Algorithms for massive data: Project 3: Link analysis

Pietro Manning

July 2024

Chapter 1

Loading and Analyzing Artworks Dataset

1.1 Loading and Preparing Data

I loaded the dataset `prado.csv` using `pandas`, ensuring that the `work_tags` column was present in the dataset.

In this code, a Pandas DataFrame is created by attempting to load a CSV file using the following command:

```
df = pd.read_csv('prado.csv')
```

If the file `prado.csv` is found, the DataFrame is loaded successfully. Otherwise, an error message is printed indicating that the file was not found.

PageRank Analysis of Prado Museum Data

Step 1: Install PySpark

To begin with, PySpark, the Python API for Apache Spark, was installed using the following command:

```
!pip install pyspark
```

Step 2: Initialize the Spark Session

A Spark session was initialized to enable the use of Spark's functionalities. The session was named *"Prado Museum PageRank"* for clarity in identifying the specific use case. This was done with the following code:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Prado Museum PageRank") \
    .getOrCreate()
```

Step 3: Load the Dataset

The dataset, `prado.csv`, was loaded into a Spark DataFrame. The file was read with headers and schema inference enabled to automatically determine the data types of the columns. This is achieved using the following command:

```
df = spark.read.csv('prado.csv', header=True, inferSchema=True)
```

Step 4: Display the First 5 Records

To verify the contents of the dataset and ensure that it was loaded correctly, the first five records of the DataFrame were displayed using:

```
df.show(5)
```

PySpark Data Processing

Objective

This script performs data processing using PySpark to transform and analyze a dataset of images and their associated tags. The goal is to create edges between images that share tags and to prepare data for further analysis, such as calculating PageRank.

Code and Explanation

- **Import Necessary Functions**

```
from pyspark.sql.functions import col, explode, collect_list,
    split

```

Import functions from PySpark's SQL module needed for data manipulation, including splitting strings, exploding arrays, and collecting lists.

- **Split Tags into Arrays**

```python
# Split the work_tags column into an array of tags
df_tags = df.withColumn("tags_array", split(col("work_tags"),
    ", "))

```

This step splits the `work_tags` column into an array of tags, where each tag is separated by a comma. The result is a new column, `tags_array`, containing these arrays.

- **Explode the Tags Array**

```python
# Explode the tags_array column
df_tags_exploded = df_tags.withColumn("tag", explode(col("
    tags_array")))

```

The `explode` function transforms each element of the `tags_array` into its own row, resulting in a DataFrame where each row contains a single tag.

- **Group by Tag and Collect Images**

```python
# Group by tag and collect all images associated with that tag
df_links = df_tags_exploded.groupBy("tag").agg(collect_list("
    work_url").alias("pictures"))

```

This step groups the DataFrame by the tag and collects all image URLs associated with each tag into a list, creating a new column, `pictures`.

- **Create Edges Between Images**

```python
# Create edges (connections) between images that share at
    least one tag
edges = df_links.select(explode("pictures").alias("src"),
    explode("pictures").alias("dst")).distinct()

```

This creates edges (or connections) between images that share at least one tag by selecting pairs of images from the `pictures` list and ensuring that each pair is unique.

- **Filter Out Self-Loops**

```python
# Filter out self-loops (connections where an image links to
    itself)
edges = edges.filter(col("src") != col("dst"))

```

Self-loops are filtered out to ensure that no image is connected to itself in the graph.

- **Convert to RDD for Further Processing**

```python
# Convert to RDD to use Spark's RDD APIs
edges_rdd = edges.rdd.map(lambda row: (row['src'], row['dst'])
    )

```

Finally, the DataFrame is converted to an RDD (Resilient Distributed Dataset) to leverage Spark's RDD APIs for further processing, such as graph algorithms or other transformations.

PageRank Calculation

Objective

This script calculates the PageRank for a set of web pages represented as an RDD of edges. PageRank is an algorithm used to rank web pages based on their link structure. The process involves iterating to update the rank of each page until convergence.

Code and Explanation

- **Set Parameters**

```python
# Number of iterations for PageRank calculation
num_iterations = 10
alpha = 0.15  # Damping factor

```

Define the number of iterations for the PageRank algorithm and the damping factor (`alpha`), which represents the probability of jumping to a random page.

- **Create an RDD of (Pages, Outgoing Links)**

