LOGO SIMILARITY

POPA ALEXIA LARISA

THE PURPOSE OF THIS PRESENTATION IS NOT TO BORE YOU; THEREFORE, I WILL AIM TO HIGHLIGHT THE KEY IDEAS AND PROVIDE DETAILED EXPLANATIONS WHERE NECESSARY :))

OVERVIEW

The following slides will illustrate the necessary steps for solving this challenge, as well as the chosen approach and why.

Additionally, I will explain the functioning of the algorithm, with the aim of highlighting the technical details.

STEP 1

THE DECRYPTION OF THE FILE

The algorithm processes a .parquet file containing domain or URL data. It loads the file into a Pandas DataFrame and extracts the column named 'domain'. Each entry is cleaned by removing prefixes like www. and https://, stripping query parameters and subpaths, and converting to lowercase. After cleaning, it filters out invalid entries, removes duplicates, and sorts the results. Finally, the cleaned list of unique domains is saved to a text file (cleaned_websites.txt). The algorithm also includes error handling for missing dependencies and incorrect column names.

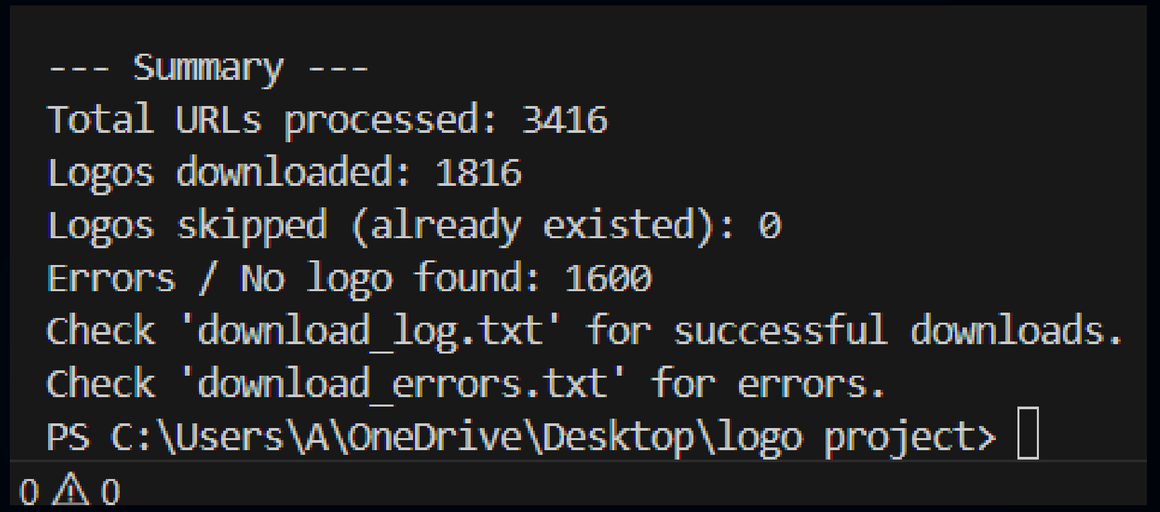
The algorithm was implemented in Python due to its rich ecosystem of data processing libraries (such as pandas and re) and its readability, which makes it well-suited for rapid development and data manipulation tasks. Additionally, I prefer this method because I believe it is significantly easier for me, which enables me to perform the task more effectively

