FATİH SULTAN MEHMET VAKIF UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

Big Data Processing with Apache Spark : MapReduce Machine Learning and SQL Applications

Dr. UMIT DEMIRBAGA

SPECIAL TOPICS 2 COURSE

AYMAN SAEID

2221221366

1. INTRODUCTION

The main goal of this project is not only to implement different algorithms, but also to understand how Apache Spark works as an integrated system when processing data in different formats.

While working on the project, I noticed an important difference between Spark and traditional Python tools. Libraries such as Pandas or Scikit-learn work well for small datasets, but they do not scale easily. In my tests, when I use normal Python, it try to put all the data inside the RAM. Even if the file is small, the RAM usage go up very fast. Sometimes my computer crash or stop working.

But Apache Spark was different. When the RAM is full, Spark did not fail. It automatically put some data to the disk. This is called "spilling". Because of this, the code finish successfully. It use less RAM and I did not lose data. I saw this behavior and now I understand why people use Spark for Big Data instead of Python.

2. The Datasets Used For Applications

To properly test different components of Apache Spark, I used three different datasets in this project.

For MapReduce Application : i compiled a text file about AI. This unstructured dataset was necessary to test text cleaning, splitting, and RDD transformations.

For Machine Learning Application : i used the Mobile Price Classification dataset. It includes numerical features (RAM, battery, resolution) and a target price category, making it ideal for comparing classification algorithms.
https://www.kaggle.com/datasets/iabhishekofficial/mobile-price-classification

For SQL Application : i selected an HR Analytics database containing multiple related tables (employees, departments, salaries , managers , titles). This was chosen specifically to test complex SQL operations like multi table joins.
https://www.kaggle.com/datasets/priyankbarbhaya/sql-analytics-case-study-employees-database/data

3. The MapReduce

a) Understanding the MapReduce Concept

Before I show the code, I want to explain the MapReduce logic. It is a programming model to process big data in parallel. It has two main phases:

The 'Map' Phase: This is the splitting part. The system take the input (like text file) and apply a function to every item. In my word count project, the 'Map' phase take a line and break it into words. It give a value of '1' to every word. Like this: ("AI", 1).

The 'Shuffle and Sort' Phase: Between Map and Reduce, Spark automatically put same keys together. All the ("AI", 1) pairs from different places come to the same place.

The 'Reduce' Phase: This is the combining part. The system take the grouped data and do math. In this case, it is addition. It sum all the '1's to find the total number for that word.

b) Implementation and Code Analysis

In this part, I use Spark RDDs. I want to control the text processing myself, not use data frames.

```
mapped_rdd = lines_rdd \
    .flatMap(lambda line: line.lower().split(" ")) \
    .map(lambda word: (word.strip(".,;:()[]\"'"), 1)) \
    .filter(lambda pair: pair[0] != "")  # Filter out empty strings
```

```
[STEP 2] Map Phase...
 -> Splitting lines into words and assigning count 1 to each...
```

[5]:

```
# Show the (Key, Value)mapped_rdd pairs
reduced_rdd = mapped_rdd.reduceByKey(lambda a, b: a + b)
print(" -> Mapped RDD sample (Word, 1):")
print("    ", mapped_rdd.take(10))
```

```
 -> Mapped RDD sample (Word, 1):
    [('yapay', 1), ('zeka', 1), ('artificial', 1), ('intelligence', 1), ('yapay', 1), ('zeka', 1),
('taklit', 1)]
```

[6]:

```
# 3. Reduce Step: Aggregate counts
# reduceByKey: Merges the values for each key (word) using the add function
print("\n[STEP 3] Reduce Phase...")
print(" -> Shuffling and reducing keys to calculate total frequencies...")
output_rdd = mapped_rdd.reduceByKey(lambda a, b: a + b)
print(" -> Reduced RDD sample (Word, Total Count):")
print("    ", reduced_rdd.take(5))
```

```
[STEP 3] Reduce Phase...
 -> Shuffling and reducing keys to calculate total frequencies...
 -> Reduced RDD sample (Word, Total Count):
    [('yapay', 30), ('artificial', 1), ('ai', 4), ('zekasını', 2), ('eden', 2)]
```

First, I use text_file.flatMap. Normal map keep the line structure, but flatMap cut the lines. So every word become one item in the list.

