CS 777 Big Data Analytics
Fall 2023

Final Project Report

Pyspark on Clash of Clans

Jialiang Ma

Contents

1. Abstract

The dynamics and strategy within mobile games offer a comprehensive and complex view of player behavior, team interactions, and performance metrics. To understand these components and to gain insights into player preferences and clan dynamics, access to rich datasets becomes pivotal. Also, the goal of this analysis could be instrumental in shaping game event strategies. By understanding the dynamics of clan performance and the inherent traits of high-performing clans, game developers can design targeted events that cater to specific clan types. To sum up, we hope the results could help us have a bigger picture and understanding of the game industry and learn about game activity planning strategies and dynamics of user experience.

2. Introduction and Research Question

The goal of this project is to use the "Clash of Clans" dataset to develop and test various classification models. We aim to predict clan characteristics, such as their type and performance, based on the data we have. By comparing these models, we want to find the most effective solution. This project will help game developers and players better understand game trends and improve the overall gaming experience.

Research Question:

1. What traits distinguish high-performing clans in terms of member levels and trophies?

2. How does the type of clan (open, closed, etc.) impact its performance metrics?

3. Can we predict a clan's type based on its performance metrics and member statistics?

We will delve into these questions in subsequent sections, employing machine learning models to uncover the answers.

For machine learning model:

First, we will conduct data cleaning, feature engineering, and train test data split.

In this case, we have three distinct classifier models.

Classifiers:

1. Logistic Regression: To predict whether a clan is likely to be of a specific type based on its performance metrics

2. Random Forest Classifier: This classifier helps to categorize clans based on their attributes, such as war statistics, average member trophies, and so on.

3. Linear Regression: To predict clans' average member trophies based on attributes like clan type, member levels and different war's statistics.

Expected outcomes and evaluation plan would come after the model training and testing. For expected outcomes, a detailed understanding of what differentiates top-performing clans from the rest. Also, we will provide the insights into regional variations in player performance and clan rankings with comprehensive knowledge of the game ecosystem but mostly focus on our research questions.

In our project, we will train predictive models that can determine not only the research questions above but forecast clan performance metrics.

For evaluation:

1. For classification tasks, metrics such as confusion matrix, accuracy, recall rate, precision, and F1 score.

2. Mean absolute error (MAE), mean squared error (MSE), and R-squared for regression.

The main goal for this project is to provide useful data insights that both game developers and players will find interesting. The project's success will be judged by the quality of the analysis and how well the machine learning models predict outcomes. This proposal lays the groundwork for an examination of the "Clash of Clans" dataset, leading to valuable findings and hope with key takeaways.

3. Description of Dataset (3.5 Million Clans)

Data: In this project, we chose the dataset with an extensive view of the ecosystem of the world-famous mobile game “Clash of Clans”. I used to be one of the players during my high school. This dataset contains comprehensive details about the clans until 2023, giving insights into clan names, member information, locations, points, event statistics and so on.

The goal of this game is for players to develop and continuously upgrade their village, while also engaging in clan wars, which can be played in multiplayer settings.

Format: Each clan is identified by a unique clan tag. Other attributes include the clan's name, type, war statistics, member levels, trophies, containing more than 3.5 million clans’ information and is expendable. (3559743 unique value – clan tag)

File: coc_clans_dataset.csv

Columns (total 27):

```
PS C:\Users\marso\Desktop\Work Space\py workspace\CS777> & C:/Users/marso/AppData/Local/Programs/Python/Python310/python.exe
 "c:/Users/marso/Desktop/Work Space/py workspace/CS777/project.py"
     clan_tag        clan_name clan_type  ... capital_league mean_member_level  mean_member_trophies
0   #UQVQRJQ0      KOJIS' CLAN    closed  ...       Unranked               83                  1254
1   #2QC9Y8CQU              uye      open  ...       Unranked               90                  1752
2   #202CJRP2U  Uprising rivals      open  ...       Unranked               30                   733
3   #2Y89RRGLY     2inchersonly      open  ...       Unranked               61                  1156
4    #99PU9QPY            aymil      open  ...       Unranked               23                   674

[5 rows x 27 columns]
PS C:\Users\marso\Desktop\Work Space\py workspace\CS777>
```

Source:
https://www.kaggle.com/datasets/asaniczka/clash-of-clans-clans-dataset-2023-3-5m-clans/data

4. Tools Used in this Project

4.1 Apache Spark

For large-scale data processing and analysis.

