# Analyzing Data with Spark

*Prepared by*

**Fares Ahmad, Shehazi Manar**


Master M2 MOSIG – Univ. Grenoble Alpes (Ensimag  UFR IM2AG)

January 2025

# Contents

# Chapter 1

# Spark Project

## 1.1 Introduction

This course aimed to help us realize more about the efficient management of the large-scale distributed systems and its importance in the modern computing infrastructure particularly in high-demand environments such as cloud computing and data centers. The project focuses on analyzing a large-scale dataset using Apache Spark. The Google Cluster Data 2011 dataset provides a unique opportunity to analyze real-world operations in a cluster environment, capturing 29 days of activity across approximately 12,500 machines.

The primary objective of this report is to illustrate more on the analysis of this dataset ad answering of the provided questions along with expanding more on our two proposed questions to explore further insights.

The report is structured into four main sections: an overview of the dataset, detailed answers to the pre-defined questions, an exploration of the two new questions proposed, and a discussion of the challenges faced during the analysis.

## 1.2 Dataset Overview

The dataset is structured as a collection of interrelated tables, each capturing specific aspects of a large-scale cluster's operations. These tables are stored as multiple CSV files, allowing efficient access and analysis. Each table includes several fields, providing details about different entities such as jobs, tasks, machines, and their resource usage.

The tables are linked by common identifiers, such as job IDs and task indices, enabling the integration of information across the dataset for comprehensive analyses.

## 1.3 Analysis of Provided Questions

### 1.3.1 Question 1:

***What is the distribution of the machines according to their CPU capacity?***

To solve this question, we began by filtering the data to remove rows where `cpu_capacity` was null. These rows do not contribute to a meaningful analysis as they represent incomplete or missing information. After filtering, we grouped the machines by their `cpu_capacity` and counted the number of machines in each group using the `.groupBy()` and `.count()` functions in Spark.

The figure below illustrates the results obtained. Our key findings were:

- The majority of machines have a normalized CPU capacity of 0.5.
- A significantly smaller number of machines have a normalized CPU capacity of 0.25 or 1.0.

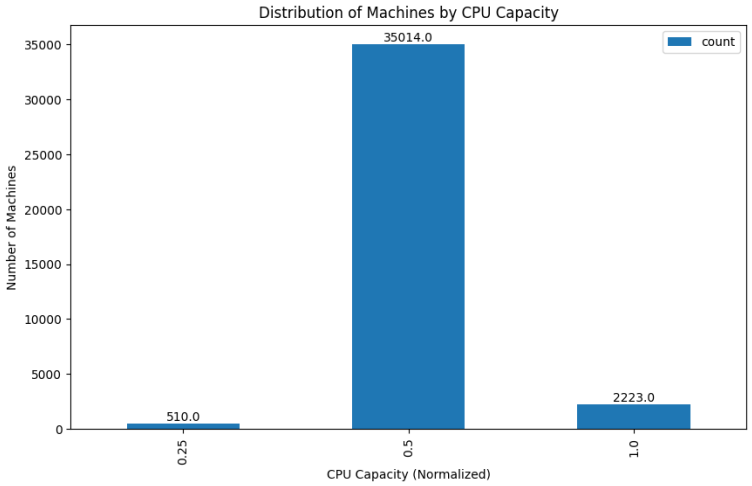


FIGURE 1.1

In conclusion, while Pandas is more suitable for smaller datasets and exploratory analysis, Spark is the preferred choice for large-scale data processing due to its scalability and distributed architecture.

### 1.3.2 Question 2:

***What is the percentage of computational power lost due to maintenance?***

To solve this question, we computed the percentage of computational power lost when machines went offline for maintenance. We used the `event_type` field to identify machines in different states:

- `event_type = 0`: Machines added to the cluster.
- `event_type = 1`: Machines removed from the cluster.
- `event_type = 2`: Machines temporarily offline for maintenance.

**Key Steps in the Analysis:**

1. First, we filtered the dataset to exclude machines that were removed from the cluster (`event_type = 1`), as they do not contribute to the total available CPU capacity. We thought that it would be not logical if we include the removed machines' cpu capacity in the total capacity as they do not really contribute to it.
2. Next, we calculated the total CPU capacity by summing the CPU capacity of all valid machines (`event_type = 0 or 2`).
3. Then, we calculated the offline CPU capacity by summing the CPU capacity of machines that were temporarily offline (`event_type = 2`). For this, we used the `.agg()` function combined with `sum()` to perform the aggregation.
4. Finally, we computed the percentage of computational power lost due to maintenance as:

$$\text{Percentage Lost} = \frac{\text{Offline CPU Capacity}}{\text{Total CPU Capacity}} \times 100$$

**Key Findings:**

- Approximately 24.76% of the computational power was lost due to maintenance during the analyzed period.
- The remaining 75.2% of the CPU capacity was available for active use.

