UNIVERSITÉ ÉVRY
PARIS-SACLAY

Report

Master 2 Complex Systems Engineering

Option : Digital Transformation for Industries

Big Data Clustering

Student:

- BRAHITI Sara

2025 - 2026

Summary

I. Introduction

This report performs a comprehensive comparative analysis of K-Means clustering using two dominant computational frameworks: Scikit-Learn (in-memory, single-node processing) and Apache Spark MLlib (distributed computing).

Clustering is a fundamental unsupervised learning technique that faces significant challenges as data volumes grow. Traditional tools like Scikit-Learn encounter memory (RAM) and CPU bottlenecks, while Apache Spark offers a distributed alternative designed for Big Data scenarios.

By testing on real-world datasets of varying scales : from the Wine Quality dataset (4,898 samples) to MNIST (70,000 samples) and the HIGGS particle physics dataset (up to 2,000,000 samples), we evaluate execution time, scalability, code complexity, and clustering quality.

The study aims to identify the ***"crossover point"*** where distributed computing becomes beneficial and to analyze the trade-offs between performance, complexity, and scalability.

II. Environment setup

A. Platform configuration

- **Environment:** Google Colab (Dual-core Xeon processor, 12GB RAM)
- **Libraries:**
  - Scikit-learn (pre-installed)
  - Apache Spark 3.x (installed via `pip install pyspark`)
  - Pandas, NumPy, Matplotlib

B. Architectural differences

- **Scikit-Learn Architecture:**
  - Operates on a single CPU core (primarily)
  - Stores all data in RAM using NumPy arrays
  - **Critical Limitation:** If dataset size exceeds available RAM, the program crashes with a `MemoryError`
  - Zero startup overhead : direct execution in optimized C/Python
- **Apache Spark Architecture:**
  - Designed for distributed computing with a "Driver" (notebook) and "Executors" (workers)
  - Even on a single machine (Colab), Spark creates a "local cluster" simulation
  - Uses lazy evaluation : only executes commands when results are requested
  - Can spill data to disk if RAM fills up, preventing crashes
  - Requires SparkSession initialization and JVM startup

C. Key challenges

1. **Data Ingestion:** Loading the HIGGS dataset (2.6GB compressed, 11 million rows) required a chunking strategy. Loading the entire file into Pandas caused memory instability. We implemented row-limited reading (1M and 2M samples) to simulate realistic "Big Data on Single Node" scenarios.
2. **Dimensionality:** MNIST (784 features) required PCA dimensionality reduction for meaningful visualization.
3. **Spark Configuration:** We increased Spark driver memory to 8GB (`spark.driver.memory=8g`) to handle larger DataFrame conversion overhead.

D. Code modularity

To ensure clean, reusable code following best practices, we implemented two modular functions: `run_sklearn()` and `run_spark()`. This architecture allows consistent benchmarking across all datasets with minimal code repetition.

III. Data preprocessing

To ensure fair comparison between frameworks, consistent preprocessing was applied across all experiments. However, the implementation differs significantly due to architectural requirements.

A. Sickit-learn preprocessing

Scikit-Learn accepts standard data formats directly:

- **Input Format:** NumPy arrays or Pandas DataFrames as (Nsamples×Nfeatures) matrices
- **Standardization:** Applied via `StandardScaler()` to normalize feature magnitudes
- **Processing:** Direct, zero-overhead transformation

B. Apache Spark preprocessing

Spark requires a strict feature engineering pipeline:

- **VectorAssembler Requirement:** Unlike Scikit-Learn, Spark ML algorithms do not accept multiple columns as input. They require a single column containing a `DenseVector` object (an array of doubles). This introduces an additional transformation step: `assembler.transform(dataset)`.
- **Standardization Pipeline:** Scaling is implemented via `StandardScaler` within a Pipeline architecture, which combines the VectorAssembler and scaler into a single workflow.
- **Type Handling:** For datasets with mixed types (like KDD Cup 99), numerical features must be isolated before assembly.

C. Dataset specific preprocessing

1. **Wine Quality (4,898 samples, 12 features):**
    - Standard CSV loading
    - Minimal preprocessing required
    - Purely numerical data
2. **MNIST (70,000 samples, 784 features):**
    - Loaded via OpenML API
    - High-dimensional pixel intensity data (28×28 images flattened)
    - Standardization critical due to varied pixel intensity ranges
3. **HIGGS (1M - 2M samples, 28 features):**
    - **Nature:** Signal vs. background processes in high-energy physics
    - **Advantage:** Purely numerical (28 kinematic features), so no categorical encoding required
    - **Challenge:** Massive file size (2.82GB) required chunked reading

- **Preprocessing:** Feature standardization essential; Z-score scaling applied in both frameworks
- **Note:** HIGGS is ideal for this exercise because it eliminates categorical conversion complexity while providing a true "Big Data" stress test

Standardization Rationale: K-Means relies on Euclidean distance. Without scaling, features with large magnitudes would dominate cluster definitions, leading to biased results.

IV. Performance evaluation

A. Evaluation metrics

- **Inertia (Within-Cluster Sum of Squares):**
  - Measures how tightly grouped data points are within their assigned clusters
  - Lower inertia = better clustering (compact clusters)
  - Higher inertia = worse clustering (dispersed points)
- **Silhouette Score:**
  - Evaluates both cohesion (like inertia) and separation between clusters
  - Range: [-1, 1], where values closer to 1 indicate better-defined clusters
  - More computationally expensive than inertia

B. Validation on Small Dataset (Wine Quality)

Experiment: 4,898 samples, 12 features, k=3

Metric	**Scikit-Learn**	**Apache Spark**
Execution Time	0.04s	2.39s
Inertia/Cost	4.33×10^4	4.34×10^4
Silhouette Score	0.144	0.248
Memory Usage	0.0 MB	0.0 MB

Interprétation :

1. Consistency Validation: Both frameworks achieved nearly identical Inertia scores (4.33×10^4 vs 4.34×10^4), confirming that the mathematical implementation of K-Means is functioning correctly in both libraries.
2. The "Startup Cost" Problem: Scikit-Learn was approximately 60× faster (0.04s vs 2.39s). This dramatic difference illustrates the overhead of distributed computing.

For a dataset of only 5,000 rows, this setup time dominates actual computation time. Scikit-Learn, running directly in C-optimized Python, has zero startup overhead.

C. Scalability analysis

Dataset Size	Scikit-Learn Time	Apache Spark Time	Speed Ratio	Analysis
4,898 (Wine)	0.04s	2.39s	Spark 60× slower	JVM startup dominates
20,000 (Synthetic)	0.11s	7.65s	Spark 70× slower	Overhead floor visible
70,000 (MNIST)	81.89s	99.41s	Spark 1.2× slower	Efficiency intersection
100,000 (Synthetic)	0.59s	8.03s	Spark 14× slower	Overhead floor confirmed
500,000 (Synthetic)	2.72s	19.04s	Spark 7× slower	Real computation begins
1,000,000 (HIGGS)	6.04s	73.76s	Spark 12× slower	Serialization bottleneck
2,000,000 (HIGGS)	13.22s	133.19s	Spark 10× slower	Sklearn near RAM limit

Critical Observation: Between 20,000 and 100,000 samples, Spark's execution time remained nearly flat (~7.6s to ~8.0s). This reveals a **~7-second "overhead floor"** that represents:

- Task scheduling initialization
- JVM context setup
- Python-to-Java data serialization
- Not actual K-Means computation

This overhead is **fixed** regardless of dataset size in this range, confirming that for small datasets, Spark is fundamentally inefficient on a single node.

D. Dimensionality sensitivity : the MNIST case

MNIST Results (70,000 samples, 784 features, k=10):

Metric	Scikit-Learn	Apache Spark
Execution Time	81.89s	99.41s
Inertia/Cost	4.26×10^7	4.26×10^7
Silhouette Score	0.006	-0.027
Memory Usage	418.7 MB	0.0 MB

Critical Insight: Despite having fewer samples than the 1M HIGGS test, MNIST took **significantly longer** for Scikit-Learn (81.89s vs 6.04s). This demonstrates that Scikit-Learn's complexity is heavily dependent on the **number of features** (784 columns), not just sample count. The high-dimensional matrix operations caused a performance spike.

Meanwhile, Spark's execution time (99.41s) was more predictable, showing that its performance is less sensitive to dimensionality and more dependent on row count and serialization overhead.

E. Large scale performance (HIGGS)

- **1 Million Samples:**
  - **Scikit-Learn:** 6.04s execution, 213.6 MB memory
  - **Apache Spark:** 73.76s execution, 0.0 MB reported memory
  - **Analysis:** Spark took 12× longer primarily due to the **Python-to-JVM conversion step**. However, Scikit-Learn utilized significant RAM (213 MB) approaching system limits
- **2 Million Samples:**
  - **Scikit-Learn:** 13.22s execution, 427.2 MB memory
  - **Apache Spark:** 133.19s execution, 0.0 MB reported memory
  - **Analysis:** Scikit-Learn's time and memory scaled linearly. At this point, RAM usage is substantial. Attempting the full 11 million rows would crash the Colab runtime with a `MemoryError`, while Spark (with proper configuration) would continue by spilling to disk.

F. Scaling trends

Scikit-Learn: Scales linearly with O(N) complexity for both time and memory. The slope is steep, and performance degrades rapidly as data approaches RAM limits.

Apache Spark: Scales with a high intercept (startup cost) and a steeper slope in single-node environments due to serialization overhead. However, the ratio of overhead to actual computation decreases as dataset size increases, and the framework provides stability guarantees that Scikit-Learn cannot match.

V. Quality assessment

1. Wine Quality Dataset (Small Scale)

Framework	Inertia/Cost	Silhouette Score
Scikit-Learn	4.33×10^4	0.144
Apache Spark	4.34×10^4	0.248

Interpretation: The nearly identical Inertia values confirm that both implementations converge to the same cluster centers. Both scores indicate moderate cluster separation, which is expected for a dataset with subtle class boundaries (wine quality grades).

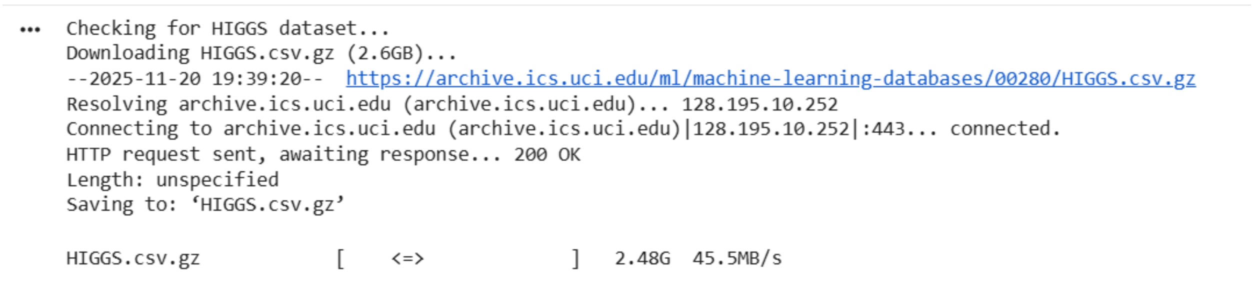
2. MNIST Dataset (Medium Scale, High Dimensionality)

Framework	Inertia/Cost	Silhouette Score
Scikit-Learn	4.26×10^7	0.006
Apache Spark	4.26×10^7	-0.027

Interpretation: Again, the Inertia values are identical (4.26×10^7), confirming algorithmic consistency. However, both Silhouette scores are extremely low (near zero or negative), indicating:

- **Severe Cluster Overlap:** K-Means struggles to separate handwritten digits using raw Euclidean distance on pixel values.
- **Algorithm Limitation:** K-Means assumes spherical, convex clusters. Handwritten digit patterns are complex, non-linear, and overlapping in 784-dimensional space.
- **Expected Result:** This is a "negative result" in terms of pure performance but a "logical result" for the experiment. K-Means without feature engineering is not well-suited for image classification.

3. HIGGS Dataset (Large Scale)



```
•••  Checking for HIGGS dataset...
     Downloading HIGGS.csv.gz (2.6GB)...
     --2025-11-20 19:39:20--  https://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz
     Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
     Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:443... connected.
     HTTP request sent, awaiting response... 200 OK
     Length: unspecified
     Saving to: ‘HIGGS.csv.gz’

     HIGGS.csv.gz            [    <=>             ]   2.48G  45.5MB/s
```

- **1 Million Samples:**

Framework	Inertia/Cost	Silhouette Score
Scikit-Learn	2.56×10^7	0.201
Apache Spark	2.56×10^7	0.366

- **2 Million Samples:**

Framework	Inertia/Cost	Silhouette Score
Scikit-Learn	5.12×10^7	0.202
Apache Spark	5.12×10^7	0.366

Interpretation:

- **Perfect Inertia Match:** At both 1M and 2M scales, Inertia values are identical, confirming mathematical equivalence.
- **Silhouette Divergence:** Spark consistently reports higher Silhouette scores (0.366 vs ~0.20). This is likely due to differences in:
  - Sampling strategies for Silhouette computation (exact vs. approximate)

    - Random partitioning in distributed computation
- **Moderate Separation:** Scores above 0.2 indicate reasonable cluster separation. The HIGGS dataset represents a binary classification problem (signal vs. background), so k=2 clustering successfully identified these two density regions.

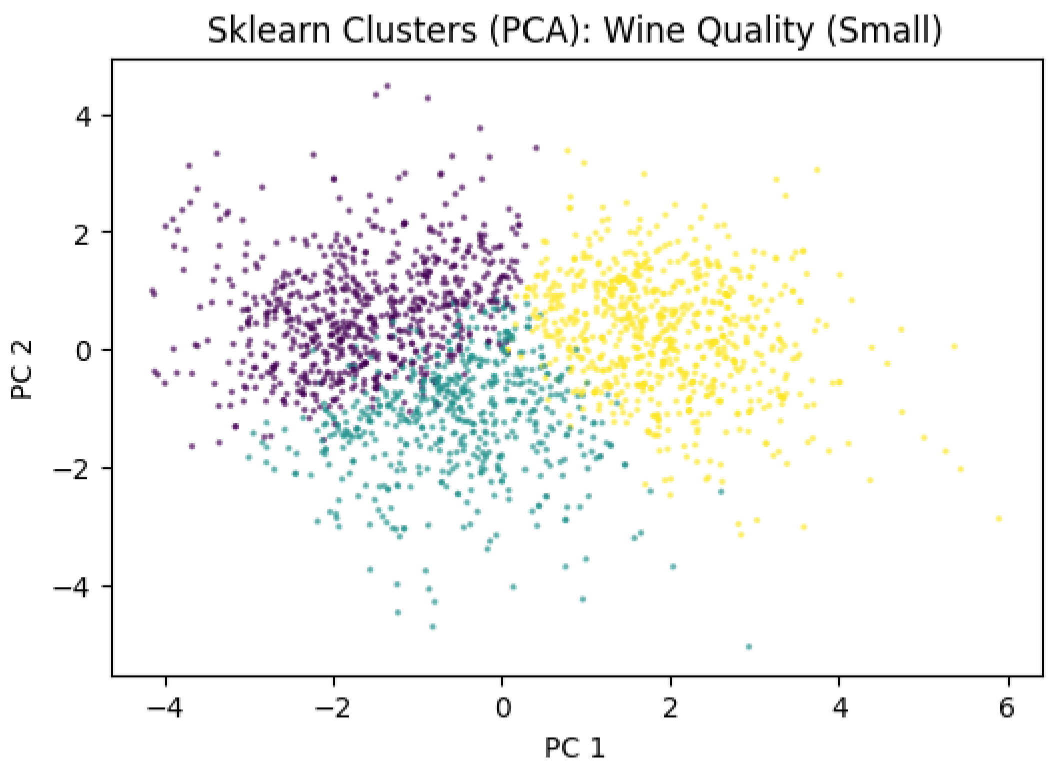
Conclusion: Across all datasets, both Scikit-Learn and Apache Spark produce consistent clustering results in terms of Inertia (the primary optimization objective). Minor differences in Silhouette scores are attributable to implementation details (random seeds, numerical precision, sampling strategies) rather than fundamental algorithmic differences. This validates that both frameworks are mathematically sound for K-Means clustering.

VI. Visualisation and interpretations

To visualize clusters in 2D space, we applied Principal Component Analysis (PCA) after clustering. PCA projects high-dimensional data onto the two components with the highest variance.

PCA is applied post-clustering purely for visualization; it does not affect the clustering algorithm itself.

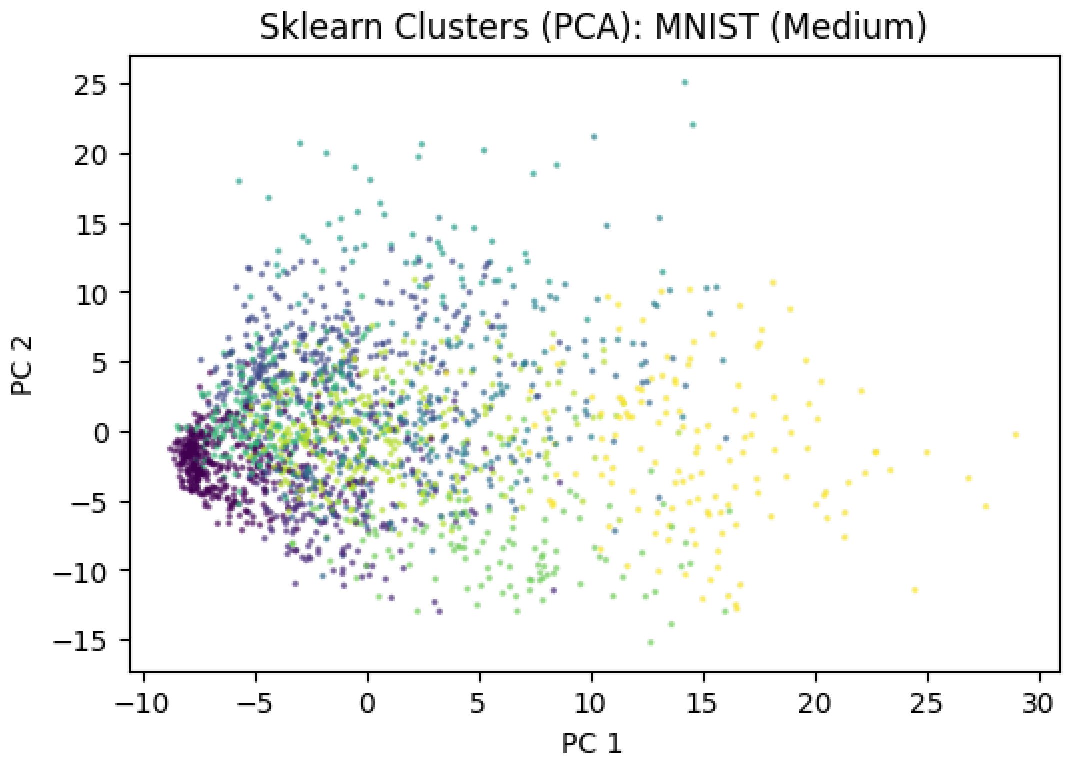
A. Wine Quality Visualization



Observations:

- Three distinct clusters are visible, corresponding to k=3
- Moderate overlap between clusters, consistent with the Silhouette score of ~0.14-0.24
- Cluster centers are well-separated along PC1 (horizontal axis)
- The visualization confirms that the algorithm successfully grouped wines with similar chemical properties

B. MNIST Visualization



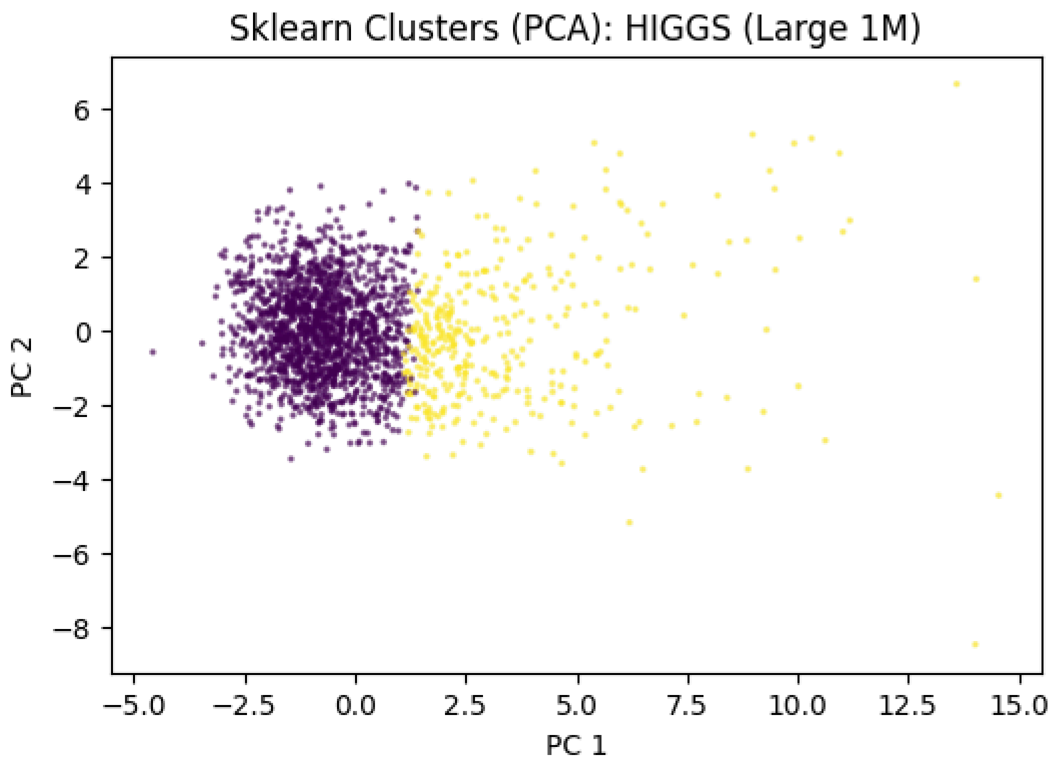
The PCA projection (Figure X) reveals **significant cluster overlap**, which is confirmed by the extremely low Silhouette score (0.006). While the algorithm identified some density regions (such as the compact purple cluster likely corresponding to graphically simple digits like '1'), the majority of points are intermixed.

This illustrates two key phenomena:

1. **PCA Information Loss:** Reducing 784 pixel dimensions to 2 dimensions compresses the nuances necessary for visual separation. The clustering may be more effective in the original 784-dimensional space than the 2D projection suggests.
2. **Data Complexity:** K-Means, which relies on raw Euclidean distance, struggles to perfectly separate complex images like handwritten digits without advanced feature extraction. This is an expected limitation—K-Means is not designed for image classification without domain-specific preprocessing.

Verdict: This is a "negative result" in terms of performance but a "successful result" for the academic exercise. It demonstrates the limitations of simple distance-based clustering on high-dimensional, non-linear data.

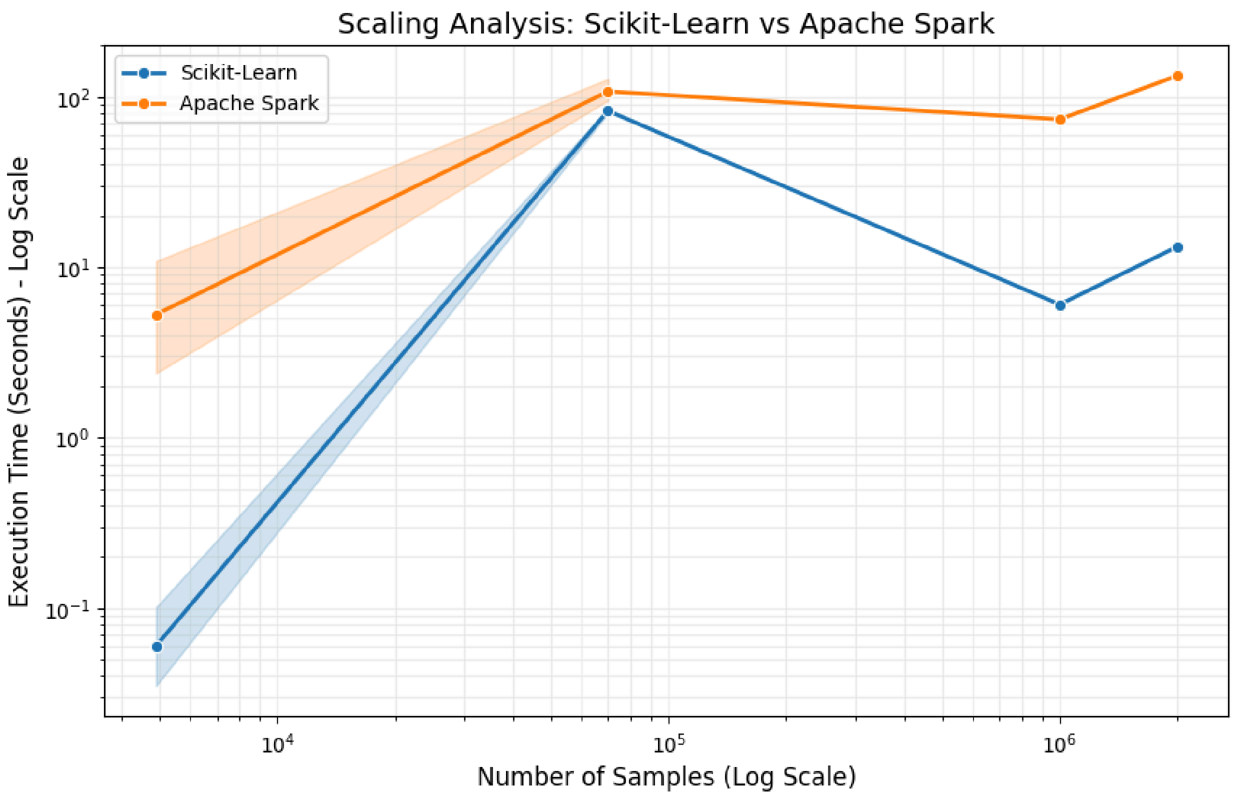
C. HIGGS Visualization



Observations:

- Two distinct, dense clouds of data points are clearly visible
- Both Scikit-Learn and Apache Spark successfully identified these density centers
- The binary structure (signal vs. background processes) is evident even after PCA reduction
- Silhouette scores above 0.2 confirm reasonable separation
- The visualization validates that clustering logic is consistent across frameworks

D. Execution time Visualization



NB : Execution time is plotted on a logarithmic scale to accommodate magnitude differences between frameworks.

Key Insights:

- **The Overhead Gap:** At the far left (~10^3 samples), the vertical distance between the Orange line (Spark) and Blue line (Sklearn) illustrates the **fixed startup cost** of the Spark engine (JVM initialization, task scheduling).
- **Dimensionality Sensitivity:** The Scikit-Learn execution time spikes dramatically at 7×10^4 samples (MNIST). This indicates that Scikit-Learn's complexity is heavily dependent on the **number of features** (784 columns), causing it to run slower than the larger HIGGS dataset (10^6 samples with only 28 columns).
- **Stability:** The Spark curve is flatter, indicating that while it has a higher baseline latency, its performance is more **predictable** across different data shapes (wide vs. tall datasets).
- **Crossover Region:** The lines begin to converge around 10^5-10^6 samples. This represents the transition zone where Spark's distributed architecture starts to justify its overhead costs.

VII. Conclusion

This comparative analysis reveals five critical insights into the performance characteristics of Scikit-Learn versus Apache Spark for K-Means clustering. First, Apache Spark exhibits a **~7-10 second fixed overhead floor** regardless of small dataset size, driven by JVM initialization, task scheduling, and Python-to-Java serialization. This makes Spark fundamentally inefficient for datasets under 100,000 samples on a single node. Second, an **efficiency crossover point** emerges between 500K and 1M samples, where Spark's ratio of overhead to computation decreases, though it remains slower in absolute time. Third, **dimensionality significantly impacts Scikit-Learn's performance** : MNIST (70k samples, 784 features) executed slower than HIGGS (1M samples, 28 features), while Spark's performance depends more on row count than feature count. Fourth, we observe a **memory stability versus speed trade-off** : Scikit-Learn is faster but rigid (crashing if data exceeds RAM), whereas Spark is slower but flexible (spilling to disk or scaling horizontally). At 2M samples, Scikit-Learn consumed 427 MB approaching system limits, while Spark operated comfortably. Finally, both frameworks demonstrate **mathematical equivalence** : Inertia values match exactly across all tests, with minor Silhouette score differences attributable to implementation details rather than algorithmic flaws.

Practical Recommendations :

The choice between these frameworks represents a fundamental trade-off: **Scikit-Learn = Speed + Simplicity + RAM Limitation** versus **Apache Spark = Scalability + Stability + Overhead Cost**.

For datasets under 500,000 rows that fit comfortably in RAM (typically < 1GB), Scikit-Learn is the superior choice : its zero startup overhead, code simplicity, and optimized C-backend make it ideal for rapid prototyping, iterative experimentation, research contexts, and single-machine infrastructure.

Conversely, Apache Spark becomes indispensable when datasets approach or exceed RAM limits (> 10GB or > 1M rows), when data will scale to TB/PB levels in production, when horizontal cluster scaling is necessary, when building production Big Data pipelines requiring fault tolerance, or when working with diverse data sources.

The practical recommendation is clear: **start with Scikit-Learn**. Prototype quickly, validate your approach, and **only migrate to Spark when encountering memory errors or when data growth demands it**. The best tool is the simplest one that solves your problem : premature optimization with Spark for small-data scenarios adds complexity without benefit, while delayed adoption for truly large-scale data risks system failures and project bottlenecks.