for the Logistic Regression classifier across six different datasets. This analysis highlights the significance of hyperparameter tuning in enhancing the model's performance. In Dataset 1, the Logistic Regression classifier achieved a test accuracy of 0.8827, indicating that using threshold 0.5 to make the prediction is good for Dataset 1. The model on this dataset has an AUC of 0.5000, indicating that using other thresholds to make the prediction is no better than random guessing. The best hyperparameters for this set was determined to be: a regularization parameter (C) of 0.001, an l1 penalty, and the liblinear solver. Compared to Dataset1, Dataset 2 showed a significant drop in test accuracy to 0.5888, with an AUC of 0.5695, suggesting that logistic regression with Dataset 2 almost has no discriminatory ability even though the best hyperparameters were adjusted to a C value of 1.000, an l2 penalty, and the saga solver. Compared to Dataset 1, Dataset 3 maintained the same test accuracy of 0.8827 but improved its AUC to 0.6755, showing moderate discrimination. The best hyperparameters reverted to those used in Dataset 1. Dataset 4 exhibited a notable improvement with an AUC of 0.8627, indicating good model performance, while maintaining a test accuracy of 0.8827. This model used the same hyperparameters as for Datasets 1 and 3, underscoring their robustness. Dataset 5 further enhanced performance with a test accuracy of 0.8847 and an AUC of 0.8031, reflecting good discrimination. The best hyperparameter adjustment included a C value of 0.010 while keeping

other parameter settings constant. Finally, Dataset 6 demonstrated the best performance with a test accuracy of 0.8845 and an AUC of 0.9078, signifying excellent discriminative ability. The best hyperparameters for this set included a C value of 0.001, an l2 penalty, and the liblinear solver.

After seeing the above result, we can see that best data among the three feature sets to predict Defensive performance using Logistic Regression are the feature data3. The final model for predicting tackle is:

$$\begin{aligned} \log \text{ odds} = & 0.290962438388591 + 0.000494070602324109 \text{ player_bmi} + 0.0038995560914854 \\ & \text{tackleSuccessRate} + 0.00751655430609551 \text{ assistSuccessRate} + 0.104944937281435 \\ & \text{forcedFumble} - 0.822553630467777 \text{ assist} - 0.00377262997020573 \text{ quarter.} \end{aligned}$$

Note that age of player is not in the final selected model. This means among the player in the given data the age difference did not contribute to the prediction of player performance. Each of the regression slope represents the log odds ratio of success over failure associated with one unit increase in the corresponding predictor variables. For example, one unit increase in player_bmi is associated with an increased odds of success in tackle by $\exp(0.000494070602324109) - 1 = 0.0004941927$. If a player made an assist tackle on the play, the odds of success in tackle changes by $\exp(-0.822553630467777) - 1 = -0.5606916$. A player's odds of success in tackle for the first quarter is higher than those in second, third and fourth quarter because the coefficient for quarter is negative. Other than assist and quarter, all other variables contribute positively to the odds of success in tackle.

Figure 2 presents the ROC curves for the Random Forest classifier applied to the six different test datasets following training and parameter tuning with the training data. The AUC values in these ROC curves range from 0.52 to 0.91, illustrating the model's varying levels of performance across different datasets. The trend of increasing AUC values from Dataset 1 to Dataset 6, moving from 0.52 to 0.91, was observed like in Figure 1. This improvement underscores the Random Forest classifier's capability to better predict tackle or attack using 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', 'quarter' than using 'gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', 'weight'. The curve for Dataset 6 reaches the best performance by edging closer to the top-left corner of the figure, highlighting a high True Positive Rate and a low False Positive Rate.

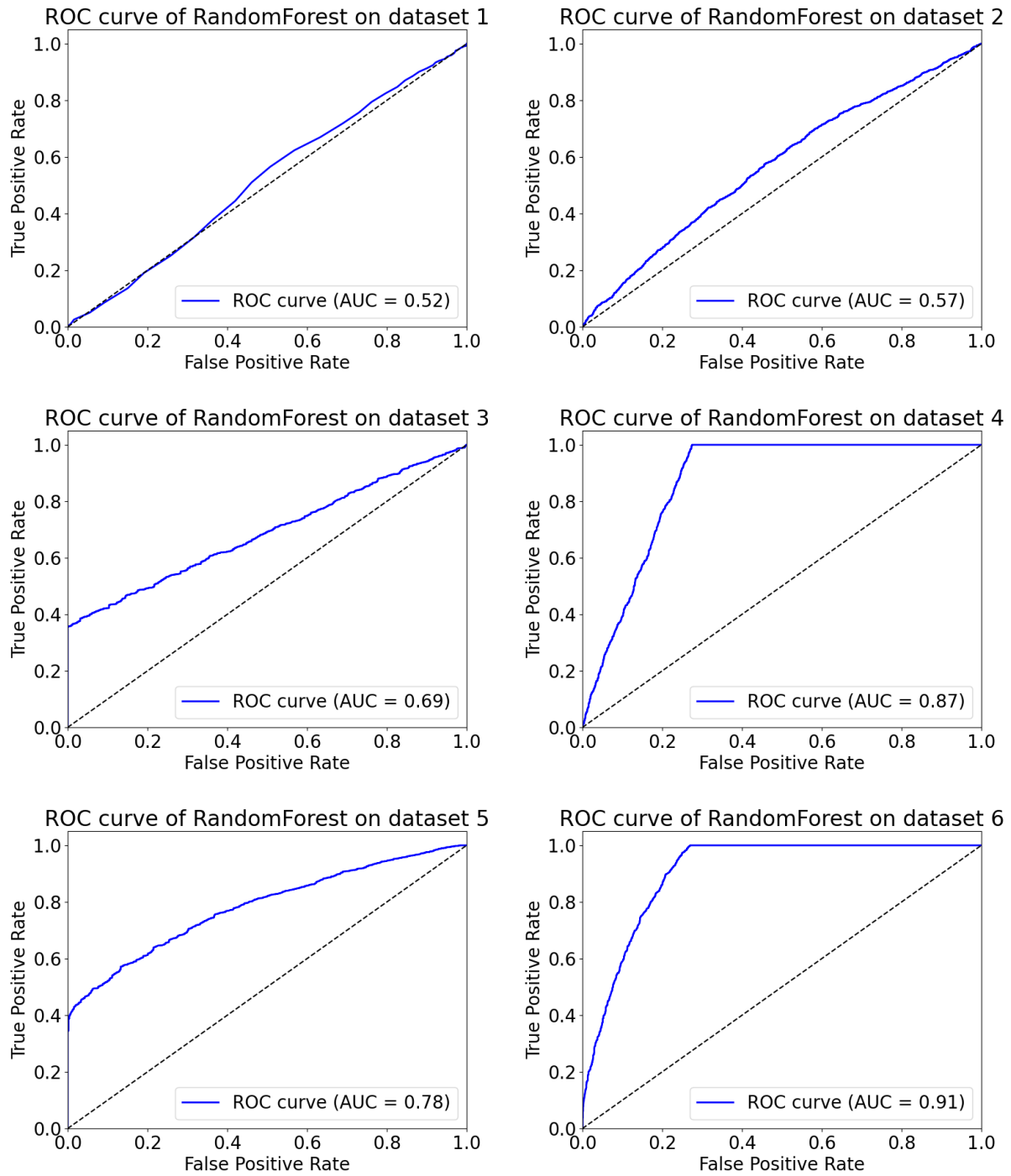


Figure 2. ROC Curve of Random Forest classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.52 to 0.91 on Dataset 1 to 6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training
1	attack	feature_data1	0.8826	0.5141 (0.487, 0.541)	max_depth=None, min_samples_leaf=3, min_samples_split=2, n_estimators=100
2	tackle	feature_data1	0.5785	0.5727 (0.556, 0.590)	max_depth=10, min_samples_leaf=5, min_samples_split=2, n_estimators=200
3	attack	feature_data2	0.8826	0.6881 (0.669, 0.707)	max_depth=None, min_samples_leaf=7, min_samples_split=2, n_estimators=100
4	tackle	feature_data2	0.8826	0.8676 (0.855, 0.880)	max_depth=None, min_samples_leaf=7, min_samples_split=2, n_estimators=100
5	attack	feature_data3	0.8845	0.7851 (0.768, 0.801)	max_depth=10, min_samples_leaf=7, min_samples_split=10, n_estimators=100
6	tackle	feature_data3	0.8845	0.9070 (0.897, 0.917)	max_depth=10, min_samples_leaf=7, min_samples_split=2, n_estimators=300

Table 3. Performance metrics and hyperparameters for Random Forest classifier across the six datasets

Table 3 shows the test accuracy, AUC values, and best hyperparameters for the Random Forest classifier across the six different datasets. In Dataset 1, the Random Forest classifier achieved a test accuracy of 0.8826, indicating that Random Forest Classifier has good predictive power using threshold 0.5 to make the prediction. Similar to Logistic Regression, the Random Forest model on this dataset has an AUC of 0.5141, reflecting minimal discriminative ability when using other thresholds to predict success or failure. The best hyperparameters for this set were determined to be: a max depth of None, a minimum samples leaf of 3, a minimum samples split of 2, and 100 estimators. Compared to Dataset 1, Dataset 2 showed a significant drop in test accuracy to 0.5785, with an AUC of 0.5727, suggesting that Random Forest with Dataset 2 almost has no discriminatory ability even though the best hyperparameters were adjusted to the

max depth of 10 and increasing the number of estimators to 200. After a lot of hyperparameter tuning, the best model is equivalent to a flipping coin random guess. This suggests that the Dataset 2's predictive variables do not have much information about tackle even though they could make reasonable prediction for attack with threshold 0.5. In Dataset 3, the test accuracy increased to 0.8826, with a moderate AUC of 0.6881, using similar hyperparameters as Dataset 2 but with fewer estimators. Dataset 4 exhibited a notable improvement with an AUC of 0.8676, indicating good model performance, while maintaining a test accuracy of 0.8826. This model used adjusted hyperparameters with a max depth of None, a minimum samples leaf of 7, and 100 estimators. Performance on Dataset 5 is further enhanced with a test accuracy of 0.8845 and an AUC of 0.7851, reflecting better discrimination on attack than using the feature variables in Dataset 3 or 1. The best hyperparameter adjustment included a max depth of 10, a minimum samples leaf of 7, a min_samples_split of 10, and 100 estimators. Finally, Dataset 6 demonstrated the best performance with a test accuracy of 0.8845 and an AUC of 0.9070, signifying excellent discriminative ability. The best hyperparameters for this model included a max depth of 10, a minimum number of samples on the leaf of 7, a min_samples_split of 2, and 300 estimators. Logistic regression has no interaction included in the model but achieved similar results as Random Forest. However, the Random Forest model allowed up to 10-way interaction to be included because max_depth= 10, which has tree depth up to 10 and allows to use potentially 10 different variables to partition the data's sample space. The interactions did not help improve the prediction. Additionally, Random Forest has many trees in the model, which makes the model interpretation problematic. On the other hand, Logistic Regression has very straight forward interpretation as the slope of regression parameter represents the log odds of success over failure associated with one unit increase in the predictive variable.

Figure 3 illustrates the ROC curves for the K-Nearest Neighbors (KNN) classifier across the six different test datasets post training and parameter tuning. The AUC values increase from 0.50 in Dataset 1 to 0.90 in Dataset 6 was observed like in Figure 1. These improvement shows that the KNN classifier performs well when predicting tackle and attack using ‘player_bmi’, ‘age’, ‘tackleSuccessRate’, ‘assistSuccessRate’, ‘forcedFumble’, ‘assist’, ‘quarter’ than using ‘gameClockSeconds’, ‘playID’, ‘preSnapHomeTeamWinProbability’, ‘preSnapVisitorTeamWinProbability’, ‘yardlineNumber’, ‘nflId’, ‘collegeName’, ‘weight’. For Dataset 6, the curve indicates best performance by approaching the top-left corner of the diagram, denoting a high True Positive Rate and a low False Positive Rate.