In our case, increase kyro size would be important to make the spark run.

```
### Load data (increased kyro size)
spark = SparkSession.builder \
    .appName("ClashOfClans_FinalProject") \
    .config("spark.kryoserializer.buffer.max", "512m") \
    .getOrCreate() # We test for different value since running
out of java heap space
```

4.2 Google Cloud Platform

Using this platform to run the Pyspark code and observe the output. In Dataproc, the cluster node configuration would recommend:
Manager node: N2 (n2-standard-4/ 16GB
Worker node: E2 (e2-standard-8/ 32GB

The CoC data also has the link on Cloud Storage:
gs://cs-777-2023/coc_clans_dataset.csv

For project (main file):

gs://cs-777-2023/project.py

project2 (k-fold validation with same model):

gs://cs-777-2023/project2.py

project_plt (generate plot in missing value part):

gs://cs-777-2023/project_plt.py

4.3 Github

For version control, and easier to allocate the files and data.

5. EDA, Data Load & Data Cleaning

5.0 Import data and required packages

```
# Packages import
import os
import sys
import numpy as np
import pandas as pd

from pyspark.sql import functions as F
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum
from pyspark.sql.types import IntegerType, DoubleType
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.feature import VectorAssembler, StandardScaler,
StringIndexer
from pyspark.ml.classification import LogisticRegression,
RandomForestClassifier
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import
MulticlassClassificationEvaluator, RegressionEvaluator
from pyspark.sql.types import IntegerType
from pyspark.ml import Pipeline
from pyspark.ml.feature import FeatureHasher

import matplotlib.pyplot as plt
```

5.1 Exploratory Data Analysis

Data structure & format:

Row(clan_tag='#UQVQRJQ0', clan_name="KOJIS' CLAN", clan_type='closed', clan_description='STRATEGY IS MUST...RESPECT THE OTHERS...FORGIVE AND FORGET', clan_location='International', isFamilyFriendly='True', clan_badge_url='https://api-assets.clashofclans.com/badges/200/NUGY9HFXNrucid3nwXmsIl29XfKLFzT488B9dR6cGxY.png', clan_level='6', clan_points='6887', clan_builder_base_points='5213', clan_versus_points='5213', required_trophies='800', war_frequency='moreThanOncePerWeek', war_win_streak='1', war_wins='93', war_ties='0', war_losses='31', clan_war_league='Unranked', num_members='11', required_builder_base_trophies='1000', required_versus_trophies='1000', required_townhall_level='1', clan_capital_hall_level='1', clan_capital_points='0', capital_league='Unranked', mean_member_level='83', mean_member_trophies='1254')

Seems all the data points are string format, we will conduct data cleaning with transformation and feature selection later to find out which columns are beneficial to our research question.

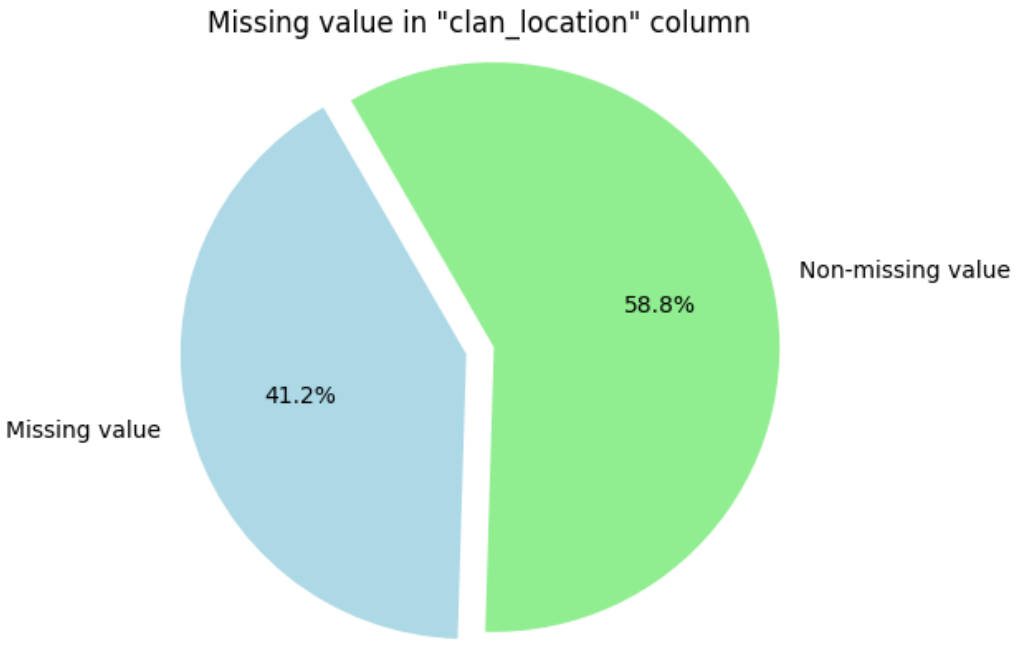
Data schema:

root
 |-- clan_tag: string (nullable = true)
 |-- clan_name: string (nullable = true)
 |-- clan_type: string (nullable = true)
 |-- clan_description: string (nullable = true)
 |-- clan_location: string (nullable = true)
 |-- isFamilyFriendly: string (nullable = true)
 |-- clan_badge_url: string (nullable = true)
 |-- clan_level: string (nullable = true)
 |-- clan_points: string (nullable = true)
 |-- clan_builder_base_points: string (nullable = true)
 |-- clan_versus_points: string (nullable = true)
 |-- required_trophies: string (nullable = true)
 |-- war_frequency: string (nullable = true)
 |-- war_win_streak: string (nullable = true)
 |-- war_wins: string (nullable = true)
 |-- war_ties: string (nullable = true)
 |-- war_losses: string (nullable = true)
 |-- clan_war_league: string (nullable = true)
 |-- num_members: string (nullable = true)
 |-- required_builder_base_trophies: string (nullable = true)
 |-- required_versus_trophies: string (nullable = true)
 |-- required_townhall_level: string (nullable = true)
 |-- clan_capital_hall_level: string (nullable = true)
 |-- clan_capital_points: string (nullable = true)
 |-- capital_league: string (nullable = true)
 |-- mean_member_level: string (nullable = true)
 |-- mean_member_trophies: string (nullable = true)

Now, let's dive into the issue of missing data. As depicted in the graph below, there's a significant proportion of missing values across various columns. For instance, the "clan_location" column has 58.8% non-missing values and 41.2% missing values. Simply discarding all the null values could heavily skew our subsequent analysis. Therefore, it's crucial to devise a strategy to address these data gaps.

(Overview of missing value)
clan_level has 7488 null values

clan_points has 3776 null values
clan_builder_base_points has 1954 null values
clan_versus_points has 1033 null values
required_trophies has 522 null values
war_win_streak has 3837 null values
war_wins has 1872 null values
war_ties has 945 null values
war_losses has 519 null values
num_members has 3866 null values
required_builder_base_trophies has 1892 null values
required_versus_trophies has 954 null values
required_townhall_level has 524 null values
clan_capital_hall_level has 262 null values
clan_capital_points has 151 null values
mean_member_level has 3747 null values
mean_member_trophies has 1834 null values



Therefore, we defined a helper function for cleaning, filling and transforming the data. For numeric columns, we will use mean as substitution for null values, since we want to maintain the general distribution of these columns. For categorical data, we use median instead, the most common category to fill up the missing value.

Also, we will convert isFamilyFriendly and clan_type from boolean and open/close into 1 or 0.

```
# Helper function to compute the mode for a given column
```

```
def compute_mode(df, column):
    return df.groupBy(column).count().orderBy('count',
ascending=False).collect()[0][0]

def clean_data(df):
    # Convert isFamilyFriendly to binary (1 = True, 0 = False)
    df = df.withColumn('isFamilyFriendly', (F.col('isFamilyFriendly')
== 'True').cast(IntegerType()))

    # Convert clan_type to binary (1 = open, 0 = closed)
    df = df.withColumn('clan_type', (F.col('clan_type') ==
'open').cast(IntegerType()))

    # Drop unuseful columns
    columns_to_drop = ['clan_tag', 'clan_name', 'clan_description',
'clan_badge_url', 'war_frequency', 'clan_war_league', 'capital_league',
'clan_location']
    df = df.drop(*columns_to_drop)

    # Convert columns from string to int data types
    columns_to_convert = [
        'clan_level', 'clan_points', 'clan_builder_base_points',
'clan_versus_points', 'required_trophies',
        'war_win_streak', 'war_wins', 'war_ties', 'war_losses',
'num_members', 'required_builder_base_trophies',
        'required_versus_trophies', 'required_townhall_level',
'clan_capital_hall_level', 'clan_capital_points',
        'mean_member_level', 'mean_member_trophies'
    ]

    for column in columns_to_convert:
        df = df.withColumn(column, df[column].cast(IntegerType()))

    # For numerical columns, impute with mean
    # Using this to maintain the general distribution
    for column in columns_to_convert:
        mean_value = df.agg({column: 'mean'}).collect()[0][0]
        df = df.na.fill({column: mean_value})

    # For categorical columns, impute with mode
    # Using most common category to replace the null value
    categorical_columns = ['isFamilyFriendly', 'clan_type']
    for column in categorical_columns:
```

```
        mode_value = compute_mode(df, column)
        df = df.na.fill({column: mode_value})

    return df
```

```
# Clean df
df = clean_data(df)

# Check for null values in the entire dataframe
for column in df.columns:
    null_count = df.filter(F.col(column).isNull()).count()
    if null_count > 0:
        print(f"{column} has {null_count} null values")

# Impute or remove null values
df = df.na.drop()

# Store the cleaned df for later use
df.cache()
#df.count()
```

