**Sentiment Analysis Pipeline for Twitter Data using Spark, Gemini, and Ray**

**1. Introduction**

This document describes an end-to-end pipeline designed to perform sentiment analysis (classifying tweets as Positive, Negative, or Neutral) on a Twitter dataset. A key challenge addressed is the lack of pre-existing sentiment labels in the raw data. The pipeline leverages several powerful technologies:

* **Apache Spark:** For scalable initial data processing, cleaning, and vocabulary generation.
* **Google Gemini API:** To programmatically generate sentiment labels for the cleaned tweets, creating a labeled dataset for training.
* **Ray (Ray Data & Ray Train):** For efficient loading of the labeled data and distributed training of the sentiment classification model.
* **PyTorch:** For defining and implementing the sentiment classification model (an LSTM with Attention).
* **Scikit-learn, Matplotlib, Seaborn:** For evaluating the trained model and visualizing results.

The goal is to demonstrate a robust workflow that handles data preparation at scale (using Spark) and distributed machine learning (using Ray) on data that initially lacks ground truth labels.

**2. Pipeline Stages**

The process is broken down into the following distinct stages, primarily orchestrated by two Python scripts (process\_label\_tweets.py and train\_sentiment\_model\_lstm.py) and a model definition file (lstm\_model.py).

**Stage 1: Data Loading and Cleaning (Spark)**

* **Script:** process\_label\_tweets.py
* **Technology:** Apache Spark (PySpark)
* **Process:**
  + A SparkSession is initialized.
  + The raw Twitter data (JSON lines format) is loaded into a Spark DataFrame, specifically reading the id\_str and full\_text fields.
  + A User-Defined Function (UDF), clean\_tweet\_text, is applied to each tweet's full\_text. This function performs normalization (lowercase) and removes URLs, user mentions, the '#' symbol from hashtags, and potentially excessive punctuation or numbers.
  + Rows where the cleaned\_text is null (due to null input or the cleaning process resulting in an empty string) are filtered out.
  + A minimum length filter (e.g., > 20 characters) is applied to remove overly short or potentially meaningless tweets after cleaning.
  + A configurable DATA\_LIMIT is applied using .limit() to select a subset of the cleaned data (e.g., 1000 records) for the subsequent, more computationally expensive steps (vocabulary building and labeling). This is crucial for managing API costs and processing time.
  + The resulting limited DataFrame (limited\_df) is cached for efficient reuse.

**Stage 2: Vocabulary Building (Spark)**

* **Script:** process\_label\_tweets.py
* **Technology:** Apache Spark (PySpark)
* **Process:**
  + Uses the cached limited\_df (containing the cleaned text for the subset).
  + The cleaned\_text is split into individual words using Spark RDD operations (flatMap).
  + Word frequencies are calculated (map, reduceByKey).
  + Words occurring less frequently than MIN\_WORD\_FREQ are filtered out.
  + The remaining words are sorted by frequency, and the top VOCAB\_SIZE words are selected.
  + A Python dictionary (vocab) is created, mapping these top words to unique integer indices. Special tokens <PAD> (for padding) and <UNK> (for unknown words) are added with indices 0 and 1, respectively.
  + This vocab dictionary is saved locally as a .pkl file using Python's pickle module. This vocabulary will be loaded later during model training.

**Stage 3: Sentiment Label Generation (Spark + Gemini API)**

* **Script:** process\_label\_tweets.py
* **Technology:** Apache Spark (PySpark), Google Gemini API
* **Process:**
  + The limited\_df RDD is processed using mapPartitions, calling the get\_sentiment\_batch\_from\_gemini function for each partition.
  + Inside get\_sentiment\_batch\_from\_gemini:
    - Tweets within the partition are collected into batches (size defined by BATCH\_SIZE).
    - For each batch, a prompt is constructed asking the Gemini model (gemini-1.5-flash-latest or similar) to classify the sentiment of each tweet (Positive, Negative, Neutral) and return the results in a specific JSON format ({"sentiments": ["Positive", "Neutral", ...]}).
    - A single API call is made per batch.
    - Error handling (retries for rate limits, parsing JSON response, handling content blocks) is implemented. If the API response format is incorrect (e.g., list length mismatch), an error label is assigned. If content is blocked, "Neutral" is assigned.
  + The function yields tuples of (id\_str, cleaned\_text, sentiment\_label) for each tweet.
  + The resulting RDD is converted back into a Spark DataFrame.

