

# README

## HTML clones

🔗 Armega Andrei

<https://github.com/AndreiArmega/HTML-clones>

### 📄 PREMISE:

Design an algorithm that will group together HTML documents which are similar from the perspective of a user who opens them in a web browser

🔗 must read sections: 1.3.2

## 1 The Task

**Group files that look similar from the user perspective.**

### 1.1 Understanding the Task

An HTML file can be analyzed from two distinct perspectives:

- **Code perspective:** This involves a structural and functional analysis of the underlying code meant to be rendered by the browser.
- **User perspective:** This focuses on the structural and functional analysis of the visual elements as rendered by the browser on the webpage.

The objective of this task is to analyze the contents of HTML files and identify **similarities**. The key point is that these similarities should be evaluated from the **user perspective**.

While the code perspective might seem redundant, at this stage of developing the solution, we will consider both perspectives at least at the idea level, as they are not mutually exclusive. The code perspective may still serve as a useful **token** for establishing categories.

#### 1.1.1 The Source Code Perspective

The **source code perspective** refers to how an HTML file is structured and written, focusing on syntax, tags, attributes, and overall organization. This view emphasizes the technical composition of the file, as opposed to how it is visually rendered.

#### 1.1.2 The User Perspective

The **user perspective** refers to how the HTML file is rendered by the browser and experienced by the user. Although the source code is accessible, this task focuses only on the **rendered output**—layout, styling, content, and interactions.

### 1.1.3 Similarity

We define **similarity** as having a **resemblance** in **appearance**, **character**, or **quantity**, without requiring exact identity.

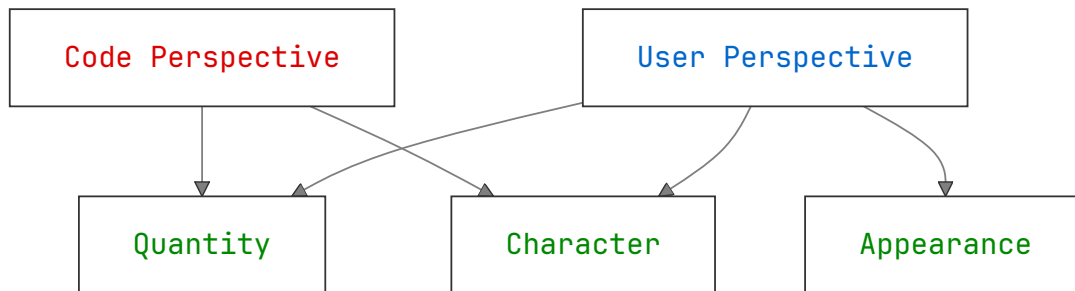
These three aspects are referred to as **tokens** of similarity—each representing a basic unit within our categorization system.

---

#### 1.1.3.1 Defining the Tokens of Similarity

We define three tokens: **appearance**, **character**, and **quantity**.

We also consider two perspectives: **code** and **user**.



##### 1.1.3.1.1 Tokens in the Code Perspective

We discard **appearance** from the code perspective, as reflected in the diagram above.

- **Quantity** is intuitive—measured by number of lines, elements, or blocks.
- **Character** is more abstract. While code lacks "feel," we treat structure, modularity, and purpose as **essence** to represent its character.

##### 1.1.3.1.2 Tokens in the User Perspective

In the user perspective, the tokens of similarity are more easily observable and often intertwined. To maintain clarity, we adopt stricter definitions:

- **Quantity**: How content-heavy or dense the page appears.
- **Appearance**: The visible aspects—fonts, structure, layout, colors, alignment, etc.
- **Character**: The emotional or psychological *effect* on the user (e.g., joy, boredom, confusion), distinct from visual properties. In simpler words, the feel of the page.

##### 1.1.3.1.3 Compound Tokens

A **compound token** is created by combining two or more basic similarity tokens.

The number of possible compound tokens is:

$$CT = \sum_{k=0}^n \binom{n}{k} = 2^n - n$$

(See section 1.1.3.2.1 for aggregate usage)

### 1.1.3.2 Scale of similarity

We consider a scale from (0% - 100%) to establish similarity between 2 HTML files. A percent of similarity can only be established by approaching the files from the same perspective, i.e. files can be only similar from the code perspective or similar from the user perspective.