The figure below provides a visual representation of the distribution of offline and available CPU capacity.

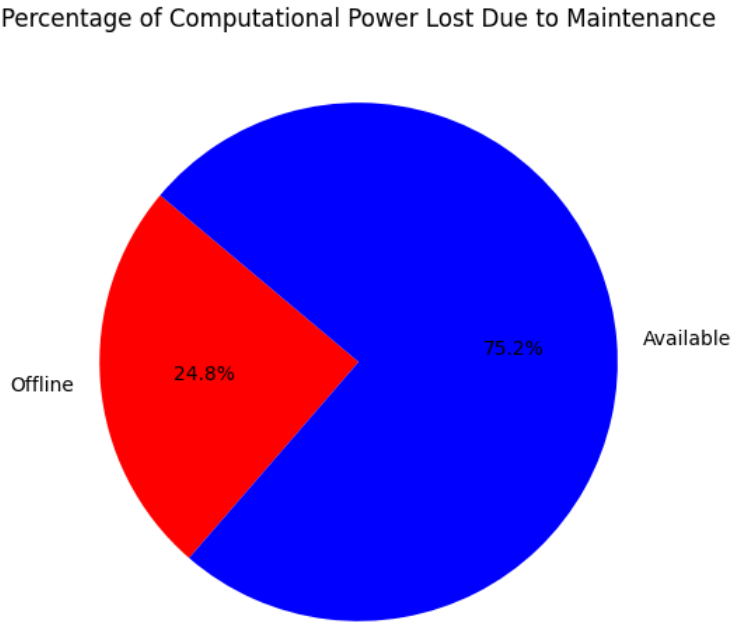


FIGURE 1.2: Percentage of Computational Power Lost Due to Maintenance

### 1.3.3 Question 3:

***What is the distribution of the number of jobs/tasks per scheduling class?***

To address this question, we analyzed the `job_events` and `task_events` datasets. Both datasets include a column named `scheduling_class`, which classifies the priority of jobs/tasks for resource allocation. The steps taken to solve this question are below:

#### 1.3.3.1 Job Events Distribution

- The dataset `part-00049-of-00500.csv.gz` was copied from Google Cloud Storage and decompressed locally.

- The file was loaded into a Spark DataFrame, and proper column names were assigned to the dataset.
- We filtered rows with null values in the `scheduling_class` column.
- Jobs were grouped by `scheduling_class` using the `groupBy()` and `count()` functions. The results were sorted by scheduling class.
- The resulting distribution was visualized using a bar chart, where the x-axis represents the scheduling classes (0, 1, 2, 3) and the y-axis represents the number of jobs in each class.

#### 1.3.3.2 Task Events Distribution

- Since the previously loaded `task_events` dataset did not include the `scheduling_class` column, a different dataset (`part-00050-of-00500.csv.gz`) was used.
- This file was loaded into a Spark DataFrame, and appropriate column names were assigned to the dataset.
- Tasks were grouped by `scheduling_class` and their counts computed using the same Spark functions as for jobs. The results were visualized with a similar bar chart.

#### 1.3.3.3 Comparison of Jobs and Tasks

A grouped bar chart was created to compare the number of jobs and tasks per scheduling class. This visualization highlights the workload associated with each class, allowing for a direct comparison of job and task distributions.

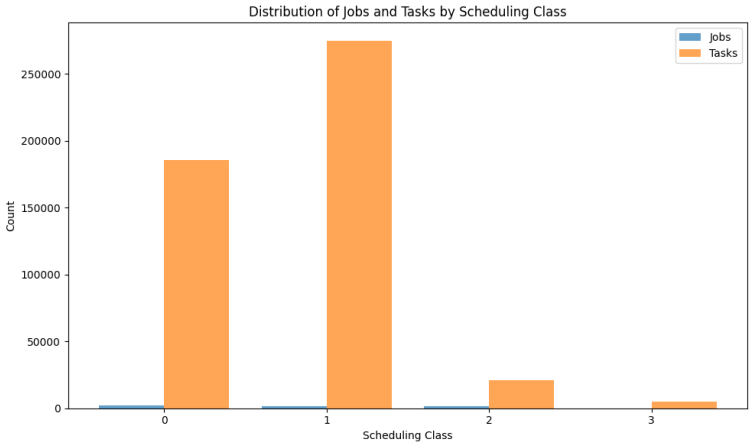


FIGURE 1.3: Distribution of Jobs and Tasks by Scheduling Class

#### 1.3.3.4 Key Observations

- Scheduling class `1` had the highest count for both jobs and tasks, indicating it handles the majority of the workload.
- Scheduling class `0` followed as the second most significant class, processing a smaller workload.
- Classes `2` and `3` represented a minimal proportion of jobs and tasks, suggesting their use for specialized or less frequent operations.

