**Veridion Logo Similarity**

**By Brodoceanu Octavian**

This project takes a list of company websites, pulls their logos, and groups them based on visual similarity. This is done without machine learning algorithms. The idea is to rely on visual features and heuritctics we can explain.

**Workflow**

The project is split into 3 scripts:

* Logo\_extraction.py
* Feature\_extraction.py
* Clustering.py

As such, it can be run either by running these 3 scripts separately or using the main.py file which calls each sequentially.

**Working steps**

* Read the parquet file
* In parallel, send requests to each page and attempt to extract the logo. Failures(due to either finding no match, 403 access denied, etc.) are logged. Logos are saved in output/logos
* Noticing the multitude of websites that belong to the same company, we propagate the logos that match domains to the sites that failed and log the results before and after the operation
* Extract visual features from each logo – edge density, dominant color, hue family, aspect ratio, sharpness, shape descriptors and save them to output/features
* Group logos into clusters based on similarity metrics - using a threshold and save them as a json file
* Paths and enabled features, as well as feature weights are set using a configuration file

**Details for each step**

**1. Logo Extraction**

We look at each company website after reading the parquet file and try to find a logo. Priority order:

* <link rel="icon">
* <meta property="og:image">
* Any <img> tag with “logo” in its filename or alt text
* Else we fall back to favicon.ico

This is done in parallel to speed up the process.

If we fail to connect or find a logo, we log this.

**Log:** output/logs/logo\_results.json

**Output**: output/logos

**2. Propagation**

If one domain under a brand (like nike.com) has a logo and others (like nike.ca) don’t, we copy the logo over to them. This saves time and avoids duplicate scraping.

**Log:** output/logs/propagation\_log.json, output/logs/stats\_before\_propagation.json, output/logs/stats\_after\_propagation.json

**Output**: output/logos

**3. Feature Extraction**

First it’s important to note that we use a mask to use only the foreground of the logo. This is done via:

* + - Using the transparency layer(where applicable) for PNG images
    - Taking the average of the EDGE pixels of the image and using a threshold

An example of foreground extraction can be seen below(note the red outline of the foreground)

We run each logo through a series of visual checks and extract:

1. **Dominant color** (RGB):

**What it is:**  
The most common color found in the logo, ignoring the background. It's returned as an RGB list, like [255, 0, 0] for red.

**Why it matters:**  
Color is a big part of brand identity —for example Coca-Cola red or Pepsi blue.

**How it's calculated:**

* First, we try to remove the background using a mask (alpha transparency or edge color detection).
* Then we run **K-means clustering** on the remaining pixels to find the most frequent color.
* The most populated cluster gives us the dominant RGB.

1. **Hue family**:

**What it is:**  
A simplified, categorical version of the dominant color. For example: red, blue, gray, black.

**Why it matters:**  
It groups colors semantically rather than numerically. Helps logos with similar colors but distant in RGB space group together, even if the exact RGBs differ.

**How it's calculated:**

* We convert the dominant RGB to **HSV** color space.
* Based on the hue, saturation, and brightness values, we assign it to a predefined label (like "red" if hue < 15, or "gray" if saturation is low).

**3. Edge Density**

**What it is:**  
A ratio showing how many edges (borders/lines) are present relative to the total number of pixels.

**Why it matters:**  
It’s a proxy for **complexity** — whether a logo is minimal and flat, or busy and detailed.

**How it's calculated:**

* We run Canny edge detection on the image.
* Count how many edge pixels are found.
* Divide by the total pixel count to get a density between 0 and 1.

**4. Aspect Ratio**

**What it is:**  
The ratio between the width and height of the **foreground bounding box**.

**Why it matters:**  
Brand logos tend to keep consistent shape proportions — square vs rectangle, wide vs tall, etc.

**How it's calculated:**

* After masking the background, we find the bounding box that wraps the non-zero (non-background) pixels.
* Divide width by height to get the ratio.

**What it is:**  
A score between 0 and 1 representing how many **sharp angles** are present in the logo contours.

**Why it matters:**  
It acts as a proxy for **logo personality** — sharper shapes are more “aggressive” or techy; rounded ones feel friendly and soft.

**How it's calculated:**

* We extract foreground contours and simplify them into polygons.
* For each polygon, we measure interior angles.
* If an angle is sharper than a threshold (e.g. < 100°), it's counted as “sharp.”
* The sharpness score is the ratio of sharp angles to all angles found.

**5. Sharpness (Angularity)**

**What it is:**A score between 0 and 1 representing how many sharp angles are present in the logo contours.

**Why it matters:**This is based on the a phenomenon called the kiki/bouba effect, where more angular shapes are associated with kiki and where fluffier shapes are associated with bouba. From this, we can determine we make assumtions based on visual angularity, so it could be an important feature.

**How it's calculated:**

* We extract foreground contours and simplify them into polygons.
* For each polygon, we measure interior angles.
* If an angle is sharper than a threshold (e.g. < 100°), it's counted as “sharp.”
* The sharpness score is the ratio of sharp angles to all angles found.

**6. Shape Descriptor (Hu Moments)**

**What it is:**A 7-dimensional vector that captures the general shape of the logo, regardless of its size, position, or rotation.

**Why it matters:**It gives a deeper comparison of overall form — logos with similar silhouettes will have similar Hu moments.

**How it's calculated:**

* We convert the image to grayscale and threshold it to find contours.
* We take the largest contour and compute its Hu Moments.
* These moments are log-scaled and normalized to form a 7-value array.

**Note: This feature is sensitive to foreground masking. If masking fails, the values might be meaningless.**

**Clustering**

Logos are compared using a custom distance function (no ML). Each logo is either added to the closest existing group or starts a new one. The threshold has been set to 0.1. We determine the closest group by computing the average of each group(similar to K-means clustering) and calculating the distance from the centroid to the sample.

Distance is based on feature differences. For vectors, we take the euclidean distance between them, while for simple values like aspect ratio, we have the direct difference.

All of these are weighted by using the weights in config.py and divide by total weight.

We track which logos were skipped due to missing data

**Configuration**

Open config.py to:

* Enable/disable features (FEATURE\_FLAGS)
* Adjust importance (FEATURE\_WEIGHTS)
* Set the distance threshold for clustering
* Adjust paths

**Input and Output Structure**

input/

└── logos.snappy.parquet

output/

├── logos/

├── features/logo\_features.json

├── clusters/clusters\_filtered.json

└── logs/

├── logo\_results.json

├── propagation\_log.json

└── null\_feature\_logos.json

Input holds the initial parquet file.

Output contains:

* logos – dir containing the extracted logos
* features – dir containing the extracted features of each logo image
* clusters – dir containing the output clusters
* logs – dir containing:
  1. logo\_results - initial log for logo extraction
  2. null\_feature\_logos - logging logos with missing features during clustering
  3. propagation\_log - logs the origin and destination domains when propagating
  4. stats\_after\_propagation – logs the stats after propagate.py(no. failures, causes, no. successes, no. propagated logos)
  5. stats\_before\_propagation – logs the stats before propagate.py(no. failures, causes, no. successes)

Configurable via **config.py**

**Results**

Each feature was tested in isolation. Using very high(>0.999) or very low(<0.001) thresholds yielded no meaningful clustering for sharpness and shape descriptors, so they were disabled. Additional tests were conducted with various values for **threshold** to determine better values for the weights. The config.py file as in the project’s current form was used for the clustering results.

**Possible Future Improvements**

Using the features close to the outputs or the middle of a CNN(prefferably something light, a MobileNet for example) for clustering to determine meaningful shape vectors.

Using multiple colors (top k for example)

Additional tests with various feature weights.