**Stage 4: Data Saving (Spark)**

* **Script:** process\_label\_tweets.py
* **Technology:** Apache Spark (PySpark)
* **Process:**
  + The DataFrame containing id\_str, cleaned\_text, and the Gemini-generated sentiment\_label is filtered to remove any rows where labeling resulted in an error.
  + The final, successfully labeled DataFrame is saved to disk in the efficient Parquet format (LABELED\_OUTPUT\_PARQUET\_PATH).

**Stage 5: Model Definition (PyTorch)**

* **Script:** lstm\_model.py
* **Technology:** PyTorch
* **Process:**
  + Defines the LSTMSentimentClassifier class, inheriting from torch.nn.Module.
  + **Embedding Layer:** An nn.Embedding layer maps word indices (from the vocabulary) to dense vectors of size EMBEDDING\_DIM. It uses padding\_idx to ignore padding tokens.
  + **LSTM Layer:** A multi-layer, bidirectional nn.LSTM processes the sequence of embeddings. batch\_first=True is used. Dropout is applied between LSTM layers if n\_layers > 1.
  + **Attention Layer:** A simple additive attention mechanism (attention method) is implemented:
    - It calculates attention scores based on the LSTM outputs.
    - It applies softmax to get attention weights.
    - It computes a context vector as a weighted sum of the LSTM outputs.
  + **Dropout:** Applied to the embedding output and the final context vector before the classification layer.
  + **Fully Connected Layer:** An nn.Linear layer maps the attention context vector to the final output dimension (NUM\_CLASSES).
  + The forward method defines the data flow through these layers.

**Stage 6: Distributed Model Training (Ray Train + PyTorch)**

* **Script:** train\_sentiment\_model\_lstm.py
* **Technology:** Ray (Ray Data, Ray Train), PyTorch
* **Process:**
  + **Initialization:** Ray is initialized (ray.init).
  + **Load Data & Vocab:** The labeled Parquet data is loaded using ray.data.read\_parquet, and the saved vocabulary (sentiment\_vocab.pkl) is loaded. The actual vocabulary size is determined.
  + **Data Splitting:** The Ray Dataset is split into training, validation, and test sets using train\_test\_split.
  + **Trainer Setup:** A ray.train.torch.TorchTrainer is configured:
    - train\_loop\_per\_worker: Specifies the function containing the training logic executed on each worker.
    - train\_loop\_config: Passes hyperparameters (learning rate, batch size, model dimensions, vocab size, etc.) to the workers.
    - scaling\_config: Defines the number of workers (NUM\_WORKERS) and whether to use GPUs (USE\_GPU).
    - run\_config: Configures experiment tracking, logging, checkpointing (storage\_path, CheckpointConfig). Uses an absolute path for storage\_path.
    - datasets: Passes the *unprocessed* Ray Dataset shards for "train" and "evaluation" to the trainer.
  + **Training Loop (**train\_loop\_per\_worker**):**
    - Each worker receives its data shard (train.get\_dataset\_shard).
    - The LSTM model is instantiated (using the imported class) and prepared for distributed training using train.torch.prepare\_model().
    - Optimizer (Adam) and Loss function (CrossEntropyLoss) are defined.
    - The code iterates through epochs.
    - **Manual Batching & Collation:** Inside each epoch, the worker iterates through its data shard using .iter\_rows(). Rows are collected into a list (current\_batch). When the batch reaches BATCH\_SIZE\_PER\_WORKER, the collate\_batch function is called.
    - collate\_batch**:** This function takes the list of row dictionaries, performs tokenization, numericalization (using the loaded global vocab), padding (pad\_sequence), and converts the data into appropriately shaped PyTorch tensors, moving them to the correct device (CPU/GPU). It returns a dictionary of tensors.
    - **Training Step:** The collated batch tensors are fed into the model, loss is calculated, backpropagation is performed, and the optimizer takes a step.
    - **Evaluation Step:** After each training epoch, the model evaluates on the validation set (also using manual batching and collation). Metrics (loss, accuracy, F1) are calculated.
    - **Reporting:** Metrics and model checkpoints (TorchCheckpoint.from\_state\_dict()) are reported back to Ray Train using train.report(). Ray handles saving the best checkpoints based on the configured checkpoint\_score\_attribute.
  + **Execution:** trainer.fit() starts the distributed training process.

