Paul Malet
Brice Convers
IASD - 2025

ML on Big Data Project
K-NN and Naive Bayes

Introduction

This report presents the implementation, analysis, and evaluation of two classical machine learning algorithms, Naive Bayes and k-Nearest Neighbors (KNN), within the Apache Spark ecosystem. The main objective is to translate traditional algorithms into a distributed computing environment using both RDD and DataFrame APIs, thereby enabling scalable data processing. The report covers a detailed understanding of the adopted algorithms, the design of the Spark-based pipelines, and the necessary transformations for distributed execution. Experimental evaluation is conducted on the Nursery dataset from the UCI Machine Learning Repository, which consists of categorical attributes related to family and child characteristics for nursery placement suitability. The study emphasizes the performance, memory usage, and accuracy of both RDD and DataFrame implementations, while also highlighting the advantages and trade-offs of each approach. Key code fragments are included and explained to illustrate the implementation choices, and a comparative analysis provides insights into the scalability and efficiency of Spark's different APIs.

Our full source code and code for the methods comparison is in the repository : https://github.com/LeptaLuma/spark-classification-algorithms

1. k-Nearest Neighbor Implementation

This implementation follows the three-phase MapReduce approach described in "A MapReduce-based k-Nearest Neighbor Approach for Big Data Classification". The solution addresses the scalability challenges of traditional k-NN algorithms by distributing computation across multiple nodes and processing data in parallel.

The algorithm has three phases :

1. Map Phase: Each mapper processes a subset of training data and finds k nearest neighbors for each test instance within that subset
2. Reduce Phase: Combines candidate sets from all mappers and selects the global k nearest neighbors
3. Cleanup Phase: Performs majority voting for final classification

The greatest aspect of this algorithm is that in the first map phase that the train data is partitioned across multiple nodes and is computed in parallel, which makes it also memory efficient since all pairwise distances are not computed, but rather each partition maintains only the k nearest neighbors.

Phase 1: Map Operation

```
def map_function(train_partition, test_data, k):
    """Map function following Algorithm 1"""
       #  For  each  test  instance,  compute  k-NN  against  this  training
partition
    for test_point in test_data:
        distances = distances between test_point and each train_point

        # Get k nearest neighbors from this partition
        k_nearest = heapq.nsmallest(k, distances, key=lambda x: x[0])

        # Create candidate set CDj for this partition
        cd_j = [(train_label, dist) for dist, train_label in k_nearest]

    # Emit candidate sets with test_idx as key and partition results
```

The Map phase processes each training partition independently. For every test instance, it:

1. Calculates Euclidean distances to all training points in the current partition
2. Selects the k nearest neighbors from this partition, with `heapq`
3. Creates a candidate set (CD_j) containing these k neighbors with their distances and labels

To get the k nearest neighbors, we use the `heapq` method to be more efficient because Python enables this, instead of a double loop like described in the paper.

Phase 2: Reduce Operation

```
def reduce_operation(cd_list, k):
    """Reduce operation following Algorithm 2"""
    # Merge all candidate sets and keep k best neighbors globally
    all_candidates = []
    for cd_j in cd_list:
        all_candidates.extend(cd_j)
    k_nearest = heapq.nsmallest(k, all_candidates, key=lambda x: x[1])
```

The Reduce phase aggregates results from all partitions:

1. Collects all candidate sets (CD_j) from different partitions for each test instance
2. Merges these candidates into a single list
3. Selects the globally k nearest neighbors by distance (again with `heapq`)

Phase 3: Cleanup Operation

```
def majority_voting(neighbors):
    """Cleanup process following Algorithm 3"""
    classes = [label for label, _ in neighbors]
    return Counter(classes).most_common(1)[0][0]
```

The Cleanup phase performs classification:

1. Extracts class labels from the k nearest neighbors
2. Applies majority voting to determine the predicted class
3. Returns the most frequent class label

DataFrame Implementation

We used a Cross Join for distance computation. The aggregate function with arrays_zip enables element-wise operations on feature vectors, computing Euclidean distance in a distributed manner.

```
cross_joined = test_df.crossJoin(train_df)
with_distances = cross_joined.withColumn(
    "distance",
    F.sqrt(F.aggregate(
        F.arrays_zip("test_features", "train_features"),
        F.lit(0.0),
             lambda  acc,  x:  acc  +  F.pow(x.getField("test_features")  -
x.getField("train_features"), 2)
    ))
)
```