Note that the `scheduling_class` column indicates the priority level of jobs/tasks for resource allocation:

- `0`: Best-effort priority, used for less critical tasks.
- `1`: Production priority, typically used for regular and high-demand workloads.
- `2`: Batch priority, assigned to jobs/tasks requiring background execution.
- `3`: Monitoring/debugging priority, reserved for system maintenance and debugging activities.

### 1.3.4 Question 4:

***Do tasks with a low scheduling class have a higher probability of being evicted?*** To answer this question, we calculated the *eviction rate* for tasks in each scheduling class. The eviction rate is defined as the ratio of the number of evicted tasks to the total number of tasks in a given scheduling class.

1. First, we filtered tasks where `event_type = 2`, indicating they were evicted.
2. Then, we calculated the total number of tasks for each scheduling class using the `groupBy()` and `count()` functions.
3. Next, we calculated the number of evicted tasks for each scheduling class using the same grouping and aggregation approach.
4. We joined the total task counts and evicted task counts on the scheduling class column and calculated the eviction rate as:

$$\text{Eviction Rate} = \frac{\text{Evicted Count}}{\text{Total Count}}$$

The bar chart below illustrates the eviction rates across scheduling classes.

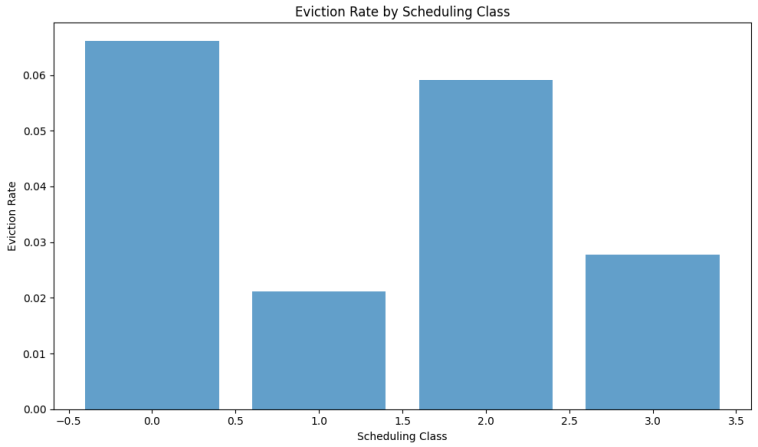


FIGURE 1.4: Eviction Rate by Scheduling Class

#### 1.3.4.1 Observations

- Tasks with a **lower scheduling class** (e.g., `0`) tend to have **higher eviction rates**. This aligns with the definition of scheduling classes, where lower classes prioritize tasks less and make them more prone to eviction.
- Tasks in higher scheduling classes (e.g., `3`) have a lower eviction rate, indicating they're critical and resilient against eviction.

### 1.3.5 Question 5:

***In general, do tasks from the same job run on the same machine?***

To answer this question, we analyzed the distribution of tasks across machines for each job. We started by grouping tasks by their `job_id` and collecting the set of unique `machine_id`s for each job. We then calculated the number of unique machines used per job.

Next, we identified jobs that exclusively ran on a single machine and compared their count with jobs that utilized multiple machines. The results are summarized below:

- Jobs running on a single machine: **1320 (59.81%)**.
- Jobs running on multiple machines: **870 (39.42%)**.

Figure 1.5 illustrates the distribution of unique machines per job, focusing on jobs that used up to 10 machines. We observe that a significant majority of jobs either run entirely on a single machine or use very few machines, indicating a high degree of locality in task allocation for many jobs.

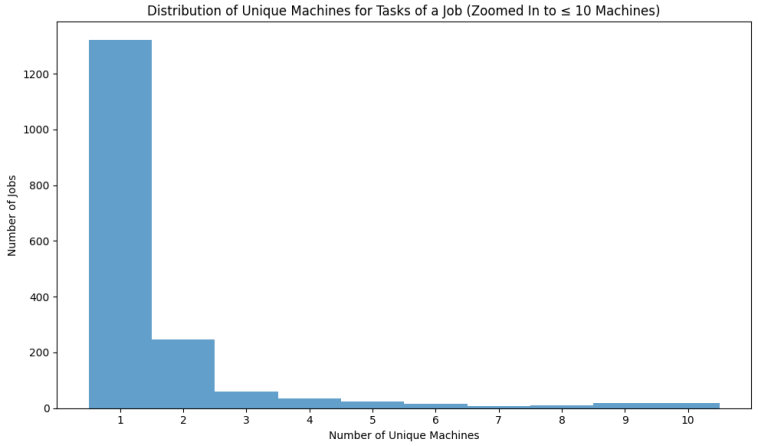


FIGURE 1.5: Distribution of Unique Machines for Tasks of a Job (Zoomed In to $\leq$ 10 Machines).