After cleaning up the dataset, we will conduct na.drop() to double check there is no null value in the dataset. Null values will cause an impact on our machine learning model, such as random forest classifiers. Simply removing rows with null values, especially when there's a significant proportion of missing data, can lead to a substantial reduction in dataset size and loss of variability.

6. Data Preprocessing

1. Handle high-cardinality columns:
   High-cardinality columns refer to columns that have a large number of unique values. Handling them is crucial as they can introduce noise and slow down the training

process. Since we employed the Random Forest model, the internal model was using a decision tree model, dealing with high-cardinality would take plenty of time to identify the root cause and set up the best parameters.

```
# Identify high cardinality col
threshold = 10000  # the problematic one which has more than
46000
high_cardinality_cols = []
for column in df.columns:
    if df.select(column).distinct().count() > threshold:
        high_cardinality_cols.append(column)

print("High cardinality columns:", high_cardinality_cols)

# Use a feature hasher for high-cardinality categorical features
hasher = FeatureHasher(inputCols=high_cardinality_cols,
outputCol="hashed_features", numFeatures=1024)
```

Results: High cardinality columns: ['clan_points', 'clan_builder_base_points', 'clan_versus_points']

These are the features have high-cardinality, we will use hasher to transform them.

2. String indexing for categorical data

First, we find those columns which suppress our unique values threshold, which is 10000. For such columns, we employ the FeatureHasher technique. This method transforms a collection of categorical or textual features to a feature vector of specified dimension (in this case, 1024). It's a beneficial technique as it can handle unseen labels gracefully. We also used the StringIndexer to achieve this transformation.

```
# Exclude high cardinality columns from StringIndexer processing
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index",
handleInvalid="skip").fit(df)
            for column in
list(set(df.columns)-set(['mean_member_trophies',
'mean_member_level']+high_cardinality_cols))]
```

3. Pipeline creation

In order to streamline the preprocessing steps, we established a pipeline that integrates both the feature hasher and the string indexers.

```
# Combine the feature hasher and other indexers
stages = indexers + [hasher]

pipeline = Pipeline(stages=stages)
df_indexed = pipeline.fit(df).transform(df)

df_indexed.cache()
```

4. Data splitting
   To evaluate the machine learning model's performance, we divided the dataset into training and testing sets. 80% of the data was allocated for training, and the remaining 20% for testing. Set up the seed for the same random split.

```
    # Split data into training and testing set: 80% train, 20% test
    # Set seed
    seed = 202310
    (train_data, test_data) = df_indexed.randomSplit([0.8, 0.2],
seed=seed)
```

5. Grouping and Aggregation
   For further understanding, we grouped the data by the clan type and computed the average member trophies and levels.

```
# Group by the clan type and calculate average member trophies and
levels
df_grouped =
train_data.groupBy("clan_type_index").agg({"mean_member_trophies":
"avg", "mean_member_level": "avg"})

print(df_grouped.head(10))
```

7. Data Modeling

7.1 Feature assembly and scaling

Before training our models, it's crucial to consolidate our datasets into a single feature vector. This process is also referred to as feature assembly. We will select all columns except target variables. Then we appended ‘hashed_features’, in this way, we can transform categorical variables with a large number of levels (high cardinality) into a numerical format that can be provided to a machine learning model. Then we conduct assembler and standard scaler to standardize and assemble data into a single vector format that spark’s MLlib can predict.

```
# Feature columns
feature_columns = [col for col in df_indexed.columns if col not
in ['clan_type', 'mean_member_trophies', 'mean_member_level']]
feature_columns.append("hashed_features")

assembler = VectorAssembler(inputCols=feature_columns,
outputCol="features", handleInvalid='skip')

# Feature scaling with standard scaler
scaler = StandardScaler(inputCol="features",
outputCol="scaledFeatures", withStd=True, withMean=True)
```

