Prado Museum Pictures - Link Analysis

Edoardo Vergani

September 12, 2024

Contents

1 Introduction

The goal of this project is to develop a ranking system for paintings in Madrid's Prado Museum, using a graph-based approach to analyze the relationships between artworks. Specifically, the relationships between paintings are defined by common tags, which represent thematic, stylistic, or descriptive attributes in the dataset. By checking these connections, the aim is to assess the relative importance of each painting within the collection.

To accomplish this, a PageRank algorithm, a method for ranking nodes in a graph, based on the structure of the relationships between them, is used. In this context, paintings are treated as nodes, and edges are created between them when they share at least one common tag. The ranking system is built using Apache Spark's distributed computing capabilities, employing the MapReduce programming model within a PySpark environment. By representing the dataset as Resilient Distributed Datasets (RDDs) and DataFrames, efficient processing and scalability for analyzing large datasets it's ensured.

2 Dataset

For this project, I have used the dataset 'Prado Museum Pictures' from Kaggle. This dataset contains thirty columns of information about the museum's paintings, including the author and their bio, the title, a description, and other technical details. The data was accessed using the Kaggle API.

For this analysis, I focused on two specific columns: *work url* and *work tag*. The *work url* column contains links to the corresponding pages on the museum's website, where the paintings can be viewed. The *work tag* column provides all the tags assigned to each piece of art, separated by semicolons. These tags include information such as the material, a brief description, a date, a technique, and more.

Since the main objective is to identify linkages among different pieces of art, we assume that two artworks are connected if they share a common tag. However, the *work tag* column required cleaning to ensure relevance, so I removed unnecessary symbols, such as '+' and extra spaces.

After cleaning, I split each line into a list of fields and transformed the dataset into a new format, where each row represents a unique combination of *work url* and *work tag*.

2.1 Data organization

PySpark, the Python API for Apache Spark, provides two main abstractions for distributed data processing: Resilient Distributed Datasets (RDDs) and DataFrames. Both have their unique characteristics and use cases.
RDDs are:

- **Immutable and Distributed:**
  - RDDs are immutable, meaning once an RDD is created, it cannot be modified. Any transformation applied to an RDD results in the creation of a new RDD.
  - RDDs are distributed across the cluster, allowing for parallel processing.
- **Fault Tolerant:**
  - RDDs are inherently fault-tolerant. They keep track of the transformations used to build the dataset (the lineage) so that if any partition of an RDD is lost, it can be recomputed from the original dataset.
- **Transformation and Action Operations:**
  - Transformations (e.g., `map`, `filter`, `flatMap`, `reduceByKey`) are lazy operations that define a new RDD from an existing one.
  - Actions (e.g., `collect`, `count`, `take`, `reduce`) trigger the execution of the transformations to return a result or write data to storage.
- **Flexibility:**
  - RDDs can handle both structured and unstructured data. They offer low-level transformation and action operations for fine-grained control over the data processing tasks.

- **No Schema:**
  - RDDs do not have a schema, making them less efficient for operations involving structured data or SQL queries.

While DataFrames are:

- **Schema-based:**
  - DataFrames are distributed collections of data organized into named columns, similar to tables in a relational database. They have a schema that defines the data types of each column.
- **Optimized Execution:**
  - DataFrames use Catalyst, Spark's query optimizer, to optimize query execution. This results in better performance, especially for complex queries.
- **Higher-Level API:**
  - DataFrames provide a higher-level API for structured data manipulation. They offer expressive domain-specific language operations (e.g., `select`, `filter`, `groupBy`, `agg`) similar to SQL.
- **Integration with Spark SQL:**
  - DataFrames can be seamlessly used with Spark SQL, enabling the execution of SQL queries on distributed data.
- **Interoperability:**
  - DataFrames can be easily converted to RDDs if needed, and vice versa. This allows for flexibility in choosing the right abstraction for the task at hand.

3 The PageRank algorithm

The algorithm employed for this analysis is the PageRank algorithm, which ranks web pages in search engine results according to their importance. This algorithm is a type of Markov chain that models the behaviour of a random surfer who follows links at random. In this project, the PageRank algorithm is applied to a dataset of artworks from the Prado Museum, using PySpark to process the data and compute the rankings efficiently.