0% similarity and 100% will not be considered as such proportions go outside the scope of the definition.

The 2 possible scales are **independent** of one another, meaning 2 files could be 99% similar from the code perspective and 1% similar from the user perspective and vice versa.

#### 1.1.3.2.1 Methods of determining a final category.

With the similarity tokens established, we now need to determine the final, relevant categories that will be produced by the solution. These categories may be composed of one or more tokens.

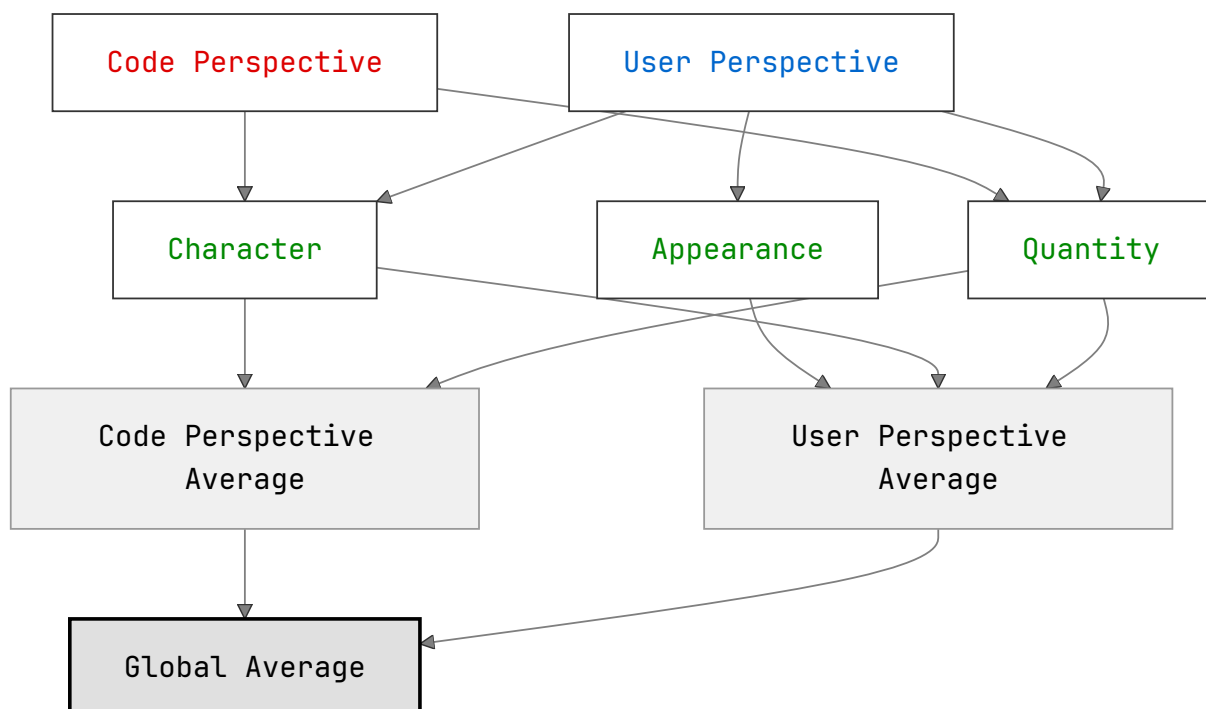
We define the **perspective aggregate** as the average value of the tokens applied within a specific perspective.

For example, for the user perspective:

**(Quantity + Appearance + Character) / 3**

We define the **global aggregate** as the average of all perspective aggregates:

**(code perspective aggregate + user perspective aggregate) / 2**



Any of the tokens or aggregates shown above can serve as valid criteria for grouping HTML files.

The most significant point of contention is whether to include the individual tokens derived from the code perspective in the final grouping logic. This concern arises from the fact that aspects of the code perspective may not directly relate to how users experience the rendered HTML.