1. Logistic Regression Model (predict clan type

```
# Logistic Regression to predict clan type
logistic_model = LogisticRegression(maxIter=20,
labelCol="clan_type_index", featuresCol="features")
pipeline_logistic = Pipeline(stages=[assembler, scaler,
logistic_model]) # Adding scaler into pipeline
# Train and predict
model = pipeline_logistic.fit(train_data)
predictions = model.transform(test_data)
```

2. Random Forest Model

```
# Random Forest model
random_forest_model =
RandomForestClassifier(labelCol="clan_type_index",
featuresCol="features", numTrees=100, maxBins=10000,
maxDepth=10)
```

```
pipeline_rf = Pipeline(stages=[assembler, scaler,
random_forest_model])
# Train and predict
model_rf = pipeline_rf.fit(train_data)
predictions_tree = model_rf.transform(test_data)
```

3. Linear Regression Model (predict mean_member_trophies

```
# Linear Regression to predict mean_member_trophies
linear_model = LinearRegression(maxIter=20,
labelCol="mean_member_trophies", featuresCol="features")
pipeline_linear = Pipeline(stages=[assembler, scaler,
linear_model])
# Train and predict
model_linear = pipeline_linear.fit(train_data)
predictions_linear = model_linear.transform(test_data)
```

8. Models Performance Evaluation

8.1 Model Evaluation

Helper function in confusion matrix:

```
# Helper function to extract TN, TP, FN, FP from cm
def extract_confusion_metrics(confusion_matrix):
    tn = next((row['count'] for row in confusion_matrix if
row['clan_type_index'] == 0 and row['prediction'] == 0), 0)
    fp = next((row['count'] for row in confusion_matrix if
row['clan_type_index'] == 0 and row['prediction'] == 1), 0)
    fn = next((row['count'] for row in confusion_matrix if
row['clan_type_index'] == 1 and row['prediction'] == 0), 0)
    tp = next((row['count'] for row in confusion_matrix if
row['clan_type_index'] == 1 and row['prediction'] == 1), 0)

    return tn, fp, fn, tp
```

Evaluator initialization:

```
evaluator =
MulticlassClassificationEvaluator(labelCol="clan_type_index")
```

Then we will use this evaluator to see the performance metrics of each model, including F1 score, accuracy, confusion matrix and recall rate. In linear regression, we use RMSE as metrics. RMSE measures the average prediction error in the same units as the target variable, allowing for easy understanding of the error magnitude.

Also, this metric emphasizes larger errors by squaring the residuals, making it a valuable metric when minimizing large prediction errors is crucial.

```
# For Logistic Regression
accuracy_logistic = evaluator.evaluate(predictions,
{evaluator.metricName: "accuracy"})
f1_logistic = evaluator.evaluate(predictions,
{evaluator.metricName: "f1"})
recall_logistic = evaluator.evaluate(predictions,
{evaluator.metricName: "weightedRecall"})

# For Random Forest
accuracy_tree = evaluator.evaluate(predictions_tree,
{evaluator.metricName: "accuracy"})
f1_tree = evaluator.evaluate(predictions_tree,
{evaluator.metricName: "f1"})
recall_tree = evaluator.evaluate(predictions_tree,
{evaluator.metricName: "weightedRecall"})

print("----- Logistic Regression -----")
print(f"Accuracy: {accuracy_logistic}")
print(f"F1 Score: {f1_logistic}")
print(f"Recall: {recall_logistic}")

print("\n----- Random Forest -----")
print(f"Accuracy: {accuracy_tree}")
print(f"F1 Score: {f1_tree}")
print(f"Recall: {recall_tree}")

# Confusion matrix for Logistic regression
print("\nConfusion matrix for Logistic Regression:")
predictions.groupBy("clan_type_index",
"prediction").count().show()

# Tn, fp, fn, tp
```

```
cm_logistic = predictions.groupBy("clan_type_index",
"prediction").count().collect()
tn, fp, fn, tp = extract_confusion_metrics(cm_logistic)

print(f"Logistic Regression: TN={tn}, FP={fp}, FN={fn}, TP={tp}")

# Confusion matrix for Random Forest
print("\nConfusion matrix for Random Forest:")
predictions_tree.groupBy("clan_type_index",
"prediction").count().show()

confusion_matrix_rf = predictions_tree.groupBy("clan_type_index",
"prediction").count().collect()
tn, fp, fn, tp = extract_confusion_metrics(confusion_matrix_rf)

print(f"Random Forest: TN={tn}, FP={fp}, FN={fn}, TP={tp}")

# For Linear Regression, we use RMSE as metrics
regression_evaluator =
RegressionEvaluator(labelCol="mean_member_trophies",
metricName="rmse")
rmse_linear = regression_evaluator.evaluate(predictions_linear)
print(f"\nRoot Mean Squared Error (RMSE) for Linear Regression:
{rmse_linear}")
```

