Report - Big Data
Page Rank

Alexis Evaristo
Hector Baril

February 11, 2026

Polytech Lille, Computer Science, Statistics, and Artificial Intelligence

Contents

1 Introduction

The objective of this project is to implement and analyze the PageRank algorithm using the power iteration method. We apply this method to a real-world dataset from the *Wikispeedia* project, provided by SNAP (Stanford Network Analysis Project).

PageRank, initially designed to rank web pages, measures the relative importance of a node within a graph. In this report, we first detail the data processing and the construction of the transition matrix. Next, we analyze the results of the standard algorithm by studying its convergence and the distribution of scores (Scan & Add). Finally, we explore a variation of the algorithm via Personalized PageRank (PPR) to highlight specific themes.

2 The Data

2.1 Description and Structure

The data used comes from the archive `wikispeedia_paths-and-graph.tar.gz`. For this study, we focused primarily on the file `paths_finished.tsv`. This file contains complete navigation paths taken by human users attempting to link two given Wikipedia articles.

Each line in the file represents a successful navigation session. The raw data requires significant preprocessing, particularly to handle backtracks (represented by the character `<`).

2.2 Cleaning and Management

The construction of the adjacency matrix L was performed in Python. The main challenge lies in interpreting the navigation paths.

We implemented a **stack** logic to simulate the user journey:

- When a page is visited, it is added to the stack.
- When the symbol `<` is encountered, the last page is removed from the stack (backtracking).
- A link is created (value 1 in the matrix) between the previous page (top of the stack before moving) and the current page.

This process yields a matrix that faithfully represents the actual clicks made by users.

2.3 Tools Used

The project was carried out using the **Python** language. The main libraries used are:

- `Numpy`: For matrix operations and linear algebra.
- `Matplotlib`: For generating graphs (convergence curves, Scan & Add).
- `Csv`: For optimized reading of data files.

3 Standard PageRank

3.1 Choice of Data Structures

We opted to use `Numpy` arrays to represent the adjacency matrix and probability vectors. A bidirectional mapping (Dictionary `Name` $\rightarrow$ `Index` and List `Index` $\rightarrow$ `Name`) was set up to link matrix indices to article titles.

The implemented algorithm follows the iterative formula of the power method:

$$q_{k+1} = \beta P q_k + (1 - \beta)\mathbf{v}\mathbf{e}^t q_k$$

Where P is the transpose of the row-normalized adjacency matrix, β is the damping factor, N is the total number of pages, $\mathbf{v} = (1/N, 1/N, ..., 1/N)^t$, and $\mathbf{e} = (1, 1, ..., 1)$.

The empty columns (filled with zeros) of P will be filled by the factor $1/N$.

Note that q_k must be normalized at the end of each iteration (we will use $\|q_k\|_1$, the L1 norm).

For simple PageRank, after simplification, we obtain the following formula:

$$q_{k+1} = \beta P q_k + \frac{(1-\beta)}{N} \sum_{i=1}^{N} q_i \ . \ (1, 1, ..., 1)^t$$

3.2 Experiments and Results

3.2.1 Convergence Analysis

We studied the impact of parameters β (damping factor) and ϵ (precision) on the speed of convergence.

