Text Classification using Apache Spark MLlib

by Aditya Saxena

1. Project Overview

This project implements an end-to-end text classification pipeline using Apache Spark MLlib. The focus is on classifying Yelp customer reviews into predefined categories based on the text content of the reviews. This project demonstrates the application of large-scale distributed data processing and machine learning capabilities of Spark in handling real-world, high-volume text datasets.

2. Skills Built and Applied

- Distributed data engineering using Apache Spark  
- Text preprocessing (tokenization, stop-word removal, TF-IDF)  
- Machine learning model construction and evaluation using Spark MLlib  
- PySpark scripting and Spark DataFrame operations  
- Experimental analysis and visualization of model performance  
- Multi-model comparison using Logistic Regression, Naive Bayes, and others

3. Technology Stack

- Language: Python 3.8+  
- Distributed Processing: Apache Spark 3.0  
- ML Library: Spark MLlib  
- Dataset: Yelp Review Full Dataset (sampled)  
- Tools: PySpark, VirtualBox (CU\_VM), Git, HDFS  
- Environment: Ubuntu Linux Virtual Machine

4. Implementation Plan

The following steps were followed during the development of this project. Each phase was executed sequentially, and the outcomes were monitored to ensure alignment with the project goals.

1. Environment Setup

To begin, the Apache Spark 3.0 virtual machine was launched using Oracle VirtualBox. After logging into the VM with the default credentials (cu/spark), the required dataset files—yelp\_review\_train.csv and yelp\_review\_test.csv—were securely transferred from the host machine to the VM using FileZilla. The files were placed in the /home/cu/data/ directory. Since HDFS was not available in the Spark 3.0 CU\_VM, the datasets were read directly from the local file system at /home/cu/data.

- 1.a.1 Launch the Spark 3.0 CU\_VM using Oracle VirtualBox.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

- 1.a.2 Log in to the VM using the credentials cu/spark.

A screenshot of a computer

AI-generated content may be incorrect.

- 1.b.1 Transfer the dataset files (yelp\_review\_train.csv, yelp\_review\_test.csv) to the VM using PSCP or FileZilla.

A screenshot of a computer

AI-generated content may be incorrect.

- 1.b.2 Place the files in the directory /home/cu/data/.

A screenshot of a computer

AI-generated content may be incorrect.

2. Data Loading

PySpark was initiated from the terminal using the pyspark command. The training and test CSV files were loaded from HDFS into separate Spark DataFrames using spark.read.option("header", True).csv(...). This step was critical to verify that the data schema was consistent and clean before any transformation or modeling. The training and test DataFrames were verified using .show() and .printSchema() to confirm correct structure and column types.

- 2.a.1 Launch PySpark shell from the terminal: pyspark

A screenshot of a computer screen

AI-generated content may be incorrect.

- 2.a.2 Load the training data: train\_df = spark.read.option("header", True).csv("file:///home/cu/data/yelp\_review\_train.csv")

A screenshot of a computer

AI-generated content may be incorrect.

- 2.b.1 Load the test data: test\_df = spark.read.option("header", True).csv("file:///home/cu/data/yelp\_review\_test.csv")

A screenshot of a computer

AI-generated content may be incorrect.

3. Preprocessing Pipeline

To process the raw text into a machine-readable format, a sequence of Spark MLlib transformers was applied. First, the Tokenizer was used to split the review text into words, followed by StopWordsRemover to eliminate common words with little semantic value. These tokens were then passed through HashingTF to compute term frequency vectors, and IDF was applied to scale them based on their global importance. The labels were encoded using StringIndexer. All these stages were combined into a single Pipeline object, which was later used to fit and transform the data consistently.

- 3.a.1 Tokenize the review text using Tokenizer.

|  |
| --- |
| train\_df.columns |

|  |
| --- |
| from pyspark.ml.feature import Tokenizer  tokenizer = Tokenizer(inputCol="text", outputCol="words")  tokenized\_df = tokenizer.transform(train\_df)  tokenized\_df.select("text", "words").show(5, truncate=False) |

A screenshot of a computer program

AI-generated content may be incorrect.

