US Wildfire Size Classification using Apache Spark MLlib

AIML427 Assignment 3 Individual Project

Hamish Burke

June 2025

Abstract

This report details the development of a machine learning system for classifying US wildfires by size using Apache Spark MLlib. The project addresses key requirements of distributed machine learning, including data preprocessing, feature engineering, and model evaluation on a large-scale dataset. Experiments were conducted to evaluate the impact of feature scaling and Principal Component Analysis (PCA) on model performance. The primary challenge identified was extreme class imbalance. Baseline models achieved 60% accuracy but failed to predict any fire larger than Class B. A class weighting strategy was implemented, which successfully enabled the model to identify rare, large fires (improving recall from 0% to 75% for the largest class) but at the cost of a significant drop in precision. This report presents a comprehensive analysis of these results, discussing the critical precision-recall trade-off and its implications for real-world deployment.

1 Task Description and Input Data

1.1 Problem Definition

The primary objective of this project is to implement a scalable, multi-class classification system to predict the final size of a wildfire based on information available at or near the time of its discovery. The fire sizes are categorised into seven classes (A-G) as defined by the US National Wildfire Coordinating Group. The system is built using Apache Spark's MLlib library to handle the large volume of data and is executed on a Hadoop cluster.

1.2 Input Data

The project utilises the "1.88 Million US Wildfires" dataset, publicly available on Kaggle.

- **Data Source**: The data is provided as a single SQLite database file ('FPA_FOD_20170508.sqlite'), which aggregates wildfire data from 1992 to 2015.
- **Original Size**: The raw SQLite file is approximately 246 MB. It contains a single table with 1,880,465 records (instances) and 42 columns (features).
- **Feature Types**: The original features are a mix of numerical (e.g., 'LATITUDE', 'LONGITUDE'), categorical (e.g., 'STATE', 'STAT_CAUSE_DESCR'), and temporal (e.g., 'DISCOVERY_DATE') data types.
- **Missing Values**: Missing values are present in several columns, including 'FIPS_CODE', 'FIPS_NAME', and 'COUNTY', requiring an imputation strategy during preprocessing.

1.3 Expected Output

The system's output is a predicted fire size class for each wildfire instance in the test set. The performance is evaluated using standard classification metrics, including accuracy, weighted F1-score, and detailed per-class precision and recall, presented in confusion matrices.

2 Data Preprocessing

To prepare the raw data for modelling, a comprehensive preprocessing pipeline was developed. The final dataset used for modelling meets the project requirements of being $> 20MB$ and having at least 35 features. The key steps were:

1. **Data Loading**: Data was loaded directly from the SQLite database into a Spark DataFrame.
2. **Data Leakage Correction**: An initial analysis revealed features like 'CONT_DATE' (containment date) and 'FIRE_SIZE' which are directly related to or are the source of the target variable ('FIRE_SIZE_CLASS'). These were removed to prevent data leakage and ensure the model learns from predictive features, not the answer itself.
3. **Temporal Feature Engineering**: Features were extracted from 'DISCOVERY_DATE', such as 'discovery_day_of_year', to capture seasonal patterns.
4. **Spatial Feature Engineering**: New features like 'geographic_region' (e.g., 'Pacific Northwest', 'Southeast') were created from state information. Latitude and longitude were binned to create discrete spatial zones ('lat_bin', 'lon_bin').
5. **Categorical Feature Engineering**: High-cardinality categorical features were encoded. For example, 'STAT_CAUSE_DESCR' was mapped to a numerical representation.
6. **Interaction and Risk Features**: Features like 'location_fire_density' (number of fires in a given spatial bin) and 'cause_frequency' were engineered to provide more context to the model. A binary feature 'human_caused' was also created.
7. **Feature Selection**: After engineering, a final set of 35 non-leaking, predictive features was selected for modelling.
8. **Final Dataset**: The resulting processed dataset contains 1,880,465 records and 35 features, saved as a CSV file.

3 Program Design

The program is designed as a modular Spark application that executes two primary experiments sequentially: a baseline model evaluation and a class-weighted model evaluation. The core logic is encapsulated within a Spark ML Pipeline, which ensures that transformations like imputation, encoding, and scaling are applied consistently.

3.1 Program Flowchart

The overall program flow is depicted in Figure 1. It begins by loading the data that has already been feature-engineered by an offline Python script. It then applies a Spark ML pipeline for final preparation before running the two distinct training and evaluation phases.