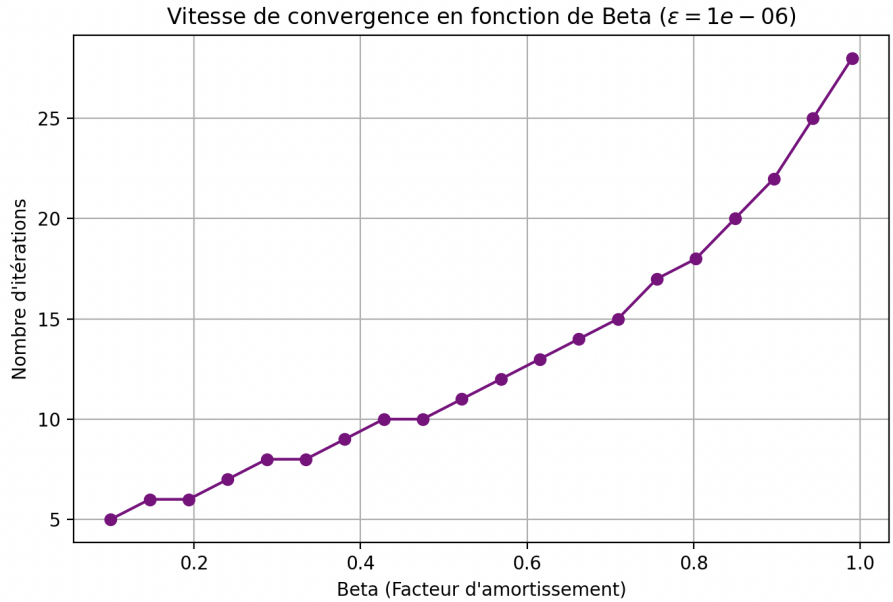

Figure 1: Convergence speed of simple PageRank as a function of β

Interpretation: *We observe that the closer β is to 1, the slower the convergence, because the weight of the structural matrix is stronger relative to the teleportation term.*

3.2.2 Precision (ϵ)

The following figure illustrates the impact of the required precision on the number of iterations.

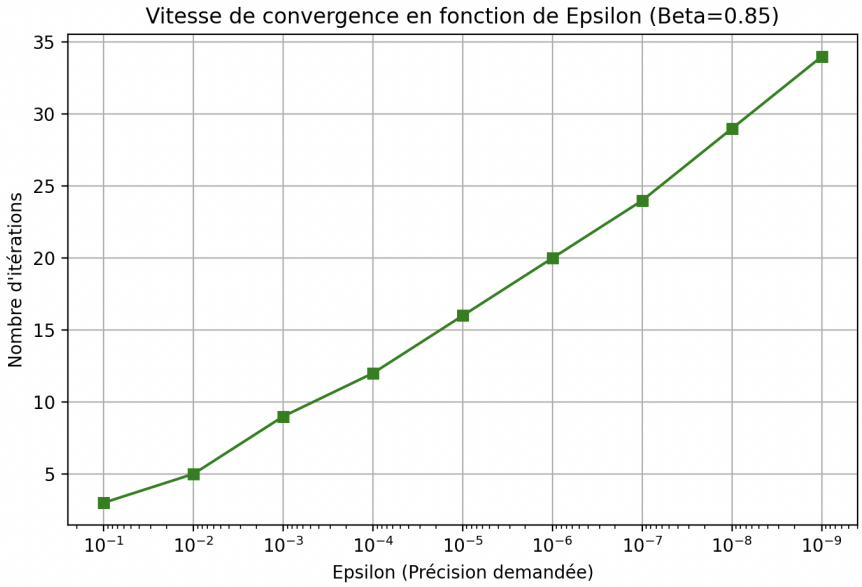

Figure 2: Number of iterations required to converge according to precision ϵ.

3.2.3 Scan & Add Analysis

The "Scan & Add" method allows visualizing the concentration of information in the PageRank vector.