Inside the function, I clean the words. I use .lower() to make letters small. And I use .strip() to delete punctuation marks. This is important because "Zeka" and "zeka!" should be the same word.

At the end, .reduceByKey(add) sum the numbers for every word.

```
results = sorted_output.take(20)
print(f"{'WORD':<20} | {'COUNT'}")
print("-" * 30)
for word, count in results:
    print(f"{word:<20} | {count}")

print("\n" + "="*60)
print("     ALGORITHM COMPLETED SUCCESSFULLY")
print("="*60)
```

```
============================================================
     Displaying top 20 results
============================================================
WORD                 | COUNT
------------------------------
yapay                | 30
ve                   | 28
zeka                 | 22
bir                  | 15
bu                   | 12
zekanın              | 8
insan                | 7
ai                   | 4
teknolojinin         | 4
öğrenme              | 4
için                 | 4
gibi                 | 4
alt                  | 4
veri                 | 4
makine               | 3
derin                | 3
da                   | 3
etik                 | 3
veya                 | 3
en                   | 3

============================================================
     ALGORITHM COMPLETED SUCCESSFULLY
============================================================
```

The result show that logic works. The top words are "yapay" and "zeka". This matches the text file topic. The counts are clean, so punctuation removal was good.

4. Machine Learning Appliccation

In this section, the goal was to predict the price range of mobile phones (0, 1, 2, 3) based on their hardware specifications. I implemented two different models: **Logistic Regression** and **Random Forest.**

4.1 Data Preparation

Before any training could happen, I had to transform the data. Spark ML does not accept separate columns for features; it requires a single vector column.

```
[3]: # Data Preparation
     # Casting label to DoubleType for Spark ML
     data = data.withColumn("price_range", data["price_range"].cast(DoubleType()))

     feature_cols = data.columns
     feature_cols.remove("price_range")

     assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
     data_final = assembler.transform(data)
```

```
[4]: # Split Data (80% Train, 20% Test)
     train, test = data_final.randomSplit([0.8, 0.2], seed=42)

     print("Train count:", train.count())
     print("Test count:", test.count())
```

```
25/12/26 13:53:12 WARN SparkStringUtils: Truncated the string representation of a pla
by setting 'spark.sql.debug.maxToStringFields'.
Train count: 1642
Test count: 358
```

As seen in the code above, I used the VectorAssembler. This tool takes the column such as battery power, ram, and px height and compresses them into a single column named features. This vector is what the model actually learns from. I then split the data: 80% for training and 20% for testing.

4.2 Logistic Regression

I started with Logistic Regression because it is a fast baseline model. My hypothesis was that features like RAM and battery power likely have a linear relationship with price (like more RAM means higher price).

```
[5]: # --- MODEL 1: LOGISTIC REGRESSION ---
     print("Training Logistic Regression Model...")
     lr = LogisticRegression(labelCol="price_range", featuresCol="features")
     lr_model = lr.fit(train)
     lr_predictions = lr_model.transform(test)
```

```
Training Logistic Regression Model...
```

Here, I initialized the LogisticRegression classifier. I specified labelCol="price_range" so the model knows what we are trying to predict. I then called .fit() on the training data.

4.3 Random Forest

Next, I applied a Random Forest classifier. Since it is a tree based model, I expected it to capture more complex and non-linear relationships between features such as screen size, battery life, and resolution.