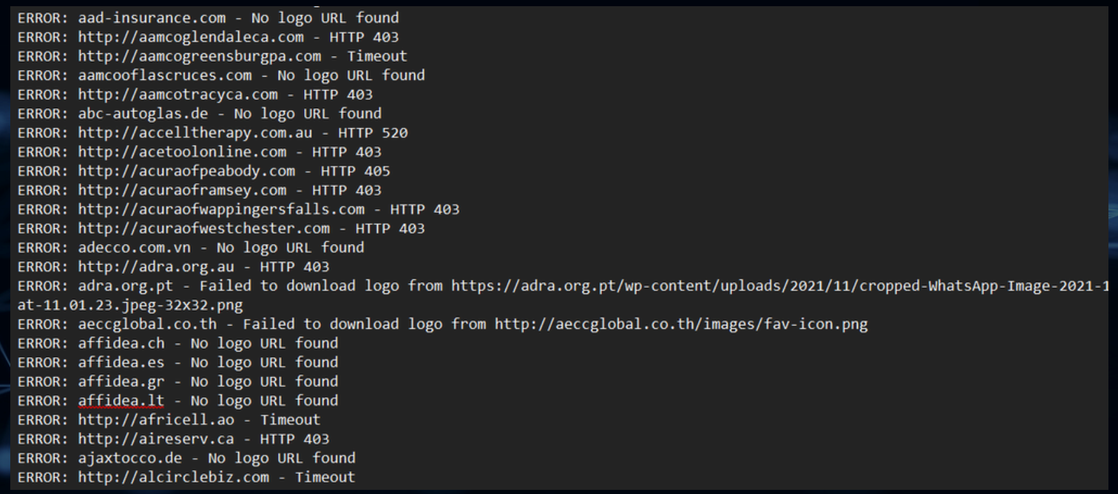
```python
import pandas as pd
import re

input_parquet_file = "logos.snappy.parquet"
output_txt_file = "cleaned_websites.txt"

try:
    df = pd.read_parquet(input_parquet_file)

    url_column_name = 'domain'
    if url_column_name not in df.columns:
        print(f"Error: Column '{url_column_name}' not found in Parquet file.")
        print("Available columns are:", df.columns.tolist())
        exit()

    raw_urls = df[url_column_name].tolist()

    def clean_url(url):
        if isinstance(url, str):
            url = url.strip()
            url = re.sub(r'^(www\.)', '', url, flags=re.IGNORECASE)
            url = re.sub(r'(https?://)', '', url, flags=re.IGNORECASE)
            url = url.split('/')[0]
            url = url.split('?')[0]
            url = url.strip().lower()
            return url
        return None

    cleaned_urls = [clean_url(url) for url in raw_urls if clean_url(url) is not None]

    unique_cleaned_urls = sorted(list(set(cleaned_urls)))

    with open(output_txt_file, "w", encoding="utf-8") as f:
        for url in unique_cleaned_urls:
            f.write(url + "\n")

    print(f"Successfully cleaned and saved {len(unique_cleaned_urls)} unique URLs to {output_txt_file}")

except Exception as e:
    print(f"An error occurred: {e}")
    print("Please ensure 'pandas' and 'pyarrow' are installed (`pip install pandas pyarrow`)")
```

STEP 2

EXTRACT LOGOS

In the initial iteration of the logo extraction script, the success rate for logo retrieval was approximately 53%. Although the script processed all specified URLs, it successfully extracted 1,816 logos, with errors reported for the remainder. This observation underscored the necessity for enhancing the script's robustness. Analysis of the error log revealed a significant prevalence of connection-related failures. Furthermore, it was hypothesized that a considerable number of websites might not store their logos in standard or easily discoverable locations within their HTML structure.



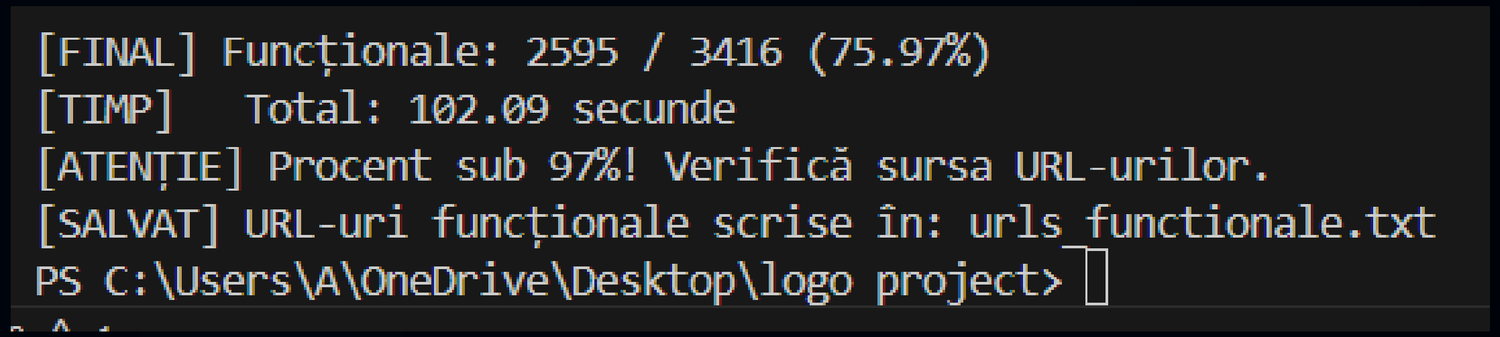


STEP 2

EXTRACT LOGOS

In order to incress the results, I manage to build an algorithm that

1. Reads URLs from an input file.
2. Normalizes each URL (adds http:// if missing).
3. For each URL:
   - Sends a HEAD request (fast check).
   - If the response is not clearly OK (e.g. 403, 301), tries a full GET request.
4. Uses multiple threads (e.g. 50) to speed up the process.
5. Prints the status of each URL (OK or error).
6. Saves all functional URLs to a new file.
7. Reports success rate and time taken.