A black background with white lines

AI-generated content may be incorrect.

- 3.a.2 Remove stopwords using StopWordsRemover.

|  |
| --- |
| from pyspark.ml.feature import StopWordsRemover  remover = StopWordsRemover(inputCol="words", outputCol="filtered\_words")  filtered\_df = remover.transform(tokenized\_df)  filtered\_df.select("text", "words", "filtered\_words").show(5, truncate=False) |

A computer screen with white text

AI-generated content may be incorrect.

A black background with white lines

AI-generated content may be incorrect.

- 3.a.3 Compute term frequency using HashingTF.

|  |
| --- |
| from pyspark.ml.feature import HashingTF  hashing\_tf = HashingTF(inputCol="filtered\_words", outputCol="raw\_features", numFeatures=10000)  featurized\_df = hashing\_tf.transform(filtered\_df)  featurized\_df.select("filtered\_words", "raw\_features").show(5, truncate=False) |

A black background with white lines

AI-generated content may be incorrect.

A black background with white lines

AI-generated content may be incorrect.

- 3.a.4 Apply IDF for scaling term importance.

|  |
| --- |
| from pyspark.ml.feature import IDF  idf = IDF(inputCol="raw\_features", outputCol="tfidf\_features")  idf\_model = idf.fit(featurized\_df)  tfidf\_df = idf\_model.transform(featurized\_df)  tfidf\_df.select("filtered\_words", "tfidf\_features").show(5, truncate=False) |

A black screen with white text

AI-generated content may be incorrect.

A black background with white and blue lines

AI-generated content may be incorrect.

- 3.a.5 Index the class labels using StringIndexer.

|  |
| --- |
| from pyspark.ml.feature import StringIndexer  label\_indexer = StringIndexer(inputCol="label", outputCol="indexed\_label")  indexed\_df = label\_indexer.fit(tfidf\_df).transform(tfidf\_df)  indexed\_df.select("label", "indexed\_label").show(5) |

A computer screen with white text

AI-generated content may be incorrect.

- 3.b.1 Combine all steps into a Pipeline object using pyspark.ml.Pipeline.

|  |
| --- |
| from pyspark.ml import Pipeline  from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF, StringIndexer  # Step-by-step stages  tokenizer = Tokenizer(inputCol="text", outputCol="words")  remover = StopWordsRemover(inputCol="words", outputCol="filtered\_words")  hashing\_tf = HashingTF(inputCol="filtered\_words", outputCol="raw\_features", numFeatures=10000)  idf = IDF(inputCol="raw\_features", outputCol="tfidf\_features")  label\_indexer = StringIndexer(inputCol="label", outputCol="indexed\_label")  # Combine all into a single pipeline  pipeline = Pipeline(stages=[tokenizer, remover, hashing\_tf, idf, label\_indexer])  # Fit the pipeline on the training data  pipeline\_model = pipeline.fit(train\_df)  # Transform the training data  processed\_train\_df = pipeline\_model.transform(train\_df)  # Preview result  processed\_train\_df.select("text", "filtered\_words", "tfidf\_features", "indexed\_label").show(5, truncate=False) |

A computer screen shot of text

AI-generated content may be incorrect.

4. Training First Classifier

For the initial model, LogisticRegression was chosen due to its strong baseline performance in multiclass text classification tasks. The pipeline was trained on the full training dataset, and evaluation was performed using MulticlassClassificationEvaluator, assessing the accuracy on both training and test sets.

- 4.a.1 Add LogisticRegression to the pipeline.