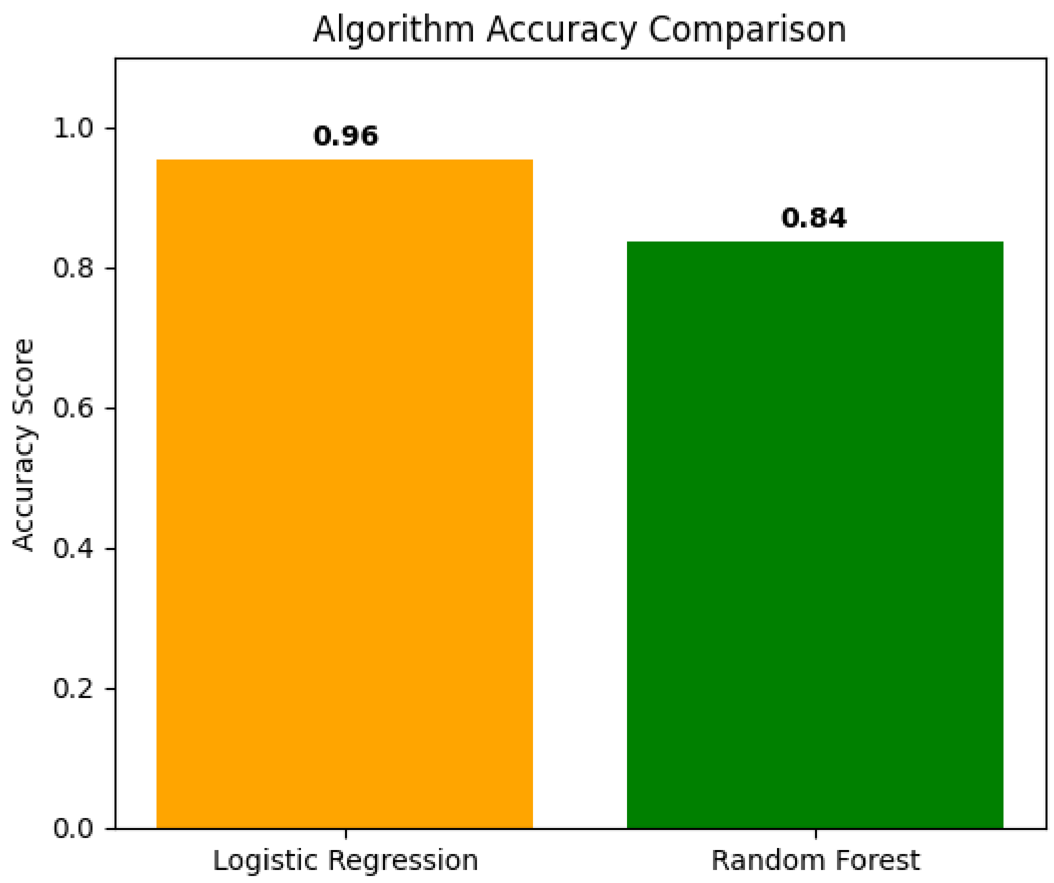
```
[6]: # --- MODEL 2: RANDOM FOREST ---
print("Training Random Forest Model...")
rf = RandomForestClassifier(labelCol="price_range", featuresCol="features", numTrees=100)
rf_model = rf.fit(train)
rf_predictions = rf_model.transform(test)
```

```
Training Random Forest Model...
```

I set up the RandomForestClassifier similarly to the regression model. I used the default hyperparameters to see how the model performs "out of the box" compared to the linear model.