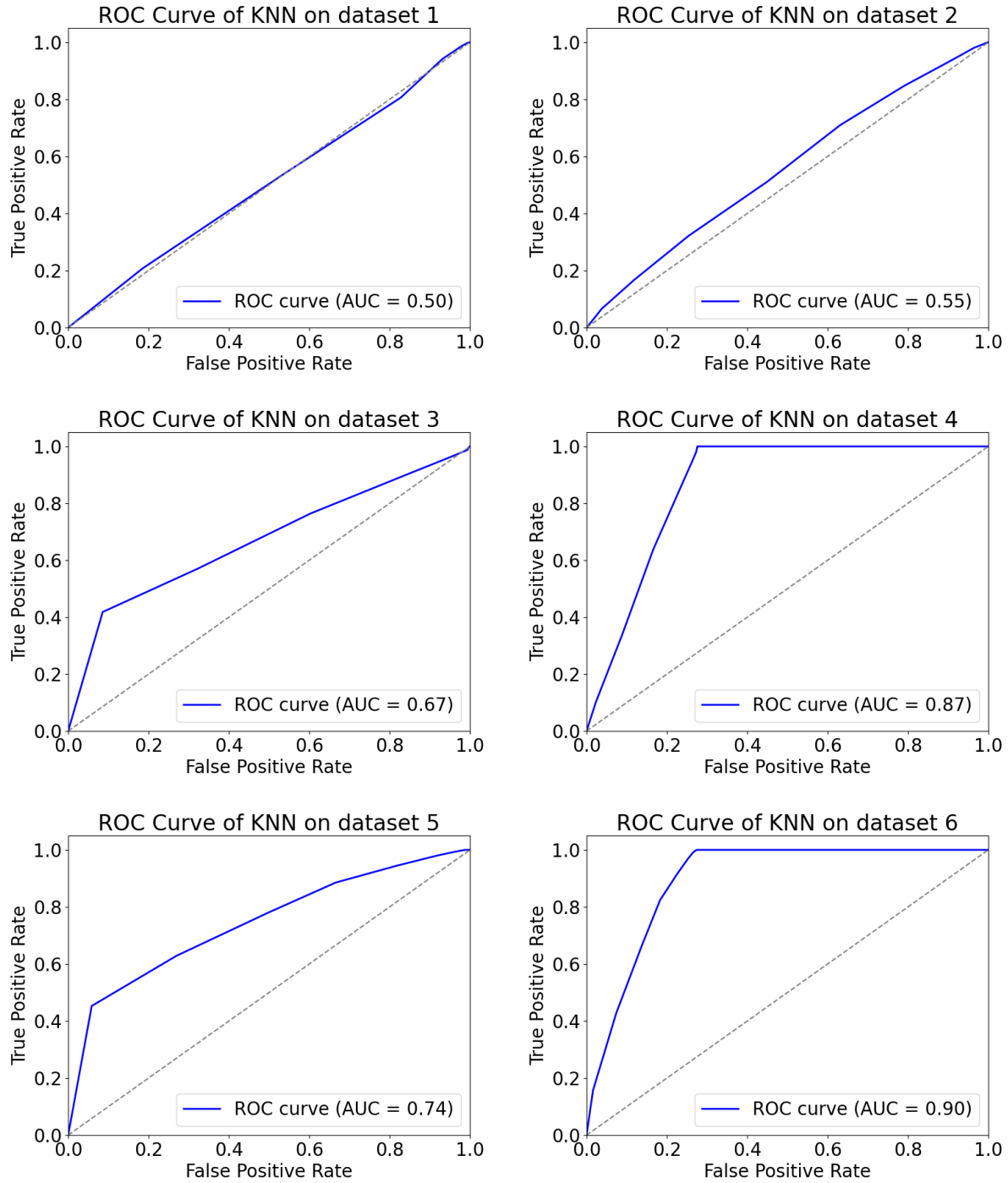


Figure 3. ROC Curve of KNN classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.50 to 0.90 on Dataset 1 to 6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training
1	attack	feature_data1	0.8824	0.5033 (0.478, 0.528)	metric=manhattan, n_neighbors=13, weights=uniform
2	tackle	feature_data1	0.5641	0.5542 (0.537, 0.571)	metric=euclidean, n_neighbors=13, weights=uniform
3	attack	feature_data2	0.8792	0.6705 (0.650, 0.691)	metric=euclidean, n_neighbors=13, weights=uniform
4	tackle	feature_data2	0.8788	0.8669 (0.855, 0.879)	metric=euclidean, n_neighbors=13, weights=uniform
5	attack	feature_data3	0.8808	0.7448 (0.722, 0.761)	metric=euclidean, n_neighbors=13, weights=uniform
6	tackle	feature_data3	0.8808	0.8963 (0.887, 0.908)	metric=euclidean, n_neighbors=13, weights=uniform

Table 4. Performance metrics and hyperparameters for KNN classifier across the six datasets.

Table 4 outlines the performance metrics and best hyperparameters for the KNN classifier across the six datasets. The test accuracy has lowest value 0.5641 for Dataset 2 and highest value 0.8824 for Dataset 1. The AUC values, which indicate the classifier's ability to discriminate between classes using different thresholds, showed significant improvement from 0.5033 in Dataset 1 to 0.8963 in Dataset 6. The following is a discussion of the performance on each Dataset one by one. For Dataset 1, the KNN classifier achieved a test accuracy of 0.8824 indicating that using threshold 0.5 to make the prediction is good for Dataset 1. The model on this dataset has an AUC of 0.5033, indicating that its performance is not better than random guessing when other thresholds are used in prediction success or failure. The best hyperparameters for this model included metric=manhattan, n_neighbors=13, weights=uniform. Dataset 2 exhibited a significant drop in test accuracy to 0.5641 but a slight improvement in

AUC to 0.5542, achieved by switching the distance metric to Euclidean. Dataset 3 saw an increase in test accuracy to 0.8792 and a moderate AUC of 0.6705, using the same hyperparameters as Dataset 2. The mode for Dataset 4 achieved a high test accuracy of 0.8788 and showed substantial improvement in AUC to 0.8669, highlighting better predictive power of the variables in feature_data2 compared to those in feature_data1. In Dataset 5, the model's test accuracy slightly improved to 0.8808 with an AUC of 0.7448, indicating good model performance. Finally, Dataset 6 achieved the best results, with a test accuracy of 0.8808 and an AUC of 0.8963, using the Euclidean distance metric, 13 neighbors, and uniform weights. This tells that the variables in feature_data3 carry more information on attack and tackle than those in feature_data2 or 1 for KNN classifier just like for other classifiers discussed earlier.

Figure 4 shows the ROC curves for the AdaBoost classifier applied to six different test datasets following comprehensive training and parameter tuning. The AUC values for these ROC curves range from 0.51 to 0.92, showcasing varying levels of classification performance. The upward trend in AUC values from Dataset 1 to Dataset 6, increasing from 0.51 to 0.92 shows the same pattern as in Figure 1. This tells that the AdaBoost classifier's makes better prediction on tackle or attack using 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', 'quarter' than using 'gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', 'weight'. The curve for Dataset 6 achieves the best performance by trending towards the top-left corner of the chart, which reflects a high True Positive Rate and a low False Positive Rate.

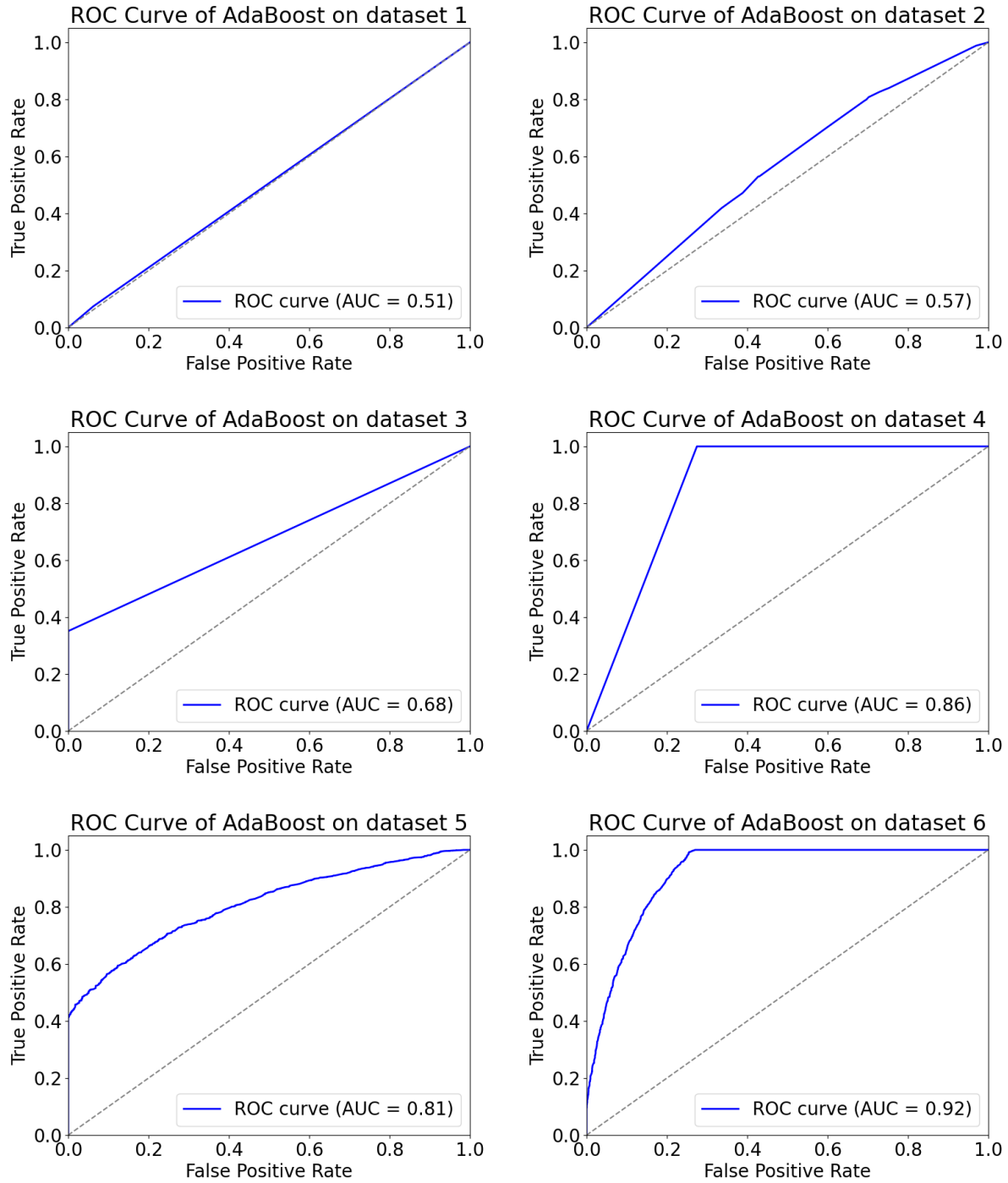


Figure 4. ROC Curve of AdaBoost classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.92 on Dataset 1 to 6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training
1	attack	feature_data1	0.8826	0.5060 (0.495, 0.517)	learning_rate=0.001, n_estimators=50
2	tackle	feature_data1	0.5900	0.5666 (0.550, 0.583)	learning_rate=0.01, n_estimators=200
3	attack	feature_data2	0.8826	0.6755 (0.668, 0.683)	learning_rate=0.001, n_estimators=50
4	tackle	feature_data2	0.8826	0.8627 (0.853, 0.873)	learning_rate=0.001, n_estimators=50
5	attack	feature_data3	0.8852	0.8083 (0.793, 0.824)	learning_rate=0.1, n_estimators=200
6	tackle	feature_data3	0.8847	0.9190 (0.910, 0.928)	learning_rate=0.1, n_estimators=200

Table 5. Performance metrics and hyperparameters for AdaBoost classifier across the six datasets.

Table 5 outlines the performance metrics and best hyperparameters for the AdaBoost classifier across the six datasets. The test accuracy ranged from a low of 0.5900 in Dataset 2 to a high of 0.8847 in Dataset 6. The AUC values, indicating the classifier's ability to discriminate between classes using different thresholds, showed significant improvement from 0.5060 in Dataset 1 to 0.9190 in Dataset 6. Below is a detailed discussion of the performance on each dataset. For Dataset 1, the AdaBoost classifier achieved a test accuracy of 0.8826, suggesting that a threshold of 0.5 is effective for predictions in this dataset. However, the model's AUC was 0.5060, indicating that its performance is not better than random guessing when other thresholds are used. Performance on the best hyperparameters included a learning rate of 0.001 and 50 estimators. Dataset 2 experienced a significant drop in test accuracy to 0.5900 but saw a slight improvement in AUC to 0.5666. This was achieved by increasing the learning rate to 0.01 and the number of estimators to 200. In Dataset 3, the test accuracy remained at 0.8826, with a moderate AUC of 0.6755, using the same hyperparameters as in Dataset 1. Performance on Dataset 4 also see a high test accuracy of

0.8826, but its AUC improved substantially to 0.8627, indicating better discriminative power with the same hyperparameters. For Dataset 5, the model's test accuracy improved slightly to 0.8852, with an AUC of 0.8083, reflecting good performance by adjusting the learning rate to 0.1 and using 200 estimators. Finally, the model achieved the best results on Dataset 6, with a test accuracy of 0.8847 and an AUC of 0.9190, using a learning rate of 0.1 and 200 estimators. This indicates that the variables in feature_data3 provide more informative features for distinguishing between success and failure in tackle than those in feature_data2 or feature_data1.

Figure 5 presents the ROC curves for the Support Vector Machine (SVM) classifier across the six different test datasets after extensive training and parameter tuning. The AUC values range from 0.51 to 0.90, reflecting varying degrees of classification performance. The trend of increasing AUC values from Dataset 1 to Dataset 6, rising from 0.51 to 0.90 shows same pattern observed in Figure 1. This tells that SVM also has superior ability to predict tackle or attack outcomes using variables 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', 'quarter' than using 'gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', 'weight'. The curve for Dataset 6 shows the peak performance by moving towards the top-left corner of the graph, indicating a high True Positive Rate and a low False Positive Rate.