```python
# Create an RDD of (pages, outgoing links)
links = edges_rdd.distinct().groupByKey().cache()

```

Create an RDD where each element is a pair consisting of a page and a list of its outgoing links. The `distinct()` method ensures that each edge is unique, and `groupByKey()` groups links by page.

- **Initialize Ranks**

```python
# Initialize each node with a rank of 1.0
ranks = links.mapValues(lambda _: 1.0)

```

Initialize each page with a rank of 1.0. This rank will be updated iteratively based on the PageRank formula.

- **Iterate to Update Ranks**

```python
for iteration in range(num_iterations):
    # Distribute the rank of each page across its outgoing
    links
    contribs = links.join(ranks).flatMap(
        lambda url_neighbors_rank: [(dest, url_neighbors_rank
    [1][1] / len(url_neighbors_rank[1][0])) for dest in
    url_neighbors_rank[1][0]]
    )

    # Recalculate the ranks using the damping factor
    ranks = contribs.reduceByKey(lambda x, y: x + y).mapValues
    (lambda rank: alpha + (1 - alpha) * rank)

```

For each iteration, distribute the rank of each page to its outgoing links. The `join()` method combines the link data with the current rank. The `flatMap()` function creates a new RDD of contributions, and `reduceByKey()` aggregates these contributions. Finally, the ranks are recalculated using the damping factor.

- **Sort and Display Results**

```python
# Sort the results by rank
sorted_ranks = ranks.sortBy(lambda x: -x[1])

# Display the top 10 results
print(sorted_ranks.take(10))

```

Sort the pages by their rank in descending order and display the top 10 pages. The `sortBy()` method orders the pages by rank, and `take()` retrieves the top results.

Chapter 2

PageRank Results

2.1 Top Results

The PageRank algorithm was applied to the dataset, and the top results are as follows:

- **Fuente: Museo Nacional del Prado** - Rank: 8.031
- **Mena Marqués** - Rank: 5.823
- **Posada Kubissa** - Rank: 4.848
- **Pérez Sánchez** - Rank: 4.336
- **Museo Nacional del Prado** - Rank: 4.062
- **Ruiz Gómez** - Rank: 4.059
- **Luna** - Rank: 4.014
- **Maurer** - Rank: 3.441
- **Azcue Brea** - Rank: 3.099
- **Albarrán** - Rank: 3.085

2.2 Tabular Results

The results are also presented in tabular format below:

picture_id	**pagerank**
Fuente: Museo Nacional del Prado	8.031017495255742
Mena Marqués	5.8232578238429005
Posada Kubissa	4.847559313804331
Pérez Sánchez	4.335954024759142
Museo Nacional del Prado	4.06153917548966
Ruiz Gómez	4.058671945290737
Luna	4.014261671170429
Maurer	3.440580580961301
Azcue Brea	3.09918514295106
Albarrán	3.0850679578532616

Table 2.1: Top 10 results of PageRank

2.3 Interpretation

The PageRank values indicate the relative importance of each entity based on their connectivity within the graph. Higher PageRank values suggest that an entity is more central or influential, as it is linked to by other high-ranking entities or has a larger number of connections.

- **Fuente: Museo Nacional del Prado** has the highest PageRank, which may suggest that it is a central or highly influential entity within the dataset. This could be due to its extensive connections or high-ranking references.
- Entities such as **Mena Marqués** and **Posada Kubissa** also exhibit high PageRank values, indicating their significant presence or influence within the dataset.
- Lower-ranked entities like **Albarrán** still hold notable PageRank values, indicating their importance but to a lesser extent compared to the top-ranked entities.

The results are consistent with the PageRank algorithm's goal of ranking entities based on their connectivity and importance. The higher PageRank values for certain entities reflect their prominence and centrality in the network.

PageRank Calculation and Normalization

Objective

This script performs PageRank calculations on a graph of web pages, normalizes the PageRank values, and saves the results to a CSV file. The PageRank algorithm is used to rank web pages based on their link structure, and normalization ensures that the ranks are proportional and can be easily interpreted.

Code and Explanation

- **Import and Initialize Spark**

```python
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder \
    .appName("PageRank") \
    .getOrCreate()

```

Import the necessary SparkSession class and create a Spark session named *"PageRank"*. This session is used to interact with Spark and perform data processing.

- **Define Parameters and Prepare Data**