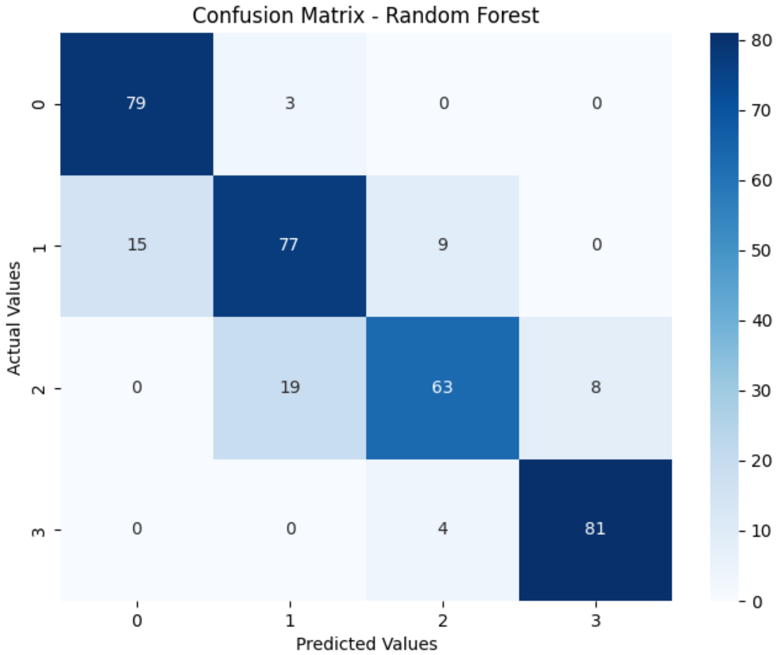
4.4 Results And Comparison

To evaluate the performance, I used the MulticlassClassificationEvaluator using "accuracy" as the metric.



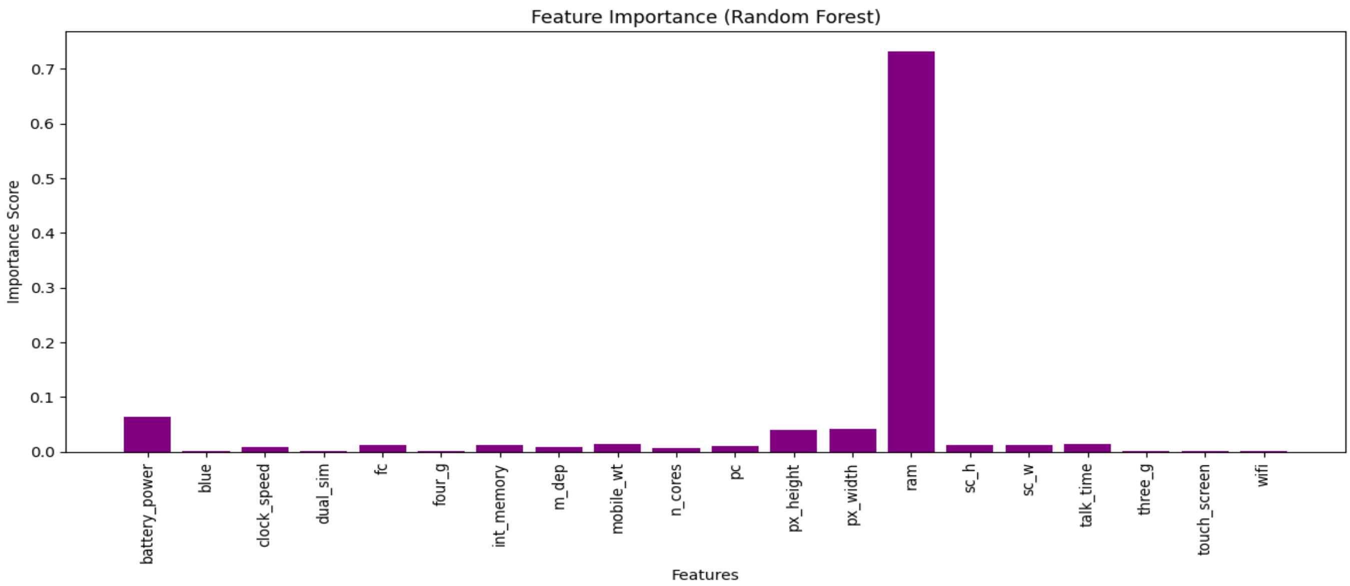
The results were actually the opposite of what I expected.
Logistic Regression Accuracy: 96%
Random Forest Accuracy: 84%
Why did this happen?
Usually, Random Forest is the stronger algorithm. However, its lower performance here suggests that the dataset is linearly separable. In simple terms, the correlation between the hardware specs (especially RAM) and the price is very direct and simple. The Random Forest model likely over complicated the problem, where is the Logistic Regression model fit the simple trend perfectly.