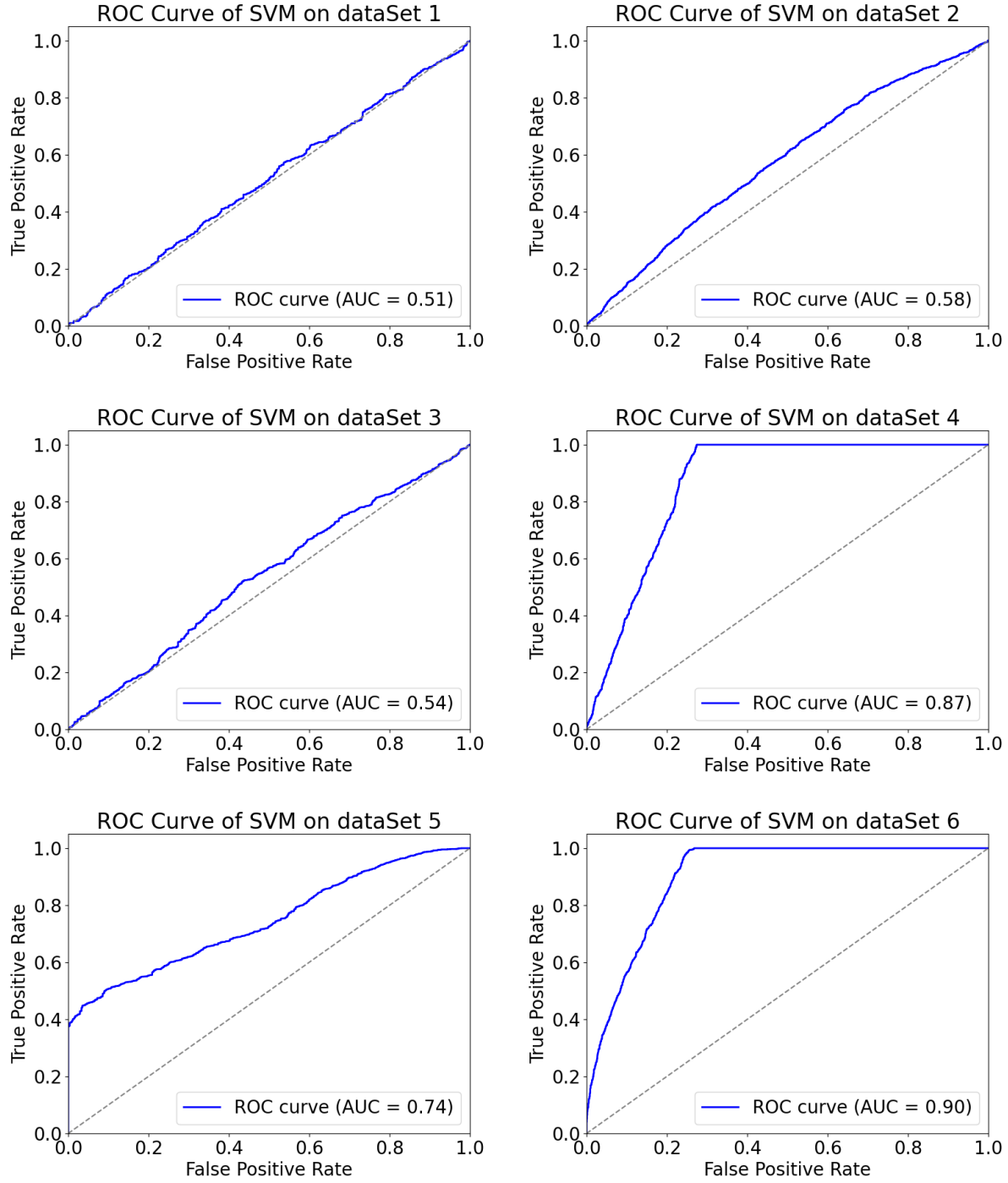


Figure 5. ROC Curve of SVM classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.90 on Dataset 1 to 6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training
1	attack	feature_data1	0.8826	0.5128 (0.486, 0.539)	'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'
2	tackle	feature_data1	0.5854	0.5763 (0.559, 0.593)	C': 0.1, 'gamma': 'auto', 'kernel': 'rbf'
3	attack	feature_data2	0.8826	0.5359 (0.509, 0.563)	'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'
4	tackle	feature_data2	0.8826	0.8663 (0.854, 0.879)	'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'
5	attack	feature_data3	0.8845	0.7443 (0.726, 0.763)	'C': 10, 'gamma': 'scale', 'kernel': 'poly'
6	tackle	feature_data3	0.8840	0.9019 (0.893, 0.912)	'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'

Table 6. Performance metrics and hyperparameters for SVM classifier across the six datasets.

Table 6 presents the performance metrics and best hyperparameters for the SVM classifier across the six Datasets. The test accuracy varied from 0.5854 in Dataset 2 to 0.8840 in Dataset 6, while the AUC values, which measure the classifier's ability to discriminate between classes, improved from 0.5128 in Dataset 1 to 0.9019 in Dataset 6. In Dataset 1, the SVM classifier achieved a test accuracy of 0.8826 with an AUC of 0.5128, indicating that its performance is no better than random guessing when different thresholds other than 0.5 are used even though it works well with threshold 0.5. The best hyperparameters for this set included a regularization parameter (C) of 0.1, 'gamma' set to 'scale', and a linear kernel. Performance on Dataset 2 exhibited a significant drop in test accuracy to 0.5854 and a slight improvement in AUC to 0.5763, achieved by adjusting the 'gamma' to 'auto' and using a rbf kernel. Comparing Dataset 1 and 2, the difference is in the response variable ('attack' versus 'tackle'). This means the same set of

predictor variables in `feature_data1` could provide information for attack but barely provide anything for predicting tackle using SVM. SVM on Dataset 3 had similar performance in that the test accuracy is 0.8826 and AUC is 0.5359, with the same hyperparameters as for Dataset 1. Performance on Dataset 4 showed significant improvement, with test accuracy of 0.8826 and an AUC of 0.8663, indicating better discriminative performance in predicting tackle with `feature_data2`. For Dataset 5, the model's test accuracy is 0.8845 with an AUC of 0.7443, using a higher regularization parameter ($C=10$) and a poly kernel with 'gamma' set to 'scale'. Finally, the model on Dataset 6 achieved the best results, with a test accuracy of 0.8840 and an AUC of 0.9019, using the same poly kernel with 'gamma' set to 'scale' but with a regularization parameter (C) of 0.1. From these results, we see that SVM model predict tackle well with the variables in `feature_data3`.

Figure 6 presents the ROC curves for the Decision Tree classifier applied the six datasets following training and parameter tuning with the training data. The AUC values for these ROC curves range from 0.51 to 0.91, indicating varying levels of classification performance. Similar to the pattern observed in Figure 1, there is an increasing trend in AUC values from Dataset 1 to Dataset 6, rising from 0.51 to 0.91, which highlights a significant improvement in model prediction of tackle or attack using 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', 'quarter' than using 'gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', 'weight'. The curve for Dataset 6 demonstrates the best performance by approaching the top-left corner of the plot, indicating a high True Positive Rate and low False Positive Rate.

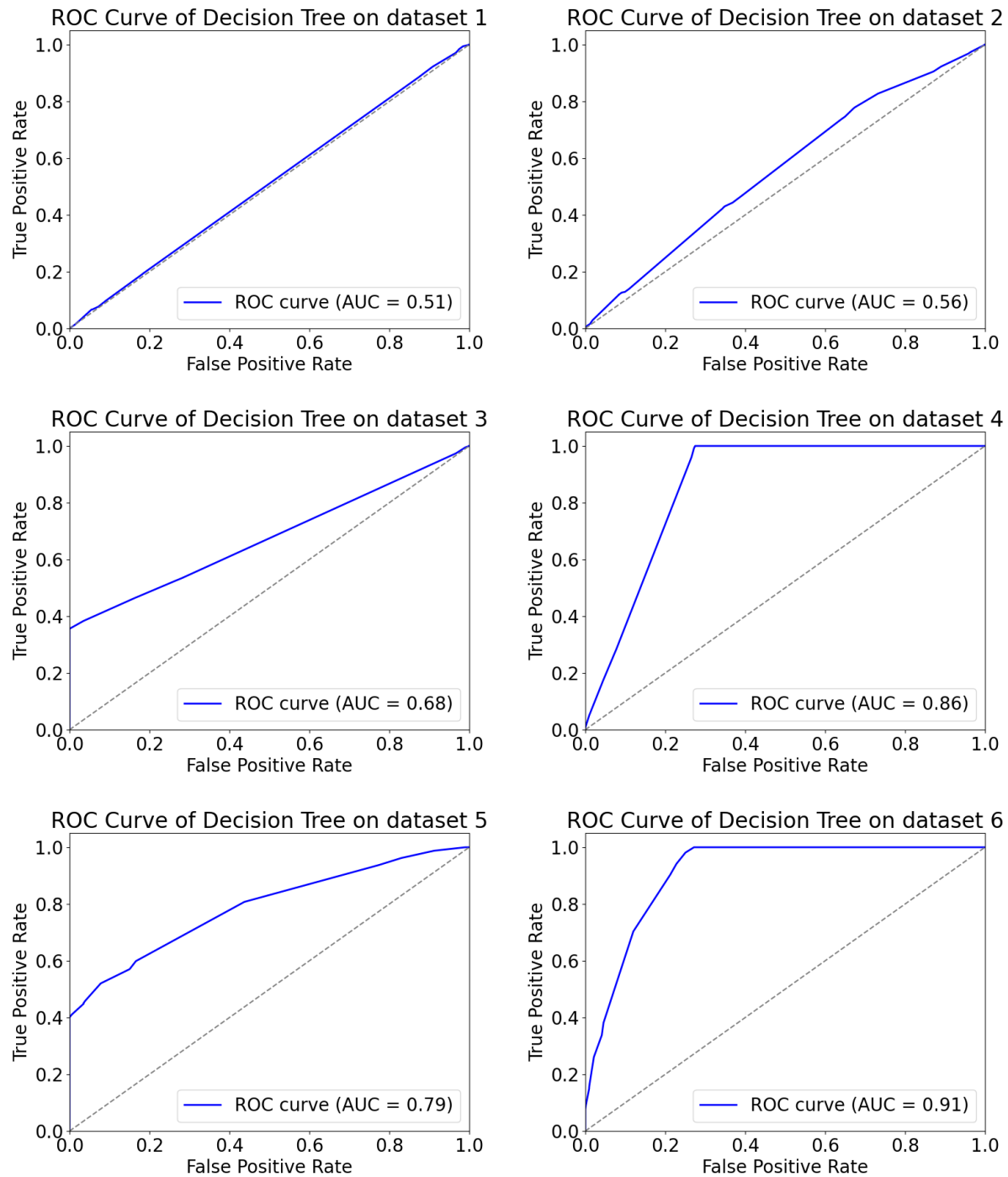


Figure 6. ROC Curve of Decision Tree classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.51 to 0.91 on Dataset 1 to

6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best hyperparameters from training
1	attack	feature_data1	0.8817	0.5098 (0.488, 0.532)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2
2	tackle	feature_data1	0.5847	0.5609 (0.544, 0.578)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2
3	attack	feature_data2	0.8820	0.6761 (0.660, 0.692)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5
4	tackle	feature_data2	0.8820	0.8629 (0.851, 0.875)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5
5	attack	feature_data3	0.8838	0.7896 (0.773, 0.806)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2
6	tackle	feature_data3	0.8838	0.9109 (0.902, 0.920)	'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2

Table 7. Performance metrics and hyperparameters for Decision Tree classifier across the six datasets.

Table 7 outlines the performance metrics and best hyperparameters for the Decision Tree classifier across the six datasets. The AUC values showed significant improvement from 0.5098 in Dataset 1 to 0.9109 in Dataset 6. Below is a detailed discussion of the performance on each dataset. For Dataset 1, the Decision Tree classifier achieved a test accuracy of 0.8817, indicating good prediction capability, and an AUC of 0.5098, suggesting that its performance is no better than random guessing when different thresholds other than 0.5 are used despite working well with threshold 0.5. The best hyperparameters for this set included 'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2. Performance on Dataset 2 exhibited a significant drop in test accuracy to 0.5847 but a slight improvement in AUC to 0.5609, using the same

hyperparameters as for Dataset 1. For Dataset 3, the test accuracy increased to 0.8820, with a moderate AUC of 0.6761, achieved by adjusting the 'min_samples_split': 5. The model on Dataset 4 has the same test accuracy 0.8820 as on Dataset 3, but showed substantial improvement in AUC to 0.8629, indicating better discriminative performance with the same hyperparameters. For Dataset 5, the model's test accuracy improved to 0.8838, with an AUC of 0.7896, using the same hyperparameters as in Dataset 1 but demonstrating better performance, possibly due to data variations. Finally, the model on Dataset 6 achieved the best results, with a test accuracy of 0.8838 and an AUC of 0.9109, using the same hyperparameters as for Dataset 1. This indicates that the variables in feature_data3 are more informative for distinguishing between attack and tackle compared to those in feature_data2 or feature_data1.

Figure 7 depicts the ROC curves for the XGBoost classifier across the six datasets following training and parameter optimization. The AUC values show an increase from 0.51 in Dataset 1 to 0.91 in Dataset 6, mirroring the pattern observed in Figures 1 to 6. This performance trend indicates that the XGBoost classifier excels at predicting tackle and attack when utilizing 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', and 'quarter' but not as well using 'gameClockSeconds', 'playID', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability', 'yardlineNumber', 'nflId', 'collegeName', and 'weight'. For Dataset 6, the curve signifies the highest performance by nearing the top-left corner of the plot, which signifies a high True Positive Rate and a low False Positive Rate.

