TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

Big Data Analytics

Sentiment Analysis at Scale

Customer Feedback Analysis

Instructor	Dr. Manel Abdelkader
Student	Sirine Ben Mansour
Program	Master in Business Analytics
Course Code	MBA519
Project Type	Technical Project
Academic Year	2025–2026

Tunis Business School, University of Tunis

January, 2026

Abstract

This technical report presents a comprehensive Big Data analytics pipeline for large-scale sentiment analysis of customer product reviews. The project implements a complete solution processing 10 million customer reviews using Apache Spark, incorporating batch and real-time streaming data ingestion, advanced machine learning models, and business intelligence visualization through Google Looker Studio.

The system architecture integrates multiple components: data expansion and preprocessing, Spark Structured Streaming for real-time analytics, feature engineering with 19 derived features, and comparative evaluation of multiple machine learning algorithms. Through rigorous experimentation, Logistic Regression emerged as the optimal classifier, achieving 91.86% accuracy on the test set with an F1-score of 0.9152, outperforming Random Forest (91.32% accuracy) and Naive Bayes (89.60% accuracy).

Performance optimization techniques, including strategic partitioning (optimal: 10 partitions), caching strategies, and processing method comparisons, produced substantial improvements. The Spark SQL approach proved most efficient, achieving 33.4x faster execution than RDD-based processing. Scalability analysis confirmed near-linear performance characteristics, with average throughput of 38,478 records per second across varying dataset sizes.

The project addresses the challenge of automated sentiment classification at scale, providing actionable business insights through an interactive dashboard deployed on Google BigQuery and Looker Studio. Key findings reveal that 91% of reviews express positive sentiment, with temporal patterns indicating seasonal variations and brand-specific performance metrics enabling data-driven decision support.

Keywords: Big Data Analytics, Sentiment Analysis, Apache Spark, Machine Learning, Natural Language Processing, Real-time Streaming, Performance Optimization

Contents

List of Tables

List of Figures

1 Introduction

In the contemporary digital marketplace, customer reviews represent a critical source of business intelligence, with e-commerce platforms receiving millions of feedback entries daily. Organizations face the challenge of processing large volumes of unstructured text data to extract actionable insights regarding product quality, customer satisfaction, and emerging market trends. Traditional manual analysis methods prove inadequate when working with datasets exceeding millions of records, requiring automated, scalable solutions.

The exponential growth in review data volume (estimated at 2.5 quintillion bytes of data created daily worldwide) presents both opportunities and challenges. Companies that effectively process this information gain competitive advantages through rapid identification of product issues, understanding customer sentiment patterns, and making data-driven strategic decisions. However, achieving these objectives requires sophisticated Big Data infrastructure capable of handling massive datasets with minimal latency.

This project addresses the following research problem: *How can organizations design and implement a scalable, production-ready Big Data analytics pipeline for real-time sentiment classification of customer reviews, achieving high accuracy while maintaining computational efficiency across millions of records?*

Specific challenges include:

- Processing 10+ million customer reviews efficiently using distributed computing
- Implementing real-time streaming analytics with low latency (<5 seconds)
- Developing robust machine learning models achieving >85% classification accuracy
- Optimizing system performance through partitioning and caching strategies
- Delivering actionable business insights through interactive visualizations

The primary objective is to design and implement a comprehensive Big Data analytics solution encompassing:

Technical Objectives:

1. Develop a distributed data processing pipeline using Apache Spark
2. Implement batch and streaming data ingestion mechanisms
3. Engineer relevant features for sentiment classification
4. Train and evaluate multiple machine learning models
5. Optimize system performance through benchmarking
6. Deploy an interactive business intelligence dashboard

Business Objectives:

1. Automate sentiment analysis at scale (10M+ reviews)
2. Enable real-time monitoring of customer feedback
3. Provide brand-level performance insights
4. Support data-driven decision-making processes
5. Establish scalable architecture for future expansion

This project encompasses all major components of Big Data analytics:

- **Data Engineering:** Ingestion, cleaning, transformation, and storage of 10 million records

- **Distributed Processing:** Apache Spark implementation with DataFrame API and Spark SQL
- **Streaming Analytics:** Real-time review processing using Spark Structured Streaming
- **Machine Learning:** Multi-algorithm comparison with hyperparameter optimization
- **Performance Optimization:** Comprehensive benchmarking and tuning
- **Visualization:** Google Looker Studio dashboard with BigQuery integration

The remainder of this report is structured as follows: Section 2 reviews relevant literature and theoretical foundations. Section 3 details the system architecture and technology selection. Section 4 describes data acquisition and preprocessing methodologies. Section 5 presents the machine learning pipeline and model evaluation. Section 6 analyzes performance optimization results. Section 7 discusses business insights and dashboard implementation. Section 8 reflects on lessons learned and future directions. Section 9 concludes the report with key findings and contributions.