UNFORTUNATELY, THIS APPROACH DID NOT YIELD A SIGNIFICANT SUCCESS RATE IN THE FINAL OUTCOME, WHICH IS WHY I CHOSE TO DISCONTINUE ITS USE.

DURING THE DEVELOPMENT OF THE LOGO EXTRACTION ALGORITHM, SEVERAL KEY ISSUES WERE ENCOUNTERED:

- MANY WEBSITES FAILED TO LOAD OR BLOCKED AUTOMATED ACCESS, LEADING TO A HIGH NUMBER OF REQUEST TIMEOUTS.
- LOGO DETECTION WAS OFTEN UNSUCCESSFUL DUE TO DYNAMIC CONTENT, NON-STANDARD HTML STRUCTURES, OR HIDDEN IMAGES.
- THE SCRAPING PROCESS WAS SLOW AND RESOURCE-INTENSIVE, ESPECIALLY WHEN SCALING TO THOUSANDS OF URLS.
- CLUSTERING LOGOS BASED ON VISUAL SIMILARITY PROVED DIFFICULT DUE TO INCONSISTENCIES IN IMAGE SIZE, FORMAT, AND QUALITY.

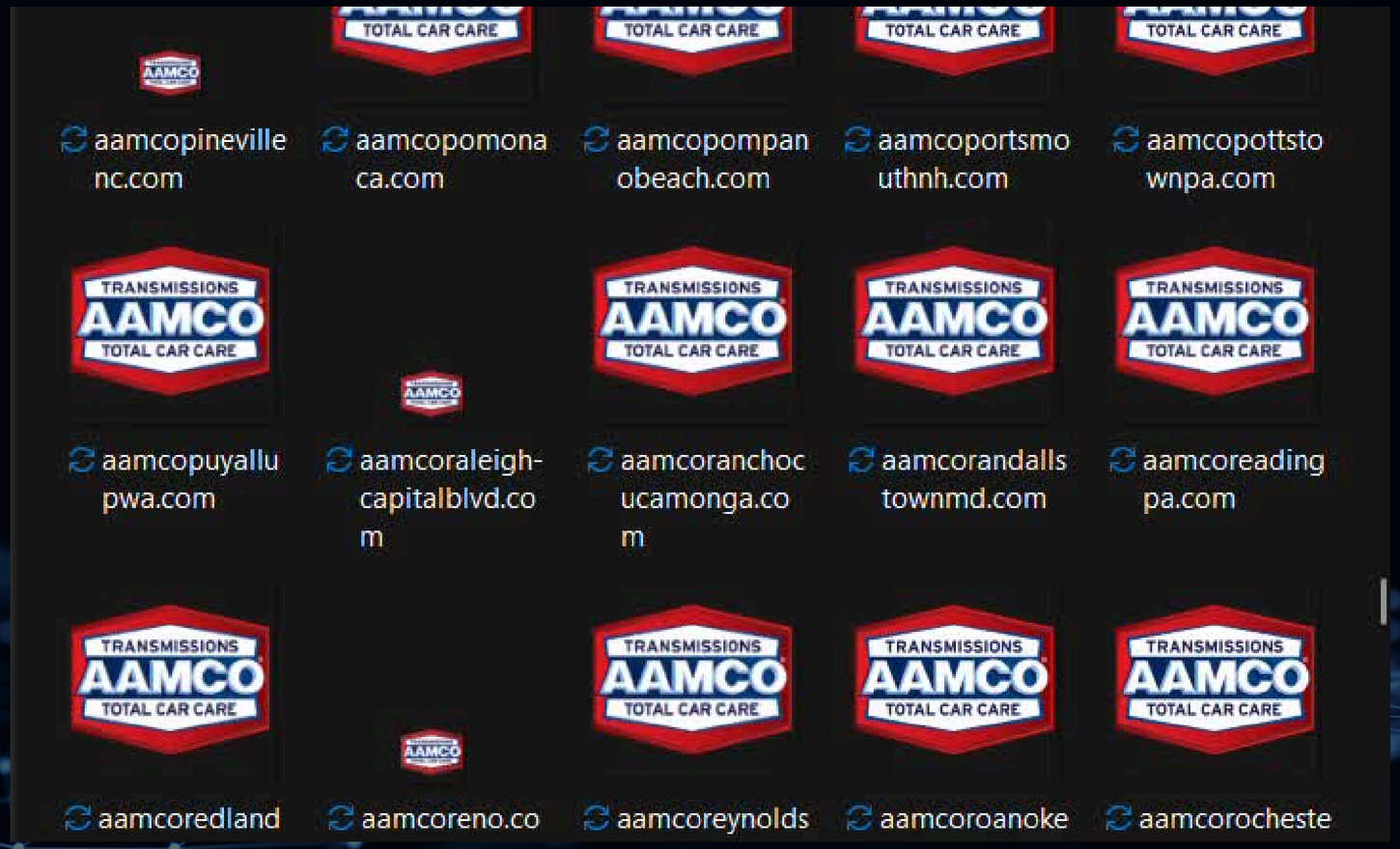
THESE CHALLENGES SIGNIFICANTLY REDUCED THE ALGORITHM'S EFFECTIVENESS AND REQUIRED RE-EVALUATION OF THE CHOSEN APPROACH.

