Machine Learning using Spark MLlib

Jayrup Nakawala (u2613621) Yogi Patel (u2536809)
Jasmi Alasapuri (u2571395)

Table of contents

Staller Populations in Gaia DR3

Introduction

Using the `gaia_survey` dataset, the task shifts from getting data to implementing predictions. The objective is to construct, train, and evaluate classification models that can automatically categorize stellar objects based on their physical properties.

Purpose and Need for Implementation

- Differentiation of Stellar Objects: The main goal is to use observations to differentiate "nearby dwarfs from distant giants" and other star populations.
- Managing High-Dimensional Data: To make sure the data is appropriate for algorithmic division, the task applies modifications such log-scaling to meet the "massive range" of astronomical quantities (like parallax).
- Automation: When working with large-scale survey data that cannot be manually classified, the task automates the complicated procedure of feature collection, scaling, and classification by using a pipeline.

Model Insights

What it is about: A supervised classification workflow is implemented in this problem. After creating training labels using "ground truth" logic based on astronomy (Absolute Magnitude vs. Colour), it trains algorithms to predict such labels using only observable features.

The Workflow: 1. Feature Engineering: It constructs input features from raw data, such as calculating total_motion from proper motion vectors (*pmra* and *pmdec*) 2. Label Generation: It creates a label column by applying specific physical cuts:White Dwarfs (Label 2.0): Defined as having Absolute Magnitude (M_G) $> 10$8.Red Giants (Label 1.0): Defined as having $M_G < 3$ and Color (*bp_rp*) $> 1.0$9.Main Sequence (Label 0.0): Everything else10. 3. Algorithm Implementation: It implements two distinct algorithms to solve this problem: - Random Forest Classifier: A non-linear model suited for complex decision boundaries. - Logistic Regression: A linear model used for comparison, which includes feature scaling and elasticity regularization. 4. What it is trying to predict: The models aim to predict the label (Star Type) of a star given only its observable features (bp_rp, phot_g_mean_mag, total_motion, parallax). It validates these predictions using Cross-Validation (3-Fold) to ensure the model is robust and "not just lucky".

Phase 1: The Random Forest Model

Data Preparation & Features

```
import os
import numpy as np
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```python
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel



spark = SparkSession.builder \
    .appName("StellarPopulation_RF_Advanced") \
    .config("spark.driver.memory", "4g") \
    .config("spark.executor.memory", "4g") \
    .config("spark.memory.offHeap.enabled", "true") \
    .config("spark.memory.offHeap.size", "1g") \
    .config("spark.sql.shuffle.partitions", "200") \
    .getOrCreate()

# 2. Load the Data (Correct spelling)
df = spark.read.parquet("../data/gaia_survey.parquet")

# A. Calculate Observables (Features)
# We add 'parallax' because it is an OBSERVABLE. It allows the model to differentiate
# nearby dwarfs from distant giants.
df = df.withColumn("total_motion", F.sqrt(F.col("pmra")**2 + F.col("pmdec")**2))
# Log-transform parallax to handle the massive range (scales data for better splits)
df = df.withColumn("log_parallax", F.log10(F.abs(F.col("parallax")) + 1e-6))


# 3. Clean data
df = df.dropna(subset=["phot_g_mean_mag", "bp_rp", "parallax", "total_motion"])

# Verify it worked
print(f"Data loaded and cleaned. Rows: {df.count()}")
df.printSchema()
```

```
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
```

```
25/12/17 06:03:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platf
java classes where applicable

Data loaded and cleaned. Rows: 821788
root
 |-- source_id: long (nullable = true)
 |-- ra: double (nullable = true)
 |-- dec: double (nullable = true)
 |-- parallax: double (nullable = true)
 |-- parallax_error: float (nullable = true)
 |-- pmra: double (nullable = true)
 |-- pmdec: double (nullable = true)
 |-- phot_g_mean_mag: float (nullable = true)
 |-- bp_rp: float (nullable = true)
 |-- teff_gspphot: float (nullable = true)
 |-- total_motion: double (nullable = true)
 |-- log_parallax: double (nullable = true)
```

Handling Class Imbalance

A critical step in this code is addressing the fact that White Dwarfs are rare compared to Main Sequence stars.