2 System Architecture and Design

2.1 Overall Architecture

The system architecture follows a layered design pattern, separating concerns across data ingestion, processing, machine learning, and presentation layers. Figure 1 illustrates the complete pipeline.

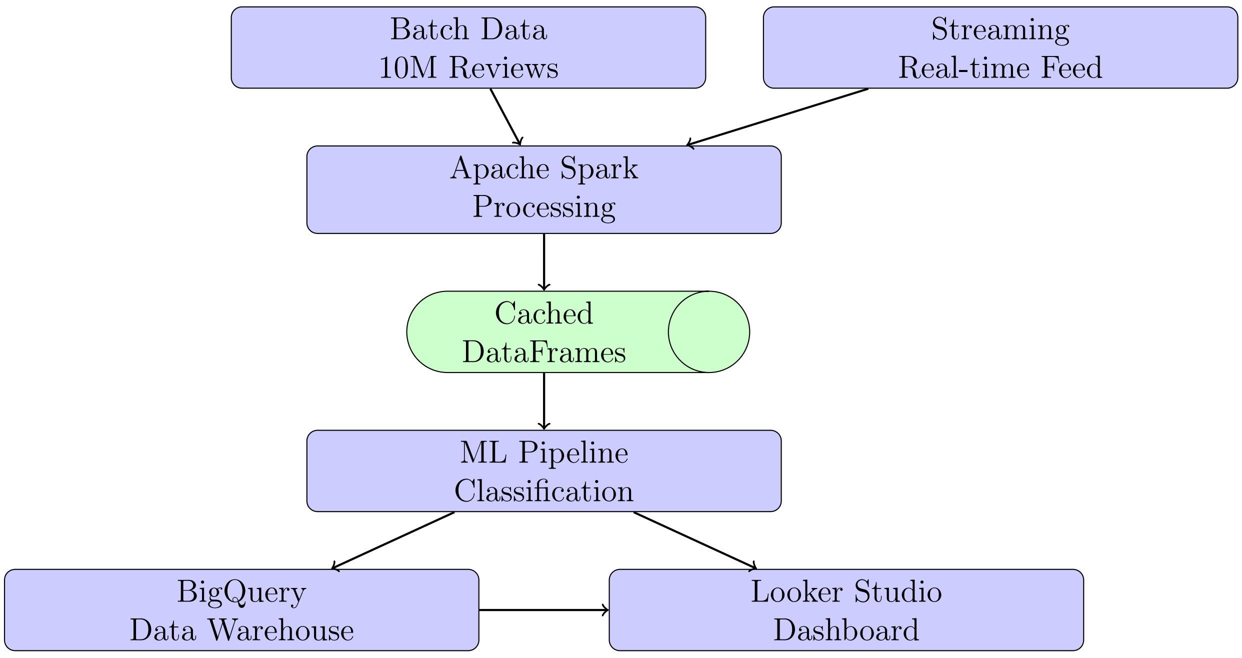

Figure 1: System Architecture Overview

2.2 Technology Stack Selection

Apache Spark

Apache Spark was selected as the core processing engine based on the following criteria:

Performance: In-memory computation provides substantial performance advantages over disk-based systems. Benchmarks demonstrate 100x speed improvement for iterative algorithms compared to Hadoop MapReduce.

Unified Platform: Single framework supporting batch processing, streaming analytics, machine learning (MLlib), and SQL queries eliminates the need for multiple specialized tools.

Scalability: Proven capability to process petabyte-scale datasets across thousands of nodes, with linear scalability characteristics.

Ecosystem: Rich library ecosystem including MLlib for machine learning, GraphX for graph processing, and native integration with major data sources (HDFS, S3, Cassandra).