To address this, I propose that we **exclude individual code perspective tokens** (e.g., code quantity, code character) as *direct* criteria for group formation. In other words, there will be no category that groups files *solely* based on code perspective character or quantity.

However, we **retain the code perspective aggregate (c.p.a.)** as a contributor to the overall global average. This means that although c.p.a. will **not independently define a group**, it may influence groupings through the **global aggregate (g.a.)**.

The rationale for partially including the code perspective is that code is rarely written in isolation. Developers often build upon existing templates or styles, which can transfer some of the **character**—as perceived by users—from one version to another. Additionally, with the increasing use of AI tools, each LLM or code generation engine tends to imprint its own *style* or *signature* on the generated HTML. These characteristics may manifest not only in code structure but also indirectly in the user-facing experience.

Therefore, while not strictly required, incorporating aspects of the code perspective can lead to a more **robust**, nuanced, and fine-grained grouping system.

#### Info

u.p. - user perspective  
c.p. - code perspective

That being said, we are left with the following set of candidate tokens:

[

- u.p. character,
- u.p. appearance,
- u.p. quantity,
- u.p. average,
- global average

]

At this point, another design decision arises: whether to use only **independent tokens** or also allow **compound tokens**. We currently have 5 independent tokens.

If we include compound tokens—combinations of 2 or more independent tokens—we're left with 27 possible combinations. However, many of these compound tokens offer little to no added value and should be discarded. For example, (u.p. character, u.p. appearance, u.p. quantity) is simply a verbose representation of u.p. average.

While introducing compound tokens could add flexibility and allow for more dynamic groupings, it also risks increasing complexity and potentially complicating the codebase unnecessarily. Compound tokens might act as a kind of wildcard grouping mechanism in situations where the available HTML files don't neatly align with existing tokens. In such edge cases, they could help fine-tune categorization.

For now, we will treat the concept of compound tokens as a **future scalability point**, not part of the initial implementation.

After a good chunk of implementing code, I came to the conclusion to discard the code perspective influence from the final result. I think a cleaner solution provides only one grouping option, but a more complex solution can take into consideration the code perspective. We are adding the code perspective analysis to the list of **starting points of future development**.

This means we discard individual tokens character, appearance and quantity and we discard the global average as we no longer consider the code perspective.

#### 1.1.3.2.2 Case study

Let's walk through two examples to illustrate how the system evaluates similarity:

##### Example Case 1

Let there be two HTML files: `a.html` and `b.html`. The intended purpose of both files is to display a simple message to the user, such as "Hello World!"

1. `a.html` contains 10 lines of code and successfully displays the message.
2. `b.html` contains 100 lines of code, also displaying the same message.
3. Both files use the same font, size, layout, and styling.
4. `b.html` includes 90 lines of redundant code that do not contribute any additional functionality.

We analyze their similarity:

##### Code Perspective:

- **Quantity**: ~10% similarity
- **Character**: ~99% similarity
- **Aggregate**: ~55% similarity

##### User Perspective:

- **Quantity**: ~99% similarity
- **Character**: ~99% similarity
- **Appearance**: ~99% similarity
- **Aggregate**: ~99% similarity

##### Global Aggregate:

- ~77% similarity
- 

##### Example Case 2

Now consider two HTML files, `a.html` and `b.html`, both created to present and advertise a restaurant's food menu.

1. Both contain approximately 1000 lines of code and include numerous high-quality images of food.
2. Both utilize vibrant colors and feature dynamic animations.

We analyze their similarity:

#### Code Perspective:

- **Quantity**: ~99% similarity
- **Character**: ~30% to 99% similarity (estimated)
- **Aggregate**: ~65% to 99% similarity

#### User Perspective:

- **Quantity**: ~50% to 99% similarity
- **Character**: ~20% to 90% similarity
- **Appearance**: ~50% to 99% similarity
- **Aggregate**: ~40% to 96.6% similarity

#### Global Aggregate:

- ~52.5% to 98.6% similarity

These two examples demonstrate both a **static and clearly defined case** (Example 1) and a **more dynamic, fluid case** (Example 2). The values provided are estimates used for illustrative purposes. In practice, the algorithm must rely on well-designed **heuristics** to calculate these similarity percentages with accuracy and consistency.