**Stage 7: Model Evaluation and Reporting (Ray + Scikit-learn + Matplotlib)**

* **Script:** train\_sentiment\_model\_lstm.py
* **Technology:** Ray, PyTorch, Scikit-learn, Matplotlib, Seaborn, Pandas
* **Process:**
  + After training completes, the best checkpoint (result.checkpoint) is retrieved.
  + The model's state\_dict is loaded from the checkpoint file path into a fresh instance of the LSTMSentimentClassifier.
  + The model is set to evaluation mode (eval\_model.eval()).
  + Predictions are made on the *test set* (using manual iteration, batching, and collation similar to the validation loop). True labels and the original text are also collected.
  + **Metrics Calculation:** A detailed classification report (precision, recall, F1 per class) and other metrics (accuracy, weighted/macro F1, precision, recall) are calculated using sklearn.metrics.
  + **Saving Results:**
    - The classification report is saved to classification\_report\_lstm.txt.
    - The calculated metrics dictionary is saved to evaluation\_metrics\_lstm.json.
    - A confusion matrix is plotted using Matplotlib/Seaborn and saved as confusion\_matrix.png.
    - A pandas DataFrame containing the cleaned\_text, true labels (ID and string), and predicted labels (ID and string) for the test set is created and saved to test\_predictions\_lstm.csv.
  + **Plotting History:** Training history (loss, accuracy over epochs) retrieved from the result object is plotted using Matplotlib/Seaborn and saved to loss\_history.png and accuracy\_history.png.

**3. Technology Roles and Benefits**

* **Apache Spark:**
  + **Role:** Handles the initial heavy lifting of processing potentially massive raw Twitter data, cleaning text efficiently across multiple cores/nodes, and performing distributed computations like word counting for vocabulary building.
  + **Benefits:** Excellent scalability for large datasets, fault tolerance during data processing, rich DataFrame API for structured data manipulation. Ideal for the ETL (Extract, Transform, Load) phase.
* **Google Gemini API:**
  + **Role:** Acts as a "human-in-the-loop" substitute, providing sentiment labels when none exist. Crucial for bootstrapping a supervised learning task from unlabeled data.
  + **Benefits:** Leverages a powerful pre-trained LLM for labeling, potentially saving significant manual effort.
  + **Caveats:** API calls incur costs and are subject to rate limits. Label quality depends on the LLM's capabilities and the clarity of the prompt. Batching API calls (as implemented) is important for efficiency.
* **Ray (Ray Data & Ray Train):**
  + **Role:** Orchestrates the distributed machine learning phase. Ray Data efficiently loads the processed data from Spark's output (Parquet). Ray Train (specifically TorchTrainer) manages the distributed training process across multiple workers (potentially on GPUs), handling data sharding, model distribution (via DDP), metric aggregation, and checkpointing.
  + **Benefits:** Simplifies distributed ML training, provides scalability for compute-intensive tasks like training deep learning models, efficient resource utilization, fault tolerance during training, seamless integration with PyTorch. The manual batching approach adopted avoids potential issues with Ray Data's automatic collation for variable-length sequences.
* **PyTorch:**
  + **Role:** Provides the framework for defining the neural network architecture (LSTM with Attention) and performing the core model training operations (forward/backward pass, optimization).
  + **Benefits:** Flexible and widely used deep learning library with strong community support and GPU acceleration.
* **Scikit-learn / Matplotlib / Seaborn / Pandas:**
  + **Role:** Standard Python libraries used for post-training analysis: calculating detailed evaluation metrics, creating visualizations (loss curves, confusion matrix), and saving prediction results in a user-friendly format (CSV).
  + **Benefits:** Provide robust and easy-to-use tools for model evaluation and reporting.

**4. Benefits Over Traditional Approaches (with Examples)**

This section highlights the specific advantages compared to a single-machine approach using libraries like Pandas, NLTK/spaCy, and scikit-learn/PyTorch running sequentially.