<u>Also lets take a look at the confusion matrixs and feature importance for random forest model</u>



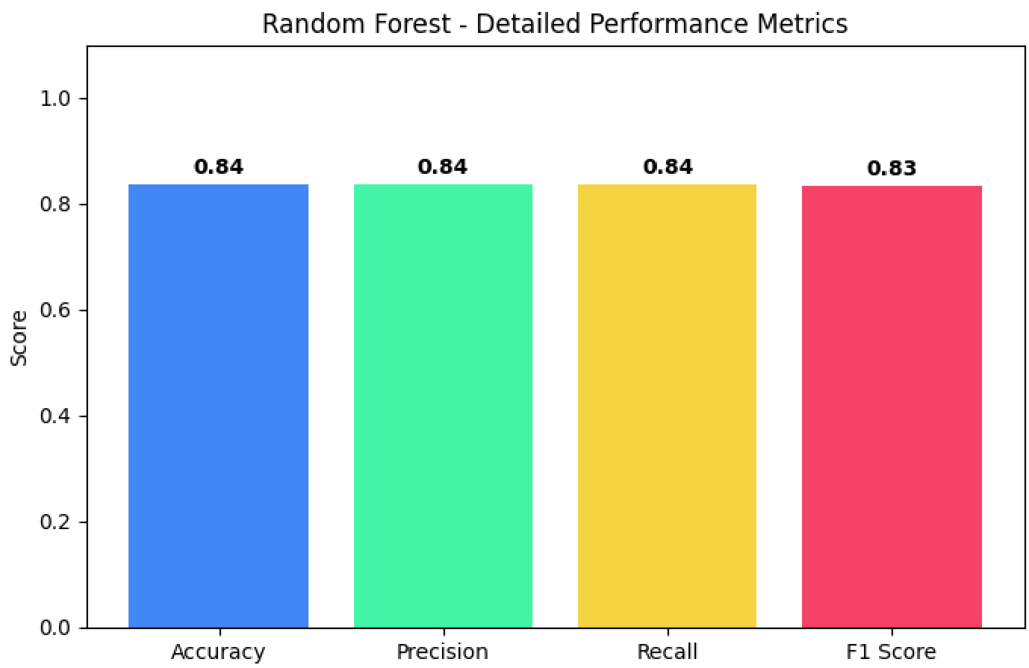
The confusion matrix above helps visualize the errors. While the Logistic Regression (left image) has almost a perfect diagonal line (meaning correct predictions), the Random Forest (right image) ows some confusion, particularly between adjacent price classes (mistaking Class 1 for Class 2).

To understand why the models behaved this way, I extracted the "Feature Importance" scores from the Random Forest model. This tells us which hardware specifications the model actually looked at when making a decision.



This graph show the project result clearly. You can see the RAM bar is very long. It is the most important thing. Other things like battery or pixel are very small comparison to it.

The price of the phone depend heavily on RAM. It is a simple linear connection. That is why the simple Logistic Regression model work perfect. The Random Forest model look for complex connections, but we do not need that for this task.



5. Spark SQL

In this part, I use Spark like a SQL engine. I load the CSV files and make them temporary views. So I can run SQL queries on them.