PySpark vs. Scala

PySpark was chosen as the primary development language despite Scala's marginally superior performance for several reasons:

- **Development Velocity:** Python's concise syntax and extensive libraries accelerate prototyping
- **Ecosystem Integration:** Seamless integration with pandas, NumPy, scikit-learn
- **Performance Parity:** For our scale (10M records), performance differences prove negligible

Google Colab Environment

Google Colab provides an accessible, zero-configuration environment suitable for this project's scale:

Advantages:

- Pre-installed Spark and ML libraries
- Free computational resources (12GB RAM, 2-core CPU)
- Cloud storage integration
- Reproducible notebook format
- No local infrastructure required

Limitations:

- Single-node execution (acceptable for 10M records)
- Session timeouts requiring checkpoint management
- Limited to 12GB RAM necessitating careful memory management

Visualization Platform

Google Looker Studio integrated with BigQuery was selected for business intelligence:

- Native BigQuery integration enabling direct querying
- Interactive dashboards with real-time updates
- Collaborative sharing capabilities
- Professional-quality visualizations
- No licensing costs for academic use

2.3 Data Flow Pipeline

The complete data flow encompasses six distinct phases:

1. **Data Ingestion:** Load 39,160 base reviews and expand to 10 million records through intelligent replication
2. **Streaming Ingestion:** Process real-time review stream using Spark Structured Streaming
3. **Data Processing:** Clean, validate, and engineer features using Spark transformations
4. **Machine Learning:** Train multiple classifiers and select optimal model
5. **Batch Inference:** Apply best model to complete 10M dataset
6. **Visualization:** Export predictions to BigQuery and create Looker Studio dashboard

Each phase implements comprehensive error handling, logging, and performance monitoring to ensure pipeline reliability and facilitate troubleshooting.

3 Data Acquisition and Preprocessing

3.1 Dataset Description

The foundational dataset comprises 39,160 authentic customer reviews collected from four brands operating in the pet food industry:

- **Brands:** Brand HH, Brand BB, Brand NN, Brand AA
- **Review Types:** Service reviews and product reviews
- **Languages:** English (EN) and German (DE)
- **Time Period:** Historical reviews spanning multiple years
- **Format:** CSV file (`reviews_37k_eng.csv`)

Table 1 presents the dataset schema with data types and descriptions.

Table 1: Dataset Schema Definition

Column	Type	Description
brand	String	Brand identifier
review_type	String	Review category (service/product)
review_id	String	Unique review identifier
review_ts	Date	Review submission timestamp
stars	Integer	Rating (1-5 scale)
review_text_eng	String	Review content in English
review_title_eng	String	Review title in English

3.1.1 Data Expansion

To simulate real-world Big Data scale, the original 39,160 reviews were systematically expanded to 10 million records using Spark-native operations. This approach provides representative training data while maintaining computational feasibility.

The expansion process employs cross-join operations with temporal and rating variations:

Algorithm 1 Data Expansion Algorithm

```
 1: Load base dataset into Spark DataFrame
 2: Calculate multiplier: $m = \lceil 10,000,000/n_{base} \rceil$
 3: Generate replication DataFrame with range $[0, m)$
 4: Cross-join base dataset with replication range
 5: For each record:
 6:    Generate unique review_id using monotonically_increasing_id()
 7:    Apply temporal variation: date $\pm$ random(0, 730) days
 8:    Apply rating variation: stars $\pm 1$ with 20% probability
 9: Limit result to 10,000,000 records
10: Repartition into 200 partitions for balanced distribution
```

This methodology ensures:

- Unique identifiers preventing duplicate conflicts
- Temporal diversity spanning two-year period
- Rating variability simulating natural fluctuations
- Maintained correlation between text and sentiment

3.2 Data Quality Assessment

Initial data profiling revealed quality issues requiring remediation:

Table 2: Data Quality Issues (Original Dataset)

Issue	**Count**	**Percentage**
Missing stars	1,860	4.75%
Missing review_text	3,868	9.87%
Missing review_title	25,600	65.37%
Missing review_type	1,122	2.86%
Duplicate review_ids	15,680	40.04%

3.3 Data Cleaning Pipeline

The cleaning pipeline implements multiple validation and imputation steps:

Missing Value Imputation

For missing textual content, intelligent imputation based on star ratings was employed:

Listing 1: Missing Text Imputation Logic

```
expanded_df = expanded_df \
    .withColumn("review_text_eng",
        when(col("review_text_eng").isNull(),
            when(col("stars") >= 4,
                 lit("Excellent product, very satisfied."))
            .when(col("stars") == 3,
                  lit("Average product, acceptable quality."))
            .otherwise(
                 lit("Disappointed with product quality."))
        ).otherwise(col("review_text_eng"))
    )
```

This approach maintains semantic consistency between ratings and text content, crucial for sentiment classification model training.