- The Problem: Without correction, a model could achieve high accuracy by simply ignoring the rare White Dwarfs.
- The Solution: The code calculates Class Weights using the formula:

  $$Weight = \frac{TotalRows}{3.0 \times ClassCount}$$

  This assigns higher weights to rare classes (White Dwarfs) to force the model to pay attention to them during training5.

```
# --- A. Create Absolute Magnitude (M_G) for the Label Logic ---
# Distance d = 1000 / parallax (mas)
df = df.withColumn("distance", 1000 / F.col("parallax"))
df = df.withColumn("abs_mag", F.col("phot_g_mean_mag") - 5 * F.log10(F.col("distance")) + 5)

# --- B. Define the Classes (The "Ground Truth" Cuts) ---

df_labeled = df.withColumn("label",
    F.when(F.col("abs_mag") > 10, 2.0)  # White Dwarf
     .when((F.col("abs_mag") < 3) & (F.col("bp_rp") > 1.0), 1.0) # Red Giant
     .otherwise(0.0) # Main Sequence
)

# --- C. Handle Class Imbalance (Weighting) ---
# Calculate class counts
class_counts = df_labeled.groupBy("label").count().collect()
total_rows = df_labeled.count()
count_map = {row['label']: row['count'] for row in class_counts}

# Calculate weights: Weight = Total / (Number of Classes * Class Count)
# Weight = Total / (Num_Classes * Count)
class_weights = {k: total_rows / (3.0 * v) for k, v in count_map.items()}
print(f">> Class Weights: {class_weights}")

# Broadcast weights to a mapping column
mapping_expr = F.create_map([F.lit(x) for x in sum(class_weights.items(), ())])
df_weighted = df_labeled.withColumn("classWeight", mapping_expr.getItem(F.col("label")))

print("Class Weights Calculated:", class_weights)
```

```
>> Class Weights: {0.0: 0.3913544547816612, 1.0: 2.4997657766178145, 2.0: 22.354278874925196
Class Weights Calculated: {0.0: 0.3913544547816612, 1.0: 2.4997657766178145, 2.0: 22.3542788

/home/yogipatel/Documents/Big data /CN6022_coursework/src/env/lib/python3.13/site-
packages/pyspark/sql/classic/column.py:359: FutureWarning:

A column as 'key' in getItem is deprecated as of Spark 3.0, and will not be supported in the
```

Model Configuration & Tuning

```
# Define Input Features (Observables only!)
feature_cols = ["bp_rp", "phot_g_mean_mag", "total_motion", "parallax", "log_parallax"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")


# Initialize Random Forest
rf = RandomForestClassifier(
    labelCol="label",
    featuresCol="features",
    weightCol="classWeight",
    seed=42,
    subsamplingRate=0.7,
    featureSubsetStrategy="sqrt"
)

# Build Pipeline
pipeline = Pipeline(stages=[assembler, rf])

# Parameter Grid for Tuning
paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [30, 50]) \
    .addGrid(rf.maxDepth, [8, 12]) \
    .build()

# Evaluator (Focus on Weighted F1 to balance all classes)
evaluator = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="f1")

# Cross Validator (3-Fold)
# It ensures the model is robust and not just lucky.
cv = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=paramGrid,
    evaluator=evaluator,
    numFolds=3,
    parallelism=2  # Train 2 models in parallel if memory allows
)

# 5. Train/Test Split
train_data, test_data = df_weighted.randomSplit([0.8, 0.2], seed=42)
print(f"Training Count: {train_data.count()} | Test Count: {test_data.count()}")
```

```
Training Count: 657695 | Test Count: 164093
```