```python
# Assume that edges_rdd is already defined

# Number of iterations and damping factor
num_iterations = 10
alpha = 0.15

# Create an RDD of (pages, outgoing links)
links = edges_rdd.distinct().groupByKey().cache()

# Initialize each node with a rank of 1.0
ranks = links.mapValues(lambda _: 1.0)

```

Define the number of iterations for the PageRank algorithm and the damping factor (`alpha`). Create an RDD where each element is a pair of a page and its outgoing links. Initialize each page with a rank of 1.0.

- **Iterate to Update Ranks**

```python
for iteration in range(num_iterations):
    # Distribute the rank of each page across its outgoing
    links
    contribs = links.join(ranks).flatMap(
        lambda url_neighbors_rank: [(dest, url_neighbors_rank
    [1][1] / len(url_neighbors_rank[1][0])) for dest in
    url_neighbors_rank[1][0]]
    )

    # Recalculate the ranks using the damping factor
    ranks = contribs.reduceByKey(lambda x, y: x + y).mapValues
    (lambda rank: alpha + (1 - alpha) * rank)

```

In each iteration, distribute the rank of each page to its outgoing links and recalculate the ranks using the PageRank formula with the damping factor.

- **Calculate and Print Total PageRank**

```python
# Find the total sum of ranks
total_rank = ranks.map(lambda x: x[1]).sum()
print(f"Total PageRank (before normalization): {total_rank}")

```

Compute the total sum of all ranks before normalization and print the result.

- **Normalize the Ranks**

```python
# Normalize the ranks
normalized_ranks = ranks.mapValues(lambda rank: rank /
    total_rank)

# Find the total sum of normalized ranks
total_normalized_rank = normalized_ranks.map(lambda x: x[1]).
    sum()
print(f"Total PageRank (normalized): {total_normalized_rank}")

```

Normalize the ranks by dividing each rank by the total rank. Compute and print the total sum of the normalized ranks to ensure correctness.

- **Sort and Save Results**

```python
# Sort the results by rank
sorted_ranks = normalized_ranks.sortBy(lambda x: -x[1])

# Display the top 10 results
sorted_ranks_df = sorted_ranks.toDF(["picture_id", "pagerank"
    ])
sorted_ranks_df.show(10)

# Save the results to a CSV file
sorted_ranks_df.write.mode("overwrite").csv('/content/
    prado_pagerank_normalized.csv')

```

Sort the normalized ranks in descending order

- **Sort and Save Results (continued)**

```python
# Sort the results by rank
sorted_ranks = normalized_ranks.sortBy(lambda x: -x[1])

# Display the top 10 results
sorted_ranks_df = sorted_ranks.toDF(["picture_id", "pagerank"
    ])
sorted_ranks_df.show(10)