Each edge $i \to j$ indicates that picture i is linked to the picture j. To determine the score of a piece of art i, denoted as r_i, I have used the following formula:

$$r_i = \sum_{j \to i} \frac{r_j}{d_j}$$

This means that the relevance score of the page i, r_i, is calculated as a weighted sum of the relevance scores of all pages linking to i. Each page linking to i is weighted according to its out-degree d_j (the number of pages it links to). Without knowing the relevance of all pages linking to i, we cannot determine the relevance of i.

We define the transition matrix M, where each entry m_{ij} in row i and column j is $1/d$ if product j has d outbound links and one of them is to node i, otherwise it takes value 0.Each entry in the matrix represents the probability distribution of a random surfer reaching product j after several steps. Starting from a vector v_0 where each component is $1/n$ for each component, after one step the distribution of the surfer will be Mv_0, after two steps M^2v_0, and so on. The probability x_i that a random surfer will be at node i at the next step, is $\sum_j m_{ij}v_j$.

Here, m_{ij} is the probability that a surfer at node j will move to node i at the next step, and v_j is the probability that the surfer was at node j at the previous step.

This iterative process leads to the concept of a Markov chain. Consequently, the surfer will approach a limit distribution v that satisfies $v = Mv$. This is true under two conditions: the nodes are strongly connected, and there are no dead ends.
It is important to note that the transition matrix is column-wise stochastic, meaning that the sum of the elements in each column is 1. As a column-wise stochastic matrix, the eigenvalue associated with the principal eigenvector is 1. The principal eigenvector v_i is the probability vector that remains unchanged at step $i + 1$.

4 Scalability

In order to address scalability issues of the code, I've used Apache Spark's distributed computing capabilities.
By choosing Spark for distributed processing, the code initializes a Spark session and uses Spark's RDD and DataFrame APIs, designed for distributed computing and handling large datasets across a cluster of machines.

The dataset is then read into an RDD with a specified minimum number of partitions, allowing Spark to parallelize the reading and processing of the file, improving performance on larger datasets.

The two types of operations that are also used on RDDs in order to address scalability are trasformation and action operations: the code makes extensive use of Spark transformations (*filter, map, flatMap*, etc.) and actions (*collect, reduceByKey*, etc.), which are fundamental for processing large datasets in a distributed manner.

5 Description of the process

The project starts with the Kaggle API credentials insert, in order to allow the use of the API for subsequently downloading the necessary data from the website.
Once that part is settled, I've proceeded with the download of the dataframe through the use of

```
kaggle.api.dataset_download_file('maparla/prado-museum-pictures','prado.csv')
```

Once the dataset is download as a .zip file, it gets extracted and loaded as a pandas dataset.
This allowed me to select the necessary columns in a more comfortable way, and after that I've transformed the pandas dataframe back to a .csv, in order to load it into pyspark with only the two necessary columns.

Let's start by creating a Spark session. For this purpose, I've used the following code line:

```
from pyspark.sql import Row
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import explode, split, regexp_replace, col, trim

spark = SparkSession.builder.appName("PradoMuseumPictures").getOrCreate()
```

The textFile method reads the CSV file into an RDD (Resilient Distributed Dataset). RDDs are the fundamental data structure of Spark. Here, we specify a minimum of 8 partitions to parallelize the data processing.
Let's define the schema for the DataFrame, specifying two fields: *id* (identifier for the artwork) and *work tags* (tags associated with the artwork).

```
rdd = spark.sparkContext.textFile('prado_selected.csv', minPartitions=8)

schema = StructType([
    StructField("id", StringType(), True),
    StructField("work_tags", StringType(), True)
])
```

5.1 Pre-processing

Let's extract and remove the header row from the RDD. Each line in the RDD is split into fields based on the comma delimiter.
With this line, I map the split fields into Row objects to prepare for DataFrame creation.

```
header = rdd.first()
data_rdd = rdd.filter(lambda row: row != header).map(lambda line: line.split(","))

row_rdd = data_rdd.map(lambda fields: Row(id=fields[0], work_tags=fields[1]))
```