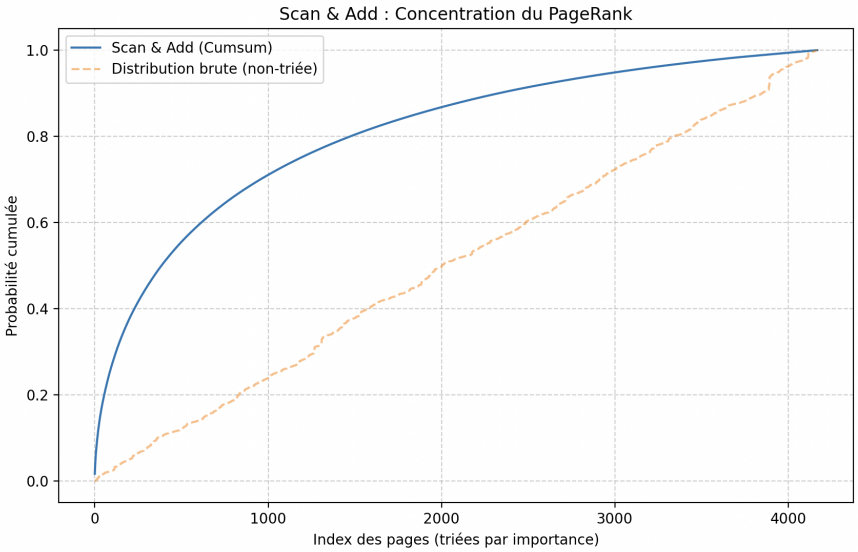

Figure 3: Scan & Add curve (Cumulative sum of sorted scores).

Analysis: The curve shows a strong concentration of importance on a reduced number of pages.

3.2.4 Computation time and iteration count vs β

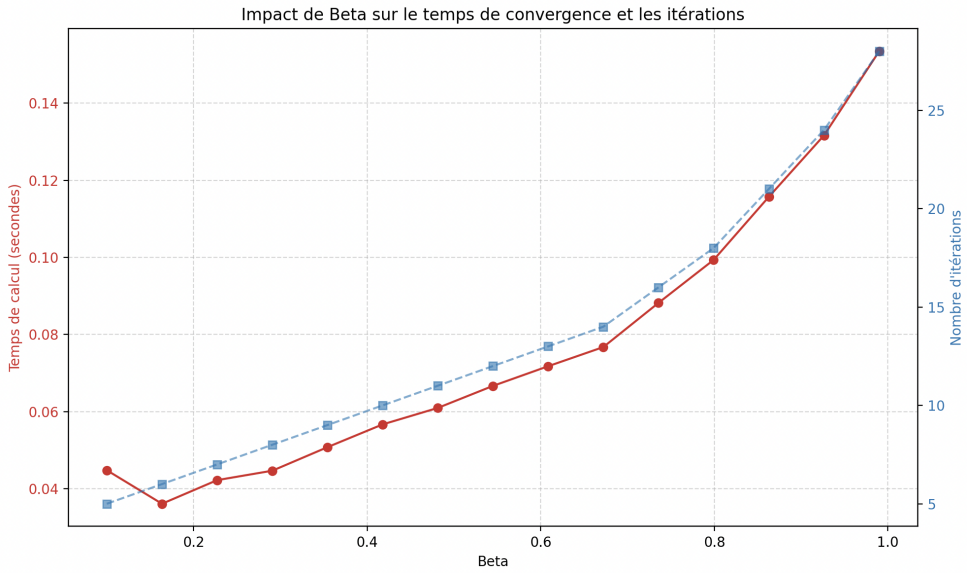

Figure 4: Evolution of convergence as a function of β (ϵ = 1e-6)

3.3 Top 20 Analysis

Here are the 20 most important pages identified by the algorithm with $\beta = 0.85$.

Rank	Page	Score
1	United_States	0.016869
2	Europe	0.009784
3	United_Kingdom	0.009187
4	England	0.007301
5	World_War_II	0.006960
6	France	0.006916
7	Germany	0.005464
8	English_language	0.005241
9	Africa	0.004917
10	India	0.004394
11	Latin	0.004352
12	Japan	0.004153
13	China	0.004069
14	United_Nations	0.003965
15	North_America	0.003824
16	Italy	0.003818
17	London	0.003752
18	Russia	0.003717
19	Earth	0.003513
20	Australia	0.003368

Table 1: Top 20 pages according to standard PageRank ($\beta = 0.85$).

We notice a predominance of countries and major geographical concepts, which is explained by the nature of the Wikispeedia game where these pages serve as navigation "hubs".

4 Personalized PageRank (PPR)

4.1 Methodology

Personalized PageRank modifies the teleportation term. Instead of redistributing uniformly across all pages in the event of teleportation, the random surfer preferentially returns to a specific subset of pages (the personalization vector v).

