Text Classification using Apache Spark MLlib

A training project 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. HDFS was then prepared by creating a directory at /user/cu/data, and the datasets were uploaded using the hadoop fs -put command.

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

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

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

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

- 1.b.3 Create a directory in HDFS: hadoop fs -mkdir /user/cu/data

- 1.b.4 Upload the files to HDFS: hadoop fs -put /home/cu/data/yelp\_review\_\*.csv /user/cu/data/

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.

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

- 2.a.2 Load the training data: spark.read.option("header", True).csv("hdfs:///user/cu/data/yelp\_review\_train.csv")

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

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.

- 3.a.2 Remove stopwords using StopWordsRemover.

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

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

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

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

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.

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

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

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

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.

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

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

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.

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

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

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

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.

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

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

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.

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

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