We used the Window functions to replicate the MapReduce logic. The partition window mimics the Map phase by ranking within partitions, while the global window implements the Reduce phase.

```
# PHASE 1: MAP - Get k-NN per partition
partition_window = (
    Window
    .partitionBy("id_test", "partition_id")
    .orderBy("distance")
)
map_output = (
    With_distances
    .withColumn(
        "rank_in_partition",
        F.row_number().over(partition_window)
    )
    .filter(F.col("rank_in_partition") <= k)
)

# PHASE 2: REDUCE - Global k nearest selection
global_window = Window.partitionBy("id_test").orderBy("distance")
reduce_output = (
    Map_output
    .withColumn("global_rank", F.row_number().over(global_window))
    .filter(F.col("global_rank") <= k)
)
```

The Majority voting is also done with the Window function.

2. Naïve Bayes Implementation

The Naive Bayes classifier was implemented following the MapReduce-inspired approach described in the paper “Naïve Bayes Classifier: A MapReduce Approach” by Songtao Zheng. This implementation leverages the distributed computation capabilities of Apache Spark to efficiently handle categorical data in both RDD and DataFrame formats. The method consists of four primary phases: counting class occurrences, estimating parameters, calculating probabilities, and classifying new instances. The following sections detail each phase with reference to the actual code implementation.

The advantage of this distributed approach is that **all counts are computed in parallel**. Each partition processes a subset of the data independently, and Spark’s aggregation operators combine results globally.

Phase 1: Counting Class Occurrences

In the first step, the algorithm computes the total number of training instances as well as the frequency of each class. For the RDD implementation, the training data is parallelized and the class counts are computed using a `map` and `reduceByKey` approach:

```
# Phase 1: Count total instances and class frequencies
train_rdd = sc.parallelize(train_data)
# Get total count and class counts
total_count = train_rdd.count()
class_counts = (
    train_rdd.map(lambda x: (x[1], 1))
    .reduceByKey(lambda a, b: a + b)
    .collectAsMap()
)
```

This phase provides the prior probabilities for each class, which are essential for the Naive Bayes calculation.

Phase 2: Parameter Estimation

Next, the algorithm estimates the conditional probabilities of each attribute value given the class. The `map_parameters` function transforms each training instance into key-value pairs of the form `(attribute_index-attribute_value-class_label, 1)`. These pairs are then aggregated using `reduceByKey` to count occurrences:

```
parameter_counts = (
    train_rdd.flatMap(map_parameters)
    .reduceByKey(lambda a, b: a + b)
    .collectAsMap()
)
```

```
def map_parameters(record):
   features, class_label = record
   results = []
   for attr_idx, attr_value in enumerate(features):
       key = f"attr-{attr_idx}-{attr_value}-{class_label}"
       results.append((key, 1))
   return results
```

This MapReduce-style operation ensures that attribute counts are computed in parallel, which significantly improves scalability compared to single-node computation.

Phase 3: Calculating Probabilities

Once the counts are available, the algorithm calculates the prior probabilities for each class and the likelihoods for each attribute value given a class. Smoothing is applied to handle zero-frequency issues, ensuring that unseen attribute-class combinations do not result in zero probabilities:

```
# Phase 3: Calculate probabilities - NOT DISTRIBUTED
priors = calculate_priors(class_counts, total_count)
likelihoods = calculate_likelihoods(parameter_counts, class_counts)
```

```
def calculate_priors(class_counts, total_count):
   return {cls: count / total_count for cls, count in
class_counts.items()}
```

```
def calculate_likelihoods(parameter_counts, class_counts):
   likelihoods = defaultdict(dict)
   for key, count in parameter_counts.items():
       parts = key.split('-')
       attr_idx, attr_value, class_label = parts[1], parts[2], parts[3]
       # Add +1 to handle zero frequency classes
       likelihood = (count + 1) / (class_counts[class_label] + 1)
       attr_key = f"attr-{attr_idx}-{attr_value}"
       likelihoods[class_label][attr_key] = likelihood
   return likelihoods
```

Here, `calculate_priors` divides each class count by the total number of instances, and `calculate_likelihoods` implements Laplace smoothing to avoid zero probabilities.