We tested this approach with a weighting factor $s = 3$ and by targeting specific pages.

4.2 Experimentation on themes

We chose to personalize the vector on the following nodes: *[Russia, Communism, Socialism]*.

Rank	Page	Score
1	Communism	0.054928
2	Russia	0.054851
3	Socialism	0.053294
4	United_States	0.013942
5	Europe	0.009572
6	World_War_II	0.009286
7	United_Kingdom	0.008054
8	Soviet_Union	0.006977
9	France	0.006590
10	China	0.005055
11	Germany	0.004870
12	World_War_I	0.004797
13	United_Nations	0.004650
14	England	0.004576
15	People_Republic_of_China	0.004546
16	Italy	0.004298
17	English_language	0.004244
18	Marxism	0.004217
19	Vladimir_Lenin	0.004118
20	Capitalism	0.004075

Table 2: Top 20 pages according to Personalized PageRank (PPR).

Page	Rank (Standard)	Rank (PPR)
Communism	89	1
Russia	18	2
Socialism	223	3
Soviet_Union	38	8
Marxism	365	18
Vladimir_Lenin	798	19

Table 3: Comparison of ranks between Standard and Personalized PageRank.

Synthesis and Analysis: *To begin with, the three selected nodes are at the top of the ranking. We can see that pages with related subjects have significantly moved up in the ranking, for example "Soviet_Union", "Marxism", "Vladimir_Lenin". Also, it is noticeable that countries are still very present in the top ranking.*

5 Conclusion

This practical work allowed us to implement the PageRank algorithm on real data. We were able to observe the robustness of the power method and analyze the impact of hyperparameters β and ϵ on convergence.

The analysis of results on Wikispeedia confirms that the human navigation structure relies heavily on general concepts (countries, world wars) to transition from one subject to another. Personalized PageRank demonstrated its ability to bias these results to favor specific themes, paving the way for contextual search engines.

A Appendices: Python Codes

The entire code was developed in Python 3. Below are the main functions used for matrix construction and PageRank calculation.

```
def get_mappings_and_pages(filepath):
    """
    Reads the file to identify all unique pages.
    Returns:
        - page_to_idx : dictionary {page_name: index}
        - pages_list : list where pages_list[i] gives the name of the page at
    index i
    """
    unique_pages = set()

    with open(filepath, "r", encoding="utf-8") as fh:
        reader = csv.reader(fh, delimiter="\t")
        for ligne in reader:
            # Verify that the line has the correct format
            if len(ligne) > 3:
                consult = ligne[3].split(";")
                for page in consult:
                    if page != "<":
                        unique_pages.add(page)

    # Sort to have a deterministic order
    pages_list = sorted(list(unique_pages))

    # Dictionary comprehension to create the reverse mapping
    page_to_idx = {page: i for i, page in enumerate(pages_list)}

    return page_to_idx, pages_list
```

Listing 1: Code to map pages to indices

```
def build_adjacency_matrix(filepath, page_to_idx, N):
    """
    Builds the adjacency matrix L using the STACK method.
    Strictly follows the original logic.
    """
    L = np.zeros((N, N), dtype=np.float64)

    with open(filepath, "r", encoding="utf-8") as fh:
        reader = csv.reader(fh, delimiter="\t")

        for ligne in reader:
            if len(ligne) <= 3:
                continue

            consult = ligne[3].split(";")
            stack = []  # Initialization of the stack for this path

            for token in consult:
                if token == "<":
                    # Backtrack: we pop
                    if stack:
                        stack.pop()
                else:
                    # It is a visited page
                    page_actuelle = token

```

```
                    # If the stack is not empty, the top is the previous page (source)
                    if stack:
                        page_precedente = stack[-1]

                        i_source = page_to_idx[page_precedente]
                        j_target = page_to_idx[page_actuelle]

                        L[i_source][j_target] = 1

                    # Push the current page (it becomes a potential source)
                    stack.append(page_actuelle)
    return L
```

