Improving Data Processing Pipelines with Apache Spark

Dhruv Nain
dn2437@nyu.edu

Tianfan Yang
ty2492@nyu.edu

Shuhao Ruan
sr5553@nyu.edu

05/08/2024

1 Introduction

The main objective of this project is to demonstrate how Spark can be used to process large volumes of data efficiently, particularly in the context of web scraping and text classification tasks, and ultimately, to show the comparison of performances. We aim to improve the data processing pipeline, available on GitHub at `https://github.com/VIDA-NYU/wildlife_pipeline`, by using Apache Spark. The pipeline involves tasks such as information extraction, data filtering, and text inference, with the goal of obtaining structured data that can be used for further analysis. Our research goal is to utilize Apache Spark to efficiently extract, filter, classify advertising data, and obtain structured datasets to support future analysis and research.

2 Related Work

In the initial phase of our project, we sought out resources to kickstart the process, with a primary focus on utilizing Apache Spark to refine our data processing pipelines. Given the reliance on ETL disk jobs for data processing within our wildlife studies, the optimization and effectiveness of these jobs are of paramount importance. Consequently, we identified resources that could provide foundational knowledge on establishing ETL pipelines utilizing Spark technology. Among the resources discovered were "Creating Basic ETL Pipelines" by João Pedro and "Creating Your First ETL Pipeline in

Apache Spark and Python" by Adnan Siddiqi. These materials have been instrumental in educating us on the design and development of ETL pipelines using Spark.

Further to creating ETL disk jobs with Spark, it is crucial to explore enhancements in this domain. We found inspiration in improving our data pipelines through Apache Spark from Sujit's work on enhancing data pipelines with Spark. These resources collectively contribute to our project by offering insights and methodologies for leveraging Apache Spark in enhancing ETL disk job performance.

3 Problem Formulation

3.1 Objective

To revamp and optimize an existing data processing pipeline we initially focused on tasks such as information extraction, data filtering, and text inference from web-scraped content. The upgraded pipeline will leverage Apache Spark to handle large volumes of data more efficiently, particularly in data extraction and classification tasks. The ultimate goal is to demonstrate Spark's capabilities in processing large datasets effectively and to compare performance improvements over the existing setup.

3.2 Specific Aims

- Apache Spark Integration
- Information Extraction Improvements
- Data filtering

3.3 Challenges

- **Scalability and Performance:** Ensuring the pipeline can scale efficiently to handle increases in data volume without compromising processing speed or accuracy.
- **Data Quality and Completeness:** Addressing issues related to data quality, such as missing values or inaccuracies in extracted information, particularly when adapting to different web markets.

3.4 Evaluation Metrics

- **Processing Time:** Comparison of data processing times before and after the integration of Apache Spark.
- **Classification Accuracy:** Measurement of the accuracy of the classification models on a benchmark dataset or through cross-validation.
- **Scalability:** Evaluation of the pipeline's ability to scale up to handle larger datasets without significant degradation in performance.