8.2 Model Optimization

In model optimization, using L2 regularization (ridge regression), can prevent overfitting by penalizing large coefficients.

```
# Logistic regression model with L2 regularization
logistic_l2 = LogisticRegression(featuresCol="features",
labelCol="clan_type_index", maxIter=10, regParam=0.1,
elasticNetParam=0)
pipeline_logistic_l2 = Pipeline(stages=[assembler, scaler,
logistic_l2])
model_logistic_l2 = pipeline_logistic_l2.fit(train_data)
predictions_l2 = model_logistic_l2.transform(test_data)

# Metrics for Logistic Regression with L2 regularization
accuracy_logistic_l2 = evaluator.evaluate(predictions_l2,
{evaluator.metricName: "accuracy"})
```

```
f1_logistic_l2 = evaluator.evaluate(predictions_l2,
{evaluator.metricName: "f1"})
recall_logistic_l2 = evaluator.evaluate(predictions_l2,
{evaluator.metricName: "weightedRecall"})

print("----- Logistic Regression with L2 Regularization -----")
print(f"Accuracy: {accuracy_logistic_l2}")
print(f"F1 Score: {f1_logistic_l2}")
print(f"Recall: {recall_logistic_l2}")
```

For enhanced Random Forest model, we increased the number of trees from 100 to 150:

```
# Optimized random forest model
rf_optimized = RandomForestClassifier(featuresCol="features",
labelCol="clan_type_index", numTrees=150, maxDepth=10,
maxBins=10000)
pipeline_rf_optimized = Pipeline(stages=[assembler, scaler,
rf_optimized])
model_rf_optimized = pipeline_rf_optimized.fit(train_data)
predictions_rf_optimized =
model_rf_optimized.transform(test_data)

# Metrics for optimized Random Forest
accuracy_rf_optimized =
evaluator.evaluate(predictions_rf_optimized,
{evaluator.metricName: "accuracy"})
f1_rf_optimized = evaluator.evaluate(predictions_rf_optimized,
{evaluator.metricName: "f1"})
recall_rf_optimized =
evaluator.evaluate(predictions_rf_optimized,
{evaluator.metricName: "weightedRecall"})

print("\n----- Optimized Random Forest -----")
print(f"Accuracy: {accuracy_rf_optimized}")
print(f"F1 Score: {f1_rf_optimized}")
print(f"Recall: {recall_rf_optimized}")

# Confusion matrix for optimized Random Forest
print("\nConfusion matrix for Optimized Random Forest:")
predictions_rf_optimized.groupBy("clan_type_index",
"prediction").count().show()
```

```
confusion_matrix_rf_optimized =
predictions_rf_optimized.groupBy("clan_type_index",
"prediction").count().collect()
tn, fp, fn, tp =
extract_confusion_metrics(confusion_matrix_rf_optimized)

print(f"Optimized Random Forest: TN={tn}, FP={fp}, FN={fn},
TP={tp}")
```

For our linear regression model predicting mean_member_trophies, we adjusted the regularization parameter and applied ElasticNet regularization, which is a combination of L1 (Lasso) and L2 (Ridge) regularization:

```
# Define the optimized linear regression model
linear_optimized = LinearRegression(featuresCol="features",
labelCol="mean_member_trophies", maxIter=10, regParam=0.1,
elasticNetParam=0.2)
pipeline_linear_optimized = Pipeline(stages=[assembler, scaler,
linear_optimized])
model_linear_optimized =
pipeline_linear_optimized.fit(train_data)
predictions_linear_optimized =
model_linear_optimized.transform(test_data)

# RMSE for optimized Linear Regression
rmse_linear_optimized =
regression_evaluator.evaluate(predictions_linear_optimized)
print(f"\nRoot Mean Squared Error (RMSE) for Optimized Linear
Regression: {rmse_linear_optimized}")
```

