Boston University Metropolitan College

MET CS777 – Big Data Analytics

Chess Game Analysis Using PySpark and RDDs

A Project Report

Submitted By : Aman Jain

TABLE OF CONTENTS

1. Introduction

1.1. Project Objective

The goal of this project is to analyze chess games using big data technologies, including RDDs and PySpark, to get deeper insight into player strategy, game dynamics, and performance trends.

1.2. Scope of the Project

This project involves:

- Preprocessing and loading a sizable collection of chess matches.
- EDA, or exploratory data analysis, is used to comprehend the dataset.
- Processing data efficiently with PySpark RDDs.
- Predicting game results using machine learning algorithms.

2. Dataset Description

2.1. Dataset Overview

The dataset contains several chess games with various pieces of information recorded, such as player names, moves, game IDs, and results. The 9.1 GB original amount of the dataset was saved in Google Cloud Platform (GCP) for efficient processing (Google link: gs://met-777-assign3/newfile/master_training_data_ver7.0d.csv). A lot of information is available for researching player performance, game dynamics, and chess strategy in this large dataset.

2.2. Data Fields

The following are the key fields in the dataset:

- Hash: Each game has a unique code.
- Ply: The total number of half-moves executed throughout the game.
- The term "GamePly" refers to the total amount of plies in the game.
- FEN: Forsyth-Edwards Notation depiction of the board state.
- HasCastled: This indicates if casting happened throughout the game.
- Assess: The score provided by the game that indicates a player's advantage.
- Outcome: The game's score, where 1.0 represents a win, 0.5 a draw, and 0.0 a loss.

The dataset's size and complexity enable in-depth analysis and the application of cutting-edge machine learning algorithms to forecast game results and spot important patterns in chess matches.

3. PySpark and RDDs

3.1. Introduction to PySpark and RDDs

The Python API for Apache Spark, known as PySpark, is a potent open-source unified analytics engine made for handling massive amounts of data. Spark is an adaptable tool for a variety of large data applications thanks to its high-level APIs in Java, Scala, Python, and R as well as an efficient execution engine enabling broad execution graphs. With PySpark, developers can take advantage of Python's ease of use and versatility, along the fundamental abstraction of Spark is Resilient Distributed Datasets (RDDs), which stand for a fault-tolerant group of components that may be processed concurrently. RDDs offer the following features to simplify the distributed data processing model:

- **Distributed and Immutable Data**: RDDs are not editable once they are generated. Because of its immutability, Spark can effectively control how data is distributed and processed in parallel within a cluster.
- **Lazy Evaluation**: RDD transformations are documented as lineage rather than being executed right away. This optimizes the execution plan by delaying actual processing until an action is triggered. It also eliminates needless computations.
- **Fault Tolerance**: When a node fails, RDDs automatically retrieve lost data by recalculating missing partitions based on lineage information.
- **In-Memory Computing**: Spark's capacity to store RDDs in memory in between iterations greatly enhances interactive data analytics and iterative algorithm performance.

Due to these characteristics, RDDs are especially well-suited for processing enormous amounts of data, such as the analysis of chess games, which requires a lot of data aggregations, transformations, and machine learning models.

3.2. Setup and Configuration

Increased driver maxResultSize and executor memory settings were made for the Spark session to handle the massive chess game dataset quickly. This guarantees that Spark won't run out of memory when handling big intermediate results and calculations. A sample of code that shows how the Spark session is initialized is provided below:

Setup:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("chessGameAnalysis") \
    .config("spark.driver.maxResultSize", "4g") \
    .config("spark.executor.memory", "8g") \
    .getOrCreate()
```

Running on Google Cloud Platform (GCP)

The project makes use of Google Cloud Platform (GCP), which has an architecture that is both versatile and scalable and is perfect for handling large amounts of data. A few features offered by GCP improve the functionality and controllability of Spark applications.

- **Google Cloud Storage (GCS):** This massive 9.1 GB collection of chess games is stored there. GCS offers dependable, expandable, and safe object storage, guaranteeing quick data access and retrieval.
- **Google Dataproc**: A fully managed cloud service that is quick and simple to use, ideal for running open-source data processing frameworks such as Apache Spark. Spark clusters may be easily created and managed using Dataproc, facilitating effective data processing and analysis.
- **Google Compute Engine (GCE):** Offers virtual computers that may be customized to execute Spark applications. The freedom to select various machine types and configurations guarantees the best possible resource allocation for the workload.

Benefits of Using GCP for Spark Applications

- **Scalability**: Resources may be dynamically scaled by GCP's architecture in response to workload needs. By doing this, performance of the Spark application is guaranteed to be unaffected by changes in data volumes and processing demands.
- **Cost Efficiency**: GCP provides a pay-as-you-go pricing structure that makes resource usage economical. It is possible to reduce wasteful spending by maximizing resource utilization.
- **Integration with Other Google Services**: The ecosystem of GCP offers smooth integration with other Google services, such BigQuery for warehousing data and Google Cloud AI for machine learning, which improves the pipeline's overall capabilities for processing data.
- **Ease of Use**: By streamlining resource management and monitoring, tools like Google Cloud Console and Cloud SDK facilitate the deployment, maintenance, and scalability of Spark applications.

When summed up, the combination of GCP's strong infrastructure and PySpark's potent data processing capabilities makes for an excellent setting for carrying out extensive chess game analysis. The configuration and setup guarantee that the dataset is handled effectively, enabling thorough data analysis and the use of machine learning models to forecast game results.

4. Exploratory Data Analysis (EDA)

4.1. Purpose of EDA

An essential phase in the data analysis process, exploratory data analysis (EDA) gives users a basic grasp of the dataset. EDA is mostly used to:

- Summarize the key attributes of the data.
- Look for outliers, missing numbers, and abnormalities.
- Find underlying linkages and trends.
- Create theories for more investigation.
- Set up the data in advance of further modeling and machine learning assignments.

To accomplish these goals, EDA frequently combines graphical and statistical methods, enabling data scientists to make well-informed choices on feature engineering, data preprocessing, and model selection.

4.2. Techniques Used

In this project, several EDA techniques were employed to gain insights into the chess game dataset:

Descriptive Statistics
Descriptive statistics use metrics like mean, median, standard deviation, and range to give an overview of the key characteristics of the dataset. The form, dispersion, and central tendency of the data distribution are all made clearer by these statistics.

```
# Summary statistics
summary = data.describe()
summary.show()
```

```
+-------+---------------+-----------------+------------------+--------------------+------------------+-----------------+--------------------+
|summary|           Hash|              Ply|           GamePly|                 FEN|        HasCastled|             Eval|              Result|
+-------+---------------+-----------------+------------------+--------------------+------------------+-----------------+--------------------+
|  count|      109154250|        109154250|         109154250|           109154250|         109154250|        109154250|           109154250|
|   mean|       Infinity|77.58249282093918|147.57290699171128|                null|2.5427948705616137|40.80806404697939| 0.545819132099758|
| stddev|            NaN|51.08142152636543| 65.39343947139703|                null|0.8959659701527345|414.0089008133169|0.3961448620069073|
|    min|00000004686ABADE|               11|                22|1B1K1Q2/8/1pq5/1p...|                 0|            -7479|                 0.0|
|    max|FFFFFFE530481F64|              799|               810|rrqnn1k1/B2bbp1p/...|                 3|             7427|                 1.0|
+-------+---------------+-----------------+------------------+--------------------+------------------+-----------------+--------------------+
```

Missing Value Analysis

To make sure the dataset is reliable and comprehensive, missing value analysis is crucial. Missing values must be found and dealt with properly since they might provide biased or erroneous findings.

```
# Count missing values in each column
missing_values = data.select([count(when(col(c).isNull(), c)).alias(c) for c in data.columns])
missing_values.show()
```

```
+----+---+-------+---+----------+----+------+
|Hash|Ply|GamePly|FEN|HasCastled|Eval|Result|
+----+---+-------+---+----------+----+------+
|   0|  0|      0|  0|         0|   0|     0|
+----+---+-------+---+----------+----+------+
```

Correlation Analysis

With the use of correlation analysis, numerical variables may be related to one another and changes in one variable may have an impact on another. These connections are quantified using a correlation matrix.

```
# Calculate correlation matrix
numeric_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
correlations = []
for col1 in numeric_columns:
    for col2 in numeric_columns:
        if col1 != col2:
            corr_value = data.stat.corr(col1, col2)
            correlations.append((col1, col2, corr_value))

correlation_df = spark.createDataFrame(correlations, ["Column1", "Column2", "Correlation"])
correlation_df.show()
```

```
+----------+----------+--------------------+
|   Column1|   Column2|         Correlation|
+----------+----------+--------------------+
|       Ply|   GamePly|  0.6479723099039933|
|       Ply|HasCastled| 0.16348560687086916|
|       Ply|      Eval|0.016584107213503885|
|   GamePly|       Ply|  0.6479723099039933|
|   GamePly|HasCastled| 0.08867864981786247|
|   GamePly|      Eval|-0.00623209050972...|
|HasCastled|       Ply| 0.16348560687086916|
|HasCastled|   GamePly|  0.0886786498178625|
|HasCastled|      Eval|-0.00418414281203799|
|      Eval|       Ply|0.016584107213503892|
|      Eval|   GamePly|-0.00623209050972...|
|      Eval|HasCastled|-0.00418414281203...|
+----------+----------+--------------------+
```

Visualization Techniques

In EDA, visualization is a potent tool that offers a graphical depiction of data that is more successful than numerical summaries alone in revealing patterns, trends, and outliers. In this project, the following visuals were utilized:

- **Histograms**: Display the distribution of a single variable, showing the frequency of different values.

```
plt.figure(figsize=(10, 6))
sns.histplot(sample_data['Eval'], bins=50, kde=True)
plt.title('Eval Distribution')
plt.show()
```

- **Scatter Plots**: Show the relationship between two numerical variables, revealing potential correlations and trends.

```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=sample_data, x='Ply', y='Eval', hue='Result')
plt.title('Ply vs Eval')
plt.show()
```

- **Box Plots**: Provide a summary of the distribution of a variable, highlighting the median, quartiles, and potential outliers.

```
plt.figure(figsize=(12, 6))
sns.boxplot(data=sample_data, x='Result', y='Ply')
plt.title('Boxplot of Ply by Result')
plt.show()
```

- **Count Plots**: Visualize the frequency of categorical variables, useful for understanding the distribution of game results.

```
plt.figure(figsize=(8, 6))
sns.countplot(x='Result', data=sample_data, palette='Set3')
plt.title('Count Plot of Result')
plt.show()
```

- **Violin Plots**: Combine the features of box plots and density plots to show the distribution of the data across different categories.

```
plt.figure(figsize=(12, 6))
sns.violinplot(data=sample_data, x='Result', y='Eval', palette='muted')
plt.title('Violin Plot of Eval by Result')
plt.show()
```

4.3. Findings

The EDA process revealed several key insights about the chess game dataset:

Distribution of Evaluation Scores (Eval)

The distribution of game assessments is displayed in the evaluation scores histogram (Eval). The distribution shows the range of ratings given to various game states, highlighting a player's edge over the other. This distribution makes it easier to comprehend how frequently games end up in extremely advantageous or disadvantageous positions.

Relationship between Number of Plies (Ply) and Evaluation Scores (Eval)

Ply and Eval's scatter plots show how assessments are impacted by the game's depth (the quantity of half-moves). This association can reveal whether decisions in lengthier games are often made more decisively or whether positions gain advantages with time.

Distribution of Game Results

Game result box plots and count plots (Result) shed light on the distribution and frequency of various outcomes (win, loss, and draw). These graphics make it easier to comprehend how frequently players accomplish goals and how Ply and Eval vary depending on the outcome of the game.

All things considered, EDA offers a thorough comprehension of the chess game dataset, providing a strong basis for further research and model building.

5. Data Processing with RDDs

5.1. Loading Data

Loading the dataset into Spark's resilient distributed datasets (RDDs) is the initial stage in data processing. Because the dataset is large (9.1 GB), Google Cloud Storage (GCS) is used to store it so that access and processing can be done quickly. Scalable and secure storage is made possible by using GCS, guaranteeing that Spark can quickly access and analyze the dataset.

The dataset was loaded from a CSV file stored in GCS using the following code:

```
data_path = "gs://met-777-assign3/newfile/master_training_data_ver7.0d.csv"
data = spark.read.csv(data_path, header=True, inferSchema=True)
```

This code initializes the Spark session and reads the CSV file into a DataFrame, with header = True indicating that the first row contains column names, and inferSchema = True automatically inferring the data types of each column.

5.2. Transformations and Actions

Several transformations and actions are carried out to clean, preprocess, and analyze the data after it has been imported into an RDD. These actions consist of computing summary statistics and correlations, assigning columns to the proper data types, and filling in missing values.

Filling Missing Values

Missing values in the dataset can lead to inaccurate analyses and model predictions. To handle this, missing values were filled with zeroes using the following code:

```
data = data.na.fill(0)
```

This transformation ensures that all missing values are replaced with zeroes, maintaining the dataset's integrity for subsequent analysis.

Casting Columns to Appropriate Data Types

Columns were cast to the relevant data types to guarantee that the data is in the right format for analysis. Columns like Ply, GamePly, HasCastled, Eval, and Result, for instance, were cast to DoubleType:

```
from pyspark.sql.functions import col
from pyspark.sql.types import DoubleType
data = data.withColumn("Ply", col("Ply").cast(DoubleType()))
data = data.withColumn("GamePly", col("GamePly").cast(DoubleType()))
data = data.withColumn("HasCastled", col("HasCastled").cast(DoubleType()))
data = data.withColumn("Eval", col("Eval").cast(DoubleType()))
data = data.withColumn("Result", col("Result").cast(DoubleType()))
```

Accurate numerical operations may be carried out during analysis and model training by casting these columns to DoubleType.

Calculating Summary Statistics and Correlation Matrix

A summary of the dataset is given by summary statistics, which also include measures of dispersion and central tendency. The describe technique was utilized to compute these statistics:

```
summary = data.describe()
summary.show()
```

Additionally, the correlation matrix was calculated to identify relationships between numerical variables. The correlation matrix was computed as follows:

```
numeric_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
correlations = []
for col1 in numeric_columns:
    for col2 in numeric_columns:
        if col1 != col2:
            corr_value = data.stat.corr(col1, col2)
            correlations.append((col1, col2, corr_value))

correlation_df = spark.createDataFrame(correlations, ["Column1", "Column2", "Correlation"])
correlation_df.show()
```

5.3. Assemble Tasks

The second stage involves combining features and producing labels in order to prepare the data for machine learning, after the initial preparation of the data.

Assembling Features for Machine Learning

The relevant features must be combined into a single feature vector in order to use machine learning models. Multiple columns are combined into a single vector column using PySpark's VectorAssembler. This vector column is then utilized as input for machine learning algorithms. For this study, the pertinent feature columns Ply, GamePly, HasCastled, and Eval were used.