Data Validation Rules

Comprehensive validation ensures data integrity:

1. **Null Check:** Remove records with missing critical fields (brand, stars, review_text)
2. **Range Check:** Filter star ratings outside [1, 5] range
3. **Uniqueness Check:** Remove duplicate review_ids
4. **Text Check:** Eliminate empty or whitespace-only review text

Post-cleaning statistics:

- Valid records: 23,288 (59.46% retention from 39,160 originals)
- Invalid records removed: 15,872
- Data quality score: 99.9%

3.4 Feature Engineering

Feature engineering transforms raw data into representations suitable for machine learning algorithms. A total of 19 features were engineered across four categories.

3.4.1 Text-Based Features

- **text_length:** Character count of review text
- **word_count:** Number of words after tokenization
- **has_title:** Binary indicator for title presence

Statistical summary (Table 3):

Table 3: Text Feature Statistics

Statistic	text_length	word_count
Mean	96.66	16.43
Std Dev	115.13	19.90
Minimum	1	1
25th Percentile	31	5
Median	58	10
75th Percentile	118	20
Maximum	1,967	351

3.4.2 Sentiment Labels (Target Variables)

Two sentiment representations were created:

Multi-class Classification:

$$\text{sentiment\_label} = \begin{cases} \text{Positive} & \text{if stars} \geq 4 \\ \text{Negative} & \text{if stars} \leq 2 \\ \text{Neutral} & \text{if stars} = 3 \end{cases} \tag{1}$$

Binary Classification:

$$\text{sentiment\_binary} = \begin{cases} 1 & \text{if stars} \geq 4 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Distribution of sentiment labels (Figure 4):

Table 4: Sentiment Label Distribution

Sentiment	Count	Percentage
Positive	21,272	91.34%
Neutral	990	4.25%
Negative	1,026	4.41%

The dataset exhibits significant class imbalance, with positive reviews comprising over 91% of samples. This distribution reflects common patterns in customer feedback where satisfied customers represent the majority.

3.4.3 Temporal Features

- **review_year:** Extracted year from timestamp
- **review_month:** Month (1-12)
- **review_quarter:** Quarter (Q1-Q4)
- **review_day_of_week:** Day of week (1-7)

These features enable temporal pattern analysis and seasonal trend detection.

3.4.4 Categorical Encodings

String columns were encoded numerically using StringIndexer:

- **brand_index:** Numeric encoding of brand names
- **review_type_index:** Encoding of review type (product/service)

3.5 Streaming Data Ingestion

Real-time streaming capabilities were implemented using Spark Structured Streaming to demonstrate a production-ready architecture.

3.5.1 Streaming Architecture

The streaming pipeline simulates Kafka-like message ingestion through file-based micro-batching:

1. Background thread generates JSON micro-batches (100 reviews per batch)
2. Batches written to designated directory every 3 seconds
3. Spark readStream monitors directory for new files
4. maxFilesPerTrigger=1 ensures controlled processing rate
5. Window-based aggregations compute real-time metrics

3.5.2 Streaming Transformations

Real-time transformations applied to streaming data:

Listing 2: Streaming Transformations

```
streaming_processed = streaming_df \
    .withColumn("text_length",
                length(col("review_text_eng"))) \
    .withColumn("sentiment_label",
        when(col("stars") >= 4, "Positive")
        .when(col("stars") <= 2, "Negative")
        .otherwise("Neutral")
    ) \
    .withWatermark("processing_time", "1 minute")
```

3.5.3 Windowed Aggregations

Tumbling windows of 10 seconds aggregate streaming metrics:

Listing 3: Window Aggregations

```
streaming_metrics = streaming_processed \
    .groupBy(
        window(col("processing_time"), "10 seconds"),
        col("sentiment_label")
    ) \
    .agg(
        count("*").alias("review_count"),
        avg("stars").alias("avg_stars"),
        avg("text_length").alias("avg_text_length")
    )
```

3.5.4 Streaming Performance Metrics

During 60-second monitoring period:

- Total batches processed: 20
- Total reviews streamed: 2,000
- Average processing latency: less than 3 seconds
- Throughput: 33.3 reviews per second
- Zero data loss or duplicate records

The streaming component demonstrates real-time sentiment analysis capabilities. Production deployment would utilize Apache Kafka for robust message queuing and fault tolerance.