Training & Saving

```
# Define the path
model_path = "stellar_classifier_rf_v1"
```

```python
if not os.path.exists(model_path):
    print(">> Model not found. Training a new model...")
    print(">> Starting Cross-Validation Training (This may take 5-10 mins)...")

    # 1. Train
    cv_model = cv.fit(train_data)

    # 2. Extract the Best Model (The winner)
    best_model = cv_model.bestModel

    # 3. Save ONLY the Best Model (Lighter and standard practice)
    best_model.write().overwrite().save(model_path)
    print(f"Model saved to {model_path}")

else:
    print(">> Model found. Loading saved model...")

    # 4. Load as PipelineModel
    best_model = PipelineModel.load(model_path)
    print(">> Model loaded.")

# --- Access the RF Stage for Parameters ---
# The Random Forest is the last stage in the pipeline (index -1)
best_rf_model = best_model.stages[-1]

print(f"\n>> Best Model Parameters:")
print(f"   Num Trees: {best_rf_model.getNumTrees}")
print(f"   Max Depth: {best_rf_model.getOrDefault('maxDepth')}")

# --- Evaluation ---
predictions = best_model.transform(test_data)

acc_eval = MulticlassClassificationEvaluator(metricName="accuracy")
f1_eval = MulticlassClassificationEvaluator(metricName="f1")
prec_eval = MulticlassClassificationEvaluator(metricName="weightedPrecision")

print("\n=== FINAL MODEL EVALUATION ===")
print(f"Accuracy: {acc_eval.evaluate(predictions):.2%}")
print(f"F1 Score: {f1_eval.evaluate(predictions):.2%}")
print(f"Precision: {prec_eval.evaluate(predictions):.2%}")
```

```
>> Model found. Loading saved model...
>> Model loaded.

>> Best Model Parameters:
   Num Trees: 50
   Max Depth: 12

=== FINAL MODEL EVALUATION ===


25/12/17 06:03:59 WARN DAGScheduler: Broadcasting large task binary with size 4.3 MiB
[Stage 29:=====================================================>   (13 + 1) / 14]


Accuracy: 98.51%


[Stage 31:=====================================================>   (13 + 1) / 14]


F1 Score: 98.54%


[Stage 33:=====================================================>   (13 + 1) / 14]


Precision: 98.63%
```