### 1.3.6 Question 6:

***Are the tasks that request the more resources the one that consume the more resources?***

To answer this question, we analyzed the relationship between the requested and consumed resources for tasks.

### 1.3.7 Resource Requests vs. Consumption

To investigate whether tasks that request more resources are also the ones that consume more, we analyzed the relationships between requested and consumed CPU and memory resources. This analysis involved calculating the ratios of consumed to requested resources and visualizing the results through scatter plots and heatmaps.

The analysis focused on the following fields:

- **Requested Resources:** `cpu_request` (CPU), `memory_request` (Memory).
- **Consumed Resources:** `cpu_rate` (CPU), `canonical_memory_usage` (Memory).

We computed the ratios of consumed to requested resources for both CPU and memory:

$$\text{CPU Ratio} = \frac{\texttt{cpu\_rate}}{\texttt{cpu\_request}},$$
$$\text{Memory Ratio} = \frac{\texttt{canonical\_memory\_usage}}{\texttt{memory\_request}}.$$

Scatter plots and heatmaps were then generated to visualize these relationships.

#### 1.3.7.1 Results and Observations

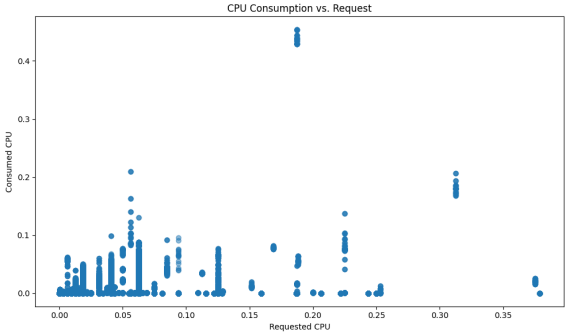


FIGURE 1.6: CPU Consumption vs. Request (Scatter Plot)

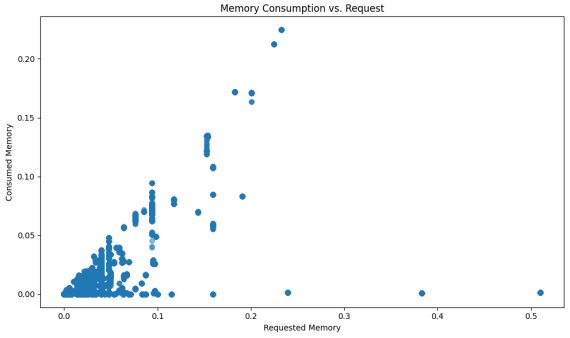


FIGURE 1.7: Memory Consumption vs. Request (Scatter Plot)

**Scatter Plots:** Figures 1.6 and 1.7 display the relationships between requested and consumed resources:

- **CPU:** The scatter plot shows a weak correlation between requested and consumed CPU. Most tasks consume significantly less CPU than requested, with sparse points in the higher request and consumption regions.

- **Memory:** The memory scatter plot indicates a stronger alignment between requested and consumed memory. Tasks that request higher memory tend to consume it more proportionally, although some tasks consume less than requested.

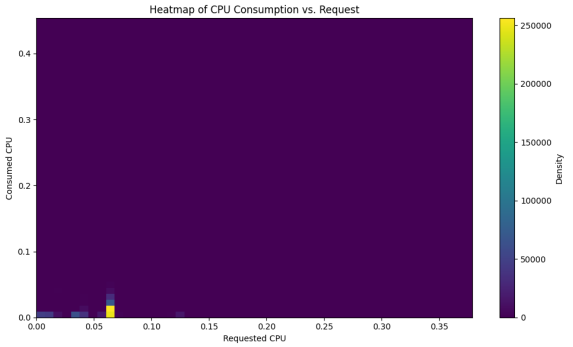


FIGURE 1.8: CPU Consumption vs. Request (Heatmap)

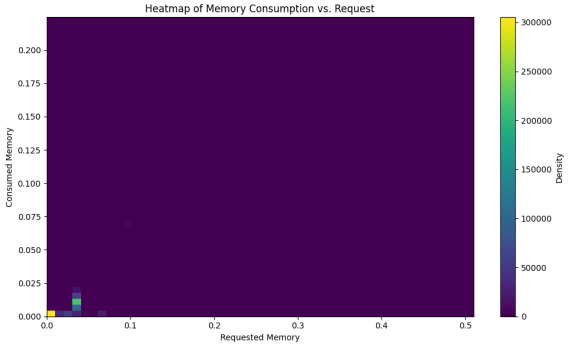


FIGURE 1.9: Memory Consumption vs. Request (Heatmap)

**Heatmaps:** Figures 1.8 and 1.9 show the density of tasks for different levels of resource requests and consumption:

- **CPU:** The density is concentrated in the lower-left corner, suggesting that most tasks request and consume low CPU resources. There is limited alignment between higher requests and consumption.
- **Memory:** The density is more aligned along the diagonal, indicating a stronger relationship between requested and consumed memory. Tasks with higher memory requests are more likely to consume proportionally higher memory.