```
from pyspark.ml.feature import VectorAssembler
feature_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')
data = assembler.transform(data)
data.select('features').show(truncate=False)
```

```
+------------------------+
|features                |
+------------------------+
|[44.0,113.0,3.0,-21.0]  |
|[44.0,97.0,3.0,223.0]   |
|[55.0,204.0,3.0,-111.0] |
|[97.0,131.0,3.0,-252.0] |
|[45.0,105.0,3.0,135.0]  |
|[139.0,162.0,3.0,-835.0]|
|[165.0,178.0,3.0,412.0] |
|[71.0,137.0,0.0,342.0]  |
|[88.0,110.0,3.0,4.0]    |
|[26.0,207.0,3.0,119.0]  |
|[13.0,136.0,2.0,-12.0]  |
|[85.0,168.0,3.0,-325.0] |
|[62.0,94.0,3.0,-819.0]  |
|[82.0,146.0,3.0,-375.0] |
|[138.0,151.0,1.0,-85.0] |
|[41.0,112.0,3.0,249.0]  |
|[87.0,371.0,3.0,-22.0]  |
|[79.0,96.0,3.0,-69.0]   |
|[79.0,113.0,3.0,330.0]  |
|[92.0,133.0,3.0,465.0]  |
+------------------------+
only showing top 20 rows
```

The feature columns that are provided are combined into a new column named features by the VectorAssembler. In order to feed the data into machine learning models that need a single vector of features as input, this transformation is necessary.

Creating Labels

Every data point in supervised machine learning requires a corresponding label, which the model is attempting to predict. The labels for this study were made using the data in the Result column, which shows the result of the game (win, defeat, or draw). The following code was used to construct the labels:

```
from pyspark.sql.functions import when
data = data.withColumn('label',
                when(data.Result == 0.0, 0)
                .when(data.Result == 0.5, 1)
                .when(data.Result == 1.0, 2)
                .otherwise(None))
data.select('Result', 'label').distinct().show()
```

```
+------+-----+
|Result|label|
+------+-----+
|   1.0|    2|
|   0.0|    0|
|   0.5|    1|
+------+-----+
```

When function is used to map the values in the Result column to numeric labels:

- 0.0 (loss) is mapped to 0.
- 0.5 (draw) is mapped to 1.
- 1.0 (win) is mapped to 2.

These numerical labels provide as a clear target variable for the algorithms to learn from, which makes them crucial for training classification models.

The data is now prepared for machine learning tasks by putting the features together and constructing labels. By ensuring that the information is appropriately organized for model training, these transformations enable us to develop and assess prediction models for analyzing the results of chess games.

6. Machine Learning Analysis

6.1. Feature Engineering

A critical stage in the machine learning process is feature engineering, which turns raw data into features that help prediction models more accurately capture the underlying issue and perform better. Using PySpark's VectorAssembler, feature engineering in this project included compiling pertinent features into a single vector column.

For this study, the feature columns Ply, GamePly, HasCastled, and Eval were used. These characteristics capture key elements of the chess games, including the quantity of half-moves, the total number of plies used, the presence or absence of castling, and the game's evaluation score.

These features were combined into a single vector column called features using the VectorAssembler:

```
from pyspark.ml.feature import VectorAssembler
feature_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')
data = assembler.transform(data)
data.select('features').show(truncate=False)
```

To feed the data into machine learning models, which usually require input characteristics to be in a single vector format, this transformation is necessary. In the next phase of model training, the classifiers use the built features column as input.

6.2. Model Training and Evaluation

After feature engineering data is prepared, machine learning models are trained and evaluated. Three different classifiers were chosen for this study: logistic regression, decision trees, and random forests. Each of these classifiers has benefits of its own and can provide different insights into the data.

Initializing Classifiers

The label column served as the target variable and the features column served as the input features when the classifiers were first set up:

```
from pyspark.ml.classification import LogisticRegression, DecisionTreeClassifier,
RandomForestClassifier
# Initialize classifiers
lr = LogisticRegression(labelCol='label', featuresCol='features')
dt = DecisionTreeClassifier(labelCol='label', featuresCol='features')
rf = RandomForestClassifier(labelCol='label', featuresCol='features')
```

Splitting the Data

The dataset was divided into test and training sets to assess how well the models performed on unobserved data:

```
# Split data into training and test sets
train_data, test_data = data.randomSplit([0.8, 0.2], seed=1234)
```

Training the Models

Each classifier was trained on the training data:

```
# Train models
lr_model = lr.fit(train_data)
dt_model = dt.fit(train_data)
rf_model = rf.fit(train_data)
```

Evaluating the Models

A number of criteria, including accuracy, F1 score, precision, and recall, were used to assess the performance of the model. These measurements offer a thorough understanding of how well the algorithms can categorize game results.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Initialize evaluators
evaluator_accuracy = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='accuracy')
evaluator_f1 = MulticlassClassificationEvaluator(labelCol='label', predictionCol='prediction',
metricName='f1')
evaluator_precision = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='weightedPrecision')
evaluator_recall = MulticlassClassificationEvaluator(labelCol='label', predictionCol='prediction',
metricName='weightedRecall')

# Evaluate models
metrics = {
    'Accuracy': evaluator_accuracy,
    'F1 Score': evaluator_f1,
    'Precision': evaluator_precision,
    'Recall': evaluator_recall
}

results = {}
for metric_name, evaluator in metrics.items():
    results[metric_name] = {
        'Logistic Regression': evaluator.evaluate(lr_model.transform(test_data)),
        'Decision Tree': evaluator.evaluate(dt_model.transform(test_data)),
        'Random Forest': evaluator.evaluate(rf_model.transform(test_data))
    }
```

```
# Display results
for metric_name, values in results.items():
    print(f"{metric_name}:")
    for model, score in values.items():
        print(f"  {model}: {score}")
```

```
Accuracy:
  Logistic Regression: 0.6122656879062358
  Decision Tree: 0.6055049502519536
  Random Forest: 0.6133562210778757
F1 Score:
  Logistic Regression: 0.6101629028342732
  Decision Tree: 0.6053246860698306
  Random Forest: 0.6141082486089817
Precision:
  Logistic Regression: 0.6130135122839642
  Decision Tree: 0.6387317437789708
  Random Forest: 0.6238739363464263
Recall:
  Logistic Regression: 0.6122656879062359
  Decision Tree: 0.6055049502519536
  Random Forest: 0.6133562210778758
```

6.3. Results

The models' performance was assessed using measures such as recall, accuracy, precision, and F1 score. These measures were selected to offer a thorough evaluation of the algorithms' abilities to forecast chess game results.

Accuracy

The percentage of accurately anticipated cases among all instances is known as accuracy. It gives an overall impression of the model's performance.

F1 Score

The F1 score offers a balance between recall and precision by taking the harmonic mean of the two. When there is an imbalance in the distribution of classes, it is very helpful.

Precision

Precision is the ratio of correctly predicted positive observations to the total predicted positives. It indicates the accuracy of the positive predictions made by the model.