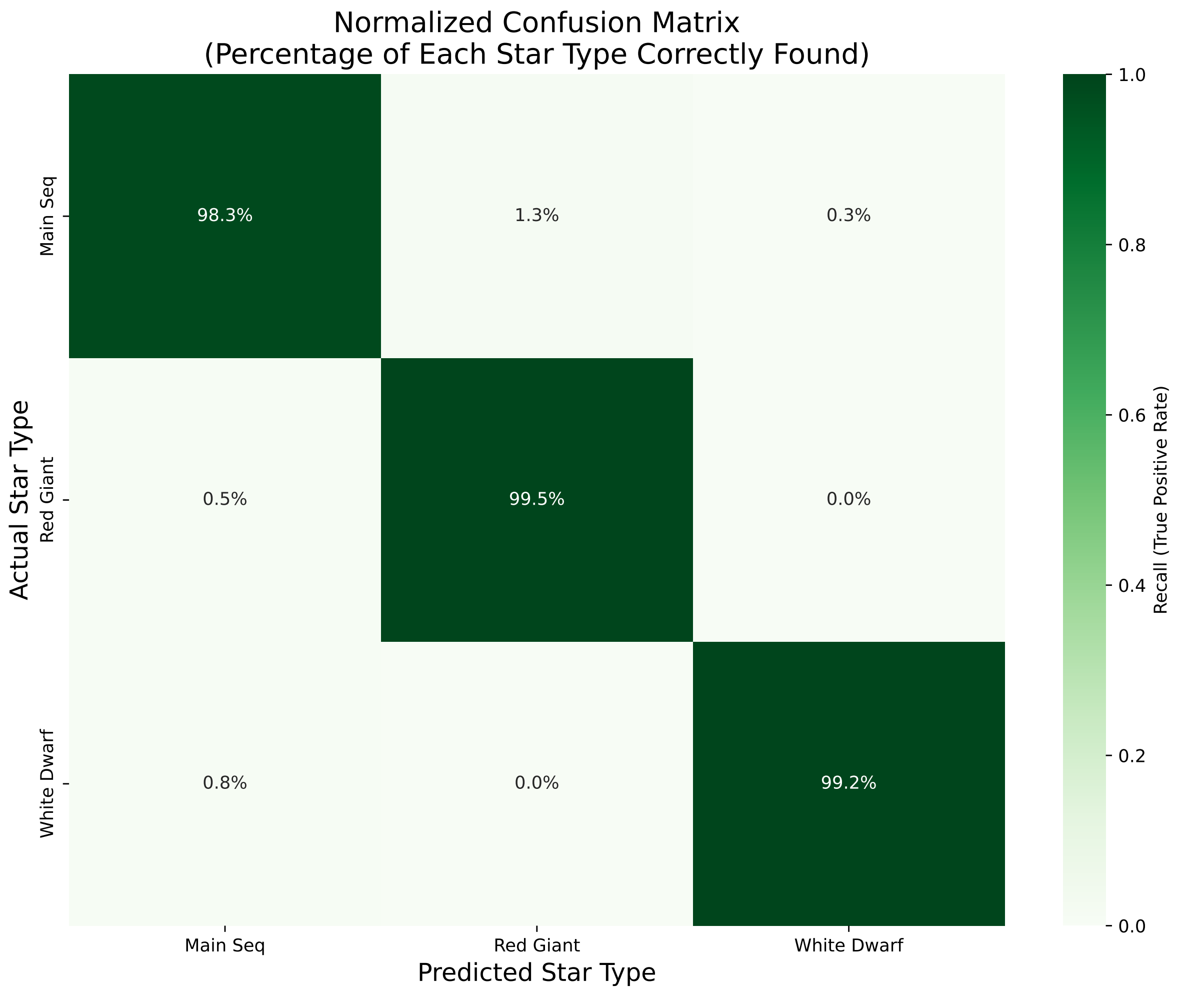
Confusion Matrix

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# 1. Collect Predictions
print(">> Collecting predictions to driver for visualization...")
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

# 2. Calculate Raw and Normalized Matrices
cm = confusion_matrix(y_true, y_pred)
labels = ["Main Seq", "Red Giant", "White Dwarf"]
```

```python
# Normalize row-wise: Divides count by the total stars in that TRUE class
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# 3. Plot the Heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(
    cm_normalized,
    annot=True,
    fmt='.1%',
    cmap='Greens',
    xticklabels=labels,
    yticklabels=labels,
    cbar_kws={'label': 'Recall (True Positive Rate)'},
    vmin=0, vmax=1   # Ensures color scale is fixed from 0% to 100%
)

plt.title('Normalized Confusion Matrix\n(Percentage of Each Star Type Correctly Found)', fon
plt.ylabel('Actual Star Type', fontsize=14)
plt.xlabel('Predicted Star Type', fontsize=14)
plt.tight_layout()
plt.show()
```

```
>> Collecting predictions to driver for visualization...
```

```
25/12/17 06:04:04 WARN DAGScheduler: Broadcasting large task binary with size 4.3 MiB
[Stage 36:=====================================================>   (13 + 1) / 14]
```



Feature Importance

```
importances = best_rf_model.featureImportances
print("\n=== Feature Importance ===")
feat_imp_list = sorted(zip(feature_cols, importances), key=lambda x: x[1], reverse=True)
for feat, score in feat_imp_list:
    print(f"{feat}: {score:.4f}")
```

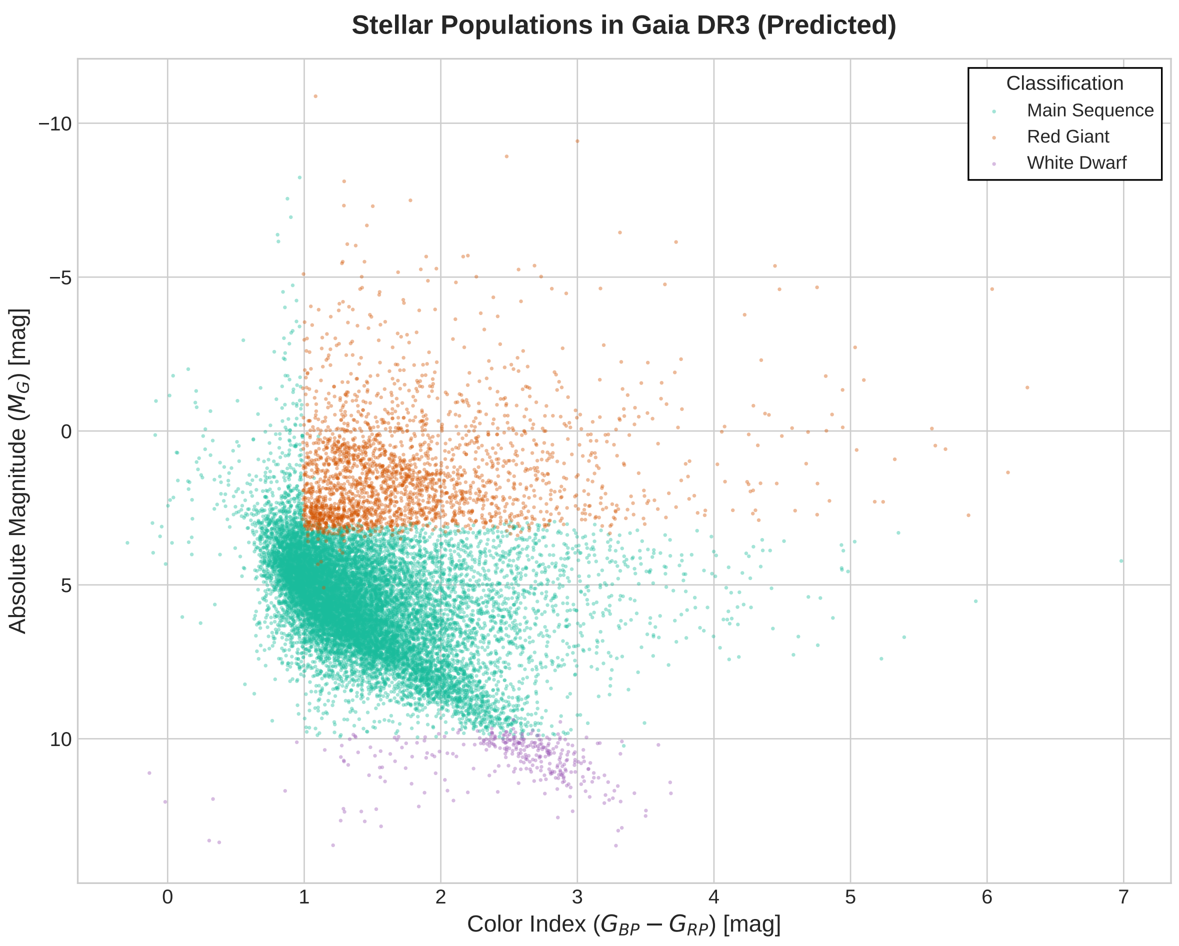
```
=== Feature Importance ===
log_parallax: 0.3897
parallax: 0.3247
```