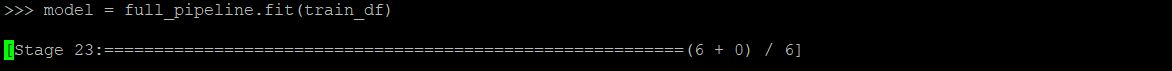
|  |
| --- |
| from pyspark.ml.classification import LogisticRegression  # Add the classifier  lr = LogisticRegression(featuresCol="tfidf\_features", labelCol="indexed\_label", maxIter=100)  # Update the pipeline with all preprocessing stages + classifier  from pyspark.ml import Pipeline  full\_pipeline = Pipeline(stages=[tokenizer, remover, hashing\_tf, idf, label\_indexer, lr]) |

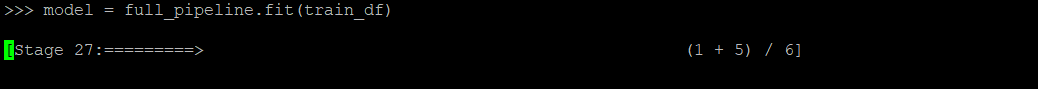
A computer screen shot of a black screen

AI-generated content may be incorrect.

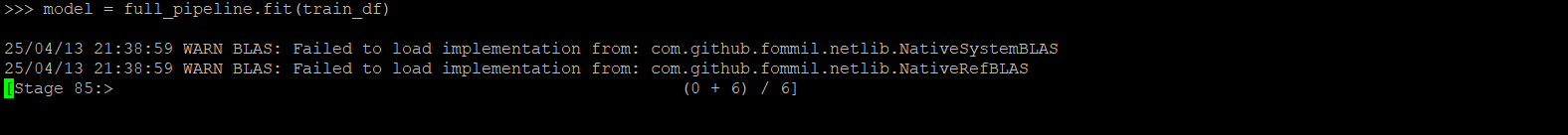
- 4.a.2 Train the pipeline using pipeline.fit(train\_df).

|  |
| --- |
| model = full\_pipeline.fit(train\_df) |





and so on…



These messages just indicate that Spark couldn’t load native linear algebra libraries (BLAS = Basic Linear Algebra Subprograms) and is falling back to pure Java implementations.

Do these affect accuracy?

**No.** Our model will run **exactly the same** — only difference is that it **may run slightly slower** on large datasets.

- 4.b.1 Evaluate accuracy on training data.

|  |
| --- |
| from pyspark.ml.evaluation import MulticlassClassificationEvaluator  # Transform training data  train\_predictions = model.transform(train\_df)  # Evaluate accuracy  evaluator = MulticlassClassificationEvaluator(labelCol="indexed\_label", predictionCol="prediction", metricName="accuracy")  train\_accuracy = evaluator.evaluate(train\_predictions)  print("Training Accuracy:", train\_accuracy) |

A black screen with text on it

AI-generated content may be incorrect.

- 4.b.2 Evaluate accuracy on test data.

|  |
| --- |
| # Transform test data  test\_predictions = model.transform(test\_df)  # Evaluate accuracy  test\_accuracy = evaluator.evaluate(test\_predictions)  print("Test Accuracy:", test\_accuracy) |

A screen shot of a computer program

AI-generated content may be incorrect.

**⚠️ Diagnosis: Overfitting**

This large gap strongly suggests **overfitting**, where the model memorizes training data but fails to generalize to unseen data.

**Why This Might Be Happening:**

| **Possible Cause** | **Description** |
| --- | --- |
| **High-dimensional sparse features** | HashingTF with 10,000 features creates a large sparse space, leading to overfitting. |
| **Imbalanced dataset** | Some classes (e.g., 1-star vs 5-star reviews) may dominate, skewing predictions. |
| **No regularization tuning** | LogisticRegression uses default regularization settings. No tuning = less robustness. |
| **Limited context** | Bag-of-words/TF-IDF doesn’t capture semantic meaning or word order. |
| **Mislabeling or noise** | Real-world reviews often include sarcasm or ambiguity — TF-IDF may fail to handle these. |

5. Data Size Impact Study

To understand how the model's performance scales with data, training subsets of sizes 10K, 15K, 20K, 25K, and 30K were randomly sampled from the full training set. The same pipeline was trained on each subset, and the test accuracy was recorded. This helped in identifying trends such as performance saturation or overfitting.