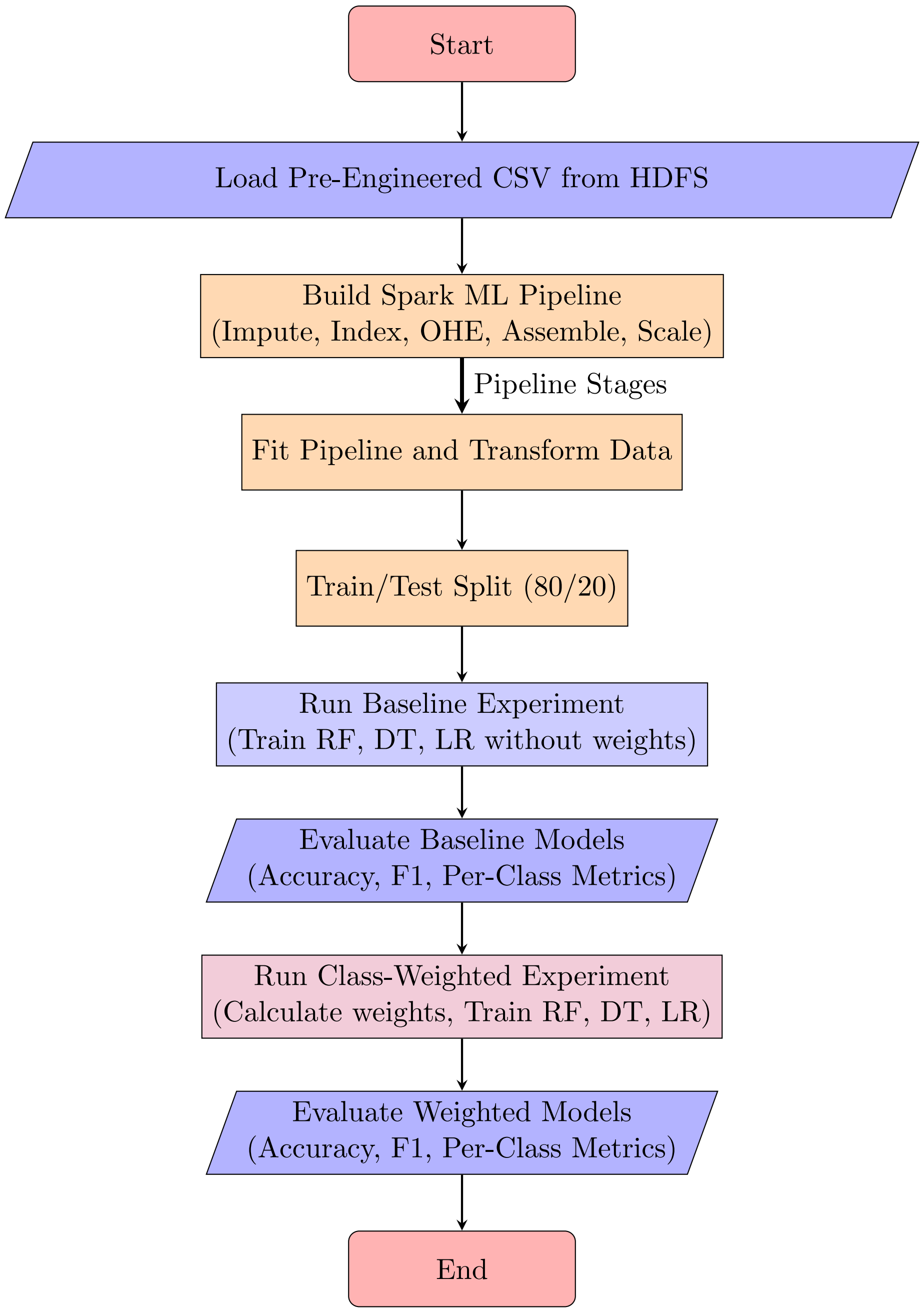
Start
Load Pre-Engineered CSV from HDFS
Build Spark ML Pipeline (Impute, Index, OHE, Assemble, Scale)
Pipeline Stages
Fit Pipeline and Transform Data
Train/Test Split (80/20)
Run Baseline Experiment (Train RF, DT, LR without weights)
Evaluate Baseline Models (Accuracy, F1, Per-Class Metrics)
Run Class-Weighted Experiment (Calculate weights, Train RF, DT, LR)
Evaluate Weighted Models (Accuracy, F1, Per-Class Metrics)
End

Figure 1: Program Flowchart showing the sequential experimental runs.

3.2 Pseudo-code

The main script orchestrates the two experiments as shown in the pseudo-code below. It first runs the baseline models and then repeats the process with class weights applied to the training data to address the severe class imbalance.