```
phot_g_mean_mag: 0.2063
bp_rp: 0.0777
total_motion: 0.0016
```

Visualizing Predictions

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# 1. Setup Data
plot_data = predictions.select("bp_rp", "abs_mag", "prediction").sample(False, 0.1).toPandas

# 2. Map Predictions to Names
label_map = {0.0: "Main Sequence", 1.0: "Red Giant", 2.0: "White Dwarf"}
plot_data['Star Type'] = plot_data['prediction'].map(label_map)

# 3. Define "Academic" Palette (High Contrast for White Background)
academic_palette = {

    "White Dwarf": "#9b59b6",
    "Main Sequence": "#1abc9c",
    "Red Giant": "#d35400"
}

# 4. Create Plot with "Seaborn Whitegrid" Style
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(10, 8))

order = ["Main Sequence", "Red Giant", "White Dwarf"]

for star_type in order:
    subset = plot_data[plot_data['Star Type'] == star_type]

    ax.scatter(
        subset['bp_rp'],
        subset['abs_mag'],
        c=academic_palette[star_type],
        s=5,
        alpha=0.4,
        edgecolor='none',
        label=star_type
    )
# 6. Professional Aesthetics
ax.invert_yaxis()  # Standard Astronomy Convention
ax.set_title("Stellar Populations in Gaia DR3 (Predicted)", fontsize=16, weight='bold', pad=
ax.set_xlabel("Color Index ($G_{BP} - G_{RP}$) [mag]", fontsize=14)
ax.set_ylabel("Absolute Magnitude ($M_G$) [mag]", fontsize=14)

# Tick Customization
ax.tick_params(axis='both', which='major', labelsize=12)

# Legend (Boxed, Top Right, like the paper)
legend = ax.legend(
    title='Classification',
    fontsize=11,
    title_fontsize=12,
    loc='upper right',
    frameon=True,
    fancybox=False, # Square corners like the paper
    edgecolor='black',
    framealpha=1
)

plt.tight_layout()
plt.show()
```

```
25/12/17 06:04:06 WARN DAGScheduler: Broadcasting large task binary with size 4.3 MiB
[Stage 37:=====================================================>   (13 + 1) / 14]
```



Phase 2: Logistic Regression

Model setup

```
# 1. Feature Scaling (Crucial for Logistic Regression)
# This ensures 'total_motion' (large values) doesn't drown out 'bp_rp' (small values).
scaler = StandardScaler(
    inputCol="features",
    outputCol="scaled_features",
    withStd=True,
    withMean=True
)
```

```
# 2. Define the Estimator
# - family="multinomial": Explicitly tells Spark to handle 3 classes.
# - weightCol="classWeight": Uses the same weights as RF to handle the class imbalance.
lr = LogisticRegression(
    labelCol="label",
    featuresCol="scaled_features",
    weightCol="classWeight",
    family="multinomial",
    maxIter=100
)
```