**Benefits of Using Spark:**

1. **Scalable Data Handling & Cleaning:**
   * **Traditional:** Loading gigabytes/terabytes with pd.read\_json(...) fails due to memory limits. Cleaning via .apply() is slow.
   * **Spark:** Reads distributed data (spark.read.json(...)) and applies cleaning UDFs in parallel across partitions (df.withColumn("cleaned", udf(col("text")))). Handles data larger than RAM.
2. **Distributed Vocabulary Building:**
   * **Traditional:** collections.Counter on large text is slow and memory-heavy.
   * **Spark:** Uses distributed RDD operations (flatMap, map, reduceByKey) for fast, parallel word counting across partitions.
   * # Spark Example
   * words\_rdd = limited\_df.select("cleaned\_text").rdd.flatMap(...)
   * word\_counts = words\_rdd.map(...).reduceByKey(...)
3. **Efficient Parallel API Calls:**
   * **Traditional:** Looping and calling the API one by one (for text in df['text']: call\_gemini(text)) is extremely slow.
   * **Spark +** mapPartitions**:** Distributes the labeling task. Each worker handles a partition and makes *batched* API calls within it, achieving significant parallelization.
   * # Spark Example
   * labeled\_rdd = limited\_rdd.rdd.mapPartitions(get\_sentiment\_batch\_from\_gemini)

**Benefits of Using Ray:**

1. **Scalable & Distributed Model Training:**
   * **Traditional:** Training time is limited by single CPU/GPU speed.
   * **Ray Train (**TorchTrainer**):** Distributes training across multiple workers/GPUs (ScalingConfig(num\_workers=...)), drastically reducing wall-clock time via data parallelism (DDP).
   * # Ray Example
   * trainer = TorchTrainer(train\_loop\_per\_worker=..., scaling\_config=...)
   * result = trainer.fit() # Starts distributed run
2. **Simplified Distributed Setup:**
   * **Traditional:** Requires manual setup of torch.distributed, process groups, DDP wrapping, etc.
   * **Ray Train:** TorchTrainer and train.torch.prepare\_model(model) handle most of the distributed boilerplate automatically.
3. **Integrated Data Handling:**
   * **Ray Data:** Efficiently reads Spark's Parquet output (ray.data.read\_parquet) and provides data shards directly to workers (train.get\_dataset\_shard). The manual iteration (iter\_rows) and collation (collate\_batch) approach bypasses potential issues with automatic batching of variable-length sequences.
4. **Unified Checkpointing & Reporting:**
   * **Traditional (Distributed):** Requires manual logic to gather metrics from all processes and save checkpoints only on the main process.
   * **Ray Train:** Workers use train.report(metrics=..., checkpoint=...). Ray automatically aggregates metrics, manages checkpoint saving based on RunConfig, and identifies the best overall checkpoint.

**Synergy:** Spark handles the large-scale data preparation optimized for I/O and distributed transformations. Ray handles the compute-intensive, iterative task of distributed model training, optimized for ML workloads and resource management.

**5. Code Implementation Overview**

* process\_label\_tweets.py**:**
  + Sets up SparkSession.
  + Defines clean\_tweet\_text UDF.
  + Loads data (full\_text), applies cleaning, filtering, and .limit().
  + Performs distributed word count and saves vocab dictionary.
  + Defines get\_sentiment\_batch\_from\_gemini and process\_single\_batch for batched API calls within mapPartitions.
  + Saves labeled data to Parquet.
* lstm\_model.py**:**
  + Contains only the LSTMSentimentClassifier class definition (LSTM with Attention), making it easily importable and serializable by Ray.
* train\_sentiment\_model\_lstm.py**:**
  + Imports LSTMSentimentClassifier from lstm\_model.py.
  + Loads vocabulary and labeled data (using Ray Data).
  + Splits data into train/validation/test sets (Ray Datasets).
  + Defines collate\_batch for manual tokenization, numericalization, padding, and tensor conversion.
  + Defines train\_loop\_per\_worker containing the PyTorch training/validation logic with manual batch iteration using iter\_rows() and collate\_batch.
  + Configures and runs TorchTrainer.
  + Implements the evaluation logic on the test set, loading the best checkpoint's state dictionary, performing inference with manual batching/collation, calculating metrics, and saving reports, metrics, plots, and predictions CSV.

**6. Conclusion**

This pipeline demonstrates a practical approach to sentiment analysis on unlabeled Twitter data by combining the strengths of different technologies. Spark handles large-scale initial data preparation, the Gemini API provides necessary labels, Ray efficiently scales the distributed training of a custom PyTorch LSTM model, and standard libraries facilitate evaluation. The use of manual batching within the Ray training loop provides fine-grained control over data handling for the sequence model. This modular and scalable design allows for processing large datasets and training effective sentiment analysis models, overcoming the common challenge of missing ground truth labels.