4 Methods, Architecture, and Design

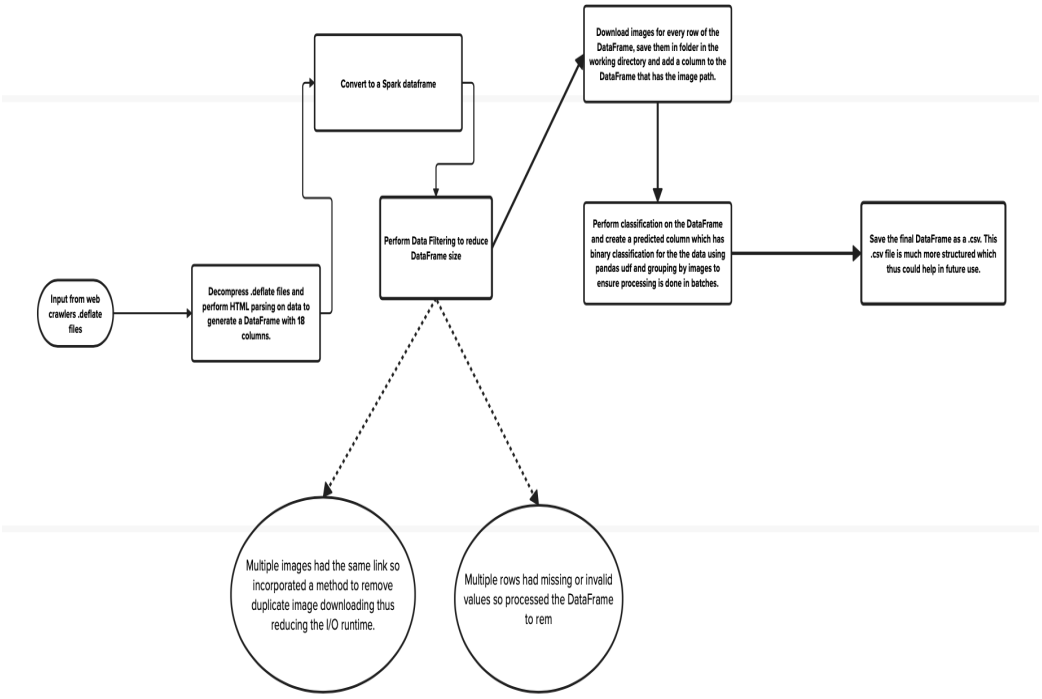

Figure 1: Pipeline Flowchart

4.1 Enhancing the Extraction Process with Spark

Distributed processing of extracted data can significantly speed up the extraction of structured data from raw HTML content, especially when processing large datasets. Integration with MLscraper and extruct allows for simultaneous extraction of product details and metadata across the dataset. After the initial data extraction, the entire process of applying transformation is done by using spark which helps in improving the runtime.

4.2 Optimizing Data Transformation

Custom UDFs for data cleaning, downloading images and performing classification worked wonders for our project.

4.2.1 Data Cleaning

On inspecting the data we noticed a lot of invalid values, missing values or NA values. To improve the runtime of the project we decided to filter data before performing classifications. We noticed that the data frame has multiple image links in the image column that weren't valid so we cleaned the data to remove any rows that had invalid links.

4.2.2 Improving I/O process by Handling Duplicate Images

On further investigation we realized that the image column had multiple duplicate values belonging to different rows of the dataframe with no other duplicate items which depicted that these entries were valid but they just pointed to same image links. Downloading and saving an image is an expensive I/O process and thus it contributed to a major chunk of our runtime. The following explains how the duplicate issue was handled.

1. **Add Unique ID to Manage Duplicates:** A unique identifier (`unique_id`) is added to the DataFrame (`df`) using the `monotonically_increasing_id()` function. This identifier helps manage duplicates and facilitate rejoining after deduplication.

2. **Deduplicate Image URLs:** Duplicate image URLs are removed from the DataFrame (`df`) using the `dropDuplicates(["image"])` method. The first unique ID (`first_unique_id`) associated with each unique image URL is retained.

3. **Download and Save Images for Unique URLs:** The `download_and_save_image_udf` User-Defined Function (UDF) is applied to download and save images for unique URLs. The UDF checks if an image already exists in the specified `image_folder`. If it does not exist, the image is downloaded from the URL and saved with a unique identifier (`image_id`) as its filename.

4. **Join Image Paths Back to Original DataFrame:** The image paths (`image_path`) corresponding to each unique image URL are joined back to the original DataFrame (`df`) using a left outer join. The join is performed based on matching image URLs between the original DataFrame and the image DataFrame (`image_df`).

4.2.3 Finely tune image classification process

1. **Data Filtering:** The function starts by filtering the DataFrame (`df`) to exclude rows where the "image_path" column is null. If the filtered DataFrame is empty, it prints a message and returns the original DataFrame without performing any further classification.

2. **Data Type Conversion:** All columns in the DataFrame are converted to strings using the `col(column_name).cast("string")` operation. This step ensures consistent data types for further processing.

3. **Schema Definition:** A specific schema (`ad_schema`) is defined for the DataFrame to ensure data consistency and enforce column types.

4. **Classifier Loading:** The function loads a classifier using the `maybe_load_classifier` method with a specified task ("zero-shot-classification").

5. **Pandas UDF Definition:** A Pandas User-Defined Function (UDF) named `convert_to_pandas` is defined using `pandas_udf` with the `GROUPED_MAP` type. This UDF takes a grouped DataFrame as input and performs inference and classification operations on it.

6. **Inference and Classification:** Inside the UDF, it retrieves inference data using the `get_inference_data` function, then runs inference using the loaded classifier (`classifier`) and the retrieved data. The inferred

predictions are added to a new column named "predicted_label" in the DataFrame.

This finely tuned image classification process ensures efficient data handling, consistent data types, and accurate inference results for image classification tasks. Our Function which utilized PySpark turned out to be more time-efficient and scalable for handling large-scale data processing and classification tasks compared to the original function that utilized pandas due to its distributed processing capabilities and parallelism.