#### 1.3.7.2 Average Ratios

The average ratios of consumed to requested resources further highlight the discrepancies:

- **CPU Ratio:** On average, tasks consume significantly less CPU than requested, reflecting over-provisioning.
- **Memory Ratio:** The average memory ratio is closer to 1, indicating a better match between requested and consumed memory.

The results reveal that:

- Tasks requesting higher CPU resources do not necessarily consume them, highlighting inefficiencies in CPU allocation.
- Memory requests are better aligned with consumption, though there is room for optimization, particularly for higher requests.
- These findings suggest opportunities to optimize resource requests, especially for CPU, to improve allocation efficiency and reduce over-provisioning.

### 1.3.8 Question 7:

***Can we observe correlations between peaks of high resource consumption on some machines and task eviction events?***

To address this question, we analyzed the relationship between resource consumption peaks and eviction events across machines. We used the following approach:

1. **Identifying Resource Peaks:** We calculated the 90th percentile threshold for CPU, memory, and disk requests for each machine. Tasks exceeding these thresholds for any of the resources were marked as having a "resource peak."
2. **Linking Resource Peaks with Eviction Events:** We filtered eviction events (`event_type = 2`) and joined them with the tasks marked with resource peaks. This allowed us to identify eviction events associated with resource consumption peaks.
3. **Correlation Analysis:** We computed the proportion of eviction events that correlated with resource peaks. The analysis revealed that approximately 25% of eviction events coincided with tasks experiencing resource peaks.

4. **Binning Machines:** To visualize the distribution of eviction events across machines, we divided machine IDs into bins. Each bin represents a range of machine IDs, making the data easier to analyze and interpret. For example:

   - **Bin 1:** Machines with lower IDs, which correspond to a specific range of machine IDs (e.g., 0–1 billion).
   - **Bin 2, Bin 3, etc.:** Subsequent ranges of machine IDs.

   This binning strategy allowed us to observe patterns of eviction events across machines without analyzing each machine individually.

#### 1.3.8.1 Key Findings:

- The majority of eviction events were concentrated in **Bin 1**, which accounted for more than half of all eviction events. This suggests that machines with lower IDs experienced more frequent evictions.
- The remaining eviction events were distributed sparsely across the other bins, with **Bin 7** and **Bin 3** showing higher counts compared to the rest.
- The observed concentration in **Bin 1** could indicate that these machines are either older, less efficient, or subject to a higher workload, resulting in more eviction events.

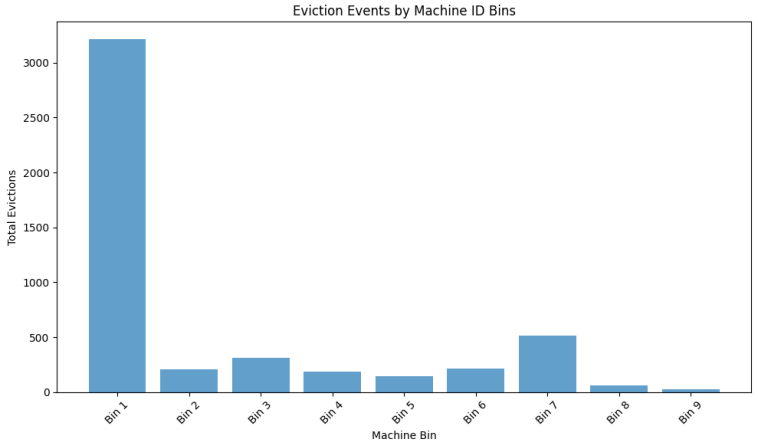


FIGURE 1.10: Eviction Events by Machine ID Bins

The analysis shows a moderate correlation (0.25) between resource peaks and eviction events. Furthermore, the distribution of eviction events across machine bins highlights disparities in machine utilization or resource allocation. Machines in **Bin 1** experienced

the most evictions, suggesting potential issues with either their capacity or workload distribution.

## 1.4 Additional Questions

### 1.4.1 Question 1: What is the average resource usage (CPU and Memory) per scheduling class?

#### Why This Question?

Understanding the average resource usage (CPU and Memory) per scheduling class is essential for evaluating how different scheduling priorities impact resource allocation and consumption. This insight can help optimize system performance by identifying underutilized or overutilized resources for specific scheduling classes. Additionally, analyzing resource usage per scheduling class can guide better provisioning strategies and improve overall system efficiency.

#### Methodology

We calculated the average CPU usage (`cpu_rate`) and memory usage (`canonical_memory_usage`) for each scheduling class. This analysis was conducted by grouping the joined dataset by `scheduling_class` and computing the mean values for the CPU and memory usage columns. The results were visualized in a bar chart to highlight resource consumption patterns across scheduling classes.