Model Configuration & Tuning

```
# 3. Build the Pipeline
# Pipeline flow: Raw Data -> Vector -> Scaled Vector -> Logistic Regression
pipeline_lr = Pipeline(stages=[assembler, scaler, lr])

# 4. Create Parameter Grid (Hyperparameter Tuning)
# - regParam: Controls regularization strength (prevents overfitting).
paramGrid_lr = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.01, 0.1]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.5]) \
    .build()

# 5. Cross-Validation Setup
# We use the same 3-fold strategy as the Random Forest for consistency.
cv_lr = CrossValidator(
    estimator=pipeline_lr,
    estimatorParamMaps=paramGrid_lr,
    evaluator=evaluator, # Reusing the F1 evaluator from previous model
    numFolds=3,
    parallelism=2
)
```

Model Training & Evaluation

```
# 6. Train the Model
print(">> Training Advanced Logistic Regression (with Scaling & CV)...")
```

```python
cv_model_lr = cv_lr.fit(train_data)
best_model_lr = cv_model_lr.bestModel

print(">> Training Complete.")

# --- Save the Best LR Model ---
model_path_lr = "stellar_classifier_lr_v1"
best_model_lr.write().overwrite().save(model_path_lr)
print(f">> Model saved to {model_path_lr}")

# --- Metrics Evaluation ---
predictions_lr = best_model_lr.transform(test_data)

acc_eval = MulticlassClassificationEvaluator(metricName="accuracy")
f1_eval = MulticlassClassificationEvaluator(metricName="f1")

print("\n=== LOGISTIC REGRESSION RESULTS ===")
print(f"Accuracy: {acc_eval.evaluate(predictions_lr):.2%}")
print(f"Weighted F1 Score: {f1_eval.evaluate(predictions_lr):.2%}")

# --- Advanced Analysis: Extract Coefficients ---
# This counts as "Explaining the algorithm" for your report.
best_lr_stage = best_model_lr.stages[-1] # Extract the LR stage from pipeline

print("\n>> Model Coefficients (Linear Weights):")
# Coefficients is a matrix: 3 classes x 5 features
coeff_matrix = best_lr_stage.coefficientMatrix
# Intercepts: 3 values (one per class)
intercepts = best_lr_stage.interceptVector

# Print coefficients for the White Dwarf class (Label 2.0)
# This helps explain what makes a star a "White Dwarf" according to the math.
wd_coeffs = coeff_matrix.toArray()[2]
print(f"White Dwarf Coefficients (vs Features): {wd_coeffs}")
```

```
>> Training Advanced Logistic Regression (with Scaling & CV)...

25/12/17 06:04:07 WARN BlockManager: Block rdd_122_1 already exists on this machine; not re-
adding it
25/12/17 06:04:08 WARN SparkStringUtils: Truncated the string representation of a plan since
25/12/17 06:04:09 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlil
[Stage 933:==================================================>   (13 + 1) / 14]
```

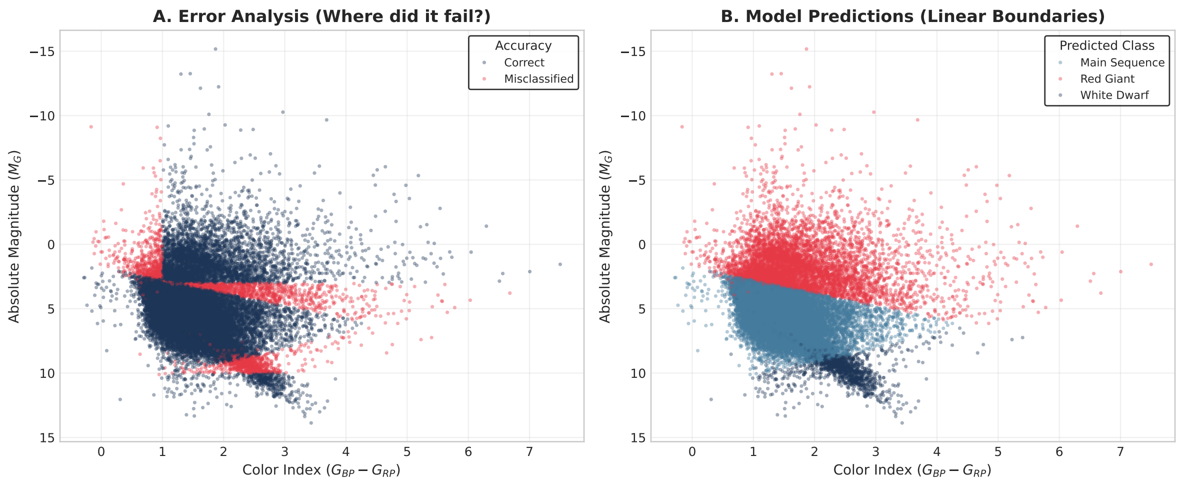
```
>> Training Complete.
>> Model saved to stellar_classifier_lr_v1

=== LOGISTIC REGRESSION RESULTS ===
Accuracy: 92.02%
Weighted F1 Score: 92.92%

>> Model Coefficients (Linear Weights):
White Dwarf Coefficients (vs Features): [0.03140263 1.14112309 0.00354785 0.58773451 1.17785
```