8.3 Models Validation by K-fold Validation

In file project2.py, we added another version of the same classifiers with k-fold validation. A 5-fold cross-validation is performed. This means the dataset is partitioned into five subsets. Then the process will repeat 5 times, we will evaluate the model by its average performance.

```
# Logistic Regression to predict clan type
logistic_model = LogisticRegression(maxIter=20,
labelCol="clan_type_index", featuresCol="features")
```

```
pipeline_logistic = Pipeline(stages=[assembler, scaler,
logistic_model]) # Adding scaler into pipeline

# Use a parameter grid for validation
paramGrid_logistic = ParamGridBuilder() \
    .addGrid(logistic_model.regParam, [0.01]) \
    .addGrid(logistic_model.elasticNetParam, [0.1]) \
    .build()

crossval_logistic = CrossValidator(estimator=pipeline_logistic,

estimatorParamMaps=paramGrid_logistic,

evaluator=MulticlassClassificationEvaluator(labelCol="clan_type_i
ndex"),
                                    numFolds=5)

# Run cross-validation
cvModel_logistic = crossval_logistic.fit(train_data)
predictions = cvModel_logistic.transform(test_data)




# Random Forest model
random_forest_model =
RandomForestClassifier(labelCol="clan_type_index",
featuresCol="scaledFeatures", numTrees=100, maxBins=10000)
pipeline_rf = Pipeline(stages=[assembler, scaler,
random_forest_model])

paramGrid_rf = ParamGridBuilder() \
    .addGrid(random_forest_model.numTrees, [100]) \
    .addGrid(random_forest_model.maxDepth, [10]) \
    .build()

crossval_rf = CrossValidator(estimator=pipeline_rf,
                             estimatorParamMaps=paramGrid_rf,

evaluator=MulticlassClassificationEvaluator(labelCol="clan_type_i
ndex"),
                             numFolds=5)
```

```
cvModel_rf = crossval_rf.fit(train_data)
predictions_rf = cvModel_rf.transform(test_data)

# Linear Regression to predict mean_member_trophies
linear_model = LinearRegression(labelCol="mean_member_trophies",
featuresCol="scaledFeatures")
pipeline_linear = Pipeline(stages=[assembler, scaler,
linear_model])

paramGrid_linear = ParamGridBuilder() \
    .addGrid(linear_model.regParam, [0.1]) \
    .addGrid(linear_model.elasticNetParam, [0.5]) \
    .build()

crossval_linear = CrossValidator(estimator=pipeline_linear,

estimatorParamMaps=paramGrid_linear,

evaluator=RegressionEvaluator(labelCol="mean_member_trophies"),
                              numFolds=5)

cvModel_linear = crossval_linear.fit(train_data)
predictions_linear = cvModel_linear.transform(test_data)
```

The evaluation part would use the same metrics.

9. Results and Evaluation

9.1 Performance measures and results of all the models

According to the results on GCP, in df_grouped()
df_grouped =
train_data.groupBy("clan_type_index").agg({"mean_member_trophies": "avg",
"mean_member_level": "avg"})
print(df_grouped.head(10))

Results:
Row(clan_type_index=0.0, avg(mean_member_level)=42.49264540288643,
avg(mean_member_trophies)=805.9769421478397), Row(clan_type_index=1.0,

avg(mean_member_level)=66.02096880178453,
avg(mean_member_trophies)=1114.6753969134702)

So, clans labeled as type 1.0 generally perform better, with members having more experience and trophies than those in type 0.0 clans. This suggests type 1.0 clans might have some advantages in the game.

Models Performance:
----- Logistic Regression -----
Accuracy: 1.0
F1 Score: 1.0
Recall: 1.0

----- Random Forest -----
Accuracy: 0.7208768185903125
F1 Score: 0.6039518714728019
Recall: 0.7208768185903125

Confusion matrix for Logistic Regression:

clan_type	prediction	count
1.0	1.0	200641
0.0	0.0	518185

Logistic Regression: TN=518185, FP=0, FN=0, TP=200641

Confusion matrix for Random Forest:

clan_type	prediction	count
1.0	0.0	200641
0.0	0.0	518185

Random Forest: TN=518185, FP=0, FN=200641, TP=0

Root Mean Squared Error (RMSE) for Linear Regression: 384.92345813807657

Optimizated Models:

----- Logistic Regression with L2 Regularization -----
Accuracy: 0.9999415713955812
F1 Score: 0.9999415732689743
Recall: 0.999941571395581

Confusion matrix for Logistic Regression with L2:

clan_type_index	prediction	count
1.0	1.0	200641
0.0	1.0	42
0.0	0.0	518143

Logistic Regression (L2): TN=518185, FP=42, FN=0, TP=200641

----- Optimized Random Forest -----
Accuracy: 0.7208768185903125
F1 Score: 0.6039518714728019
Recall: 0.7208768185903125