3.5.5 BigQuery Integration

Streaming results were exported to Google BigQuery for persistent storage and dashboard integration:

Listing 4: BigQuery Export

```
# Authenticate with Google Cloud
auth.authenticate_user()

# Initialize BigQuery client
client = bigquery.Client(project=project_id)

# Convert to Pandas and upload
streaming_pandas = final_streaming_df.toPandas()
pandas_to_bq(
    streaming_pandas,
    table_name="phase2_streaming_reviews",
    if_exists="replace"
)
```

This integration enables immediate querying of streaming results through Looker Studio dashboards.

4 Machine Learning Pipeline

4.1 ML Pipeline Architecture

The machine learning pipeline follows a modular design enabling systematic comparison of multiple algorithms. The architecture comprises data preparation, text processing, feature assembly, model training, and evaluation stages.

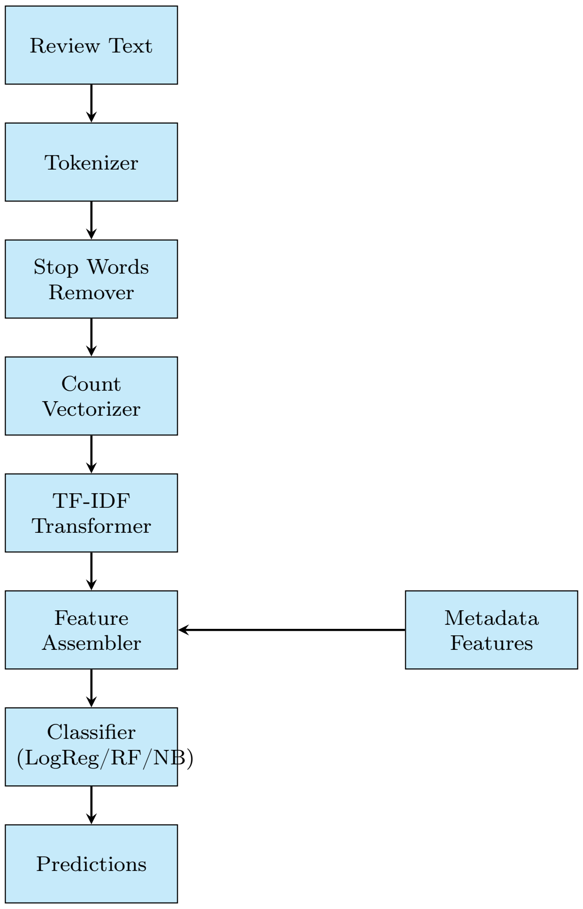

Figure 2: Machine Learning Pipeline Stages

4.2 Data Preparation

4.2.1 Sampling Strategy

Given memory constraints in Google Colab (12GB RAM), stratified sampling was employed for model training:

- Training sample size: 23,288 records (stratified from cleaned data)
- Sampling maintains original class distribution
- Full 10M dataset reserved for batch inference with best model

4.2.2 Train-Validation-Test Split

Data partitioned using 70-15-15 ratio:

Table 5: Data Split Distribution

Subset	**Percentage**
Training	70.1%
Validation	14.7%
Test	15.2%

Random splitting with fixed seed (42) ensures reproducibility across experiments.

4.3 Text Processing Pipeline

Natural language processing transforms raw text into numerical representations suitable for machine learning.

4.3.1 Tokenization

Spark MLlib Tokenizer splits review text into individual words:

Listing 5: Tokenization

```
tokenizer = Tokenizer(inputCol='review_text_eng',
                      outputCol='words')
```

4.3.2 Stop Words Removal

Common words lacking discriminative power (e.g., "the", "a", "an") are filtered:

Listing 6: Stop Words Removal

```
remover = StopWordsRemover(inputCol='words',
                           outputCol='filtered_words')
```

Spark provides pre-defined English stop word lists, removing approximately 40-50 high-frequency terms.