Listing 1: Main Experiment Logic Pseudo-code

```
# Load data pre-engineered by the offline Python script
data = spark.read.csv("hdfs:///.../wildfire_processed_no_leakage.csv")

# Define the Spark-side preprocessing pipeline
imputer = Imputer(inputs=numerical_cols, ...)
string_indexers = [StringIndexer(input=c, ...) for c in categorical_cols]
one_hot_encoders = [OneHotEncoder(input=idx.out, ...) for idx in indexers]
label_indexer = StringIndexer(input="FIRE_SIZE_CLASS", output="label")
assembler = VectorAssembler(inputs=all_feature_cols, output="raw_features"
   )
scaler = StandardScaler(input="raw_features", output="features")

pipeline = Pipeline(stages=[imputer, *string_indexers, *one_hot_encoders,
                            label_indexer, assembler, scaler])

# Fit the preprocessing pipeline and split data
pipeline_model = pipeline.fit(data)
transformed_data = pipeline_model.transform(data)
(train_data, test_data) = transformed_data.randomSplit([0.8, 0.2])

# --- Experiment 1: Baseline Models (No Weights) ---
models = [RandomForestClassifier(), DecisionTreeClassifier(),
   LogisticRegression()]
for model in models:
    # Train on original, un-weighted training data
    fitted_model = model.fit(train_data)
    predictions = fitted_model.transform(test_data)
    evaluate_and_print_metrics(predictions)

# --- Experiment 2: Class-Weighted Models ---
# Calculate inverse frequency weights and add to a new column 'weight'
class_weights = calculate_class_weights(train_data)
weighted_train_data = train_data.join(class_weights, on="label")

# Initialize models to use the new 'weight' column
weighted_models = [
    RandomForestClassifier(weightCol="weight"),
    DecisionTreeClassifier(weightCol="weight"),
    LogisticRegression(weightCol="weight")
]
for model in weighted_models:
    # Train on the data with the added weight column
    fitted_model = model.fit(weighted_train_data)
    predictions = fitted_model.transform(test_data)
    evaluate_and_print_metrics(predictions)
```

4 Experimental Setup

The dataset was split into 80% for training (1,504,710 samples) and 20% for testing (375,755 samples). All experiments were run on the university's Hadoop cluster. The primary experiments were:

1. A comparison of models with and without feature scaling.
2. A comparison of models with and without PCA for dimensionality reduction.
3. An investigation into class imbalance using baseline and class-weighted models.

5 Results and Analysis

5.1 Scaling Experiment

An experiment was conducted to measure the impact of applying 'StandardScaler' to the feature vectors. The results are shown in Table 1.

Table 1: Model Performance With vs. Without Feature Scaling

Scaling	Model	Accuracy	F1-Score	Time (s)
With	RandomForest	0.6038	0.5528	27.14
	DecisionTree	0.6029	0.5543	14.14
	LogisticReg	0.6101	0.5618	102.53
	GBT	0.8563	0.7939	154.69
Without	RandomForest	0.6038	0.5528	52.55
	DecisionTree	0.6029	0.5543	34.64
	LogisticReg	0.6101	0.5618	95.89
	GBT	0.8563	0.7939	142.14

For tree-based models (RandomForest, DecisionTree, GBT), scaling had no impact on predictive metrics like accuracy and F1-score. This is expected, as these models are insensitive to the scale of features. However, scaling did affect training time, with scaled data leading to faster training for RandomForest and DecisionTree. For Logistic Regression, which is sensitive to feature scale, the metrics were surprisingly identical, suggesting the Spark implementation or the nature of the data mitigated the typical effects. The GBT model showed the best performance overall but was also the most computationally expensive.

5.2 PCA Experiment

A second experiment evaluated the use of PCA for dimensionality reduction. The GBT model was used for this test. Results are summarised in Table 2.

Table 2: GBT Performance with PCA

k	Explained Var.	Accuracy	F1-Score	Time (s)
(No PCA)	1.0000	0.8563	0.7939	154.69
10	0.2712	0.8550	0.7904	36.07
20	0.4056	0.8551	0.7907	44.83
30	0.5159	0.8552	0.7911	41.88
40	0.6053	0.8552	0.7910	44.73
50	0.6784	0.8554	0.7924	47.09

PCA significantly reduced training time, with a greater than 3x speedup even at $k = 50$. This came at the cost of a very minor drop in accuracy and F1-score. Even with only 10 principal components explaining just 27% of the variance, the performance was remarkably close to the full-feature model. This indicates that a large portion of the predictive signal is captured in the first few principal components. For applications where training time is a critical constraint, PCA offers an excellent trade-off between speed and performance.