Listing 2: Code to create the adjacency matrix

```

def normalize_matrix(L, N):
    """
    Handles 'dangling nodes' (rows of zeros) and normalizes by row.
    """
    # 1. Handling Dead-ends (Dangling nodes)
    # If a page points to nothing, we assume it points to everyone (random surf)
    for i in range(N):
        if np.sum(L[i]) == 0:
            L[i, :] = 1.0 / N

    # 2. Stochastic Normalization (the sum of each row must be 1)
    # Divide each row by its sum
    row_sums = L.sum(axis=1)
    # Numpy trick: divide matrix by the column vector of sums
    L = L / row_sums[:, np.newaxis]

    return L
```

Listing 3: Code to obtain the transition matrix

```

def compute_pagerank(beta, epsilon, P, v, s, N, max_iter=1000):
    """
    Computes the PageRank vector q.
    P : Transpose of the normalized adjacency matrix (L.T)
    v : Personalization vector
    s : Sum of v (or normalization factor of v)
    """

    # Start timer
    start_time = time.perf_counter()

    # Uniform initialization
    q = np.ones(N) / N
    q = q / nla.norm(q, 1)

    iterations = 0
    diff = epsilon + 1.0  # To enter the loop

    while diff > epsilon and iterations < max_iter:
        q_old = q.copy()
        sum_q = np.sum(q)

        # Original formula:
        # q = beta * P * q + teleportation term
        q = beta * (P @ q)
        q = q + ((1 - beta) / s * sum_q * v)
```

```
28
29          # Renormalization (numerical safety)
30          q = q / nla.norm(q, 1)
31
32          # Calculate difference for convergence
33          diff = nla.norm(q - q_old, 1)
34          iterations += 1
35
36      # Stop timer
37      end_time = time.perf_counter()
38      execution_time = end_time - start_time
39
40      return q, iterations, execution_time
```

Listing 4: PageRank Code

```
1
2  if __name__ == "__main__":
3      FILE_PATH = "./paths_finished.tsv"
4
5      print("1. Loading and Mapping...")
6      # Retrieve the dictionary (name->idx) and the list (idx->name)
7      page_to_idx, idx_to_page = get_mappings_and_pages(FILE_PATH)
8
9      N = len(pages_list := idx_to_page)  # Walrus syntax (python 3.8+)
10     print(f"   Number of unique pages : {N}")
11
12     print("2. Building matrix L...")
13     L = build_adjacency_matrix(FILE_PATH, page_to_idx, N)
14
15     print("3. Normalization...")
16     L = normalize_matrix(L, N)
17
18     # Preparation for PageRank
19     # Use the transpose for equation P @ q
20     P = L.T
21
22     # Configuration of personalization vector (v)
23     v = np.ones(N)  # = np.zeros(N) in PPR
24     indices_cles = [3237, 3422, 919]  # Specific indices
25
26     # Verify that indices exist (to avoid crash if file is different)
27     valid_indices = [k for k in indices_cles if k < N]
28     if valid_indices:
29         v[valid_indices] = 1
30     else:
31         # Fallback if indices out of bounds, set everything to 1
32         v = np.ones(N)
33
34     s = np.sum(v)  # Sum for normalization in the formula
35     if s == 0:
36         s = 1  # Division by zero safety
37
38     # --- Main Calculation ---
39     print("4. Computing PageRank...")
40     beta_val = 0.85
41     epsilon_val = 0.000001
42
43     final_q, iterations, duration = compute_pagerank(beta_val, epsilon_val, P, v
       , s, N)
44
45     print(f"   Convergence in {iterations} iterations and {duration:.4f} seconds
       .")
46
```

```
    # --- Display Results (Top 20) ---
    print("\n--- TOP 10 PAGES ---")
    # Associate score and index, sort, then retrieve name
    top_indices = np.argsort(final_q)[:20]
    for idx in top_indices:
        print(f"Score: {final_q[idx]:.6f} | Page: {idx_to_page[idx]} (Index: {
    idx})")
```

Listing 5: Main Execution Code