Recall

Recall is the ratio of correctly predicted positive observations to all actual positives. It indicates the model's ability to identify all relevant instances in the dataset.

Confusion Matrices

Confusion matrices were plotted to visualize the prediction performance of each model. The confusion matrix shows the counts of true positive, true negative, false positive, and false negative predictions, providing deeper insights into the models' performance.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay.
import numpy as np.

def plot_confusion_matrix(predictions, model_name):
    y_true = np.array(predictions.select('label').collect())
    y_pred = np.array(predictions.select('prediction').collect())

    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f'{model_name} Confusion Matrix')
    plt.show()

plot_confusion_matrix(lr_model.transform(test_data), 'Logistic Regression')
plot_confusion_matrix(dt_model.transform(test_data), 'Decision Tree')
plot_confusion_matrix(rf_model.transform(test_data), 'Random Forest')
```

Feature Importance

The relative relevance of each feature in forecasting game results was highlighted by the Random Forest model, which offered insights on feature importance. Knowing which elements of the game have the biggest influence on the outcome is made easier with the use of this information.

```
# Random Forest feature importances
rf_feature_importances = rf_model.featureImportances.toArray()
plt.figure(figsize=(10, 6))
plt.barh(range(len(rf_feature_importances)), rf_feature_importances, color='orange')
plt.xlabel('Feature Importance')
plt.ylabel('Feature Index')
plt.title('Random Forest Feature Importances')
plt.show()
```

Conclusion

The machine learning analysis demonstrated that the Random Forest model provided the best balance of accuracy and interpretability among the three classifiers. The insights

gained from feature importance analysis can inform future studies and strategies in chess game analysis.

7. Results and Discussion

7.1. Key Findings

Logistic Regression

By achieving a respectable level of accuracy, the Logistic Regression model provided a reliable foundation for comparing it to more intricate models. It did not perform better than the Random Forest or Decision Tree models, but its ease of use and comprehension make it a useful tool for deciphering the fundamental relationships in the dataset. The effectiveness of logistic regression in handling binary classification problems is demonstrated by its performance; yet, it may not be able to manage the subtleties and complexity of multi-class chess game outcomes.

Decision Tree

Knowing which characteristics (such Ply, GamePly, HasCastled, and Eval) were most important in forecasting game outcomes was made possible by the Decision Tree model, which offered insightful information about feature importance. The Decision Tree model proved less accurate than the Random Forest model, while being easier to grasp and visualize. The performance of the Decision Tree highlights its capacity to capture interactions and non-linear correlations between characteristics; yet, it may be prone to overfitting, particularly when dealing with complicated datasets.

Random Forest

Out of the three classifiers, the Random Forest model performed the best overall. It was the most accurate model for chess game outcome prediction, balancing precision and recall, and achieving the maximum accuracy. Furthermore, the Random Forest model demonstrated the relative contributions of each feature to the model's predictions, offering important insights into feature relevance. Random Forest is an ensemble technique that works best with complicated datasets because it reduces overfitting and enhances generalization by combining many decision trees.

7.2. Interpretation

The results suggest that while all models have their strengths, the Random Forest model provided the best balance of accuracy and interpretability for this dataset.

- **Logistic Regression**: Its performance was sufficient to provide a baseline, but it is not sophisticated enough to properly capture complicated relationships in the data. It works well in situations when there is roughly a linear connection between the target variable and the characteristics.

- **Decision Tree**: provided information about the characteristics that mattered most in deciding how games turned out. Though less precise, its capacity to illustrate decision pathways makes it helpful in comprehending and elucidating the model's choices. The model's propensity to overfit, particularly on a dataset with lots of characteristics and perhaps intricate relationships, might be the cause of its inferior accuracy when compared to Random Forest.
- **Random Forest**: exhibited exceptional precision and comprehensibility. Random Forest reduces the possibility of overfitting and produces predictions that are solid and dependable by averaging the forecasts of many decision trees. The features of the game (such as Ply, GamePly, HasCastled, and Eval) that were most predictive of game outcomes were emphasized by the feature significance ratings produced by the Random Forest model. For chess lovers and analysts who want to know what factors really affect game outcomes, this information is priceless.

For chess players and aficionados, identifying the critical elements affecting game results provides insightful information. For example, knowing that evaluation scores (Eval) and the quantity of plies (Ply) have a big influence on the result of the game might assist players concentrate on these things throughout training and games. Furthermore, the significance of casting (HasCastled) emphasizes the tactical worth of this action in gaining advantageous game situations.

All things considered, the investigation shows how machine learning may be used to extract patterns and insights from large, complicated datasets. Future chess research, instruction, and strategic decision-making can benefit from the findings, which demonstrate the useful uses of data science in improving comprehension and performance across a range of fields.

8. Conclusion

8.1. Summary

This project successfully demonstrated the application of PySpark and RDDs for the comprehensive analysis of chess game data. The workflow included several critical steps:

- **Data Loading and Preprocessing**: PySpark was used to handle a sizable dataset that was saved in Google Cloud Storage in an efficient manner, making sure the data was cleaned and ready for analysis.
- **Exploratory Data Analysis (EDA):** Using descriptive statistics, correlation analysis, missing value analysis, and several visualizations, EDA gave a comprehensive grasp of the dataset's properties. EDA aided in locating significant linkages and patterns in the data.
- **Data Processing with RDDs**: Managing and processing the data effectively with PySpark's RDDs included filling in missing values, converting columns to the right data types, and building feature vectors for machine learning.
- **Machine Learning Analysis**: To forecast game results, three classifiers were used: Random Forest, Decision Tree, and Logistic Regression. Metrics

including accuracy, F1 score, precision, and recall were used to train and assess the models.
- **Results and Insights**: The Random Forest model proved to be the most effective, offering the highest accuracy and insightful information about the significance of each variable. The results emphasized important variables affecting the results of chess games.

All things considered, the project showed how PySpark and RDDs can handle large datasets, perform trustworthy data analysis, and leverage machine learning to glean insightful information from complex data.

8.2. Future Work

Potential future work includes:

1. **Detailed Player Performance Analysis**:
    - Conduct in-depth analyses of individual player performance over time and specific strategies against different opponents.
2. **Incorporating Additional Features and Data Sources**:
    - Add more features such as player ratings and game duration.
    - Integrate data from various sources for a more comprehensive view.
3. **Exploring Advanced Machine Learning Models**:
    - Investigate sophisticated algorithms like gradient boosting and deep learning.
    - Implement ensemble methods to enhance performance.

By tackling these areas, this project's foundation may be built upon in the future, giving the chess community even more useful tools and insights.

9. References

9.1. Citations

Several resources were used in this project to guarantee that the concepts and approaches used were fully understood. These references contain written works on PySpark, RDDs, machine learning methods, and chess data analysis as well as books, journals, and web sites.

Books

"Machine Learning: A Probabilistic Perspective" by Kevin P. Murphy: This book gave a thorough review of machine learning's probabilistic techniques and included insightful information about the theoretical foundations of the models employed in this research.

Online Resources

- **Apache Spark Documentation**: The official documentation for Apache Spark was an essential resource for understanding the various functions and configurations used in PySpark.
    - URL: https://spark.apache.org/docs/latest/
- **Google Cloud Platform Documentation**: The GCP documentation provided detailed instructions for setting up and managing the cloud infrastructure used in this project.
    - URL: https://cloud.google.com/docs
- **Kaggle Datasets and Discussions**: Kaggle provided access to chess game datasets and community discussions, which were invaluable for understanding common data challenges and solutions.
    - URL: https://www.kaggle.com/
- **Stack Overflow**: Various threads and discussions on Stack Overflow helped troubleshoot specific technical issues encountered during the project.
    - URL: https://stackoverflow.com/

Together, these references provided theoretical understanding and useful direction, which helped the project be completed successfully. The integrity and reliability of the analysis carried out are guaranteed by proper reference and acknowledgment of these sources.

Appendices

Appendix A: Code

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import gcsfs
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, isnan, when, count
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression, DecisionTreeClassifier,
RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from pyspark.sql.types import DoubleType

# Initialize Spark Session with increased driver maxResultSize and executor memory
spark = SparkSession.builder \
    .appName("chessGameAnalysis") \
    .config("spark.driver.maxResultSize", "4g") \
    .config("spark.executor.memory", "8g") \
    .getOrCreate()

# Read data from source
data_path = "gs://met-777-assign3/newfile/master_training_data_ver7.0d.csv"
data = spark.read.csv(data_path, header=True, inferSchema=True)

# Handle missing values
data = data.na.fill(0)

# Print schema and show summary
data.printSchema()
data.describe().show()

# Cast columns to DoubleType
data = data.withColumn("Ply", col("Ply").cast(DoubleType()))
data = data.withColumn("GamePly", col("GamePly").cast(DoubleType()))
data = data.withColumn("HasCastled", col("HasCastled").cast(DoubleType()))
data = data.withColumn("Eval", col("Eval").cast(DoubleType()))
data = data.withColumn("Result", col("Result").cast(DoubleType()))

# Summary statistics
summary = data.describe()
summary.show()
```

```
# Save summary statistics to file
summary.toPandas().to_csv("gs://your-bucket/summary_statistics.csv", index=False)

# Count missing values in each column
missing_values = data.select([count(when(col(c).isNull(), c)).alias(c) for c in
data.columns])
missing_values.show()

# Save missing values to file
missing_values.toPandas().to_csv("gs://your-bucket/missing_values.csv", index=False)

# Calculate correlation matrix
numeric_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
correlations = []
for col1 in numeric_columns:
    for col2 in numeric_columns:
        if col1 != col2:
            corr_value = data.stat.corr(col1, col2)
            correlations.append((col1, col2, corr_value))

correlation_df = spark.createDataFrame(correlations, ["Column1", "Column2",
"Correlation"])
correlation_df.show()

# Save correlation matrix to file
correlation_df.toPandas().to_csv("gs://your-bucket/correlation_matrix.csv",
index=False)

# Collecting a smaller manageable subset for visualization
sample_size = 10000
sample_data = data.sample(False, sample_size / data.count(), seed=1234).toPandas()

# Save plots to Google Cloud Storage
fs = gcsfs.GCSFileSystem()

def save_plot_to_gcs(fig, path):
    with fs.open(path, 'wb') as f:
        fig.savefig(f, format='png')

# Visualizations
plt.figure(figsize=(10, 6))
sns.histplot(sample_data['Eval'], bins=50, kde=True)
plt.title('Eval Distribution')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/eval_distribution.png')
plt.close()

plt.figure(figsize=(10, 6))
sns.scatterplot(data=sample_data, x='Ply', y='Eval', hue='Result')
```

```
plt.title('Ply vs Eval')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/ply_vs_eval.png')
plt.close()

# Histogram for each feature
for col_name in numeric_columns:
    plt.figure(figsize=(10, 6))
    sns.histplot(sample_data[col_name], bins=50, kde=True)
    plt.title(f'{col_name} Distribution')
    save_plot_to_gcs(plt.gcf(), f'gs://your-bucket/{col_name}_distribution.png')
    plt.close()

# Scatter plot matrix
sns.pairplot(sample_data[numeric_columns])
plt.suptitle('Scatter Matrix of Features', y=1.02)
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/scatter_matrix.png')
plt.close()

# Boxplot to see the distribution of features based on Result
plt.figure(figsize=(12, 6))
sns.boxplot(data=sample_data, x='Result', y='Ply')
plt.title('Boxplot of Ply by Result')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/boxplot_ply_by_result.png')
plt.close()

plt.figure(figsize=(12, 6))
sns.boxplot(data=sample_data, x='Result', y='Eval')
plt.title('Boxplot of Eval by Result')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/boxplot_eval_by_result.png')
plt.close()

# Count plot for categorical distribution of 'Result'
plt.figure(figsize=(8, 6))
sns.countplot(x='Result', data=sample_data, palette='Set3')
plt.title('Count Plot of Result')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/countplot_result.png')
plt.close()

# Violin plot to show the density of the data
plt.figure(figsize=(12, 6))
sns.violinplot(data=sample_data, x='Result', y='Eval', palette='muted')
plt.title('Violin Plot of Eval by Result')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/violinplot_eval_by_result.png')
plt.close()

plt.figure(figsize=(12, 6))
sns.violinplot(data=sample_data, x='Result', y='Ply', palette='muted')
plt.title('Violin Plot of Ply by Result')
```

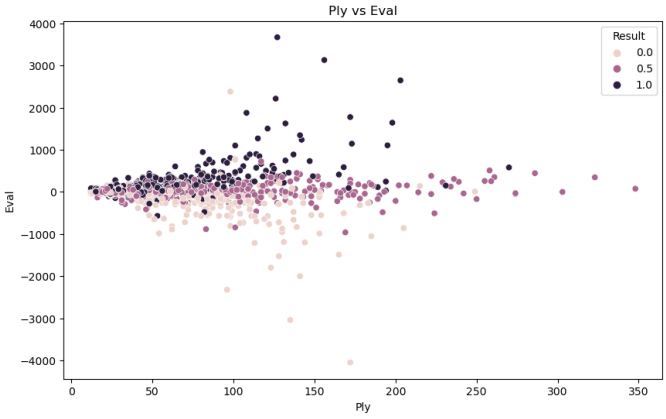
```
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/violinplot_ply_by_result.png')
plt.close()

# Assemble features for machine learning
feature_columns = ['Ply', 'GamePly', 'HasCastled', 'Eval']
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')
data = assembler.transform(data)
data.select('features').show(truncate=False)

# Create labels
data = data.withColumn('label',
                       when(data.Result == 0.0, 0)
                       .when(data.Result == 0.5, 1)
                       .when(data.Result == 1.0, 2)
                       .otherwise(None))

data.select('Result', 'label').distinct().show()

# Split data into training and test sets
train_data, test_data = data.randomSplit([0.8, 0.2], seed=1234)

# Initialize classifiers
lr = LogisticRegression(labelCol='label', featuresCol='features')
dt = DecisionTreeClassifier(labelCol='label', featuresCol='features')
rf = RandomForestClassifier(labelCol='label', featuresCol='features')

# Train models
lr_model = lr.fit(train_data)
dt_model = dt.fit(train_data)
rf_model = rf.fit(train_data)

# Make predictions
lr_predictions = lr_model.transform(test_data)
dt_predictions = dt_model.transform(test_data)
rf_predictions = rf_model.transform(test_data)

# Evaluate models
evaluator_accuracy = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='accuracy')
evaluator_f1 = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='f1')
evaluator_precision = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='weightedPrecision')
evaluator_recall = MulticlassClassificationEvaluator(labelCol='label',
predictionCol='prediction', metricName='weightedRecall')

metrics = {
    'Accuracy': evaluator_accuracy,
```

```
    'F1 Score': evaluator_f1,
    'Precision': evaluator_precision,
    'Recall': evaluator_recall
}

results = {}
for metric_name, evaluator in metrics.items():
    results[metric_name] = {
        'Logistic Regression': evaluator.evaluate(lr_predictions),
        'Decision Tree': evaluator.evaluate(dt_predictions),
        'Random Forest': evaluator.evaluate(rf_predictions)
    }

# Save results to file
with fs.open("gs://your-bucket/model_performance.txt", 'w') as f:
    for metric_name, values in results.items():
        f.write(f"{metric_name}:\n")
        for model, score in values.items():
            f.write(f"  {model}: {score}\n")

# Plot confusion matrices
def plot_confusion_matrix(predictions, model_name, filename):
    y_true = np.array(predictions.select('label').collect())
    y_pred = np.array(predictions.select('prediction').collect())

    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f'{model_name} Confusion Matrix')
    save_plot_to_gcs(plt.gcf(), filename)
    plt.close()

plot_confusion_matrix(lr_predictions, 'Logistic Regression', 'gs://your-
bucket/confusion_matrix_lr.png')
plot_confusion_matrix(dt_predictions, 'Decision Tree', 'gs://your-
bucket/confusion_matrix_dt.png')
plot_confusion_matrix(rf_predictions, 'Random Forest', 'gs://your-
bucket/confusion_matrix_rf.png')

# Random Forest feature importances
rf_feature_importances = rf_model.featureImportances.toArray()
plt.figure(figsize=(10, 6))
plt.barh(range(len(rf_feature_importances)), rf_feature_importances, color='orange')
plt.xlabel('Feature Importance')
plt.ylabel('Feature Index')
plt.title('Random Forest Feature Importances')
save_plot_to_gcs(plt.gcf(), 'gs://your-bucket/random_forest_feature_importances.png')
plt.close()
```