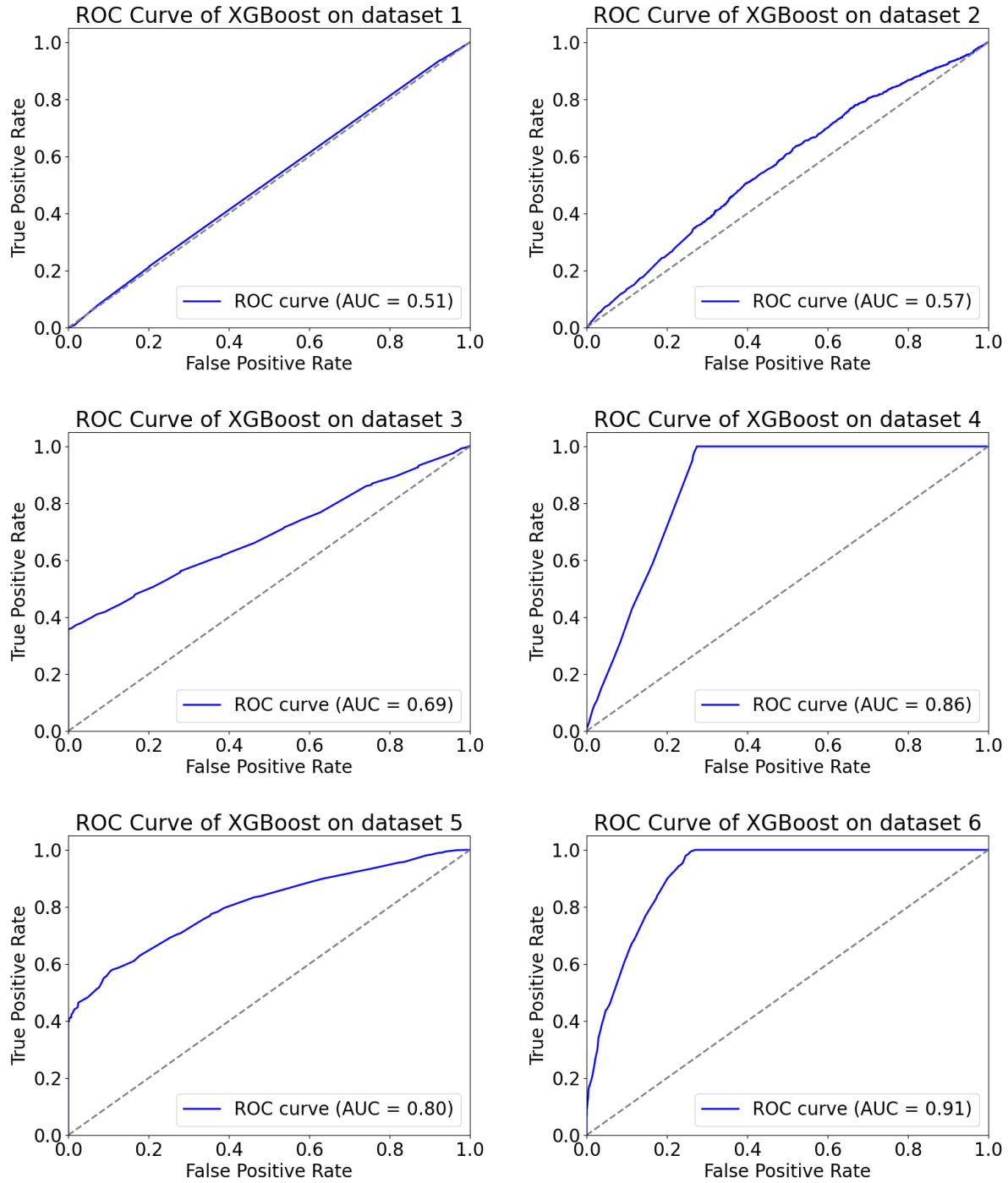


Figure 7. ROC Curve of XGBoost classifier applied to test datasets after training and parameter tuning with the training data. AUC value increases from 0.50 to 0.92 on Dataset 1 to 6.

Dataset	Y	X	Test accuracy	AUC & 95% CI	Best Hyperparameters
1	attack	feature_data1	0.8826	0.5103 (0.489, 0.531)	'learning_rate': 1.0, 'max_depth': 1, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 4, 'reg_lambda': 0.1
2	tackle	feature_data1	0.5833	0.5682 (0.551, 0.585)	'learning_rate': 0.1, 'max_depth': 3, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 20, 'reg_lambda': 10
3	attack	feature_data2	0.8826	0.6906 (0.671, 0.710)	'learning_rate': 1.0, 'max_depth': 1, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 20, 'reg_lambda': 0.5
4	tackle	feature_data2	0.8826	0.8634 (0.851, 0.876)	'learning_rate': 1.0, 'max_depth': 2, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 6, 'reg_lambda': 0.1
5	attack	feature_data3	0.8856	0.8034 (0.787, 0.820)	'learning_rate': 1.0, 'max_depth': 2, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 10, 'reg_lambda': 0.5
6	tackle	feature_data3	0.8856	0.9147 (0.906, 0.924)	'learning_rate': 1.0, 'max_depth': 2, 'min_child_weight': 2, 'min_split_loss': 0, 'n_estimators': 12, 'reg_lambda': 0.1

Table 8. Performance metrics and hyperparameters for XGBoost classifier across the six datasets.

Table 8 presents the performance metrics and best hyperparameters for the XGBoost classifier across the six datasets. The test accuracy varied from 0.5833 in Dataset 2 to 0.8856 in Dataset 6 while the AUC values, which measure the classifier's ability to discriminate between classes, improved from 0.5103 in Dataset 1 to 0.9147 in Dataset 6. In Dataset 1, the XGBoost classifier achieved a test accuracy of 0.8826 with an AUC of 0.5103, indicating that its performance is no better than random guessing when using thresholds other than 0.5, even though it performs well at a threshold of 0.5. The best hyperparameters for this set included a 'learning_rate' of 1.0, 'max_depth' of 1, 'min_child_weight' of 2, 'min_split_loss' of 0, 'n_estimators' of 4, and 'reg_lambda' of 0.1. Dataset 2 exhibited a significant drop in test accuracy to 0.5833 and a slight improvement in AUC to 0.5682, achieved by adjusting the 'learning_rate' to 0.1, 'max_depth' to 3, and 'reg_lambda' to 10. Comparing Dataset 1 and 2, the difference in performance may be attributed to the response variable ('attack' versus 'tackle'). Performance on Dataset 3 had a similar test accuracy of 0.8826, but with a notable increase in AUC to 0.6906, indicating better discriminative performance with feature_data2 and hyperparameters similar to those of Dataset 1. The model on Dataset 4 showed significant improvement, with a test accuracy of 0.8826 and an AUC of 0.8634, indicating much better performance in predicting tackle with feature_data2. For Dataset 5, the model's test accuracy increased to 0.8856 with an AUC of 0.8034, using a 'learning_rate' of 1.0, 'max_depth' of 2, and 'n_estimators' of 10. Finally, the model on Dataset 6 achieved the best results, with a test accuracy of 0.8856 and an AUC of 0.9147, using the same 'max_depth' of 2 and 'n_estimators' of 12, but with a 'reg_lambda' of 0.1. From these results, it is clear that the XGBoost model predicts both attack and tackle well with the variables in feature_data3, which provides the most informative features compared to feature_data1 and feature_data2.

To have a closer look at the model under the best parameter settings for each of the six datasets, we examine the negative loglikelihood and classification error for training and test data in Figures 8 and 9. It can be seen that for Dataset 1, the training Negative Loglikelihood curve is lower than the curve for the test data. This indicates that overfitting problem exists. Moving on to Datasets 2 to 6, the separation between the training and test Negative Loglikelihood curves incrementally reduces suggesting that less overfitting occurred compared to Dataset 1. The final models for Datasets 5 and 6 had the least overfitting. This also explains why the performance for Datasets 5 and 6 are better than for the other datasets. For the classification error curves, the values for the training curves and test curves are close to each other for all except Dataset 2. This confirms that classification accuracy or error is not sensitive enough for model training. We only look at the overfitting problem for XGBoost and will not do so for the other models because the performance from different models do not differ a lot.

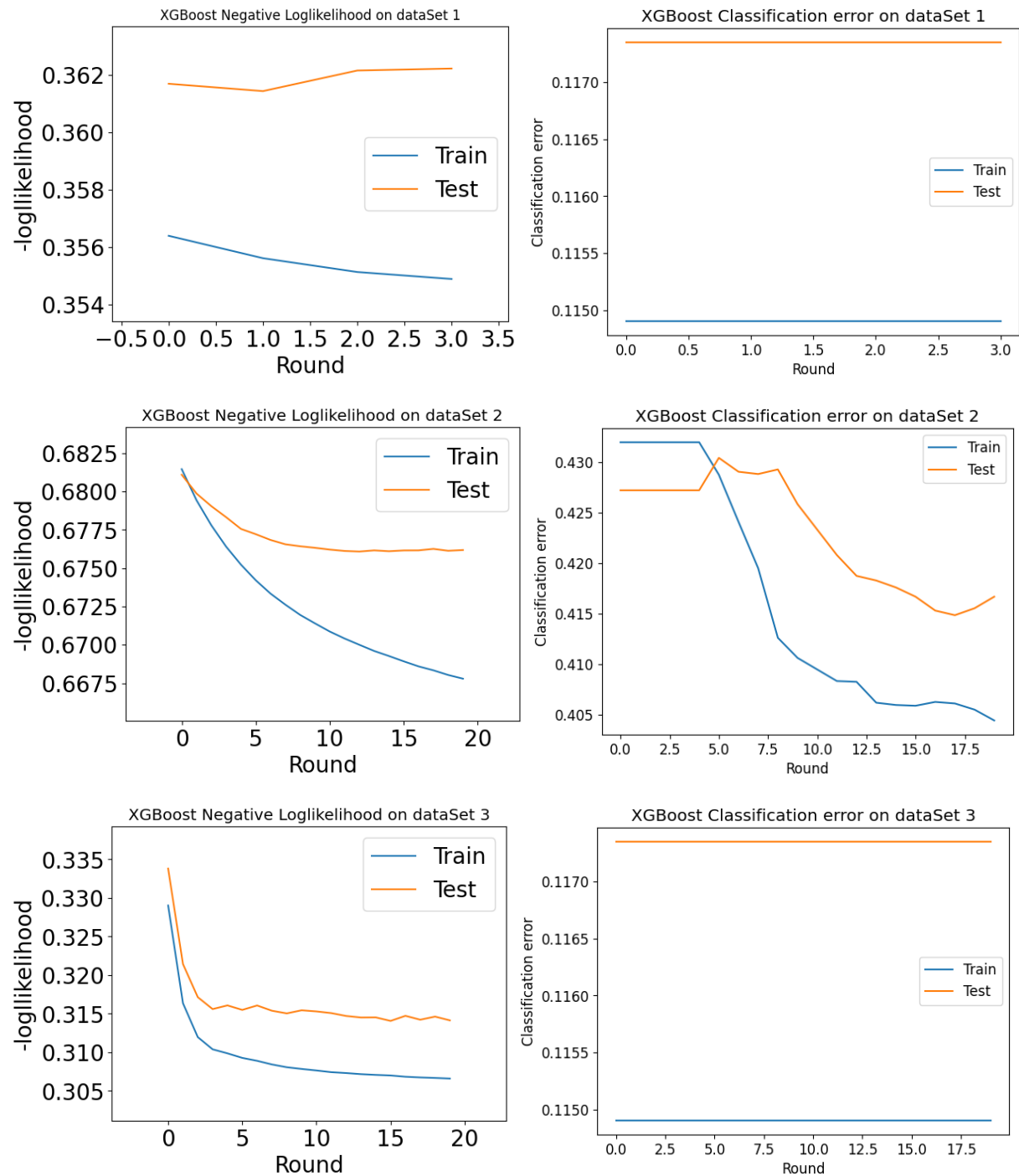


Figure 8. Negative Loglikelihood and Classification error Curve of XGBoost classifier on Datasets 1 to 3.

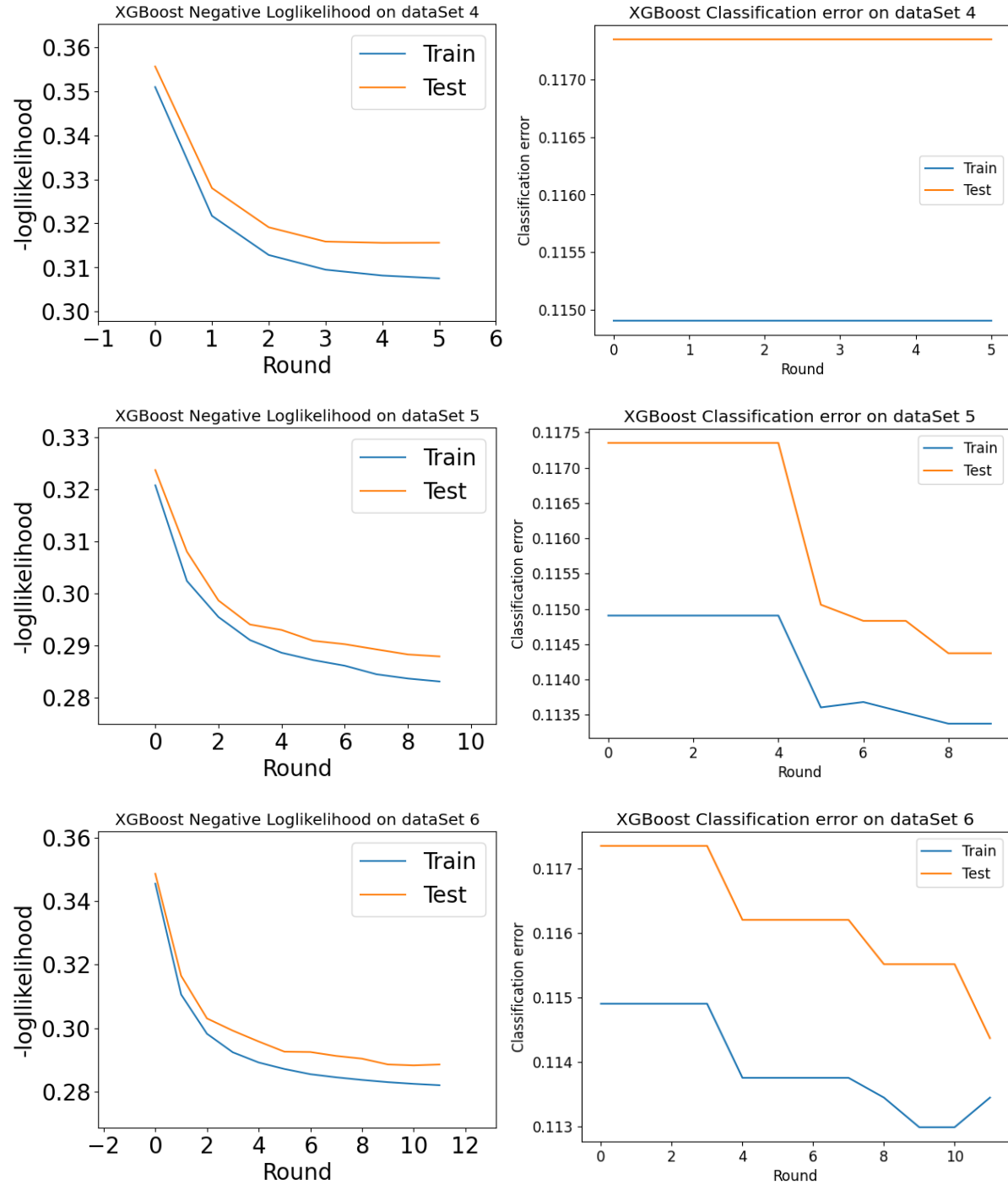


Figure 9. Negative Loglikelihood and Classification error Curve of XGBoost classifier on Datasets 4 to 6.

Conclusion and Discussion

In this report the performance of several machine learning classifiers was comprehensively analyzed. The models include Logistic Regression, Random Forest, K-Nearest Neighbors (KNN), AdaBoost, Support Vector Machine (SVM), Decision Tree, and XGBoost, across six distinct datasets. Each dataset comprised of either different combinations of features or different outcome variable tackle or attack in NFL defensive performance. Overall, the analysis indicates that feature_data3, which includes 'player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assist', and 'quarter', provides the most predictive power for both tackle and attack outcomes across all classifiers. Among the classifiers, Logistic Regression, XGBoost and AdaBoost consistently achieved the highest performance metrics, demonstrating their robustness and effectiveness in predicting NFL defensive performance. The AUCs and 95% confidence intervals of these three models are 0.919 (0.910, 0.928), 0.919 (0.910, 0.928), and 0.915 (0.906, 0.924), respectively. Decision Tree, SVM, and Random Forest have slightly lower AUC than the top three, but the difference is not statistically significant. The AUCs and 95% confidence interval are 0.911 (0.902, 0.920), 0.902 (0.893, 0.912), and 0.907 (0.897, 0.917), respectively. KNN has lowest performance and the difference to the top three is significant even though the difference with the middle three classifiers is not significant. The AUC and 95% confidence interval are 0.896 (0.887, 0.908). The results shows that appropriate set of feature variables play more important roles in prediction performance than different classifiers. Hyperparameter tuning could help to optimize each model for NFL defensive performance.

Even though the aforementioned performance is encouraging, there are limitations in the study. One limitation is that the dataset only contains one season's NFL data. The insight of this analysis only applies to interpretation for one season with similar athletic conditions. It will not work well if the prediction involves multiple seasons. In between two seasons, there are 6 months of down time, during which some players train excessively and improve their skills while some players' condition deteriorates. When multiple seasons are involved, the athletes are not in consistent condition. Another limitation is that this study did not do an extensive feature selection. The variables in feature_data1 come from feature importance of a random forest classifier conducted by Han 2023. The variables in feature_data 2 and 3 were selected by using correlation between predictive variables and tackle (Chen 2023, DuBois 2023). The variables selected based on random forest may have the following problems:

1. **Bias Towards High Cardinality Features:** Random Forest tends to give higher importance to features with many unique values (high cardinality), such as continuous variables or categorical variables with many levels, potentially overestimating their importance compared to features with fewer unique values.
2. **Correlation Bias:** If two features are highly correlated, random forest may split on one feature and reduce the importance of the other, even though both features might be important. This can lead to biased importance scores.
3. **Non-Interpretable Rankings:** Feature importance scores from random forests can be difficult to interpret, especially in the presence of complex interactions and nonlinear relationships among features.
4. **Model-Specific Importance:** The feature importance scores are specific to the random forest model and may not generalize well to other types of models. What is important for

a random forest might not be as important for a linear model or another machine learning algorithm.

5. **Sensitivity to Hyperparameters:** The importance scores can be sensitive to the hyperparameters of the random forest. Changes in the number of trees, depth of trees, or other parameters can affect the feature importance rankings.
6. **Overfitting Risk:** In small datasets, random forest can overfit, leading to inflated importance scores for features that capture noise rather than true signal.
7. **Randomness:** Despite being an ensemble method, the random component of random forests (random subset selection for each tree) can introduce variability in the feature importance scores, leading to instability in the rankings unless a large number of trees are used.
8. **Unaccounted Interactions:** Random forests can capture interactions between variables, but the feature importance metric does not directly reflect these interactions. A variable might appear less important on its own, even if it plays a crucial role in interaction with other variables.
9. **Dependence on Feature Scaling:** Although random forests are generally robust to feature scaling, differences in scale between features can still affect the importance scores to some extent, especially in proximity-based importance measures.

To mitigate these limitations, it is often beneficial to use a combination of feature selection methods, consider domain knowledge, and validate the chosen features through cross-validation or other resampling techniques. Additionally, examining the importance scores from multiple models can provide a more comprehensive understanding of feature importance.

Correlation only assesses the strength of linear relationship. It may miss nonlinear relationships that could be important. In addition, there are other problems using correlation to select variables. They are listed below:

1. **Restricted to bivariate relationship:** Correlation looks at the relationship between pairs of variables. It does not account for multicollinearity or interactions among multiple variables. This can lead to selecting redundant variables or missing important combinations of variables.
2. **Sensitivity to Outliers:** Correlation can be highly sensitive to outliers, which can distort the strength and direction of the relationship between variables.
3. **Binary and Categorical Variables:** Correlation coefficients are generally used for continuous variables and may not be appropriate or interpretable for binary or categorical variables without proper transformation or using specialized methods like point-biserial correlation for binary variables.
4. **Magnitude Interpretation:** The interpretation of correlation magnitude can be subjective. A moderate correlation in one context may be considered strong or weak in another, depending on the domain and specific problem.
5. **Influence of Sample Size:** In small samples, correlation coefficients can be unstable and not representative of the population. Larger samples provide more reliable estimates, but they can also detect trivial correlations that are not practically significant.
6. **Ignoring Data Distribution:** Correlation does not take into account the underlying distribution of the data. Variables with similar distributions may have low correlation even if they are related in a non-linear way.

7. **Dimensionality Issues:** In high-dimensional datasets, the likelihood of finding spurious correlations increases. This can lead to overfitting if such variables are included in a model based solely on their correlation.

Due to these limitations, it is often necessary to use more sophisticated techniques, such as multiple regression analysis, principal component analysis (PCA), or machine learning algorithms, to select and evaluate the importance of variables in a dataset.

In summary, this report contributes to the Sports analytics field by extending the application of advanced machine learning techniques to NFL defensive performance prediction. By exploring a wider range of models and tuning parameters, this report provides valuable insights that could influence team strategies, betting market, and fan engagement. Furthermore, the analysis in this report could be adapted to other sports analytics domains, broadening the impact of this work.

Bibliography

- Friedman, Jerome H., Trevor Hastie, and Rob Tibshirani. 2010. "Regularization Paths for Generalized Linear Models via Coordinate Descent". *Journal of Statistical Software* 33 (1):1-22. <https://doi.org/10.18637/jss.v033.i01>.
- Breiman Leo (2001). "Random Forests". *Machine Learning*. **45** (1): 5–32. Bibcode:2001MachL..45....5B. doi:10.1023/A:1010933404324.
- Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785-794. 2016.
- Cortes, Corinna; Vapnik, Vladimir (1995). "Support-vector networks" (PDF). *Machine Learning*. **20** (3): 273–297. CiteSeerX 10.1.1.15.9362. doi:10.1007/BF00994018. S2CID 206787478.
- Cover, Thomas M.; Hart, Peter E. (1967). "Nearest neighbor pattern classification" (PDF). *IEEE Transactions on Information Theory*. **13** (1): 21–27. CiteSeerX 10.1.1.68.2616. doi:10.1109/TIT.1967.1053964. S2CID 5246200.
- Freund, Yoav; Schapire, Robert E. (1995). "A decision-theoretic [sic] generalization of on-line learning and an application to boosting". *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 23–37, doi:10.1007/3-540-59119-2_166, ISBN 978-3-540-59119-1, retrieved 2022-06-24
- Fürnkranz, Johannes. (2011). "Decision Tree". In: Sammut, C., Webb, G.I. (eds) *Encyclopedia of Machine Learning*. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-30164-8_204
- Han, Jennifer. (2023). "Defensive performance prediction on a player/play". <https://www.kaggle.com/code/jenniferhann/defensive-performance-prediction-on-a-player-play/notebook>
- Chen, Annie. (2023). "Predictive Analysis of Defensive Player Performance". <https://www.kaggle.com/code/anniechensy00/predictive-analysis-of-defensive-player-performanc/notebook>
- DuBois, Brynee (2023). "Analyzing NFL Defensive Player Performance with ML". <https://www.kaggle.com/code/brynned/analyzing-nfl-defensive-player-performance-with-ml>

Appendix A - Python code for Data Preparation

```
pip install pandas
pip install seaborn
pip install numpy
pip install matplotlib
pip install plotly
pip install colorama
pip install scikit-learn

import pandas as pd
import seaborn as sns
import numpy as np

import matplotlib.pyplot as plt
import plotly.io as pio

import datetime
from colorama import Fore, Style, init
#import lightgbm as lgb

from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier,
VotingClassifier
from sklearn.metrics import accuracy_score, recall_score, classification_report,
confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,
BaggingRegressor, RandomForestRegressor, GradientBoostingRegressor,
AdaBoostClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier,
export_graphviz
from sklearn.metrics import mean_squared_error, confusion_matrix,
classification_report, roc_curve, auc, accuracy_score, roc_auc_score
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

import warnings
warnings.filterwarnings('ignore')

# Load the datasets
```

```

games_path = '/Users/jihyukgweon/Desktop/kaggle/input/nfl-big-data-bowl-2024/games.csv'
players_path = '/Users/jihyukgweon/Desktop/kaggle/input/nfl-big-data-bowl-2024/players.csv'
plays_path = '/Users/jihyukgweon/Desktop/kaggle/input/nfl-big-data-bowl-2024/plays.csv'
tackles_path = '/Users/jihyukgweon/Desktop/kaggle/input/nfl-big-data-bowl-2024/tackles.csv'

# Read the datasets into dataframes
games = pd.read_csv(games_path)
players = pd.read_csv(players_path)
plays = pd.read_csv(plays_path)
tackles = pd.read_csv(tackles_path)
# Merge tackles data with player data
combined_df_1 = pd.merge(tackles, players, on = 'nflId')

# Then merge with play data
combined_df_2 = pd.merge(combined_df_1, plays, on = ['gameId', 'playId'])

# Further merge with games data
data = pd.merge(combined_df_2, games, on = 'gameId')

data2 = data.sort_values(by = ['gameId', 'playId'])
data2
data2['attack'] = data2[['tackle', 'assist', 'forcedFumble']].max(axis=1)

non_numeric_columns_final_df = data2.select_dtypes(exclude=[np.number]).columns
# Display the non-numeric columns in the merged data
print(non_numeric_columns_final_df)
def convert_height_to_inches(height):
    if isinstance(height, int):
        return height # Return the height unchanged if it's already an integer
    else:
        feet, inches = map(int, height.split('-'))
        return feet * 12 + inches

# Apply the function to the 'height' column
data2['height'] = data2['height'].apply(convert_height_to_inches)

# Convert gameClock from the given format to seconds
def convert_gameclock_to_seconds(gameclock):
    if pd.isna(gameclock): # Handle missing values
        return np.nan

    minutes, seconds = map(int, gameclock.split(':'))
    total_seconds = minutes * 60 + seconds
    return total_seconds

```

```

# 'birthDate' is in string format, convert it to datetime
data2['birthDate'] = pd.to_datetime(data2['birthDate'], errors='coerce')
# Calculate age from 'birthDate'
data2['age'] = (datetime.datetime.now() - data2['birthDate']).dt.days // 365
# Handle NaN values by replacing them with the average age
average_age = data2['age'].mean()
data2['age'].fillna(average_age, inplace=True)
# Drop the original 'birthDate' column
data2.drop('birthDate', axis=1, inplace=True)
# Encoding categorical values to numerical values
# List of columns to be encoded
columns_to_encode = ['collegeName', 'position', 'displayName',
                     'ballCarrierDisplayName', 'playDescription', 'possessionTeam',
                     'defensiveTeam', 'yardlineSide', 'passResult',
                     'playNullifiedByPenalty', 'offenseFormation', 'foulName1', 'foulName2',
                     'gameDate', 'gameTimeEastern', 'homeTeamAbbr', 'visitorTeamAbbr']
# Dictionary to store mappings
encoding_mappings = {}
# Encode each column
label_encoder = LabelEncoder()
for column in columns_to_encode:
    # Fill NaN values with a placeholder before encoding
    data2[column].fillna('NaN', inplace=True)

    # Fit and transform with LabelEncoder
    data2[column] = label_encoder.fit_transform(data2[column])

    # Store the mappings
    encoding_mappings[column] = dict(zip(label_encoder.classes_,
    label_encoder.transform(label_encoder.classes_)))
# Feature 1: Interaction term between 'down' and 'yardsToGo'
data2['down_yards_interaction'] = data2['down'] * data2['yardsToGo']

# Feature 2: Body Mass Index (BMI)
def calculate_bmi(weight, height_in_inches):
    # BMI formula: 703 * weight in pounds / (height in inches)^2
    return 703 * weight / (height_in_inches ** 2)
data2['player_bmi'] = data2.apply(lambda row: calculate_bmi(row['weight'],
row['height']), axis=1)
data2["tackleSuccess"] = data2.groupby("nflId")["tackle"].transform(lambda x :
x.loc[x==1].count())
data2["tackleFail"] = data2.groupby("nflId")["tackle"].transform(lambda x :
x.loc[x==0].count())
data2["tackleSuccessRate"] = data2["tackleSuccess"] / (data2["tackleFail"] +
data2["tackleSuccess"]) * 100

```

```

data2["assistSuccess"] = data2.groupby("nflId")["assist"].transform(lambda x :
x.loc[x==1].count())
data2["assistFail"] = data2.groupby("nflId")["assist"].transform(lambda x :
x.loc[x==0].count())
data2["assistSuccessRate"] = data2["assistSuccess"] / (data2["tackleFail"] +
data2["assistSuccess"]) * 100
data2
data2["tackleSuccess"] = data2.groupby("nflId")["tackle"].transform(lambda x :
x.loc[x==1].count())
data2["tackleFail"] = data2.groupby("nflId")["tackle"].transform(lambda x :
x.loc[x==0].count())
data2["tackleSuccessRate"] = data2["tackleSuccess"] / (data2["tackleFail"] +
data2["tackleSuccess"]) * 100
data2["assistSuccess"] = data2.groupby("nflId")["assist"].transform(lambda x :
x.loc[x==1].count())
data2["assistFail"] = data2.groupby("nflId")["assist"].transform(lambda x :
x.loc[x==0].count())
data2["assistSuccessRate"] = data2["assistSuccess"] / (data2["tackleFail"] +
data2["assistSuccess"]) * 100
feature_data1 =
data2[['gameClockSeconds', 'playId', 'preSnapHomeTeamWinProbability', 'preSnapVisitorTeam
WinProbability', 'yardlineNumber', 'nflId',
        'collegeName', 'weight', 'attack']]
feature_data2 = data2[['down_yards_interaction', 'player_bmi', 'quarter', 'assist',
'forcedFumble', 'tackle']]

feature_data3 =
data2[['player_bmi', 'age', 'tackleSuccessRate', 'assistSuccessRate', 'forcedFumble', 'assi
st', 'quarter', 'tackle']]

feature_data1['rowID'] = range(len(feature_data1))
feature_data2['rowID'] = range(len(feature_data2))
feature_data3['rowID'] = range(len(feature_data3))

feature_data1.drop(columns=['rowID'], inplace=True)
feature_data2.drop(columns=['rowID'], inplace=True)
feature_data3.drop(columns=['rowID'], inplace=True)

#feature 1
import random
random.seed(42)

def generate_binary_variables(num_variables, probab_success):
    train_index = []
    test_index = []
    for i in range(num_variables):
        if random.random() < probab_success:
            train_index.append(i)

```

```

        else:
            test_index.append(i)
    return train_index, test_index
# Generating 17426 binary random variables with probability of success = 0.75
num_variables = 17426
probability_of_success = 0.75
train_index, test_index = generate_binary_variables(num_variables,
probability_of_success)
print(feature_data1.iloc[train_index])
print(feature_data1.iloc[test_index])
train_index1 = feature_data1.iloc[train_index]
test_index1 = feature_data1.iloc[test_index]

#feature 2
import random
random.seed(42)
def generate_binary_variables(num_variables, prob_success):
    train_index = []
    test_index = []
    for i in range(num_variables):
        if random.random() < prob_success:
            train_index.append(i)
        else:
            test_index.append(i)
    return train_index, test_index
# Generating 17426 binary random variables with probability of success = 0.75
num_variables = 17426
probability_of_success = 0.75
train_index, test_index = generate_binary_variables(num_variables,
probability_of_success)
print(feature_data2.iloc[train_index])
print(feature_data2.iloc[test_index])
train_index2 = feature_data2.iloc[train_index]
test_index2 = feature_data2.iloc[test_index]

#feature 3
import random
random.seed(42)
def generate_binary_variables(num_variables, prob_success):
    train_index = []
    test_index = []
    for i in range(num_variables):
        if random.random() < prob_success:
            train_index.append(i)
        else:
            test_index.append(i)
    return train_index, test_index
# Generating 17426 binary random variables with probability of success = 0.75

```

```

num_variables = 17426
probability_of_success = 0.75
train_index, test_index = generate_binary_variables(num_variables,
probability_of_success)
print(feature_data3.iloc[train_index])
print(feature_data3.iloc[test_index])
train_index3 = feature_data3.iloc[train_index]
test_index3 = feature_data3.iloc[test_index]

#1
# attack + set1
# Apply get_dummies to each split separately
X_train1 = pd.get_dummies(train_index1.drop('attack', axis=1), drop_first=True)
y_train1 = train_index1['attack']
X_test1 = pd.get_dummies(test_index1.drop('attack', axis=1), drop_first=True)
y_test1 = test_index1['attack']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
are now either numerical or one-hot encoded
scaler = StandardScaler()
X_train1 = scaler.fit_transform(X_train1)
X_test1 = scaler.transform(X_test1)

#2
#Tackle + set1
# Apply get_dummies to each split separately
X_train2 = pd.get_dummies(train_index1.drop('attack', axis=1), drop_first=True)
y_train2 = train_index2['tackle']
X_test2 = pd.get_dummies(test_index1.drop('attack', axis=1), drop_first=True)
y_test2 = test_index2['tackle']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
are now either numerical or one-hot encoded
scaler = StandardScaler()
X_train2 = scaler.fit_transform(X_train2)
X_test2 = scaler.transform(X_test2)

#3
# attack + set2
# Apply get_dummies to each split separately
X_train3 = pd.get_dummies(train_index2.drop('tackle', axis=1), drop_first=True)
y_train3 = train_index1['attack']
X_test3 = pd.get_dummies(test_index2.drop('tackle', axis=1), drop_first=True)
y_test3 = test_index1['attack']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
are now either numerical or one-hot encoded
scaler = StandardScaler()
X_train3 = scaler.fit_transform(X_train3)

```

```

X_test3 = scaler.transform(X_test3)

#4
# tackle + set2
# Apply get_dummies to each split separately
X_train4 = pd.get_dummies(train_index2.drop('tackle', axis=1), drop_first=True)
y_train4 = train_index2['tackle']
X_test4 = pd.get_dummies(test_index2.drop('tackle', axis=1), drop_first=True)
y_test4 = test_index2['tackle']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
# are now either numerical or one-hot encoded
scaler = StandardScaler()
X_train4 = scaler.fit_transform(X_train4)
X_test4 = scaler.transform(X_test4)

#5
# attack + set3
# Apply get_dummies to each split separately
X_train5 = pd.get_dummies(train_index3.drop('tackle', axis=1), drop_first=True)
y_train5 = train_index1['attack']
X_test5 = pd.get_dummies(test_index3.drop('tackle', axis=1), drop_first=True)
y_test5 = test_index1['attack']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
# are now either numerical or one-hot encoded
scaler = StandardScaler()
X_train5 = scaler.fit_transform(X_train5)
X_test5 = scaler.transform(X_test5)
from sklearn.impute import SimpleImputer
# Instantiate the imputer with your chosen strategy
imputer = SimpleImputer(strategy='mean')
# Impute missing values in your feature matrix X_train5
X_train5 = imputer.fit_transform(X_train5)
X_test5 = imputer.fit_transform(X_test5)

#6
# tackle + set3
# Apply get_dummies to each split separately
X_train6 = pd.get_dummies(train_index3.drop('tackle', axis=1), drop_first=True)
y_train6 = train_index3['tackle']
X_test6 = pd.get_dummies(test_index3.drop('tackle', axis=1), drop_first=True)
y_test6 = test_index3['tackle']
# Standardize the numerical features
# Note: Only standardize numerical columns. For simplicity, I'm assuming all columns
# are now either numerical or one-hot encoded
scaler = StandardScaler()

```

```
X_train6 = scaler.fit_transform(X_train6)
X_test6 = scaler.transform(X_test6)
from sklearn.impute import SimpleImputer
# Instantiate the imputer with your chosen strategy
imputer = SimpleImputer(strategy='mean')
# Impute missing values in your feature matrix X_train5
X_train6 = imputer.fit_transform(X_train6)
X_test6 = imputer.fit_transform(X_test6)
```

Appendix B - Python code for Logistic Regression

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_curve, auc
import matplotlib.pyplot as plt
from plotnine import *

# Define the parameter grid for Logistic Regression
parasetting = []
for C in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
    for P in ['l1', 'l2', 'elasticnet']:
        for S in ['liblinear', 'lbfgs', 'saga']:
            for T in [1e-4, 1e-3, 1e-2]:
                for M in [100, 200, 300]:
                    if P == 'elasticnet':
                        for l in [0, 0.5, 1]:
                            if S != 'saga':
                                pass
                            else:
                                parasetting.append({'logistic__penalty' : [P],
'logistic__C' : [C], 'logistic__solver' : [S], 'logistic__tol' : [T],
'logistic__max_iter' : [M], 'logistic__l1_ratio' : [l]})
                    else:
                        parasetting.append({'logistic__penalty' : [P], 'logistic__C' :
[C], 'logistic__solver' : [S], 'logistic__tol' : [T], 'logistic__max_iter' : [M],
'logistic__l1_ratio' : [0]})

logistic = Pipeline([('scaler', StandardScaler()), ('logistic',
LogisticRegression(random_state=10))])

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# To store results
results = []
coefficients = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list,
X_test_list, y_test_list), 1):
    # Perform GridSearchCV for Logistic Regression
```

```

grid_search_logistic = GridSearchCV(estimator=logistic, param_grid=parasetting,
cv=3, scoring='accuracy', n_jobs=-1, return_train_score=True, pre_dispatch='2*n_jobs')
grid_search_logistic.fit(X_train, y_train)

# Best cross-validation accuracy
best_cv_score = grid_search_logistic.best_score_

# Best model from grid search
best_model = grid_search_logistic.best_estimator_

# Extract best parameters
best_params = grid_search_logistic.best_params_

# Evaluate on training data
y_train_pred = best_model.predict(X_train)
train_accuracy = accuracy_score(y_train, y_train_pred)

# Evaluate on test data
y_test_pred = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Compute AUC-ROC
y_test_prob = best_model.predict_proba(X_test)[: , 1]
fpr, tpr, _ = roc_curve(y_test, y_test_prob)
roc_auc = auc(fpr, tpr)

# Collect the results
results.append({
    'Dataset': f'Set {i}',
    'Best CV Accuracy': best_cv_score,
    'Training Accuracy': train_accuracy,
    'Test Accuracy': test_accuracy,
    'AUC': roc_auc,
    'param_logistic__C': best_params.get('logistic__C', None),
    'param_logistic__penalty': best_params.get('logistic__penalty', None),
    'param_logistic__solver': best_params.get('logistic__solver', None),
    'param_logistic__tol': best_params.get('logistic__tol', None),
    'param_logistic__max_iter': best_params.get('logistic__max_iter', None),
    'param_logistic__l1_ratio': best_params.get('logistic__l1_ratio', None)
})

# Extract and store coefficients
coefficients.append({
    'Dataset': f'Set {i}',
    'Coefficients': best_model.named_steps['logistic'].coef_
})

# Plot AUC-ROC Curve

```

```

plt.figure(figsize=(8, 6))
plt.rcParams['font.size'] = 20
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:0.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curve of Logistic Regression on dataset {i}')
plt.legend(loc='lower right')
plt.savefig(f'/Users/jihyukgweon/Desktop/Logistic/Logistic_AUC_ROC_Set{i}.png')
plt.show()