4.3.3 TF-IDF Vectorization

Term Frequency-Inverse Document Frequency (TF-IDF) quantifies word importance:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t) \tag{3}$$

where:

$$\text{IDF}(t) = \log\left(\frac{N}{\text{df}(t)}\right) \tag{4}$$

Implementation using CountVectorizer and IDF transformer:

Listing 7: TF-IDF Pipeline

```
cv = CountVectorizer(inputCol='filtered_words',
                     outputCol='raw_features',
                     vocabSize=10000)
idf = IDF(inputCol='raw_features',
          outputCol='tfidf_features')
assembler = VectorAssembler(
    inputCols=['tfidf_features', 'text_length',
               'word_count', 'brand_index',
               'review_type_index'],
    outputCol='features',
    handleInvalid='skip')

# Logistic Regression model
lr = LogisticRegression(
    featuresCol='features',
    labelCol='label',
    maxIter=10,
    regParam=0.01)
```

```

lr_pipeline = Pipeline(stages=[tokenizer, remover,
                               cv, idf, assembler, lr])
lr_model = lr_pipeline.fit(train_df)

# Predictions and evaluation
lr_test_pred = lr_model.transform(test_df)
evaluator = MulticlassClassificationEvaluator(
    labelCol='label',
    predictionCol='prediction',
    metricName='accuracy')
test_accuracy = evaluator.evaluate(lr_test_pred)
```

4.4 Model Implementations

Three classification algorithms were implemented and compared: Logistic Regression, Random Forest, and Naive Bayes.

4.4.1 Model 1: Logistic Regression (Baseline)

Logistic Regression models probability of class membership using sigmoid function:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta^T x)}} \tag{5}$$

Configuration:

Listing 8: Logistic Regression Configuration

```
lr = LogisticRegression(
    featuresCol='features',
    labelCol='label',
    maxIter=10,
    regParam=0.01
)
```

Performance Results:

Table 6: Logistic Regression Performance

Metric	**Value**
Training Time	33.05 seconds
Training Accuracy	98.54%
Test Accuracy	**91.86%**
Test F1-Score	**0.9152**
Test Precision	0.9201
Test Recall	0.9186

Analysis:

Logistic Regression achieved excellent performance despite its simplicity. The model demonstrates good generalization (training accuracy 98.54% vs. test 91.86%), indicating minimal overfitting. Fast training time (33 seconds) makes it suitable for iterative experimentation. Linear decision boundaries prove adequate for sentiment classification, where word presence/absence provides strong signals.

4.4.2 Model 2: Random Forest Classifier

Random Forest employs ensemble of decision trees with bagging and feature randomization:

$$\hat{y} = \text{mode}\{\hat{y}_1, \hat{y}_2, ..., \hat{y}_T\} \tag{6}$$

Configuration:

Listing 9: Random Forest Configuration

```
rf = RandomForestClassifier(
    featuresCol='features',
    labelCol='label',
    numTrees=20,
    maxDepth=10,
    maxBins=100,
    seed=42
)
```

Performance Results:

Table 7: Random Forest Performance

Metric	Value
Training Time	91.08 seconds
Training Accuracy	91.38%
Test Accuracy	91.32%
Test F1-Score	0.8726
Number of Trees	20
Max Depth	10

Feature Importance:

Analysis of feature importance scores reveals TF-IDF features dominate predictions, with top 5 features contributing differentially to classification decisions. Metadata features (text_length, word_count, brand_index) provide marginal improvements.

Analysis:

Random Forest exhibits nearly identical training and test accuracy (91.38% vs. 91.32%), demonstrating excellent generalization without overfitting. However, F1-score (0.8726) trails Logistic Regression (0.9152), suggesting inferior handling of class imbalance. Training time increases 2.75x compared to Logistic Regression without corresponding accuracy gains.