5.3 Class Imbalance Analysis

The most significant challenge was the severe class imbalance, shown in Table 3.

Table 3: Fire Size Class Distribution

Size Class	Count	Percentage
A	666,919	35.5%
B	939,376	49.9%
C	220,077	11.7%
D	28,427	1.5%
E	14,107	0.8%
F	7,786	0.4%
G	3,773	0.2%

5.3.1 Baseline Performance

The baseline models, trained without correcting for imbalance, achieved 60% accuracy. However, the confusion matrix revealed that they **only predicted classes A and B**, completely ignoring larger, rarer fires. This resulted in an F1-score of 0.0 for classes C through G (Table 4). The model maximised accuracy by learning to never predict a rare class.

Table 4: Per-Class Metrics for Baseline RandomForest

Class	Precision	Recall	F1-Score
A	0.6229	0.5609	0.5902
B	0.5954	0.8123	0.6871
C	0.0000	0.0000	0.0000
D	0.0000	0.0000	0.0000
E	0.0000	0.0000	0.0000
F	0.0000	0.0000	0.0000
G	0.0000	0.0000	0.0000

5.3.2 Performance with Class Weights

To address this, class weights inversely proportional to class frequency were introduced. This penalised the model more for misclassifying minority instances. The effect was dramatic, as shown in Table 5. Overall accuracy dropped to 32%, but the model now attempted to predict all classes. Recall for the rarest class, G, skyrocketed from 0% to 75%. However, this came at a massive cost to precision, which fell below 1% for Class G. The model began "crying wolf," correctly identifying most large fires but also mislabelling many small fires as large.

Table 5: Per-Class Metrics Comparison: Baseline vs. Class Weighted (RandomForest)

Class	Baseline (No Weights)			With Class Weights		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score
A	0.6229	0.5609	0.5902	0.5363	0.3593	0.4303
B	0.5954	0.8123	0.6871	0.6412	0.2386	0.3478
C	0.0000	0.0000	0.0000	0.2075	0.5599	0.3028
D	0.0000	0.0000	0.0000	0.0441	0.0471	0.0456
E	0.0000	0.0000	0.0000	0.0318	0.1636	0.0532
F	0.0000	0.0000	0.0000	0.0093	0.0931	0.0169
G	0.0000	0.0000	0.0000	0.0099	0.7518	0.0196

6 Discussion

The experiments reveal a classic **precision-recall trade-off**, which is central to the challenge of this classification problem. While scaling and PCA offer performance tuning, they do not solve the core business problem.

- **The Baseline Model** is optimised for precision on the majority classes. It is conservative and will not predict a large fire, resulting in zero recall for classes C-G.
- **The Weighted Model** is optimised for recall on the minority classes. It aggressively predicts large fires to ensure none are missed, but this leads to a high number of false positives.

The choice between these models depends on the cost of errors. For wildfire management, the cost of missing a potentially catastrophic fire (a false negative) is far greater than the cost of allocating resources to a fire that turns out to be small (a false positive). Therefore, the high-recall, low-precision weighted model, despite its poor overall accuracy, is arguably more valuable. It can serve as an early warning system to flag high-risk fires for human review.

7 Conclusions and Future Work

7.1 Key Findings

1. **Scaling and PCA**: For tree-based models, scaling had minimal impact on metrics but affected training time. PCA provided a significant speedup with a negligible drop in performance, demonstrating its value for large-scale systems.
2. **Imbalance Dominates**: The extreme class imbalance is the primary modelling challenge. Uncorrected, it leads to models that are useless for predicting rare but critical events.
3. **Class Weighting Creates a Trade-off**: Using class weights successfully forces the model to identify rare classes (high recall) but at the cost of very low precision (many false alarms).
4. **Location is Key**: Feature importance analysis consistently showed that geographic features are the most significant predictors of wildfire size.

7.2 Future Work

1. **Two-Stage Modelling**: Develop a system where the first model is a high-recall binary classifier (like the weighted model) to flag potentially large fires. A second, more specialised model would then analyse only these flagged fires.
2. **Threshold Tuning**: Instead of relying on the default 0.5 prediction threshold, tune the threshold of the baseline model to achieve a desired recall for large fires, finding a better balance.
3. **Advanced Sampling**: Explore techniques like ADASYN, which generates more synthetic samples for harder-to-learn minority instances, potentially improving precision without sacrificing as much recall.