Phase 4: Classification

For each test instance, the algorithm computes the logarithm of posterior probabilities for every class. The predicted class is the one with the highest posterior score. Logarithms are used to prevent numerical underflow when multiplying many small probabilities:

```
def classify_instance(features, priors, likelihoods, class_counts):
   class_scores = {}
   for class_label in priors:
       # Start with log prior
       log_score = math.log(priors[class_label])
       # Add log likelihoods
       for attr_idx, attr_value in enumerate(features):
           attr_key = f"attr-{attr_idx}-{attr_value}"
           if attr_key in likelihoods[class_label]:
               log_score += math.log(likelihoods[class_label][attr_key])
           else:
               # Handle unseen data
               log_score += math.log(1 / (class_counts[class_label] + 1))
       class_scores[class_label] = log_score
   return max(class_scores.items(), key=lambda x: x[1])[0]
```

The RDD version applies this function to all test instances using `map`, while the DataFrame version uses a Spark UDF to apply the same logic in a distributed manner:
predictions_df = test_df.withColumn("predicted_class", classify_spark_udf(F.col("features")))

```
def classify_udf(features):
    priors = broadcast_priors.value
    likelihoods = broadcast_likelihoods.value
    class_counts = broadcast_class_counts.value
    class_scores = {}
    for class_label in priors:
        # Start with log prior (matches RDD)
        log_score = math.log(priors[class_label])
        # Add log likelihoods (matches RDD)
        for attr_idx, attr_value in enumerate(features):
            attr_key = f"attr-{attr_idx}-{attr_value}"
            if class_label in likelihoods and attr_key in
likelihoods[class_label]:
                log_score += math.log(likelihoods[class_label][attr_key])
            else:
                # Handle unseen data exactly like RDD implementation
                log_score += math.log(1 / (class_counts[class_label] + 1))
        class_scores[class_label] = log_score
    return max(class_scores.items(), key=lambda x: x[1])[0]
```

Both implementations follow the same logical steps, but differ in how data is represented and processed. The RDD approach directly operates on tuples of features and labels, applying transformations and aggregations using Spark’s low-level API. The DataFrame implementation leverages Spark SQL functions and higher-level APIs, including `groupBy`, `posexplode`, and `withColumn`, to achieve the same results in a more declarative and

SQL-like style. Broadcast variables are used in the DataFrame version to efficiently distribute model parameters across worker nodes during classification.

3. Results and Analysis

In this section, we are going to describe the test results of the both algorithms through the Nursery dataset[1]. This dataset is provided by the UCI Machine Learning Repository and was originally developed to assist in the placement of children in nursery schools. It contains categorical attributes that describe both the family's situation and the child's needs. The dataset includes eight descriptive features: parents (quality of parental background), has_nurs (the importance of nursery school), form (the structure of the family), children (the number of children), housing (living conditions), finance (financial status), social (social conditions), and health (health conditions).

The target variable is class, which indicates the suitability of the nursery placement, categorized into five possible outcomes (not_recom, recommend, very_recom, priority, and spec_prior).

The dataset consisted of 9,072 training samples and 3,888 test samples. The class distribution in the training set was moderately imbalanced, with three dominant classes (*not_recom*, *priority*, *spec_prior*) each representing around one-third of the data, and one minority class (*very_recom*) accounting for only 2.5%. This imbalance is important to keep in mind, as it can influence classification performance.

3.1. Naive Bayes Performance

Two implementations of Naive Bayes were tested: one using the RDD and one using the DataFrame. Both achieved an identical accuracy of 90.3%, indicating that the choice of API did not affect predictive performance.

For the RDD Implementation:
- Time: 3.39s
- Memory: 1.2 MB

For the DataFrame Implementation:
- Time: 4.96s
- Memory: 2.5 MB

The RDD version demonstrated clear advantages in execution time (≈31% faster) and memory consumption (≈50% less), making it more efficient for this specific workload.

Despite the class imbalance, Naive Bayes performed strongly overall, suggesting that the feature representation captured enough signal to separate the majority classes effectively.