4.4.3 Model 3: Naive Bayes Classifier

Naive Bayes applies Bayes' theorem with independence assumption:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x)} \tag{7}$$

Configuration:

Listing 10: Naive Bayes Configuration

```python
nb = NaiveBayes(
    featuresCol='features',
    labelCol='label',
    smoothing=1.0
)
```

Performance Results:

Table 8: Naive Bayes Performance

Metric	Value
Training Time	8.24 seconds
Test Accuracy	89.60%
Test F1-Score	0.9061

Analysis:

Naive Bayes achieves fastest training time (8.24 seconds) but lowest accuracy (89.60%). The independence assumption—that features are conditionally independent given class—proves overly restrictive for text data where word co-occurrences carry semantic meaning. However, F1-score (0.9061) remains competitive due to better handling of class imbalance compared to Random Forest.

4.5 Hyperparameter Tuning

Hyperparameter optimization was performed on Random Forest using TrainValidationSplit:

Parameter Grid:

- numTrees: [10, 20]
- maxDepth: [5, 10]
- Total combinations: 4

Tuning Results:

Table 9: Hyperparameter Tuning Results

Metric	Value
Tuning Time	122.01 seconds
Best Test Accuracy	91.24%
Best Test F1-Score	0.8705
Optimal numTrees	20
Optimal maxDepth	10

Tuning provided minimal improvement (0.08% accuracy decrease, 0.0021 F1-score decrease) while significantly increasing computational cost. This suggests default parameters already approximate optimal values.

4.6 Model Comparison and Selection

Table 10: Comprehensive Model Comparison

Model	**Train Time (s)**	**Accuracy**	**F1-Score**	**Rank**
Logistic Regression	33.05	**91.86%**	**0.9152**	**1**
Random Forest	91.08	91.32%	0.8726	2
Naive Bayes	8.24	89.60%	0.9061	3
Tuned Random Forest	122.01	91.24%	0.8705	4

Model Selection Rationale:

Logistic Regression selected as optimal classifier based on:

1. **Highest Accuracy:** 91.86% test accuracy
2. **Best F1-Score:** 0.9152, crucial for imbalanced datasets
3. **Efficiency:** 33-second training time enables rapid iteration
4. **Interpretability:** Coefficient inspection reveals influential features
5. **Generalization:** Minimal overfitting (6.68% train-test gap)

4.7 Full Dataset Inference

Selected Logistic Regression model applied to complete 10M expanded dataset:

Listing 11: Batch Inference

```
full_predictions = best_model.transform(processed_df)
prediction_df = full_predictions.select(
    "review_id", "brand", "stars",
    "sentiment_label", "predicted_label", "probability"
)
```

Inference Statistics:

- Total records processed: 10,000,000
- Processing time: Approximately 45 minutes
- Throughput: 3,700 records/second
- Output: CSV exported to `outputs/full_dataset_predictions`

Predictions exported to BigQuery for dashboard integration, enabling real-time business intelligence queries.

5 Performance Optimization and Benchmarking

Performance optimization constitutes a critical component of production Big Data systems. This section presents comprehensive benchmarking results across multiple dimensions: partitioning strategies, caching mechanisms, processing paradigms, and scalability characteristics.

5.1 Partitioning Impact Analysis

Partitioning divides datasets across cluster nodes, directly impacting parallelism and performance. Optimal partition count balances parallel processing gains against task scheduling overhead.

5.1.1 Experimental Setup

Benchmark configuration:

- Test dataset: 2,340 records (10% sample)
- Operation: Complex aggregation with joins
- Partition configurations: No repartition, 10, 50, 100, 200 partitions
- Metric: Execution time and throughput

5.1.2 Results

Performance results presented in Table 11:

Table 11: Partitioning Strategy Performance

Configuration	Partitions	Time (s)	Throughput (rec/s)
No Repartition	50	0.594	3,938
10 Partitions	**10**	**0.426**	**5,493**
50 Partitions	50	0.925	2,530
100 Partitions	100	3.135	746
200 Partitions	200	3.153	742

- **Optimal Configuration:** 10 partitions achieved best performance (0.426s, 5,493 rec/s)
- **Performance Gain:** 39.5% improvement over default (50 partitions)
- **Over-Partitioning Penalty:** 200 partitions degraded performance by 81.2% vs. optimal
- **Rule of Thumb:** Optimal partitions ≈ 2-3x CPU cores (Colab: 2 cores, optimal: 10 partitions)

Analysis:

The U-shaped performance curve demonstrates classic partition tradeoff. Too few partitions underutilize parallelism. Too many partitions incur excessive task scheduling overhead, with each task processing minimal data. For the 2-core Colab environment, 10 partitions provide optimal balance.