#### Results

The bar chart below illustrates the average CPU and memory usage (normalized) for each scheduling class:

- Scheduling class **0**: Minimal resource usage was observed, reflecting the lowest priority tasks with limited resource demands.
- Scheduling class **1**: Tasks exhibited moderate CPU and memory consumption, showing a balanced usage pattern.

- Scheduling class **2**: CPU usage was lower than memory usage, indicating that tasks prioritized memory resources.
- Scheduling class **3**: This class demonstrated the highest memory consumption (**0.039**), with CPU usage remaining relatively low (**0.002**). This highlights tasks in this class as having high memory demands relative to their CPU consumption.

### Conclusion

The analysis revealed distinct resource usage patterns across scheduling classes. Scheduling class 3 demonstrated significantly higher memory usage compared to CPU usage, likely due to tasks requiring substantial memory resources while being CPU-light. These findings provide insights into resource allocation trends, enabling more efficient system management and resource provisioning.

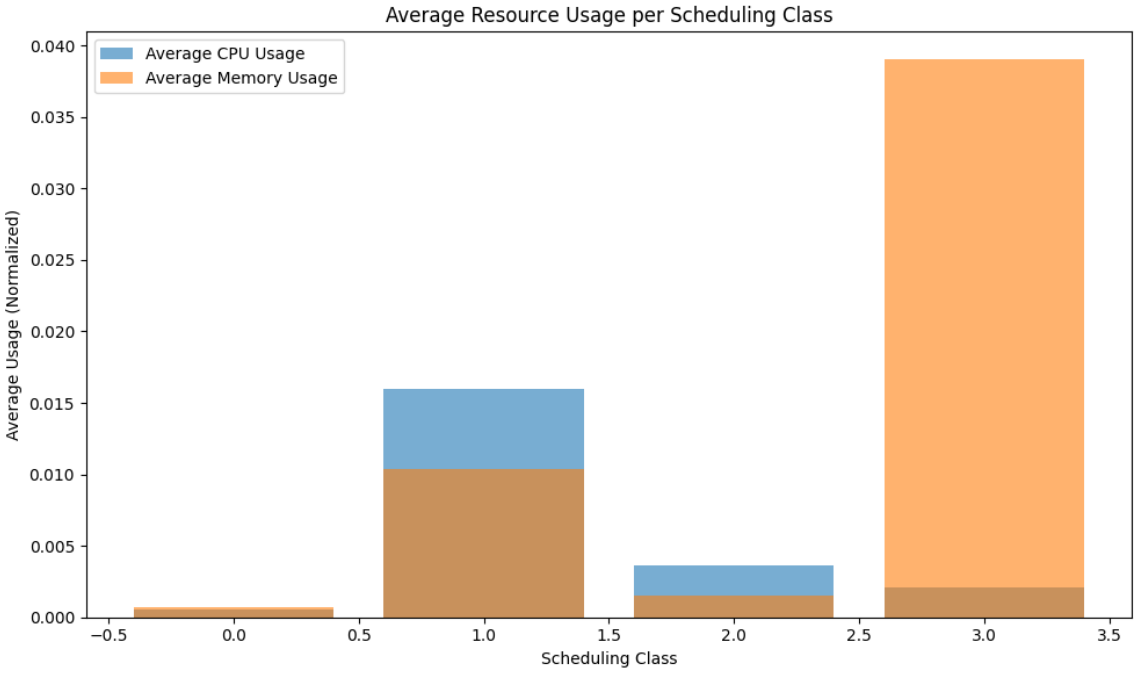


FIGURE 1.11: Average Resource Usage (CPU and Memory) per Scheduling Class

### 1.4.2 Questions 2: Predicting Task Eviction Using Machine Learning

Task eviction in distributed systems can significantly affect performance and resource utilization. Predicting which tasks are likely to be evicted enables proactive resource allocation, improving system reliability. Leveraging the capabilities of Spark's ML library,

we implemented a machine learning pipeline to predict task eviction using resource requests and utilization as input features. This analysis provides insights into key factors influencing eviction.

### Methodology

We structured the problem as a binary classification task, where the label `evicted` indicates whether a task was evicted (`1`) or not (`0`). The process involved the following steps:

1. **Feature Selection:** We selected the following features:
   - `cpu_request`: CPU resources requested by the task.
   - `memory_request`: Memory resources requested by the task.
   - `cpu_rate`: Actual CPU usage of the task.
   - `canonical_memory_usage`: Actual memory usage of the task.
   - `disk_io_time`: Disk input/output time for the task.
2. **Feature Engineering:** A feature vector was created using Spark's `VectorAssembler`, combining the selected features into a single column.
3. **Data Splitting:** The dataset was split into 80% training and 20% testing subsets to evaluate model performance.
4. **Model Training:** Two models were trained and compared:
   - **Logistic Regression:** A simple linear model for binary classification.
   - **Random Forest Classifier:** A non-linear model based on an ensemble of decision trees.
5. **Evaluation:** Both models were evaluated using the Area Under the ROC Curve (AUC) metric to measure their ability to distinguish between evicted and non-evicted tasks.