4.3 Data Storage

The final .csv file that is generated is much more structured as it only included the data where predictions could be made and thus is suitable for any future analysis unlike the .csv file generated by the original pipeline.

5 Results

5.1 Original pipeline runtime for 1 file

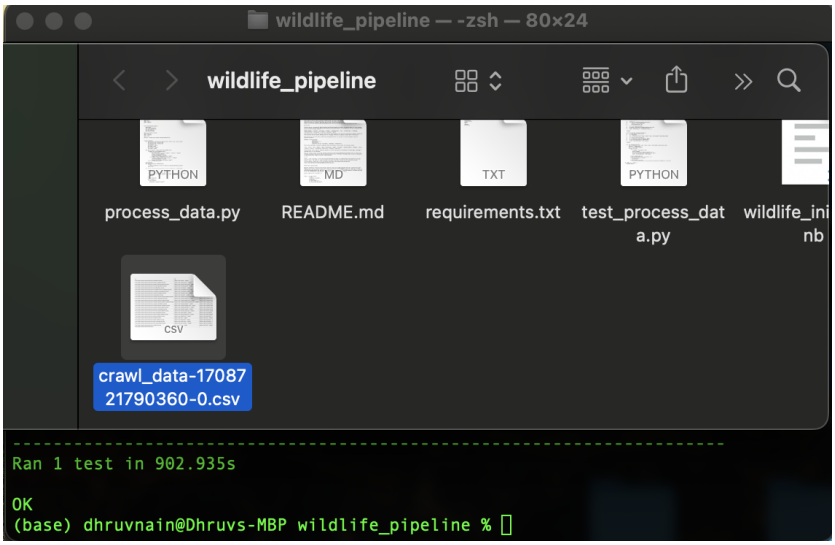

Figure 2: Initial Pipeline takes 902 seconds to process

5.2 Spark pipeline runtime for the same file

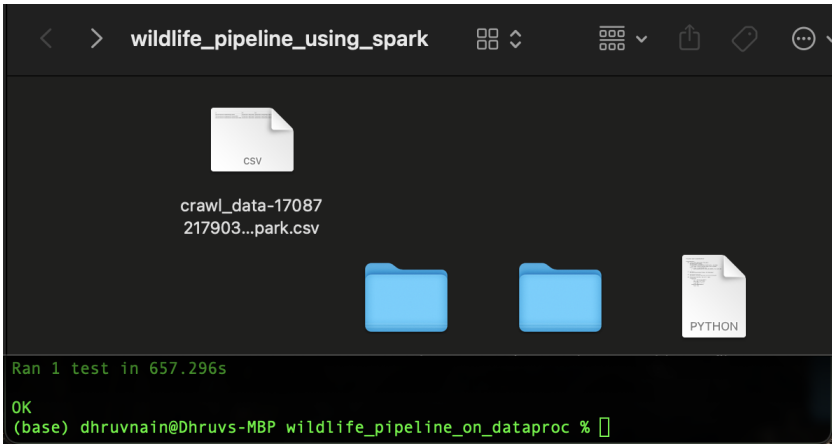

Figure 3: Spark Pipeline takes 657 seconds to process

The images depict the comparison between two data processing pipelines for a wildlife data file: the original and a version optimized with Apache Spark. In the original pipeline, running on a local system without Spark, it takes approximately 902 seconds to process a single data file. However, with the implementation of Spark in the pipeline, the processing time is reduced to 657 seconds. This improvement translates to a reduction in runtime of about 27%, which is substantial considering the complexity of data processing tasks involved. This acceleration is notable on a local machine; thus, one can anticipate even more significant improvements when deployed on a cluster with distributed computing resources. Spark's ability to manage and process large datasets efficiently across multiple nodes can vastly enhance performance, highlighting the potential benefits for scalability and efficiency in data processing tasks.

References

[1] Sujit Code90. Enhancing data pipelines with apache spark. https://medium.com/@sujit.code90/enhancing-data-pipelines-with-apache-spark-4f40a9016ea6, 2024.

[2] Joao Pedro. Creating a simple etl pipeline with apache spark. https://joaopedro214.medium.com/

creating-a-simple-etl-pipeline-with-apache-spark-825cc17c8cf6, 2021.

[3] Pknerd. Create your first etl pipeline in apache spark and python. https://pknerd.medium.com/create-your-first-etl-pipeline-in-apache-spark-and-python-ec3d12e2c169, 2019.

[1] [2] [3]