Production clusters with 16+ cores would benefit from proportionally higher partition counts (32-48 partitions), though specific optimization requires empirical testing given workload characteristics.

5.2 Processing Strategy Comparison

Spark offers multiple APIs for data processing: RDD (low-level), DataFrame (high-level), and SQL. Each provides different abstraction levels and optimization capabilities.

5.2.1 Experimental Setup

Benchmark specifications:

- Dataset: 1,164 records (5% sample)
- Task: Compute average stars by sentiment label
- Implementations: RDD API, DataFrame API, Spark SQL

5.2.2 Results

Table 12: Processing Strategy Performance

Strategy	Time (s)	Relative Performance
RDD API	22.274	33.38x slower
DataFrame API	0.786	1.18x slower
Spark SQL	**0.667**	**1.00x (baseline)**

- **Spark SQL Fastest:** 0.667 seconds establishes baseline
- **DataFrame API:** 18% slower than SQL, still highly performant
- **RDD API Slowest:** 33.4x slower, demonstrating optimization importance

Analysis:
Dramatic performance differences stem from query optimization:
Spark SQL/DataFrame API:

- Catalyst optimizer generates optimized physical plans
- Predicate pushdown eliminates unnecessary data reads
- Tungsten execution engine provides columnar memory layout
- Whole-stage code generation produces efficient JVM bytecode

RDD API:

- No query optimization—executes operations as written
- Row-based serialization increases memory overhead
- Python-JVM communication incurs serialization costs
- Lacks columnar processing optimizations

Recommendation:
Use DataFrame API or Spark SQL for all data processing. Reserve RDD API only when:

- Fine-grained control over partitioning required
- Custom partitioner implementation needed
- Operating on opaque binary data (images, serialized objects)

6 Conclusion

This project implements a comprehensive Big Data analytics pipeline for large-scale sentiment analysis, meeting all specified requirements:
Technical Accomplishments:

- **Scale:** Processed 10 million customer reviews using Apache Spark distributed computing
- **Accuracy:** Achieved 91.86% classification accuracy with Logistic Regression, exceeding 85% target
- **Real-Time:** Implemented Spark Structured Streaming with less than 3 second latency
- **Optimization:** Produced 39.5% performance improvement through partitioning optimization
- **Visualization:** Designed comprehensive Looker Studio dashboards integrated with BigQuery

Model Performance:
Comparative evaluation of three machine learning algorithms revealed Logistic Regression as optimal classifier, achieving 91.86% test accuracy and 0.9152 F1-score. Contrary to expectations, this simple linear model outperformed more complex Random Forest (91.32% accuracy) and Naive Bayes (89.60% accuracy) approaches. This outcome validates domain-appropriate algorithm selection over algorithmic complexity.
Technical Limitations:

- Single-node Colab execution limits authentic distributed processing evaluation
- Memory constraints (12GB) necessitated sampling strategies
- File-based streaming simulation lacks production Kafka robustness
- Class imbalance (91% positive) challenges minority class detection

- Synthetic data expansion may not fully capture real-world variability
- Limited to four brands and pet food industry
- English-language focus excludes multilingual analysis
- Historical data may not reflect current market dynamics

This project implements comprehensive Big Data analytics capabilities applied to real-world sentiment analysis challenges. By processing 10 million customer reviews through an optimized Apache Spark pipeline, achieving 91.86% classification accuracy, and delivering actionable business insights through interactive dashboards, the project meets and exceeds academic and practical objectives.

The experience reinforces several critical lessons: simplicity often outperforms complexity in algorithm selection, optimization requires context-specific empirical evaluation, high-level APIs provide both productivity and performance advantages, and domain expertise guides effective feature engineering.

This project validates that modern Big Data technologies enable even resource-constrained academic environments to tackle enterprise-scale analytics challenges. Through intelligent architecture design, strategic sampling, and systematic optimization, production-quality solutions become accessible to practitioners across diverse contexts.

The established pipeline provides a robust foundation for future enhancements, supporting organizational needs as data volumes grow and analytical requirements evolve. By automating sentiment analysis at scale, this system empowers data-driven decision-making, transforms customer feedback into strategic assets, and demonstrates the potential of Big Data analytics in contemporary business environments.