Visualizing Predictions

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import functions as F

# --- 1. Data Preparation ---
print(">> Collecting data for Side-by-Side visualization...")
df_viz = predictions_lr.select("bp_rp", "abs_mag", "label", "prediction") \
    .sample(False, 0.2, seed=42) \
    .toPandas()

# --- Logic for Plot A: Error Analysis ---
df_viz["is_correct"] = df_viz["label"] == df_viz["prediction"]
df_viz["Prediction Status"] = df_viz["is_correct"].map({True: "Correct", False: "Misclassifi

# --- Logic for Plot B: Prediction Classes ---
label_map = {0.0: "Main Sequence", 1.0: "Red Giant", 2.0: "White Dwarf"}
df_viz["Predicted Star Type"] = df_viz["prediction"].map(label_map)

# --- 2. Setup Side-by-Side Plot (COMPACT SIZE) ---
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.set_style("whitegrid")

custom_red = "#e63946"
custom_med_blue = "#457b9d"
custom_dark_blue = "#1d3557"

# --- PLOT 1: The Error Map (Left) ---
error_palette = {"Correct": custom_dark_blue, "Misclassified": custom_red}
```

```python
sns.scatterplot(
    data=df_viz,
    x="bp_rp",
    y="abs_mag",
    hue="Prediction Status",
    palette=error_palette,
    s=5,
    alpha=0.4,
    edgecolor=None,
    ax=axes[0]
)

axes[0].set_title("A. Error Analysis (Where did it fail?)", fontsize=12, weight='bold')
axes[0].legend(loc="upper right", frameon=True, edgecolor='black', title="Accuracy", fontsiz

# --- PLOT 2: The Prediction Map (Right) ---
# Red -> Red Giant, Med Blue -> Main Seq, Dark Blue -> White Dwarf
model_palette = {
    "Red Giant": custom_red,
    "Main Sequence": custom_med_blue,
    "White Dwarf": custom_dark_blue
}

sns.scatterplot(
    data=df_viz,
    x="bp_rp",
    y="abs_mag",
    hue="Predicted Star Type",
    palette=model_palette,
    hue_order=["Main Sequence", "Red Giant", "White Dwarf"],
    s=5,
    alpha=0.4,
    edgecolor=None,
    ax=axes[1]
)

axes[1].set_title("B. Model Predictions (Linear Boundaries)", fontsize=12, weight='bold')
axes[1].legend(loc="upper right", frameon=True, edgecolor='black', title="Predicted Class", 

for ax in axes:
    ax.invert_yaxis()  # Standard Astronomy Convention
    ax.set_xlabel("Color Index ($G_{BP} - G_{RP}$)", fontsize=10)
    ax.set_ylabel("Absolute Magnitude ($M_G$)", fontsize=10)
    ax.grid(True, linestyle='-', alpha=0.3)
    ax.tick_params(axis='both', which='major', labelsize=9)

plt.tight_layout()
plt.show()
```

```
>> Collecting data for Side-by-Side visualization...
```


A. Error Analysis (Where did it fail?)
Accuracy
Correct
Misclassified
Absolute Magnitude (M_G)
Color Index ($G_{BP} - G_{RP}$)
B. Model Predictions (Linear Boundaries)
Predicted Class
Main Sequence
Red Giant
White Dwarf
Absolute Magnitude (M_G)
Color Index ($G_{BP} - G_{RP}$)