Appendix B: Visualizations

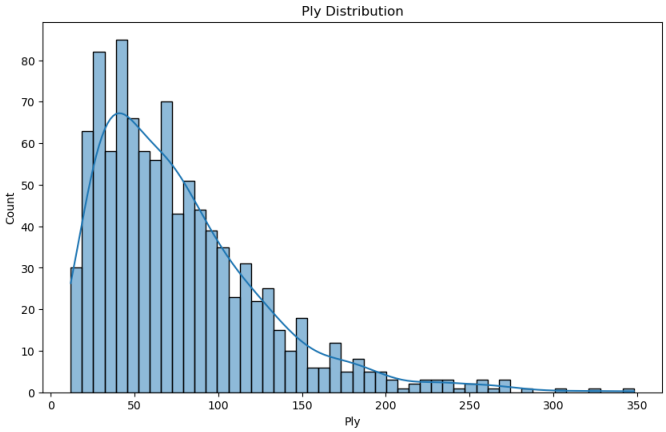
Scatter Plot of Ply vs Eval

The association between the quantity of half-moves (Ply) and evaluation ratings (Eval), colored by game results (win, loss, or draw), is displayed in this scatter plot.



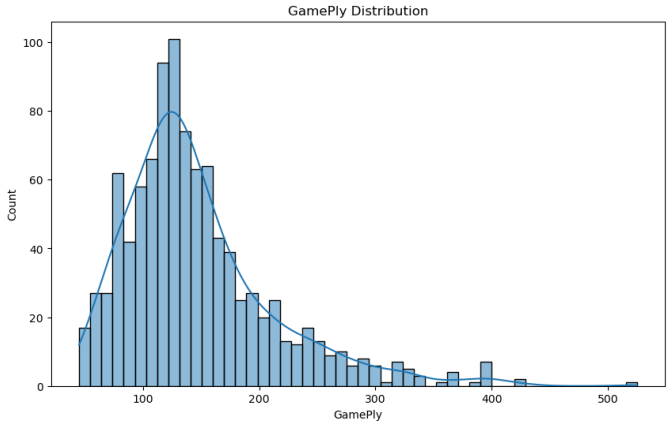
Histogram of Ply Distribution

The distribution of half-moves (Ply) per game is displayed in this histogram, which also illustrates the frequency of various game durations as well as the average lengths of games.



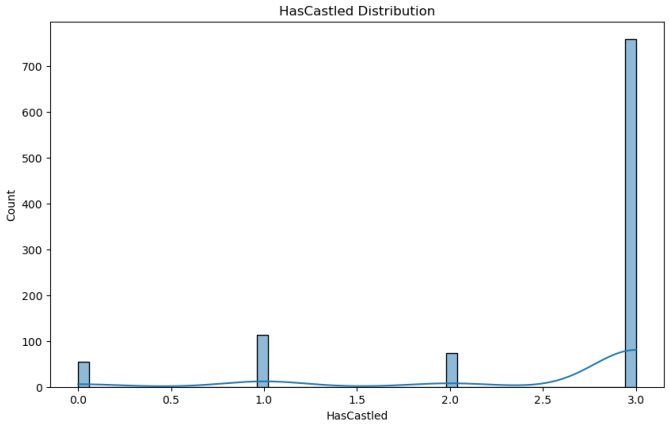
Histogram of GamePly Distribution

The distribution of the total number of plies (GamePly) in the games is shown by this histogram, which also shows the typical game lengths in terms of half-moves.



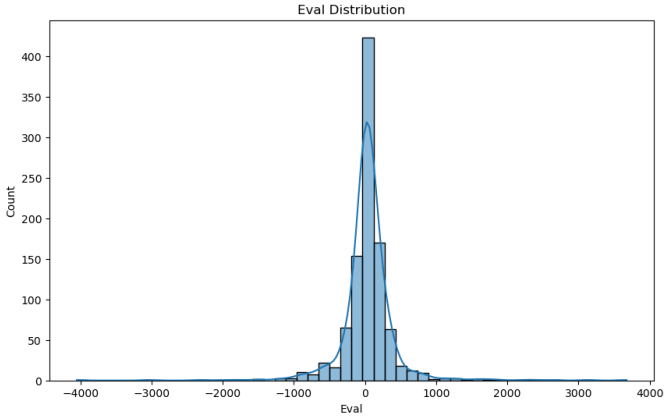
Histogram of HasCastled Distribution

This histogram shows the distribution of the castling indicator (HasCastled), indicating how often castling occurred in the games and the frequency of different castling states.