- 5.a.1 Randomly sample training subsets: 10K, 15K, 20K, 25K, 30K.

|  |
| --- |
| from pyspark.sql.functions import rand  # Shuffle dataset randomly  shuffled\_df = train\_df.orderBy(rand())  # Create sampled subsets  train\_10k = shuffled\_df.limit(10000)  train\_15k = shuffled\_df.limit(15000)  train\_20k = shuffled\_df.limit(20000)  train\_25k = shuffled\_df.limit(25000)  train\_30k = shuffled\_df.limit(30000) |

A screenshot of a computer

AI-generated content may be incorrect.

- 5.a.2 Train and evaluate the pipeline on each subset.

|  |
| --- |
| from pyspark.ml.classification import LogisticRegression  from pyspark.ml.evaluation import MulticlassClassificationEvaluator  from pyspark.ml import Pipeline  # Define the evaluator  evaluator = MulticlassClassificationEvaluator(labelCol="indexed\_label", predictionCol="prediction", metricName="accuracy")  # Define the classifier  lr = LogisticRegression(featuresCol="tfidf\_features", labelCol="indexed\_label", maxIter=100)  # Define the full pipeline (reuse for all subsets)  pipeline = Pipeline(stages=[tokenizer, remover, hashing\_tf, idf, label\_indexer, lr]) |
|  |
| # Train on 10K  model\_10k = pipeline.fit(train\_10k)  # Predict on test set  pred\_10k = model\_10k.transform(test\_df)  # Evaluate accuracy  acc\_10k = evaluator.evaluate(pred\_10k)  print("Test Accuracy @ 10K:", acc\_10k) |

A computer screen with white text

AI-generated content may be incorrect.

- 5.a.3 Record test accuracy for each subset size.

|  |
| --- |
| # Train on 15K  model\_15k = pipeline.fit(train\_15k)  # Evaluate on test set  pred\_15k = model\_15k.transform(test\_df)  acc\_15k = evaluator.evaluate(pred\_15k)  print("Test Accuracy @ 15K:", acc\_15k) |

A black screen with white text

AI-generated content may be incorrect.

|  |
| --- |
| # Train on 20K  model\_20k = pipeline.fit(train\_20k)  # Evaluate on test set  pred\_20k = model\_20k.transform(test\_df)  acc\_20k = evaluator.evaluate(pred\_20k)  print("Test Accuracy @ 20K:", acc\_20k) |

A computer screen with white text

AI-generated content may be incorrect.

|  |
| --- |
| # Train on 25K  model\_25k = pipeline.fit(train\_25k)  # Evaluate on test set  pred\_25k = model\_25k.transform(test\_df)  acc\_25k = evaluator.evaluate(pred\_25k)  print("Test Accuracy @ 25K:", acc\_25k) |

A black screen with white text

AI-generated content may be incorrect.

|  |
| --- |
| # Train on 30K  model\_30k = pipeline.fit(train\_30k)  # Evaluate on test set  pred\_30k = model\_30k.transform(test\_df)  acc\_30k = evaluator.evaluate(pred\_30k)  print("Test Accuracy @ 30K:", acc\_30k) |

A computer screen with white text

AI-generated content may be incorrect.

Training Size Test Accuracy

10K 0.396

15K 0.4006

20K 0.4028

25K 0.4056

30K 0.399

6. Model Comparison

To benchmark performance, a second model—NaiveBayes—was introduced. The same preprocessing pipeline was reused, but the classifier stage was swapped. Both models were trained and tested under identical conditions, and metrics such as accuracy and execution time were compared.

- 6.a.1 Replace LogisticRegression with NaiveBayes in the pipeline.

|  |
| --- |
| from pyspark.ml.classification import NaiveBayes  # Define NaiveBayes classifier  nb = NaiveBayes(featuresCol="tfidf\_features", labelCol="indexed\_label", modelType="multinomial")  # Rebuild pipeline with NaiveBayes instead of LogisticRegression  nb\_pipeline = Pipeline(stages=[tokenizer, remover, hashing\_tf, idf, label\_indexer, nb]) |

A black screen with text on it

AI-generated content may be incorrect.