Let's convert the Row RDD into a DataFrame using the predefined schema.
The explode function is used to split the work tags column into individual tags, creating a new row for each tag. This is necessary for associating each tag separately with its corresponding artwork.

```
df = spark.createDataFrame(row_rdd, schema)

exploded_df = df.withColumn("work_tag", explode(split(col("work_tags"), ';')))
```

I've then cleaned the tags by using the regexp_replace function that removes unwanted characters (+ and ") from the work_tag column, and then I've removed any row where the work_tag column is empty after the whitespace trimming.

```
cleaned_df = exploded_df.withColumn('work_tag', regexp_replace('work_tag', r'[+"\"]', ''))

cleaned_df = cleaned_df.filter(trim(col('work_tag')) != '')
```

Now I will convert the cleaned DataFrame back into an RDD for further processing.
The groupByKey function groups the artworks by their tags, collecting all artwork IDs associated with each tag into a list.
I've filtered out tags that are associated with only one artwork, as these do not contribute to the linkage needed for PageRank computation.

```
exploded_rdd = cleaned_df.rdd

grouped_rdd = exploded_rdd.map(lambda row: (row.work_tag,
    row.id)).groupByKey().mapValues(list)

filtered_rdd = grouped_rdd.filter(lambda x: len(x[1]) > 1)
```

The next step is to transform the list of products in (k, v) pairs where k and v are existing links in the network (e.g., we start from $(k, [v_1, v_2, v_3])$ and we end up with $(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_2), (v_3, v_1), (v_2, v_1)$). To do that, we define a new custom function that returns each possible combination of the value list given as input.

```
import itertools
from itertools import combinations

def combination(row):
    pairs = list(combinations(row[1], 2))
    return pairs + [(b, a) for a, b in pairs]

linkage_rdd = filtered_rdd.flatMap(combination)
```

As it is possible to notice from the above section, I used the *flatMap* function to transform a list of lists into a single list of tuples. This step allowed me to convert the data into the desired form: transitioning from a list of lists to a flat list. These transformations were necessary to prepare the data for input into the PageRank algorithm.

5.2 PageRank Implementation

Now, I need to define two important variables: the total number of nodes and the out-degree of each node.
The number of nodes is easily obtained by counting the number of distinct tuples grouped by key, which allows me to determine the cardinality of the nodes.

```
total_nodes = linkage_rdd.flatMap(lambda x: [x[0], x[1]]).distinct().count()
```

The degree of a node represents the number of connections (edges) it has. I mapped each node to a count of 1 for each occurrence and then summed these counts using *reduceByKey* to get the degree of each node. I collected the results as a dictionary for easy lookup.

```
id2degree = linkage_rdd.map(lambda x: (x[0], 1)).reduceByKey(lambda x, y: x +
    y).collectAsMap()
```

Since the probability that a random surfer starts from any random node is $\frac{1}{\text{tot. nodes}}$, I initialized the probability vector with $\frac{1}{\text{tot. nodes}}$ for each entry. I used a dictionary, where the key represents the product ID and the value represents the probability. This allows me to update the probability vector v during each iteration of the PageRank algorithm..

```
p2diz = {node: 1 / total_nodes for node in id2degree.keys()}
```

Next, I computed the sparse transition matrix as a list of tuples in the form (i, j, m_{ij}).

```
P = linkage_rdd.map(lambda x: (x[0], x[1], 1 / id2degree[x[0]]))
PT = P.map(lambda x: (x[1], x[0], x[2]))
```

Finally, I implemented the PageRank iteration to rank the products in relation to one another.
I've also introduced a damping factor to prevent dead-end cycles, and a stopping condition based on comparison between rank vector through the different iterations.
The tolerance level, epsilon, is a small value used to determine when the PageRank algorithm has converged. It sets a threshold for how small the difference between the PageRank values from one iteration to the next can be before stopping the iterations. If the difference is less than epsilon for all nodes, the algorithm is considered to have converged.
I've set the number of maximum iterations to 10 anyway due to computation limitations on Colab, but it's possible to change it in order to make it any value.

```
epsilon = 1e-6 # Tolerance level
alpha = 0.85 # Damping factor
max_iterations = 10 # to prevent infinite loops in case of non-convergence

iteration_count = 0

while True:
    iteration_count += 1
    new_p = PT.map(lambda x: (x[0], alpha * x[2] * p2diz.get(x[1], 0)))\
             .reduceByKey(lambda x, y: x + y)\
             .collect()

    converged = True
    for idx, prb in new_p:
        if abs(p2diz[idx] - prb) > epsilon:
            converged = False
        p2diz[idx] = prb + (1 - alpha) / total_nodes

    if converged:
        print(f"Converged after {iteration_count} iterations.")
        break
    if iteration_count >= max_iterations:
        print(f"Reached maximum iterations ({max_iterations}) without convergence.")
        break
```