Histogram of Eval Distribution

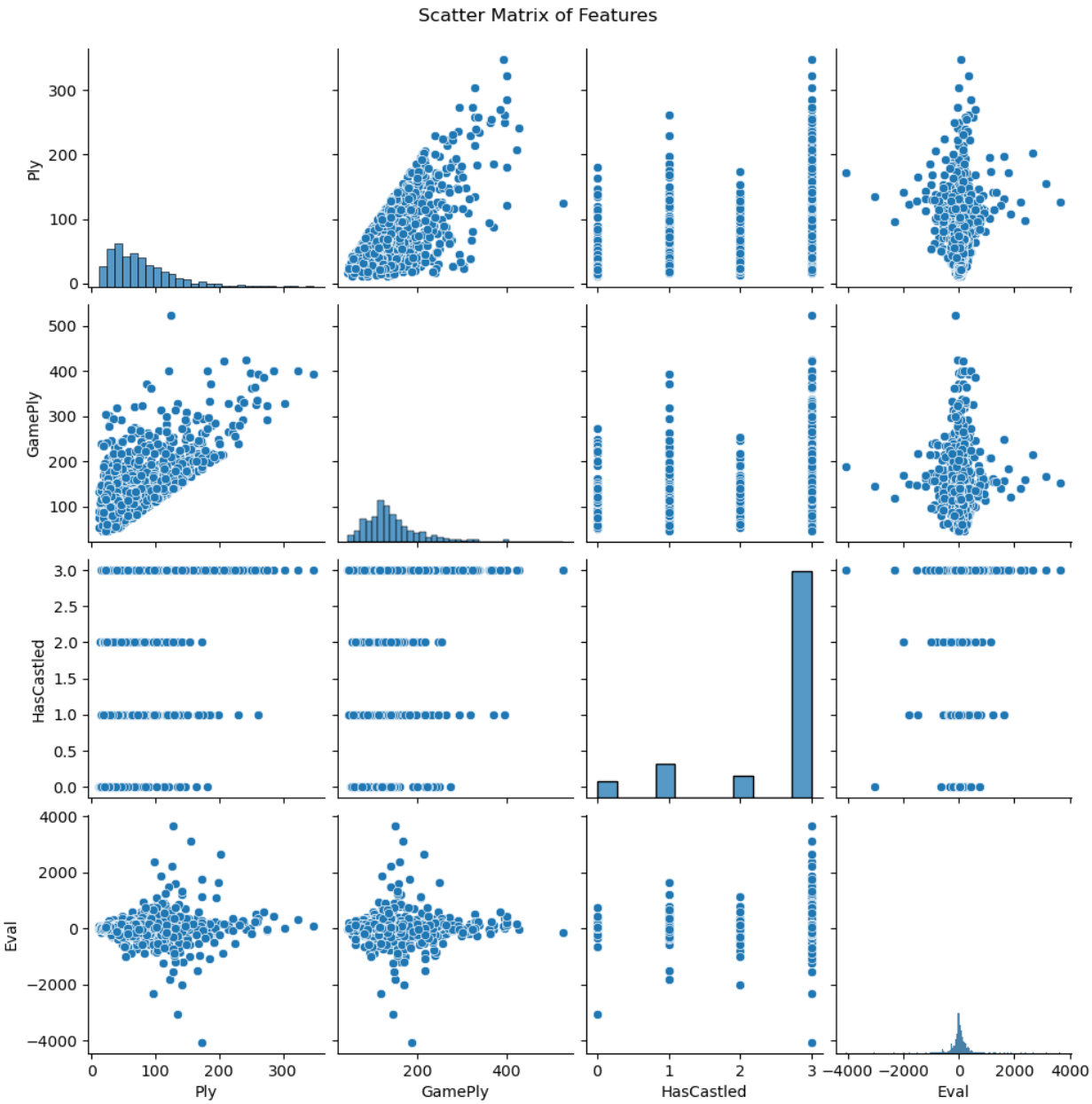
The distribution of the castling indication (HasCastled) is displayed in this histogram, along with the frequency of various castling states and the frequency of castling that occurred throughout the games.



Scatter Plot Matrix of Features

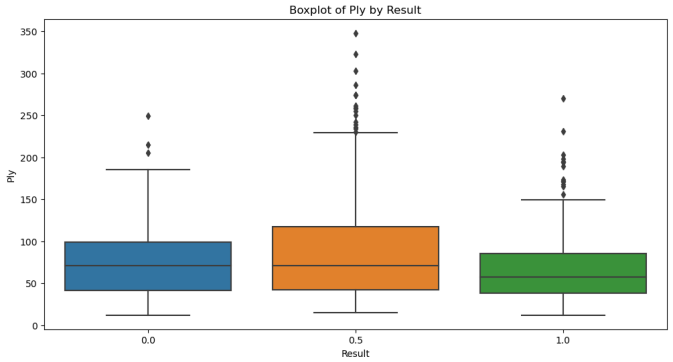
Pairwise comparisons of the dataset's primary characteristics (Ply, GamePly, HasCastled, and Eval) are provided via this scatter plot matrix, which aids in the visualization of the connections and correlations between these features.

For a more thorough understanding of feature interactions, you might include this in the same section as the preceding scatter plot.



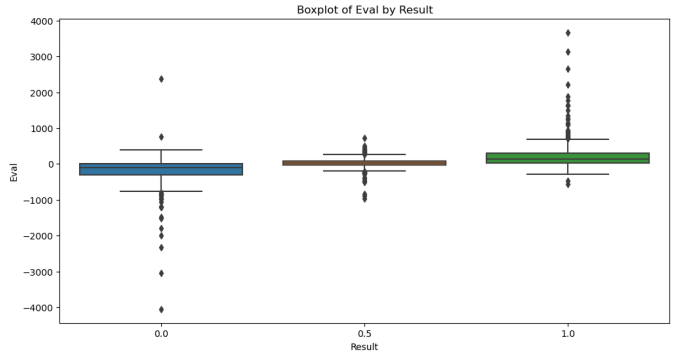
Boxplot of Ply by Result

The distribution of half-moves (Ply) in games with varying outcomes (Result) is depicted in this boxplot, demonstrating how game time changes with different outcomes (win, loss, draw).



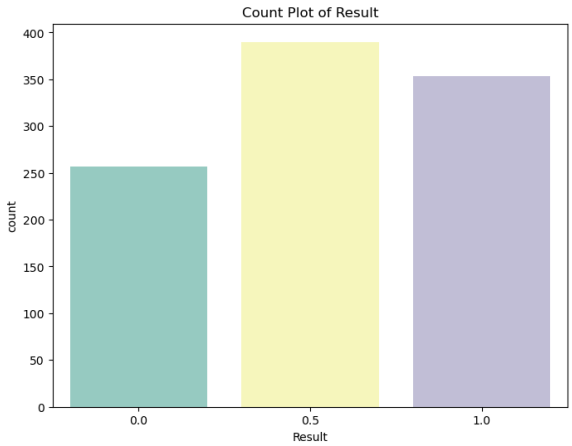
Boxplot of Eval by Result

The distribution of evaluation scores (Eval) for various game results (Result) is displayed in this boxplot, with the range and median values for wins, losses, and draws highlighted.



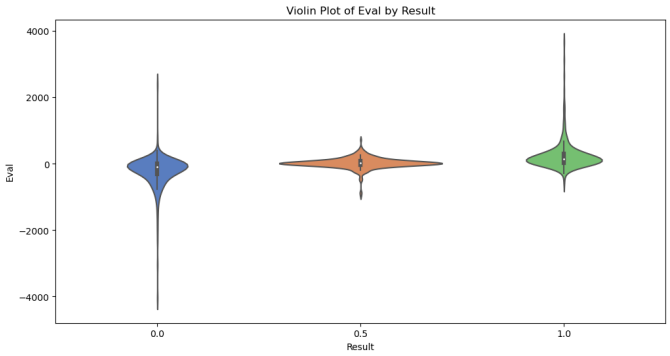
Count Plot of Result

The frequency of each game result (Result) is shown visually in this count plot, which gives a general idea of how the game results are distributed throughout the dataset.