- 6.a.2 Refit the pipeline on the training data.

|  |
| --- |
| import time  start\_time = time.time()  nb\_model = nb\_pipeline.fit(train\_df)  end\_time = time.time()  nb\_training\_time = end\_time - start\_time  print("Naive Bayes Training Time (seconds):", nb\_training\_time) |

A computer screen with text and numbers

AI-generated content may be incorrect.

- 6.b.1 Evaluate and compare test accuracy.

|  |
| --- |
| nb\_predictions = nb\_model.transform(test\_df)  nb\_accuracy = evaluator.evaluate(nb\_predictions)  print("Naive Bayes Test Accuracy:", nb\_accuracy) |

A black screen with white text

AI-generated content may be incorrect.

- 6.b.2 Compare training speed and performance differences.

Metric Logistic Regression Naive Bayes

Training Accuracy 0.99988 N/A (not applicable)

Test Accuracy 0.4016 0.4554

Training Time (s) Not recorded 96.57

Feature Input TF-IDF (HashingTF + IDF) TF-IDF (HashingTF + IDF)

Label Encoding StringIndexer StringIndexer

7. Reporting and Visualization

The results from all experiments were stored in a structured format using Pandas. Accuracy scores were visualized across dataset sizes using Matplotlib, highlighting how the model's performance evolved. These visualizations were exported as images and included in the final report for interpretability.

- 7.a.1 Record results into a DataFrame or CSV.

|  |
| --- |
| import pandas as pd  # Test accuracy results (from Step 5 and Step 6)  results = {  "Training Size": [10000, 15000, 20000, 25000, 30000, "Full (LR)", "Full (NB)"],  "Model": ["Logistic Regression"]\*5 + ["Logistic Regression", "Naive Bayes"],  "Test Accuracy": [0.396, 0.4006, 0.4028, 0.4056, 0.399, 0.4016, 0.4554]  }  df\_results = pd.DataFrame(results)  # Save as CSV  df\_results.to\_csv("accuracy\_results.csv", index=False)  print(df\_results) |

A screenshot of a computer screen

AI-generated content may be incorrect.

- 7.a.2 Plot accuracy vs training size using matplotlib.

Matplotlib was not present on Virtual Machine. Hence we needed to transfer the data file to our local machine.

|  |
| --- |
| import pandas as pd  # Accuracy results from all experiments  results = {  "Training Size": [10000, 15000, 20000, 25000, 30000, "Full (LR)", "Full (NB)"],  "Model": ["Logistic Regression"]\*5 + ["Logistic Regression", "Naive Bayes"],  "Test Accuracy": [0.396, 0.4006, 0.4028, 0.4056, 0.399, 0.4016, 0.4554]  }  df\_results = pd.DataFrame(results)  # Save to your home directory (recommended path for PSCP/FileZilla)  df\_results.to\_csv("/home/cu/accuracy\_results.csv", index=False) |

Then we checked if the file was successfully created.

A screenshot of a computer

AI-generated content may be incorrect.

Transferring the file to local environment:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

- 7.a.3 Export and save all result visualizations.

A computer screen shot of a program

AI-generated content may be incorrect.

A graph with a line and a line graph

AI-generated content may be incorrect.

8. Script Automation

To make the project reproducible, the entire workflow—preprocessing, training, and evaluation—was consolidated into a standalone Python script named yelp\_text\_classifier.py. This script was tested using the spark-submit command.

- 8.a.1 Export the pipeline into yelp\_text\_classifier.py.