# Save the results to a CSV file
sorted_ranks_df.write.mode("overwrite").csv('/content/
    prado_pagerank_normalized.csv')

```

Sort the normalized ranks in descending order to prioritize higher-ranked pages. Convert the sorted RDD to a DataFrame with columns `picture_id` and `pagerank`. Display the top 10 results and save the DataFrame to a CSV file for further use or analysis.

Chapter 3

Normalized PageRank Results

3.1 Total PageRank

The total PageRank before normalization was 2087.0, and after normalization, the total PageRank is approximately 1.0. This normalization process ensures that the sum of all PageRank values is scaled to 1, making it easier to compare the relative importance of entities in the dataset.

3.2 Top Normalized PageRank Results

The normalized PageRank results are presented below:

picture_id	**pagerank**
Fuente: Museo Nacional del Prado	0.003848115714066
Mena Marqués	0.002790252910322
Posada Kubissa	0.002322740447438
Pérez Sánchez	0.002077601353502
Museo Nacional del Prado	0.001946113644221
Ruiz Gómez	0.001944739791706
Luna	0.001923460312012
Maurer	0.001648577183019
Azcue Brea	0.001484995276929
Albarrán	0.001478230933326

Table 3.1: Top 10 normalized PageRank results

3.3 Interpretation

The normalized PageRank values reflect the relative importance of each entity in a way that is scaled to a total of 1. This normalization process is essential for comparing the influence of different entities on a common scale.

3.3.1 Why are the Values Lower?

Normalized PageRank values are generally lower than their raw PageRank counterparts because they are scaled relative to the total PageRank of all entities. The raw PageRank values are summed to a larger total (2087.0 in this case), while after normalization, the total is scaled to 1. This scaling makes the individual values smaller but ensures they sum to one, which helps in analyzing their relative importance more effectively.

3.3.2 Sum Close to One

The sum of normalized PageRank values being close to one is a direct result of the normalization process. By dividing each PageRank value by the total PageRank, the values are adjusted so that their total is exactly 1. This is a standard practice in PageRank algorithms to ensure that the results are proportionate and easy to interpret, with the sum providing a meaningful comparison across different datasets.

3.3.3 Relative Importance

Despite the smaller values, the relative importance of entities remains the same. For instance, **Fuente: Museo Nacional del Prado** still holds the highest normalized PageRank, indicating its relative influence compared to other entities in the dataset.

Chapter 4

Scalability

4.1 Introduction

Scalability is a critical factor in the performance and efficiency of distributed computing systems, particularly when handling large datasets. In the context of graph algorithms like PageRank, scalability ensures that the algorithm can handle growing amounts of data without significant degradation in performance. This chapter discusses the scalability of the PageRank algorithm implemented using Apache Spark, highlighting how Spark's architecture and features contribute to its ability to scale effectively.

4.2 Scalability in Apache Spark

Apache Spark is designed to scale horizontally, which means that it can handle increasing amounts of data by adding more computing resources (i.e., nodes) to the cluster. Several factors contribute to Spark's scalability:

4.2.1 Distributed Computing

Spark distributes data and computation across multiple nodes in a cluster. When executing the PageRank algorithm, the dataset is divided into partitions, each of which is processed independently on different nodes. This distribution of data and tasks allows Spark to leverage the combined processing power of all nodes, making it capable of handling large-scale datasets efficiently.

4.2.2 Resilient Distributed Datasets (RDDs)

RDDs are a fundamental abstraction in Spark that provide fault tolerance and parallel processing. RDDs enable the PageRank algorithm to be expressed in a distributed manner. The algorithm operates on RDDs of edges and ranks, performing transformations and actions that are executed in parallel across the cluster. This parallelism is key to achieving scalability, as it allows the algorithm to process large volumes of data concurrently.

4.2.3 Efficient Data Storage and Processing

Spark's in-memory processing capability significantly improves performance by reducing the need for disk I/O. Intermediate results and data are cached in memory, which accelerates iterative algorithms like PageRank that require multiple passes over the data. Additionally, Spark's use of efficient data storage formats and compression techniques further enhances its ability to handle large datasets.

4.2.4 Load Balancing and Fault Tolerance

Spark automatically handles load balancing and fault tolerance. If a node fails, Spark can recompute lost data from other nodes or use lineage information to recover from failures. This capability ensures that the PageRank algorithm can continue to execute smoothly even in the presence of hardware or software failures, contributing to its overall scalability.

4.3 Scalability of PageRank Algorithm

The PageRank algorithm benefits from Spark's scalability features in several ways:

4.3.1 Parallelization of Computation

PageRank involves iterative computation where each iteration updates the rank of pages based on their outgoing links. Spark's ability to parallelize this computation across multiple nodes allows it to handle large graphs efficiently. Each iteration is executed in parallel, with contributions from multiple pages aggregated and updated concurrently.

4.3.2 Handling Large Graphs

Large-scale graphs can be processed effectively due to Spark's ability to partition the graph data and distribute it across the cluster. This partitioning ensures that the computation is distributed and that each node processes a manageable subset of the data. As the size of the graph grows, additional nodes can be added to the cluster to maintain performance and scalability.

4.3.3 Efficient Rank Updates

The PageRank algorithm requires frequent updates to the ranks of pages. Spark's in-memory processing and efficient data management enable rapid updates and recalculations of ranks during each iteration. This efficiency is crucial for handling large numbers of iterations and ensuring that the algorithm converges in a reasonable time frame.

4.4 Conclusion

In summary, the PageRank algorithm's scalability is well-supported by Apache Spark's distributed computing architecture, RDDs, in-memory processing, and fault tolerance mechanisms. These features enable Spark to handle large-scale graph data and iterative computations efficiently, making it suitable for applications requiring high scalability and performance.