### Model Performance

The evaluation results for both models are summarized in Table 1.1.

| Model | ROC-AUC Score |
|---|---|
| Logistic Regression | 0.66 |
| Random Forest Classifier | 0.69 |

TABLE 1.1: Model Performance Comparison

This result indicates that the Random Forest model was better at distinguishing between evicted and non-evicted tasks, likely due to its ability to capture non-linear relationships in the data.

### Feature Importance

The feature importance values extracted from the Random Forest model are visualized in Figure 1.12. The results highlight the following:

- `canonical_memory_usage`: The most significant feature, contributing over 70% to the model's predictions. This shows the importance of memory consumption in task eviction.
- `memory_request`: The second most important feature, reflecting the critical role of memory allocation policies.
- `cpu_request`: While less impactful than memory-related features, CPU requests still played a role in predicting eviction.
- `disk_io_time` and `cpu_rate`: These features had minimal importance, suggesting that disk I/O and CPU consumption were less influential in this context.

The analysis revealed that memory-related features (`canonical_memory_usage` and `memory_request`) are the primary determinants of task eviction. This emphasizes the importance of effective memory management strategies in distributed systems.

## Conclusion

By leveraging Spark's ML library, we successfully implemented a machine learning pipeline to predict task eviction and identified key resource-related factors influencing eviction. The insights gained can guide resource allocation and system optimization efforts, contributing to more stable and efficient distributed systems.

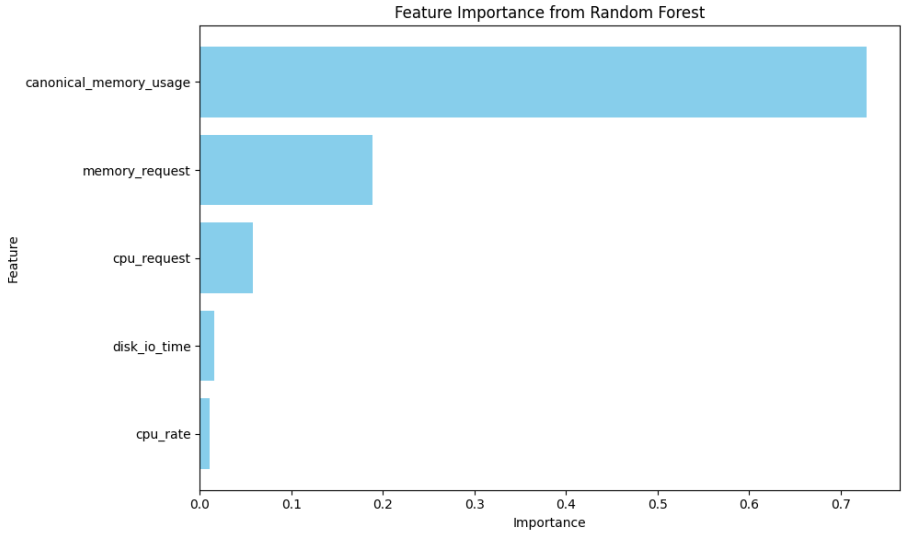


FIGURE 1.12: Feature Importance from Random Forest Classifier

## 1.5 Extensions:

### 1.5.1 Comparing Spark with Pandas

In this section, we aimed to compare the performance of Spark with Pandas for the same analysis. We took a subset of the data (e.g., the first 10000 rows) and performed the same grouping and counting operations using Pandas (specifically for **Question 1**). We measured runtime and memory usage for both Spark and Pandas, and the following observations were made:

1. **Pandas:** Pandas performed the analysis quickly for the subset of 10,000 rows. This is because Pandas operates in-memory, making it efficient for small to medium-sized datasets. However, it becomes significantly slower and more memory-intensive when processing larger datasets due to its inability to handle distributed computation.

2. **Spark:** Spark demonstrated higher runtime overhead for the subset due to its distributed nature, which involves partitioning the data and scheduling tasks across workers. However, this overhead becomes negligible as the dataset size increases. Spark is optimized for large-scale data processing and can efficiently handle datasets that exceed the memory capacity of a single machine.

### 1.5.2 Model Training and Evaluation

The goal of this extension (in additional Question 2) was to assess the performance of different machine learning frameworks and models, namely Scikit-learn, TensorFlow, and PyTorch, for predicting task evictions. These frameworks were evaluated and compared against the Spark MLlib models used earlier. The evaluation metrics included Area Under the ROC Curve (AUC), precision, recall, and F1-score. Additionally, this extension aimed to demonstrate how Spark's preprocessing capabilities could complement more advanced model training frameworks.

#### 1.5.2.1 Implementation and Results

#### 1.5.2.2 Scikit-learn Models

Two models were implemented using Scikit-learn: Logistic Regression and Random Forest. The performance metrics for these classifiers are summarized in Table 1.2.

| **Model** | **AUC** | **Precision (Class 0)** | **Recall (Class 0)** | **Precision (Class 1)** | **Recall (Class 1)** |
|---|---|---|---|---|---|
| Logistic Regression | 0.502 | 0.77 | 0.99 | 0.31 | 0.01 |
| Random Forest | 0.454 | 0.75 | 0.90 | 0.02 | 0.01 |

TABLE 1.2: Performance Metrics for Scikit-learn Models

**Overall Observation:** Both models were significantly affected by class imbalance, resulting in biased predictions towards the majority class. These results highlight the limitations of traditional models for imbalanced datasets and justify exploring neural network-based approaches.

#### 1.5.2.3 Neural Networks

We implemented neural networks using TensorFlow and PyTorch to explore their ability to model complex, nonlinear relationships in the dataset. The performance metrics for these models are summarized in Table 1.3.

| **Model** | **AUC** |
|---|---|
| TensorFlow | 0.70092 |
| PyTorch | 0.70057 |

TABLE 1.3: Performance Metrics for Deep Learning Models

- **TensorFlow:** The neural network demonstrated consistent improvement in AUC over training epochs. It outperformed the Scikit-learn models and Spark MLlib, achieving the highest AUC among all frameworks.
- **PyTorch:** The PyTorch neural network achieved comparable performance to TensorFlow, confirming the robustness of neural networks in handling the dataset.

#### 1.5.2.4 Comparison of Frameworks

This section highlights the comparison of performance across Spark MLlib, Scikit-learn, and deep learning frameworks (TensorFlow and PyTorch). Spark was used for preprocessing and feature engineering in all cases, while the other frameworks handled the model training.

| Framework | Logistic Regression (AUC) | Random Forest (AUC) | Neural Network (AUC) |
|---|---|---|---|
| Spark MLlib | 0.66 | 0.69 | - |
| Scikit-learn | 0.502 | 0.454 | - |
| TensorFlow | - | - | 0.70092 |
| PyTorch | - | - | 0.70057 |

TABLE 1.4: Comparison of Model Performance Across Frameworks

#### 1.5.2.5 Key Findings

- **Neural Networks Outperform Traditional Models:** TensorFlow and PyTorch achieved an AUC of 0.701, outperforming traditional models (Logistic Regression and Random Forest) implemented in both Spark MLlib and Scikit-learn.
- **Spark's Preprocessing Advantage:** Spark's distributed computing capabilities efficiently handled the preprocessing and feature engineering of large-scale data, showing its key role in a hybrid workflow.
- **Spark MLlib vs. Neural Networks:** Spark MLlib provided reasonable performance with AUCs of 0.66 (Logistic Regression) and 0.69 (Random Forest), but it was outperformed by TensorFlow and PyTorch. This highlights the potential of neural networks to model complex, nonlinear relationships in the dataset.
- **Hybrid Workflow Potential:** Combining Spark's scalability with the modeling capabilities of TensorFlow and PyTorch provides a powerful hybrid workflow for large-scale predictive modeling tasks.

## 1.6 Conclusion

This project provided a comprehensive analysis of the Google Cluster Data 2011 dataset, showcasing the power of Apache Spark in processing large-scale data. Key findings include:

- **Efficient Use of Spark:** Spark's scalability enabled efficient filtering, aggregation, and analysis of large-scale distributed data, outperforming single-machine frameworks like Pandas for such tasks.
- **Resource Allocation Insights:** CPU requests often exceeded actual consumption, highlighting over-provisioning, while memory usage was more aligned with requests. Optimizing resource allocation can improve system performance.
- **Task and Job Behavior:** Lower-priority tasks were more prone to eviction, and most jobs exhibited high task locality. Resource consumption peaks moderately correlated with eviction events.
- **Predictive Modeling:** Neural networks using TensorFlow and PyTorch achieved superior performance in predicting task eviction, with memory-related features being the most influential predictors.
- **Framework Comparison:** While Spark MLlib provided reasonable results, combining Spark for preprocessing with advanced frameworks like TensorFlow and PyTorch demonstrated a powerful hybrid approach for large-scale predictive tasks.

We acknowledge that reading the entire dataset directly from the cloud would have provided a more comprehensive analysis and better utilization of Spark's distributed capabilities. However, due to resource and logistical constraints, we worked with a subset of the data.

Future work could address dataset imbalance, improve predictive models, and explore real-time implementations for monitoring and optimization. This project highlights the potential of integrating distributed processing with advanced machine learning to manage complex systems effectively.