IN THE LIGHT OF THE ABOVE, I CONSIDERED THE BEST OPTION TO CHOOSE THE VERSION WITH 53% RESULTS AND MOVE FORWARD WITHT THE NEXT STEP:CLUSTERING

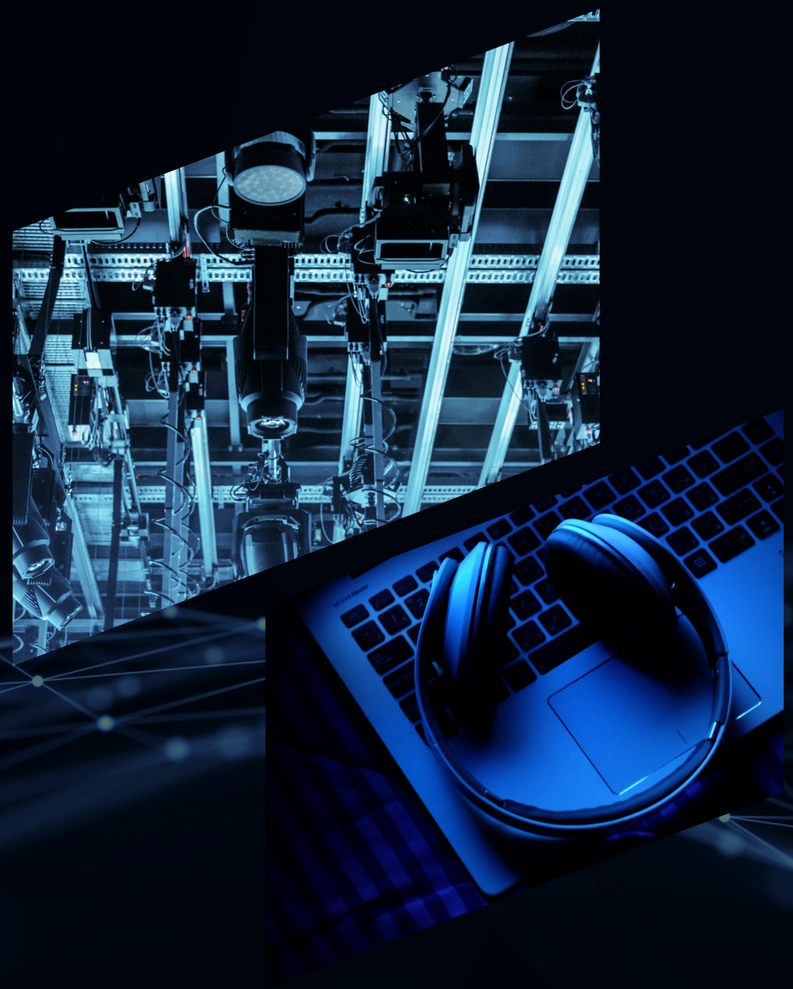
STEP 3

CLUESTERING

- Feature Extraction (Non-ML): It uses imagehash.phash to calculate a perceptual hash for each logo. This hash is a unique "fingerprint" that represents the image's visual content. Visually similar images will have similar hashes.
- Grouping (Non-ML Heuristic): It iterates through all logos. For each logo, it compares its hash to the "representative hash" of existing groups. If the Hamming distance (number of differing bits) between the current logo's hash and a group's representative hash is below a configurable MAX_HASH_DISTANCE threshold, the logo is added to that group. Otherwise, a new group is created.
- Output: It creates separate subdirectories for each identified group (cluster) and copies the respective logos into them. The MAX_HASH_DISTANCE parameter can be adjusted to control how strictly logos are grouped based on visual similarity.



THANK YOU!



I appreciate your time and consideration for this position. Please do not hesitate to contact me should you require any additional information. This challenge provided a valuable learning experience.