Confusion matrix for Optimized Random Forest:

clan_type	prediction	count
1.0	0.0	200641
0.0	0.0	518185

Optimized Random Forest: TN=518185, FP=0, FN=200641, TP=0

Root Mean Squared Error (RMSE) for Optimized Linear Regression:
387.13558150199685

Models Performance with Cross-Validation:

----- Logistic Regression -----
Accuracy: 0.7208768185903125
F1 Score: 0.6039518714728019
Recall: 0.7208768185903125

----- Random Forest -----
Accuracy: 0.7208768185903125
F1 Score: 0.6039518714728019
Recall: 0.7208768185903125

Confusion matrix for Logistic Regression:

clan_type	prediction	count
1.0	0.0	200641
0.0	0.0	518185

Confusion matrix for Random Forest:

clan_type	prediction	count
1.0	0.0	200641
0.0	0.0	518185

Root Mean Squared Error (RMSE) for Linear Regression: 384.80338811162494

9.2 Justification for selection of the best model

Brief Summary:

Logistic Regression perfectly predicted clan types, as verified by its 100% accuracy and the confusion matrix.

Random Forest underperformed, categorizing all clans as type 0.0, resulting in an accuracy of about 72%.

Linear Regression had an RMSE of 384.92, indicating the prediction error for "mean_member_trophies".

The Optimized Logistic Regression almost replicated the original's perfection but had a minor misclassification.

The Optimized Random Forest didn't show any improvement over the original.

The Optimized Linear Regression had a slightly worse RMSE of 387.14.

Using Cross-Validation didn't enhance the performance for either the logistic regression or the random forest model.

Therefore, Logistic Regression has the highest accuracy in initial testing, and in the L2 version, the accuracy is also higher than other classifiers.

The Random Forest model, both in its initial and optimized states, failed to capture the nuances of the data, predicting all clans as type 0.0.

The Linear Regression model's RMSE values indicate its predictive capability regarding "mean_member_trophies". The best one is the k-fold cross-validation one, which is slightly smaller than the original one, 384.8.

9.3 Research Questions Answered

Research Question 1:
What traits distinguish high-performing clans in terms of member levels and trophies?

Ans:
When we look into data_grouped():
Row(clan_type_index=0.0, avg(mean_member_level)=42.49264540288643, avg(mean_member_trophies)=805.9769421478397), Row(clan_type_index=1.0, avg(mean_member_level)=66.02096880178453, avg(mean_member_trophies)=1114.6753969134702)

Open clans (labeled as type 1) have an average member level of 66.02 and average member trophies of 1114.68.

Closed clans (labeled as type 0) have an average member level of 42.49 and average member trophies of 805.98.

All in all, clans of type 1 typically have members with higher levels and more trophies compared to clans of type 0.

Research Question 2:
How does the type of clan (open, closed, etc.) impact its performance metrics?

The type of clan appears to have a significant impact on its performance metrics:

Open clans (type 1.0) tend to have members with higher levels and more trophies, indicating better performance.

Closed clans (type 0.0) have members with lower levels and fewer trophies on average.

Thus, open clans generally outperform closed clans in terms of member levels and trophies

Research Question 3:
Can we predict a clan's type based on its performance metrics and member statistics?

Based on the model evaluations earlier, Logistic Regression has the perfect score on accuracy, F1 score, and recall rate, all at 1. This suggests that it can perfectly predict a clan's type based on the provided metrics. However, such perfection may indicate overfitting, especially when this performance is compared with other models or when applied to unseen data.

The Random Forest model mostly guessed clans as type 0 and had about 72% accuracy. The Logistic Regression model, even with some tweaks, was almost perfect with about 100% accuracy. Based on this, the Logistic Regression model is the best for predicting clan type, but we should test it more to be sure.

10. Conclusion

Clans labeled as 1 (open) are better than those labeled as 0 (closed), with members who play more and win more. The Logistic Regression model was best at predicting this. To sum up, we should dig deeper to see what makes these clans successful.

References:
https://stackoverflow.com/questions/45420112/cross-validation-in-pyspark
https://kb.databricks.com/machine-learning/kfold-cross-validation

https://medium.com/@big_data_landscape/a-beginners-guide-to-random-forest-in-pyspark-9045caac1087
https://medium.com/@uzzaman.ahmed/pyspark-hash-functions-a-comprehensive-guide-fb1bb069aac9
https://medium.com/swlh/logistic-regression-with-pyspark-60295d41221
https://www.analyticsvidhya.com/blog/2019/11/build-machine-learning-pipelines-pyspark/

Spark History: