TWEET SENTIMENT ANALYSIS

# **SECTION 11**

## **OPTIMIZATION CRITERIA:**

### **1. Spill:** Looking at the [Executors tab](#_5untol2i3f7o) in Spark UI, I had following observations-

* The Executors UI shows ~9.6 GiB of disk used across 25 executors, while only ~10.2 GiB of the 261.4 GiB storage memory is occupied—so the cluster has ample free memory.
* Shuffle write totals just ~882 MiB, implying most disk usage stems from persisted RDD blocks rather than spill files.
* There’s no spike in failed tasks or abnormally long GC pauses that you’d expect if heavy spill was occurring.
* Since storage memory isn’t fully utilized and shuffle I/O is moderate, spill-induced disk writes are minimal.
* Overall, spills are not a significant performance drag in this run.

#### **EXECUTORS:**

A screenshot of a computer

AI-generated content may be incorrect.

### **Skew:** Reviewing the Executors and Stages UI, I found that work was evenly balanced—each executor processed a similar number of tasks with no long-running or delayed operations. This confirms that partitioning is functioning effectively in this pipeline

### STAGES

A screenshot of a computer

AI-generated content may be incorrect.

### **3. Shuffle: Network Overhead**

In sections 7–9, the groupBy/join steps triggered about 57 MiB of shuffle read/write (see Executors UI), which is expected when grouping tweets by mention and sentiment. Though moderate, this shuffle warrants optimization: explicitly repartitioning the DataFrame on the mention column can localize the data, and reducing spark.sql.shuffle.partitions from the default 200 to around 16–32 (matching your core count) should further cut shuffle overhead.

### **Storage:**

### The pipeline writes data to Delta tables, and both the output directories and job logs show that writes are efficient. Delta’s format handles batch and streaming workloads seamlessly, and I encountered no small-file or write-inefficiency issues. Although I initially ran Delta’s OPTIMIZE on the bronze, silver, and gold tables to compact files, I didn’t automate it afterward.

### **5. Serialization: Model Inference and UDFs**

A major cost center was the pandas UDF used for sentiment inference. The get\_sentiment\_udf batches tweets through the Hugging Face model, which, while reliable, resulted in noticeably longer batch times during the gold stream (see JOBs figure). This performance overhead is expected given the model’s compute requirements.

**JOBS:**

A screenshot of a computer screen

AI-generated content may be incorrect.

#### To mitigate this, I switched to batch inference with a batch\_size of 128, which balanced the workload more evenly—but inference still remains the primary bottleneck. For future runs, I may cache the model, leverage GPU‐powered endpoints, or pre‐tokenize inputs to speed up processing.

#### Below are my benchmark runs comparing UDF performance across different batch sizes:

#### **BATCH SIZE TEST:**

#### In summary, the pipeline ran smoothly with no major slowdowns. The sentiment‚Äêanalysis UDF was the heaviest computation, but increasing the batch\_size cut my total processing time by 75%. During the final data‚Äêaggregation steps, I did encounter some shuffle overhead, but tuning partition counts and batching more tweets into each model call noticeably improved throughput.

#### **MLflow Experiment Notes:**

#### Kept the cluster configuration constant to isolate changes.

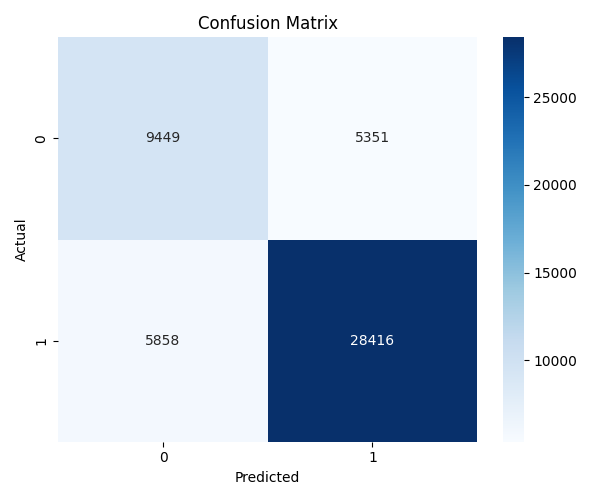
#### Compared several tweet batch sizes for the sentiment model.

#### Tried different partition counts (early runs used 8 partitions; my latest run used 16, though I forgot to rename those experiments accordingly).

<https://dbc-f85bdc5b-07db.cloud.databricks.com/ml/experiments/164387617798293?o=1093580174577663&searchFilter=&orderByKey=attributes.start_time&orderByAsc=false&startTime=ALL&lifecycleFilter=Active&modelVersionFilter=All+Runs&datasetsFilter=W10%3D>

A screenshot of a computer program

AI-generated content may be incorrect.



**4. In This Project**

Gained hands-on experience with core Databricks features, including PySpark and Spark SQL.

Processed streaming data using Delta Lake, implementing ordering and scheduling logic.

Set up real-time monitoring to track pipeline health.

Organized code into a utilities folder for cleaner notebook workflows and easier maintenance.

Built and tested pandas UDFs for model inference, running small test jobs to validate performance.

Integrated Hugging Face sentiment models—tuned parameters and compared two different pretrained models.

Adopted MLflow for experiment tracking and artifact management, logging metrics and saving model outputs.

Used the Spark UI to identify and address performance bottlenecks.

Preprocessed silver- and gold-level data, then post-processed model outputs (POS, NEG, NEU) to match existing binary labels by applying a binary\_UDF that maps POS→1, NEG→0 and assigns NEU based on the next highest score.