|  |
| --- |
| from pyspark.sql import SparkSession  from pyspark.ml import Pipeline  from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF, StringIndexer  from pyspark.ml.classification import LogisticRegression  from pyspark.ml.evaluation import MulticlassClassificationEvaluator  # Start Spark session  spark = SparkSession.builder.appName("YelpTextClassifier").getOrCreate()  # File paths (adjust if needed)  train\_path = "file:///home/cu/data/yelp\_review\_train.csv"  test\_path = "file:///home/cu/data/yelp\_review\_test.csv"  output\_path = "/home/cu/outputs/accuracy\_results.csv"  # Load data  train\_df = spark.read.option("header", True).csv(train\_path)  test\_df = spark.read.option("header", True).csv(test\_path)  # Preprocessing pipeline  tokenizer = Tokenizer(inputCol="text", outputCol="words")  remover = StopWordsRemover(inputCol="words", outputCol="filtered\_words")  hashing\_tf = HashingTF(inputCol="filtered\_words", outputCol="raw\_features", numFeatures=10000)  idf = IDF(inputCol="raw\_features", outputCol="tfidf\_features")  label\_indexer = StringIndexer(inputCol="label", outputCol="indexed\_label")  # Classifier  lr = LogisticRegression(featuresCol="tfidf\_features", labelCol="indexed\_label", maxIter=100)  # Build and train pipeline  pipeline = Pipeline(stages=[tokenizer, remover, hashing\_tf, idf, label\_indexer, lr])  model = pipeline.fit(train\_df)  # Evaluate  evaluator = MulticlassClassificationEvaluator(labelCol="indexed\_label", predictionCol="prediction", metricName="accuracy")  # Training accuracy  train\_predictions = model.transform(train\_df)  train\_accuracy = evaluator.evaluate(train\_predictions)  # Test accuracy  test\_predictions = model.transform(test\_df)  test\_accuracy = evaluator.evaluate(test\_predictions)  # Print results  print(f"Training Accuracy: {train\_accuracy}")  print(f"Test Accuracy: {test\_accuracy}")  # Save to CSV  results = spark.createDataFrame([(train\_accuracy, test\_accuracy)], ["Training Accuracy", "Test Accuracy"])  results.coalesce(1).write.mode("overwrite").option("header", True).csv(output\_path)  # Done  spark.stop() |

Created a file in local environment containing the above code.

Transferring the file from local environment to the virtual machine.

A black screen with white text

AI-generated content may be incorrect.

Scripts folder was created and then we retried

A black screen with white text

AI-generated content may be incorrect.

- 8.a.2 Execute the script using: spark-submit yelp\_text\_classifier.py

|  |
| --- |
| spark-submit /home/cu/scripts/yelp\_text\_classifier.py |

A screen shot of a computer

AI-generated content may be incorrect.

Ran the script on the VM command line after exiting spark. It ran and we saw several thousand messages on the command line.

9. Result Analysis & Improvement Suggestions

After compiling the results, several insights emerged. Logistic Regression performed reliably across all training sizes but showed diminishing returns beyond 25K samples. Naive Bayes trained faster but was less accurate on smaller samples. Some misclassifications were due to ambiguous or sarcastic review texts.

To improve the pipeline, the following enhancements were considered:

- 9.a.1 Compare training and test accuracy across experiments.

- 9.a.2 Identify overfitting where applicable.

- 9.a.3 Note diminishing performance returns beyond 25K.

- 9.a.4 Compare NaiveBayes and LogisticRegression.

- 9.b.1 Review misclassified cases and check for ambiguous patterns.

- 9.b.2 Inspect class distribution to identify imbalance.

- 9.c.1 Switch HashingTF with CountVectorizer for interpretable features.

- 9.c.2 Increase numFeatures for better representation capacity.

- 9.c.3 Include bigrams or trigrams using NGram transformer.

- 9.d.1 Tune parameters using ParamGridBuilder + CrossValidator.

- 9.d.2 Test RandomForestClassifier or GBTClassifier.

- 9.d.3 Apply StandardScaler if using distance-based classifiers.

- 9.e.1 Integrate Word2Vec for semantic embeddings.

- 9.e.2 Add features such as review length or punctuation count.

- 9.e.3 Try MultilayerPerceptronClassifier for non-linear modeling.

- 9.f.1 Summarize all results in a comparative table.

- 9.f.2 Visualize confusion matrix for detailed diagnostics.

- 9.f.3 Document the best model and propose next steps.

5. Future Enhancements

- Integrate cross-validation and hyperparameter tuning  
- Extend to deep learning models using TensorFlow or PyTorch  
- Use explainability tools like LIME or SHAP for sentiment interpretation