# Display the results
results_df = pd.DataFrame(results)
print(results_df)

# Plot accuracy results using plotnine
logistic_params = results_df[['param_logistic__C', 'param_logistic__penalty',
'param_logistic__solver', 'param_logistic__tol', 'param_logistic__max_iter',
'param_logistic__l1_ratio']]
logistic_train_accuracy = results_df['Training Accuracy']
logistic_test_accuracy = results_df['Test Accuracy']

logistic_result_train = logistic_params.copy()
logistic_result_train['Accuracy'] = logistic_train_accuracy
logistic_result_train['whichdat'] = 'train'
logistic_result_train['Dataset'] = results_df['Dataset']

logistic_result_test = logistic_params.copy()
logistic_result_test['Accuracy'] = logistic_test_accuracy
logistic_result_test['whichdat'] = 'test'
logistic_result_test['Dataset'] = results_df['Dataset']

logistic_result = pd.concat((logistic_result_train, logistic_result_test))

print(logistic_result['Accuracy'])

for i in range(1, 7):
    current_result = logistic_result[logistic_result['Dataset'] == f'Set {i}']

    plt = (ggplot(current_result)
          + aes(x='param_logistic__C', y='Accuracy')
          + facet_grid('param_logistic__penalty + param_logistic__l1_ratio ~
param_logistic__solver + param_logistic__max_iter')
          + geom_point(aes(color='whichdat'))
          + ggtitle(f'Accuracy Plot for Dataset {i}')
          )

```

```
plt.draw()

# Display coefficients
coefficients_df = pd.DataFrame(coefficients)
print(coefficients_df)

# Extract and display coefficients for dataset 6
dataset_6_coefficients = coefficients_df[coefficients_df['Dataset'] == 'Set
6']['Coefficients'].values[0]
print(f'Coefficients for the best model from dataset 6:\n{dataset_6_coefficients}')
```

Appendix C - R code for Logistic Regression

```
setwd('/Users/jihyukgweon/Desktop/Logistic/pythonimputeddata')
#reading the data
library(glmnet)
#install.packages("pROC")
library(pROC)
coefs = list()
auci = list()
aucs = list()
accuracy = list()
pdf('AUCLogistic.pdf', height = 18, width = 8)
par(mfrow = c(6,2),mar = c(4,5,4,1))
lambdamin = list()
for (i in 1:6){
  Xtrain <- read.csv(paste0('dataset',i,'xtrain.csv'),row.names = 1)
  ytrain <- read.csv(paste0('dataset',i,'ytrain.csv'),row.names = 1)
  Xtest <- read.csv(paste0('dataset',i,'xtest.csv'),row.names = 1)
  ytest <- read.csv(paste0('dataset',i,'ytest.csv'),row.names = 1)
  imputeX = makeX(Xtrain, test = Xtest, na.impute = TRUE)
  Xtrainimpute = imputeX$x
  Xtestimpute = imputeX$xtest
  ytrain = unlist(ytrain)
  ytest = unlist(ytest)
  mod = glmnet(Xtrainimpute, ytrain)
  cvfit = cv.glmnet(Xtrainimpute, ytrain)
  result2 = predict(cvfit,newx = Xtestimpute, s = 'lambda.min')
  rocdd = roc(response = ytest, predict = result2, ci = TRUE, plot = FALSE)

  res = data.frame(fpr = 1- rocdd$specificities,tpr = rocdd$sensitivities)
  plot(tpr~fpr,data=res, main = paste('ROC Curve of Logistic Regression on dataset',i),
```

```

      xlab = 'False positive Rate',
      ylab = 'True Positive Rate', xlim = c(0,1), ylim = c(0,1), col = 'blue',font.axis = 2,
      cex.lab = 1.5)
text(.7,.2,paste('AUC = ',round(rocdd$auc,3)),cex = 2)
abline(a = 0, b = 1)

  aucci[[i]] = rocdd$ci
  accuracy[[i]] = mean(abs((result2 > .5) - ytest) < 1e-5)
  aucs[[i]] = rocdd$auc
  coefs[[i]] = coef(cvfit, s = 'lambda.min')
  lambdamin[[i]] = cvfit$lambda.min
}
dev.off()
round(unlist(lambdamin),5)
aucci
auc_ci = cbind(round(unlist(aucs),3),round(matrix(unlist(aucci),ncol = 2, byrow = TRUE),3))
paste(auc_ci[1,], '(' ,auc_ci[2,],auc_ci[3,],')')
asd = matrix(paste0(auc_ci[1,], ' (',auc_ci[2,],', ',auc_ci[3,],')'))
for (i in 1:6){
  print.noquote(asd[i])
}
matrix(round(unlist(accuracy),4))
aucci
accuracy
aucs
paste(paste(coefs[[6]],c(",colnames(Xtrain))), collapse = '+ ')
coefs[[6]]

```

Appendix D - Python code for Random Forest

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, roc_curve, auc
import json
import matplotlib.pyplot as plt
from plotnine import ggplot, aes, facet_grid, geom_point, labs, theme_minimal

# Define hyperparameters grid for grid search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [3, 5, 7]
}

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Function to plot results
def plot_results(df, dataset_num):
    plt = (ggplot(df)
          + aes(x='param_n_estimators', y='mean_test_score')
          + facet_grid('param_max_depth ~ param_min_samples_split')
          + geom_point(aes(color='param_min_samples_leaf'))
          + labs(title=f'Random Forest Accuracy - Dataset {dataset_num}',
                 x='Number of Estimators',
                 y='Accuracy')
          + theme_minimal()
          )
    plt.draw()

# To store results
results = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list,
X_test_list, y_test_list), 1):
    # Define Random Forest model
    random_forest = RandomForestClassifier(random_state=42)

    # Perform grid search with cross-validation
```

```

    grid_search = GridSearchCV(random_forest, param_grid, cv=3, scoring='accuracy',
n_jobs=-1, return_train_score=True)
    grid_search.fit(X_train, y_train)

    # Save results
    cv_results = pd.DataFrame(grid_search.cv_results_)
    cv_results['dataset'] = f'Set {i}'
    results.append(cv_results)

    # Get the best hyperparameters
    best_params = grid_search.best_params_
    print(f"Best Hyperparameters for Set {i}: {best_params}")

    # Evaluate on test set
    best_model = grid_search.best_estimator_
    test_accuracy = best_model.score(X_test, y_test)
    print(f"Test Accuracy for Set {i}: {test_accuracy:.4f}")

    # Compute AUC-ROC & CI
    y_test_prob = best_model.predict_proba(X_test)[:, 1]
    fpr, tpr, _ = roc_curve(y_test, y_test_prob)
    roc_auc, ci = AUCCI(y_test,
                        y_test_prob,
                        confidence_level=0.95)
    print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
    print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")

    # Save the evaluation results to a file
    eval_result = {
        'dataset': f'Set {i}',
        'best_params': best_params,
        'test_accuracy': test_accuracy,
        'roc_auc': roc_auc,
        'fpr': fpr.tolist(),
        'tpr': tpr.tolist(),
        'ci': ci
    }

    #with
open(f'/Users/jihyukgeon/Desktop/RandomForest/RandomForestResultSet{i}.json', 'w') as
outfile:
    #json.dump(eval_result, outfile)

    # Plot AUC-ROC Curve
    plt.figure(figsize = (8,6))
    plt.rcParams['font.size'] = 20
    plt.plot(fpr, tpr, color = 'blue', lw = 2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k--')

```

```

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC curve of RandomForest on dataset {i}')
plt.legend(loc='lower right')

plt.savefig(f'/Users/jihyukgweon/Desktop/RandomForest/RandomForest_AUC_ROC_Set{i}.png'
)
plt.show()

# Combine all results for plotting
combined_results = pd.concat(results, ignore_index=True)

# Plot accuracy results using plotnine
for i in range(1, 7):
    current_result = combined_results[combined_results['dataset'] == f'Set {i}']
    plot_results(current_result, i)

```

Appendix E - Python code for K-Nearest Neighbor

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, roc_curve, auc
import pandas as pd
import json
from plotnine import ggplot, aes, facet_grid, geom_point, labs, theme_minimal
import matplotlib.pyplot as plt

# Define the parameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11, 13],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Function to plot results
def plot_results(df, dataset_num):
    plt = (ggplot(df)
          + aes(x='param_n_neighbors', y='mean_test_score')
          + facet_grid('param_weights ~ param_metric')
          + geom_point(aes(color='dataset'))
          + labs(title=f'KNN Accuracy by Number of Neighbors, Weights, and Metric - Dataset {dataset_num}',
                 x='Number of Neighbors',
                 y='Accuracy')
          + theme_minimal()
          )
    plt.draw()

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Initialize the KNeighborsClassifier
knn_clf = KNeighborsClassifier()

# To store results
results = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list, X_test_list, y_test_list), 1):
    # Perform GridSearchCV
```

```

grid_search = GridSearchCV(estimator=knn_clf, param_grid=param_grid, cv=3,
scoring='accuracy', n_jobs=-1, return_train_score=True)
result = grid_search.fit(X_train, y_train)

# Save results
cv_results = pd.DataFrame(result.cv_results_)
cv_results['dataset'] = f'Set {i}'
results.append(cv_results)

# Get the best hyperparameters
best_params = grid_search.best_params_
print(f"Best Hyperparameters for Set {i}: {best_params}")

# Evaluate the best model on the test set
best_knn_clf = grid_search.best_estimator_
test_accuracy = best_knn_clf.score(X_test, y_test)
print(f"Test Accuracy with Best Estimator for Set {i}: {test_accuracy:.4f}")

# Compute AUC-ROC & CI
y_test_prob = best_knn_clf.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_test_prob)
roc_auc, ci = AUCCI(y_test,
                    y_test_prob,
                    confidence_level=0.95)
print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")
# Save the evaluation results to a file
eval_result = {
    'dataset': f'Set {i}',
    'best_params': best_params,
    'test_accuracy': test_accuracy,
    'roc_auc': roc_auc,
    'fpr': fpr.tolist(),
    'tpr': tpr.tolist(),
    'ci' : ci
}

#with open(f'/Users/jihyukgweon/Desktop/KNN/KNNresultdat{i}.json', 'w') as
outfile:
    #json.dump(eval_result, outfile)

# Plot AUC-ROC Curve
plt.figure(figsize=(8,6))
plt.rcParams['font.size'] = 20
plt.plot(fpr, tpr, color = 'blue', lw =2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color = 'gray',linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])

```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curve of KNN on dataset {i}')
plt.legend(loc='lower right')
plt.savefig(f'/Users/jihyukgweon/Desktop/KNN/KNN_AUC_ROC_Set{i}.png')
plt.show()

# Combine all results for plotting
combined_results = pd.concat(results, ignore_index=True)

# Plot accuracy results using plotnine
plot_results(combined_results, i)
```

Appendix F - Python code for AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, auc, accuracy_score
import pandas as pd
import json
import matplotlib.pyplot as plt

# Define hyperparameters grid for grid search
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.001, 0.01, 0.1, 1.0]
}

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Function to plot AUC-ROC Curve
def plot_roc_curve(fpr, tpr, roc_auc, dataset_num):
    plt.figure(figsize=(8, 6))
    plt.rcParams['font.size'] = 20
    plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve of AdaBoost on dataset {dataset_num}')
    plt.legend(loc="lower right")

plt.savefig(f'/Users/jihyukgweon/Desktop/AdaBoost/AdaBoost_AUC_ROC_Set{dataset_num}.png')
plt.show()

# To store results
results = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list,
X_test_list, y_test_list), 1):
    # Define AdaBoost model
    adaboost = AdaBoostClassifier(random_state=42)

    # Perform grid search with cross-validation
```

```

    grid_search = GridSearchCV(adaboost, param_grid, cv=3, scoring='accuracy',
n_jobs=-1, return_train_score=True)
    grid_search.fit(X_train, y_train)

    # Get the best hyperparameters
    best_params = grid_search.best_params_
    print(f"Best Hyperparameters for Set {i}: {best_params}")

    # Evaluate on test set
    best_model = grid_search.best_estimator_
    test_accuracy = best_model.score(X_test, y_test)
    print(f"Test Accuracy for Set {i}: {test_accuracy:.4f}")

    # Compute AUC-ROC & CI
    y_test_proba = best_model.predict_proba(X_test)[: , 1]
    fpr, tpr, _ = roc_curve(y_test, y_test_proba)
    roc_auc, ci = AUCCI(y_test,
                        y_test_proba,
                        confidence_level=0.95)
    print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
    print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")

    # Save the evaluation results to a file
    eval_result = {
        'dataset': f'Set {i}',
        'best_params': best_params,
        'test_accuracy': test_accuracy,
        'roc_auc': roc_auc,
        'fpr': fpr.tolist(),
        'tpr': tpr.tolist(),
        'ci' : ci
    }

    #with open(f'/Users/jihyukgweon/Desktop/AdaBoost/AdaBoostResultSet{i}.json', 'w')
as outfile:
        #json.dump(eval_result, outfile)