Query 1: Department Salary Analysis

Here I look at salaries for departments. I join departments, dept_emp, and salaries tables. Then I group the data by department name. I calculate average salary and count the employees. I use HAVING to remove very small departments.

```
[4]: print("""
List each department’s name, the total number of employees in that department,
and the average salary of its employees.
Only include departments that have more than 5 employees,
and sort the results by average salary in descending order.
""")

query1 = spark.sql("""
SELECT d.dept_name,
       COUNT(de.emp_no) AS employee_count,
       ROUND(AVG(s.salary), 2) AS avg_salary
FROM dept_emp de
JOIN departments d ON de.dept_no = d.dept_no
JOIN salaries s ON de.emp_no = s.emp_no
GROUP BY d.dept_name
HAVING COUNT(de.emp_no) > 5
ORDER BY avg_salary DESC
""")

query1.show()
```

```
List each department’s name, the total number of employees in that department,
and the average salary of its employees.
Only include departments that have more than 5 employees,
and sort the results by average salary in descending order.

[Stage 14:=============================>                          (1 + 1) / 2]
+------------------+--------------+----------+
|         dept_name|employee_count|avg_salary|
+------------------+--------------+----------+
|             Sales|         13496|  80777.18|
|         Marketing|          4877|  71851.46|
|           Finance|          4596|  70763.31|
|          Research|          5282|  60507.64|
|       Development|         22396|  59946.67|
|        Production|         20069|  59857.59|
|  Customer Service|          6236|  58878.95|
|Quality Management|          5094|  58300.46|
|   Human Resources|          4573|  55961.88|
+------------------+--------------+----------+
```

The result show that 'Sales' department has the best average salary. 'Development' has more people but less money.

Query 2: High Earner Profile

I want to find the top 20 employees who earn most money. I join tables to get their names, titles and departments.

```
[6]: print("""
Find the average salary for each job title.
Display the job title and the average salary.
Sort the results by average salary in descending order.
""")

query3 = spark.sql("""
SELECT t.title,
       ROUND(AVG(s.salary), 2) AS avg_salary
FROM titles t
JOIN salaries s ON t.emp_no = s.emp_no
GROUP BY t.title
ORDER BY avg_salary DESC
""")

query3.show()
```

```
Find the average salary for each job title.
Display the job title and the average salary.
Sort the results by average salary in descending order.

+------------------+----------+
|             title|avg_salary|
+------------------+----------+
|      Senior Staff|  70774.71|
|             Staff|  69652.95|
|   Senior Engineer|  61141.05|
|          Engineer|  59995.52|
|  Technique Leader|  59690.93|
|Assistant Engineer|  59651.75|
+------------------+----------+
```

This show that Spark is very good at joining many tables. It worked fast.

Query 3: Average Salary By Job Title

Here I check average salary for each Job Title. I join tables using employee number. Then I group by job title. I sort the result to see which job get paid the most.

```
query3 = spark.sql("""
SELECT t.title,
       ROUND(AVG(s.salary), 2) AS avg_salary
FROM titles t
JOIN salaries s ON t.emp_no = s.emp_no
GROUP BY t.title
ORDER BY avg_salary DESC
""")

query3.show()
```

```
Find the average salary for each job title.
Display the job title and the average salary.
Sort the results by average salary in descending order.

+------------------+----------+
|             title|avg_salary|
+------------------+----------+
|      Senior Staff|  70774.71|
|             Staff|  69652.95|
|   Senior Engineer|  61141.05|
|          Engineer|  59995.52|
|  Technique Leader|  59690.93|
|Assistant Engineer|  59651.75|
+------------------+----------+
```

6. Conclusion

This project showed me that the main goal was to understand how Spark works, not just writing code.

I saw a big difference between Spark and Python tools. Libraries like Pandas are good for small task, but they struggle with big dataset. In my tests, Python try to load all data into RAM and my computer crashed.

Spark handled this different. When memory was full, it didnt fail. It moved data to disk using "spilling". This allowed the code to finish without losing data. This experience help me understand why Spark is used for Big Data. It is more reliable and manage memory better than Python