As shown in the code above, I updated the dictionary of probabilities during each iteration. After 10 rounds, I was able to determine the random surfer's position within the network of products.

6 Comments and discussion on the experimental results

I will subsequently explore with the given list the results of the PageRank algorithm, as explained in the paper, applied to the works from the Prado Museum in Madrid. It is a dataset of considerable size, and despite this, the algorithm is able to process it, provided it is given the computational capabilities to do so.

Here, I've listed the result of the probabilities of the first 20 nodes of the network.

```
Highest-ranked pictures based on PageRank:
With PageRank: 0.00014992517978630073, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/sagrada-familia-con-san-juanito-y-santa-catalina/ad7a7cd
With PageRank: 0.00014647587259316984, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/apolo-servido-por-las-ninfas/222697ef-7345-446a-b0e1-033
With PageRank: 0.00014459240145153445, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/virgen-con-el-nio-sentado-en-su-regazo-enmarcada/243bb02
With PageRank: 0.00014459240145153445, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/el-sueo-de-san-jose-o-la-muerte-de-san-francisco/bef45e1
With PageRank: 0.00014459240145153442, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/anotacion-sobre-la-boda-del-artista-referencia-a/d67b5f8
With PageRank: 0.00014367876995063418, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/estragos-de-la-guerra/9a90b597-ff67-4032-bfb4-aa06f711d0
With PageRank: 0.00014367876995063418, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/y-no-hai-remedio/1e681912-6be1-4104-9c2a-f3ba2f93edae
With PageRank: 0.00014355554957207732, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/proyecto-de-decoracion-arquitectonica-para-un/81710e09-8
With PageRank: 0.00014291473902351768, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/busto-de-rodrigo-calderon/dec05112-e484-4798-b209-115dc3
With PageRank: 0.00014291290652545337, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/angeles-nios-volando/9fac7fd3-eb0c-4098-94a7-94e8c920512
With PageRank: 0.0001425955491098106, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/figura-sedente-de-fraile-tonsurado-y-barbado/a7e477cf-aa
With PageRank: 0.0001425955491098106, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/registro-de-nacimiento-y-bautismo-de-dos-hijos-de/c8724c
With PageRank: 0.00014231723281380698, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/cuaderno-italiano-iii/95d80046-3e26-44ff-8350-1cdd13b9ba
With PageRank: 0.00014231723281380698, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/cuaderno-italiano-i/9f201d18-a7a0-4a39-bbe9-bf797cdd71b6
With PageRank: 0.00014231723281380696, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/cuaderno-italiano-ii/fc1e7d2f-6896-44bd-a547-aed54f23bd7
With PageRank: 0.0001423044072633783, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/traza-de-arquitectura-para-retablo-o-baldaquino/bcc18817
With PageRank: 0.00014220303839510885, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/un-religioso-curando-o-resucitando-a-un-hombre/bd9fa2e4-
With PageRank: 0.00013860960099747364, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/el-sacrificio-de-ifigenia-caricaturas/09fea329-40ca-4ffd
With PageRank: 0.00013658089467508483, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/retrato-mortuorio-del-periodista-pedro-avial/e0ee4b55-a0
With PageRank: 0.00013506916494018443, Picture ID:
    https://www.museodelprado.es/coleccion/obra-de-arte/cuaderno-de-dibujos/aa8de096-669c-4470-b269-2963f8f7cd82
```

The PageRank returns rather low probabilities even in the cases with the highest values, both due to the number of works present and the quality of the existing tags in the dataset, which are often very detailed and therefore difficult to associate with a large number of works. In any case, the algorithm performs its task computationally efficiently and provides us with understandable results.