    # Plot AUC-ROC Curve
    plot_roc_curve(fpr, tpr, roc_auc, i)

```

Appendix G - Python code for Support Vector Machine

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, auc, accuracy_score
import pandas as pd
import json
import matplotlib.pyplot as plt

# Define hyperparameters grid for grid search
param_grid = {
    'C': [0.1, 1, 10], # Regularization parameter
    'kernel': ['linear', 'poly', 'rbf'], # Kernel type
    'gamma': ['scale', 'auto'] # Kernel coefficient for 'rbf' and 'poly'
}

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Function to plot AUC-ROC Curve
def plot_roc_curve(fpr, tpr, roc_auc, dataset_num):
    plt.figure(figsize=(8, 6))
    plt.rcParams['font.size']=20
    plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve of SVM on dataSet {dataset_num}')
    plt.legend(loc="lower right")
    plt.savefig(f'/Users/jihyukgeon/Desktop/SVM/SVM_AUC_ROC_Set{dataset_num}.png')
    plt.show()

# To store results
results = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list,
X_test_list, y_test_list), 1):
    # Define SVM model
    svm = SVC(probability=True)

    # Perform grid search with cross-validation
```

```

    grid_search = GridSearchCV(svm, param_grid, cv=3, scoring='accuracy', n_jobs=-1,
return_train_score=True)
    grid_search.fit(X_train, y_train)

    # Get the best hyperparameters
    best_params = grid_search.best_params_
    print(f"Best Hyperparameters for Set {i}: {best_params}")

    # Evaluate on test set
    best_model = grid_search.best_estimator_
    test_accuracy = best_model.score(X_test, y_test)
    print(f"Test Accuracy for Set {i}: {test_accuracy:.4f}")

    # Compute AUC-ROC & CI
    y_test_score = best_model.decision_function(X_test)
    fpr, tpr, _ = roc_curve(y_test, y_test_score)
    roc_auc, ci = AUCCI(y_test,
                        y_test_score,
                        confidence_level=0.95)
    print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
    print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")
    # Save the evaluation results to a file
    eval_result = {
        'dataset': f'Set {i}',
        'best_params': best_params,
        'test_accuracy': test_accuracy,
        'roc_auc': roc_auc,
        'fpr': fpr.tolist(),
        'tpr': tpr.tolist(),
        'ci': ci
    }
    #with open(f'/Users/jihyukgweon/Desktop/SVM/SVMResultSet{i}.json', 'w') as
outfile:
        #json.dump(eval_result, outfile)

    # Plot AUC-ROC Curve
    plot_roc_curve(fpr, tpr, roc_auc, i)

```

Appendix H - Python code for Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, auc, roc_auc_score, accuracy_score
import pandas as pd
import json
import matplotlib.pyplot as plt

# Define the hyperparameters grid
param_grid = {
    'max_depth': [None, 5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Function to plot AUC-ROC Curve
def plot_roc_curve(fpr, tpr, roc_auc, dataset_num):
    plt.figure(figsize=(8, 6))
    plt.rcParams['font.size'] = 20
    plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve of Decision Tree on dataset {dataset_num}')
    plt.legend(loc='lower right')

plt.savefig(f'/Users/jihyukgweon/Desktop/Decisontree/DecisionTree_AUC_ROC_Set{dataset_num}.png')
plt.show()

# To store results
results = []

for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list, X_test_list, y_test_list), 1):
    # Create a decision tree classifier
    tree_clf = DecisionTreeClassifier(random_state=42)
```

```

# Perform grid search with cross-validation
grid_search = GridSearchCV(tree_clf, param_grid, cv=3, scoring='accuracy',
n_jobs=-1, return_train_score=True)
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print(f"Best Hyperparameters for Set {i}: {best_params}")

# Get the best model
best_tree_clf = grid_search.best_estimator_

# Evaluate the best model on test set
test_accuracy = best_tree_clf.score(X_test, y_test)
print(f"Test Accuracy for Set {i}: {test_accuracy:.4f}")

# Make predictions on the test set
y_pred_proba = best_tree_clf.predict_proba(X_test)[:, 1]

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

# Compute AUC-ROC & CI
roc_auc, ci = AUCCI(y_test,
                    y_pred_proba,
                    confidence_level=0.95)
print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")

# Save the evaluation results to a file
eval_result = {
    'dataset': f'Set {i}',
    'best_params': best_params,
    'test_accuracy': test_accuracy,
    'roc_auc': roc_auc,
    'fpr': fpr.tolist(),
    'tpr': tpr.tolist(),
    'ci' : ci
}

#with
open(f'/Users/jihyukgweon/Desktop/Decisiantree/DecisionTreeResultSet{i}.json', 'w') as
outfile:
    #json.dump(eval_result, outfile)

# Plot AUC-ROC Curve
plot_roc_curve(fpr, tpr, roc_auc, i)

```

Appendix I - Python code for XGBoost

```
from confidenceinterval import roc_auc_score as AUCCI
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, roc_auc_score, roc_curve, auc
import json
from plotnine import ggplot, aes, facet_grid, geom_point, labs, theme_minimal
import matplotlib.pyplot as plt

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

param_grid = {
    'n_estimators' : [2,4,6,8,10,12,14,16,18,20],
    'max_depth' : [1,2,3,4,5,6,7],
    'learning_rate' : [0.001, 0.01, 0.1, 1.0],
    'reg_lambda': [0.1, 0.5, 1, 10],
    'min_split_loss' : [0],
    'min_child_weight' : [2]
}

model = XGBClassifier()

# Function to plot results
def makesubplot(round, results, what, ylab, main, filename):
    fig, ax = plt.subplots()
    ax.plot(round, results['validation_0'][what], label='Train')
    ax.plot(round, results['validation_1'][what], label='Test')
    ax.legend()
    plt.rcParams['font.size'] = 10
    plt.xlabel('Round')
    plt.ylabel(ylab)
    plt.title(main)
    plt.savefig(filename)
    plt.show()

# Function to plot ROC-AUC curve
def plot_roc_auc(y_test, y_pred_prob, i, filename):
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
    roc_auc = roc_auc_score(y_test, y_pred_prob)
```

```

plt.figure(figsize=(8,6))
plt.rcParams['font.size'] = 20
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curve of XGBoost on dataset {i}')
plt.legend(loc='lower right')
plt.savefig(filename)
plt.show()

# To store results
results = []

# Loop through each dataset
for i, (X_train, y_train, X_test, y_test) in enumerate(zip(X_train_list, y_train_list,
X_test_list, y_test_list), 1):

    # Perform GridSearchCV
    grid_search = GridSearchCV(model, param_grid=param_grid, cv=3,
scoring='neg_log_loss', n_jobs=-1, return_train_score=True)
    result = grid_search.fit(X_train, y_train)

    # Get the best hyperparameters
    best_params = grid_search.best_params_
    print(f"Best Hyperparameters for Set {i}: {best_params}")

    best_XGB_clf = grid_search.best_estimator_
    test_accuracy = best_XGB_clf.score(X_test, y_test)
    print(f"Test Accuracy with Best Estimator for Set {i}: {test_accuracy:.4f}")

    y_pred_prob = best_XGB_clf.predict_proba(X_test)[:, 1]
    # Compute AUC-ROC
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
    roc_auc, ci = AUCCI(y_test,
                        y_pred_prob,
                        confidence_level=0.95)
    print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
    print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")

    # Save the evaluation results
    eval_result = {
        'dataset': f'Set {i}',

```

```

        'test_accuracy': accuracy,
        'roc_auc': roc_auc,
        'fpr': fpr.tolist(),
        'tpr': tpr.tolist(),
        'ci': ci
    }

    # Plot AUC-ROC Curve
    plot_roc_auc(y_test, y_pred_prob, i,
f'/Users/jihyukgweon/Desktop/XGBoost/XGBoost_AUC_ROC_Set{i}.png')

    # Store results for final plotting
    results_dict['dataset'] = f'Set {i}'
    results.append(pd.DataFrame(results_dict))

# Plot accuracy results using plotnine
def plot_results(df):
    plt = (ggplot(df)
          + aes(x='validation_0', y='mean_test_score')
          + facet_grid('param_weights ~ param_metric')
          + geom_point(aes(color='dataset'))
          + labs(title=f'XGBoost Accuracy by Round, Weights, and Metric',
                  x='Round',
                  y='Accuracy')
          + theme_minimal()
          )
    plt.draw()

```

Appendix J - XGBoost result model over training assessment for the selected best hyperparameters

```
import pandas as pd
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, roc_auc_score, roc_curve, auc
import json
from plotnine import ggplot, aes, facet_grid, geom_point, labs, theme_minimal
import matplotlib.pyplot as plt

# Training and test datasets
X_train_list = [X_train1, X_train2, X_train3, X_train4, X_train5, X_train6]
y_train_list = [y_train1, y_train2, y_train3, y_train4, y_train5, y_train6]
X_test_list = [X_test1, X_test2, X_test3, X_test4, X_test5, X_test6]
y_test_list = [y_test1, y_test2, y_test3, y_test4, y_test5, y_test6]

# Define the XGBoost model
mod1 = XGBClassifier(
    n_estimators=4,
    learning_rate=1.0,
    reg_lambda=0.1,
    min_split_loss=0,
    min_child_weight=2,
    max_depth = 1,
    objective='binary:logistic'
)
mod2 = XGBClassifier(
    n_estimators=20,
    learning_rate=0.1,
    reg_lambda=10,
    min_split_loss=0,
    min_child_weight=2,
    max_depth = 3,
    objective='binary:logistic'
)
mod3 = XGBClassifier(
    n_estimators=20,
    learning_rate=1.0,
    reg_lambda=0.5,
    min_split_loss=0,
    min_child_weight=2,
    max_depth = 1,
    objective='binary:logistic'
)
mod4 = XGBClassifier(
```

```

        n_estimators=6,
        learning_rate=1.0,
        reg_lambda=0.1,
        min_split_loss=0,
        min_child_weight=2,
        max_depth = 2,
        objective='binary:logistic'
    )
mod5 = XGBClassifier(
    n_estimators=10,
    learning_rate=1.0,
    reg_lambda=0.5,
    min_split_loss=0,
    min_child_weight=2,
    max_depth = 2,
    objective='binary:logistic'
)

mod6 = XGBClassifier(
    n_estimators=12,
    learning_rate=1.0,
    reg_lambda=0.1,
    min_split_loss=0,
    min_child_weight=2,
    max_depth = 2,
    objective='binary:logistic'
)

modlist = [mod1, mod2, mod3, mod4, mod5, mod6]

# Function to plot results
def makesubplot(round, results, what, ylab, main, filename, marginvalue):
    fig, ax = plt.subplots()
    ax.plot(round, results['validation_0'][what], label='Train')
    ax.plot(round, results['validation_1'][what], label='Test')
    ax.margins(marginvalue)
    ax.legend()
    plt.rcParams['font.size'] = 12
    plt.xlabel('Round')
    plt.ylabel(ylab)
    plt.title(main)
    plt.savefig(filename)
    plt.show()

# Function to plot ROC-AUC curve
def plot_roc_auc(y_test, y_pred_prob, i, filename):
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)

```

```

roc_auc = roc_auc_score(y_test, y_pred_prob)

plt.figure(figsize=(8,6))
plt.rcParams['font.size'] = 20
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC Curve of XGBoost on dataset {i}')
plt.legend(loc='lower right')
plt.savefig(filename)
plt.show()

# To store results
results = []

# Loop through each dataset
for i, (mod, X_train, y_train, X_test, y_test) in enumerate(zip(modlist, X_train_list,
y_train_list, X_test_list, y_test_list), 1):

    evalset = [(X_train, y_train), (X_test, y_test)]

    mod.fit(
        X_train, y_train,
        eval_metric=['logloss', 'error'],
        eval_set=evalset,
        verbose=True
    )

    results_dict = mod.evals_result()
    y_pred = mod.predict(X_test)
    y_pred_prob = mod.predict_proba(X_test)[:, 1]
    accuracy = accuracy_score(y_test, y_pred)
    print(f'Accuracy on the test set data (Set {i}): {100 * accuracy:.2f}%')

    # Save the evaluation results to a file
    with open(f'/Users/jihyukgeon/Desktop/XGBoost/XGBoostresultdat{i}.json', 'w') as
outfile:
        json.dump(results_dict, outfile)

    rounds = len(results_dict['validation_0']['error'])
    round_range = range(0, rounds)

# Plot and save the results

```

```

    makesubplot(round_range, results_dict, 'logloss', '-loglikelihood', f'XGBoost
Negative Loglikelihood on dataSet {i}', f'xgboost_loglik_set{i}.png', marginvalue =
0.2)

    makesubplot(round_range, results_dict, 'error', 'Classification error', f'XGBoost
Classification error on dataSet {i}', f'xgboost_classif_error_set{i}.png', marginvalue
= 0.05)

# Compute AUC-ROC
fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
roc_auc, ci = AUCCI(y_test,
                    y_pred_prob,
                    confidence_level=0.95)
print(f"AUC-ROC for Set {i}: {roc_auc:.4f}")
print(f"Confidence Interval for AUC {i}: {ci[0]:.3f},{ci[1]:.3f}")

# Save the evaluation results
eval_result = {
    'dataset': f'Set {i}',
    'test_accuracy': accuracy,
    'roc_auc': roc_auc,
    'fpr': fpr.tolist(),
    'tpr': tpr.tolist(),
    'ci': ci
}

# Plot AUC-ROC Curve
plot_roc_auc(y_test, y_pred_prob, i,
f'/Users/jihyukgweon/Desktop/XGBoost/XGBoost_AUC_ROC_Set{i}.png')

# Store results for final plotting
results_dict['dataset'] = f'Set {i}'
results.append(pd.DataFrame(results_dict))

# Plot accuracy results using plotnine
def plot_results(df):
    plt = (ggplot(df)
          + aes(x='validation_0', y='mean_test_score')
          + facet_grid('param_weights ~ param_metric')
          + geom_point(aes(color='dataset'))
          + labs(title=f'XGBoost Accuracy by Round, Weights, and Metric',
                  x='Round',
                  y='Accuracy')
          + theme_minimal()
          )
    plt.draw()

```