These results also highlight the RDD API's lower overhead compared to the DataFrame API for relatively small datasets. On much larger datasets, DataFrames often become more advantageous due to query optimization, even if they appear slower here. The DataFrame

[1] The website link of the Nursery dataset is: https://archive.ics.uci.edu/dataset/76/nursery

API is often preferred in practice due to its ease of use, better integration with Spark SQL, and potential optimizations for larger-scale data.

3.2. Analysis of Results for KNN

The experiments with the K-Nearest Neighbors (KNN) algorithm show clear differences between the RDD and DataFrame implementations in terms of execution time, memory usage, and accuracy.

For the RDD Implementation:

- Execution Time: 9.34s
- Memory Usage: 0.9 MB
- Accuracy: 94.57%

For the DataFrame Implementation:

- Execution Time: 13.97s
- Memory Usage: 2.5 MB
- Accuracy: 93.70%

The RDD implementation outperformed the DataFrame approach across all metrics. It achieved 33% faster execution time, 64% lower memory usage, and slightly higher accuracy (+0.87%). The superior performance of RDDs for KNN can be attributed to the algorithm's inherently simple computational pattern that benefits from the low-level RDD operations without requiring the overhead of DataFrame optimizations.

3.3. Comparative Analysis: Naive Bayes vs K-Nearest Neighbors

The KNN outperformed Naive Bayes with 94.57% vs 90.3% accuracy on the nursery dataset. However, Naive Bayes executed 2.75 times faster than KNN (3.39s vs 9.34s for RDD implementations). This reflects Naive Bayes' O(n) prediction complexity compared to KNN's O(n×k) distance calculations across the entire training set. Memory usage was comparable between algorithms, with both RDD implementations using under 1.2 MB.

About the differences between RDD and DataFrame implementations, both algorithms showed identical patterns. The RDD implementations were faster and used less memory than DataFrame versions for both of them. However, KNN exhibited accuracy variation between APIs (94.57% vs 93.70%), while Naive Bayes maintained identical accuracy across both implementations. This suggests KNN is more sensitive to Spark's execution optimizations.

The results reveal a clear speed-accuracy trade-off. Naive Bayes provides 90% of KNN's accuracy in one-third of the execution time, making it optimal for time-sensitive applications. KNN's 4% accuracy gain comes at nearly triple the computational cost, justified only when maximum classification performance is required.

3.4. Limitations

This work has some limitations that constrain the generalizability of findings. First of all, there is a limit in terms of dataset scale and diversity. Evaluation was limited to a single, relatively small categorical dataset (12,960 instances). Performance characteristics may differ significantly on larger datasets or those with numerical features, where Spark's distributed advantages become more apparent. Another limit is that the experimental process was not rigorous enough, performance metrics are based on single runs without statistical validation.

Some additional work needs to be done to be able to generalize those findings with more confidence.

Conclusion

This work demonstrates the feasibility and effectiveness of implementing classical machine learning algorithms in a distributed environment using Apache Spark. Both Naive Bayes and KNN were successfully translated into RDD and DataFrame-based pipelines, achieving high accuracy on the Nursery dataset. The comparative analysis shows that RDD implementations generally offer faster execution and lower memory usage, while DataFrames provide a more expressive and higher-level interface that can simplify code maintenance. Experimental results highlight the trade-offs between performance and usability, emphasizing the importance of choosing the appropriate Spark API based on dataset characteristics and computational requirements. Overall, this report illustrates the potential of Spark for scaling traditional algorithms to large datasets.

References

Zheng, S. (Year). *Naïve Bayes classifier: A MapReduce approach* (Master’s thesis). North Dakota State University of Agriculture and Applied Science. Retrieved from https://library.ndsu.edu/ir/bitstream/handle/10365/31406/NaiveBayesMapReduce.pdf

Jesús Maillo, Isaac Triguero, and Francisco Herrera. 2015. *A MapReduce-Based k-Nearest Neighbor Approach for Big Data Classification*. In Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 02 (TRUSTCOM-BIGDATASE-ISPA '15). IEEE Computer Society, USA, 167–172. https://doi.org/10.1109/Trustcom.2015.577

Dua, D., & Graff, C. (2019). *UCI Machine Learning Repository: Nursery Data Set*. University of California, Irvine, School of Information and Computer Sciences. Retrieved from https://archive.ics.uci.edu/ml/datasets/nursery