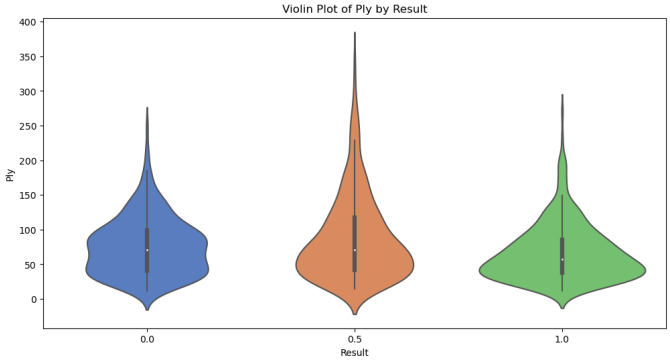
Violin Plot of Eval by Result

In contrast to boxplots, this violin plot displays the distribution and density of evaluation scores (Eval) across various game results (Result), offering a more in-depth look at the distribution.



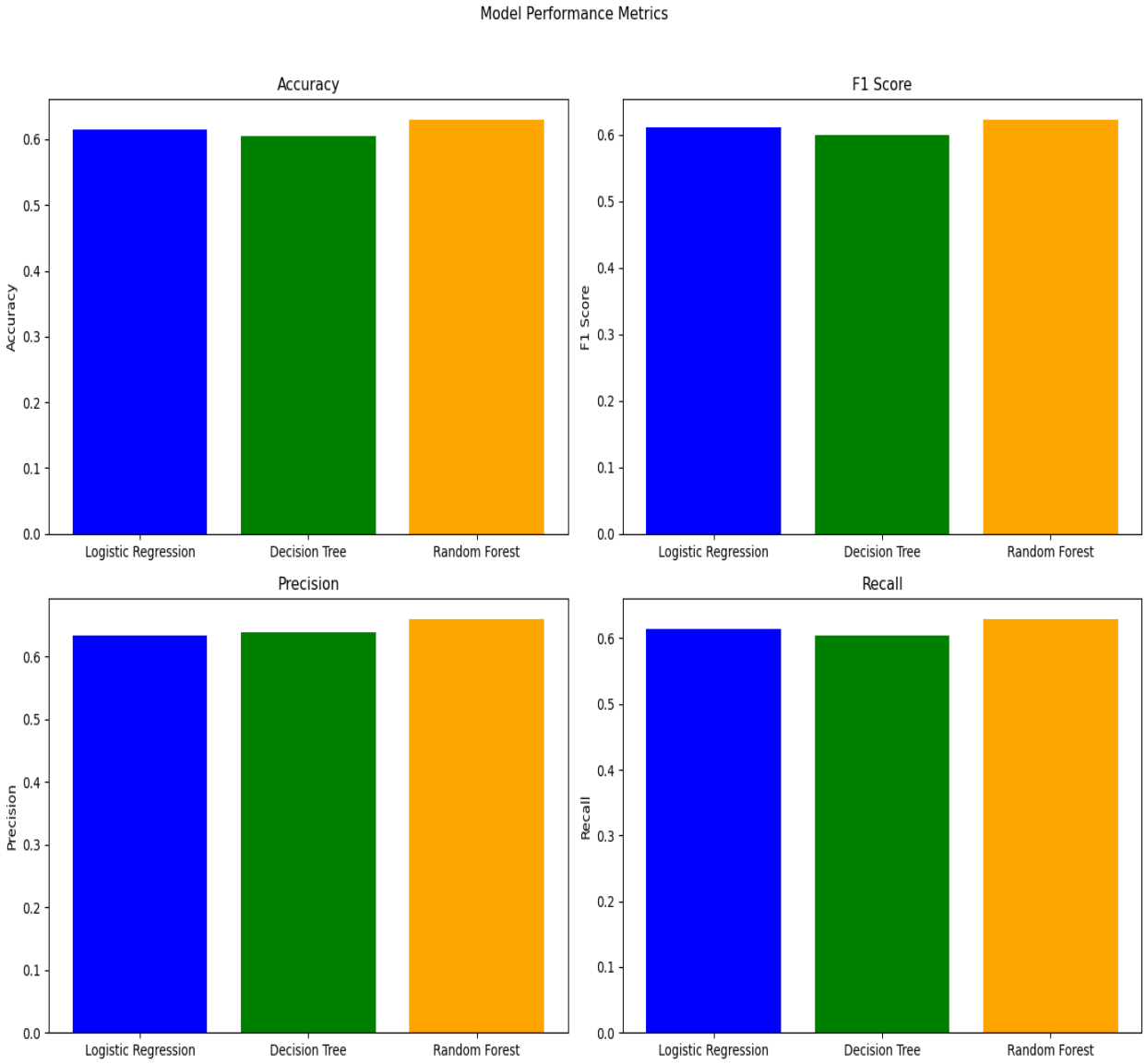
Violin Plot of Ply by Result

The distribution and density of half-moves (Ply) in games with various outcomes (Result) are shown in this violin plot, giving a thorough understanding of how game time changes with outcomes (win, loss, draw).



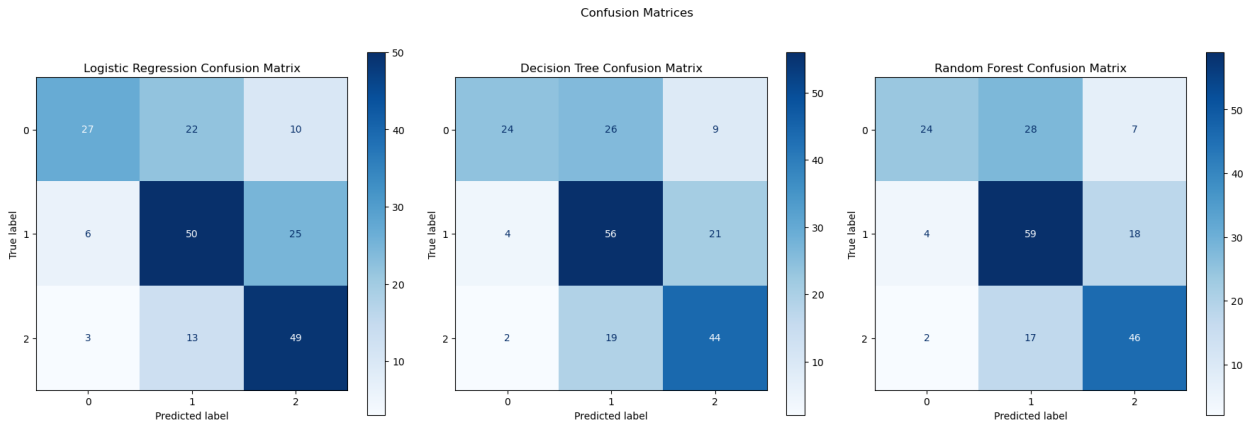
Model Performance Metrics

Based on accuracy, F1 score, precision, and recall measures, the performance of the Random Forest, Decision Tree, and Logistic Regression models is compared in this series of bar charts.



Confusion Matrices

These confusion matrices show the real vs expected labels for game results and compare the prediction performance of the Random Forest, Decision Tree, and Logistic Regression models.



Random Forest Feature Importances

This bar chart illustrates which factors have the most effects on the prediction of game results by showing the relative relevance of various attributes as assessed by the Random Forest model.