As discussed in 1.1.3.2.1 we discard the code perspective from the final grouping and we will only consider u.p. average as a defining grouping mechanic. The involvement of each u.p. single token will be fluid and not evenly divided. This means that Quantity is not 33% determinative of the final result.

### 1.1.3.3 Judging the Similarity

Returning to the core task, the solution must be able to **compare rendered HTML pages**. But how does a piece of software interpret a rendered HTML page? The same way a human does—**visually**.

So how can we make a program "see"? Naturally, by giving it an image.

The core algorithm ingests a large set of images—screenshots of rendered HTML pages. Based on these images, **a series of sub-algorithms and heuristics** will work together to determine final groupings of visually similar pages.

---

#### 1.1.3.3.1 The Sub-Algorithms Selection Process

Given the variety of **visual similarity cues** the system must evaluate, it's clear that a single image processing algorithm won't suffice.

While it's possible to include a vast number of algorithms to support the decision-making process, I chose not to. To explain this choice, I use the analogy of a **courtroom**:

Imagine that the main algorithm is the court, and each image recognition method is a **juror**. These jurors are brought in to reach a verdict. In our case, the "verdict" is whether a page is **guilty** (it visually resembles another) or **innocent** (it does not).

Each juror (algorithm) brings its own perspective. The task of the main algorithm is to reconcile these opinions and reach a reliable conclusion.

So, how many jurors do we need?

Imagine a courtroom with 100 jurors. Most would agree that's excessive—deliberations. It would be slow, and reaching consensus would be difficult. In real-world courts, we've settled on 12 jurors. Similarly, in our solution, we don't need dozens of image comparison algorithms—especially since many overlap in what they detect.

The number and selection of algorithms are based on a few core principles:

- **Diversity:** We need enough algorithms to bring varied perspectives and examine different visual features.
- **Efficiency:** Too many algorithms slow down processing and introduce redundancy without added value.
- **Redundancy (within reason):** Some overlap is useful. It allows multiple algorithms to verify the same feature, reducing the risk of misjudgment by any single one.
- **Specialization:** Each algorithm should target a distinct visual trait, ensuring we cover all relevant aspects of similarity.

By following this approach, we build a reliable, efficient, and fair "jury" of algorithms to evaluate visual similarity between HTML pages.

#### 1.1.3.3.2 The Jurors

After reasearching many algorithms to cover all the nuance in judging similarity by the decided tokens I selected the following:

- Perceptual hashing
- Histogram comparison
- ORB(Oriented FAST and Rotated BRIEF)
- SSIM (Structural Similarity Index)
- Deeplearning approach

For understaning theese decision please refer to the definitions in 1.1.3.1.2. We will take each algorithm and explain its role in the process.

##### 1. *SSIM*

This algorithm was chosen for detecting structural similarity which is the crux of judging appearance. This algorithm mostly helps determine [appearance](#).

##### 2. *Histogram comparison*

This algorithm was chosen to satisfy the color tone comparison and overall visual mood. This algorithm mostly helps determine [appearance](#).

##### 3. *Perceptual hashing*

This algorithm was chosen for 2 reasons.

Reason 1: We need a fast decently reliable algorithm for the **initial grouping**(see 1.2.1 bellow)

Reason 2: It offers some redundancy to SSIM and ORB

This algorithm helps determine [appearance and quantity](#).

##### 4. *ORB*

This algorithm was chosen for its strength of detecting distinctive visual features such as corners, edges, and blobs. In the context of HTML this can be

logos, text boxes, navigation bars and such. It also offers some redundancy for the SSIM and hashing. This algorithm helps determine **appearance, quantity and character**.

#### 5. **Deeplearn**

For the purpose of determining the character(as defiend) of a HTML page, the deeplearning approach was the obvious first approach. Evidently, a model that has been trained on html data is difficult to produce, especially on a small data set liek the one provided. I chose to use a pre trained model. This algorithm helps determine **character**.

I approached this with appearance as being the most important token to be analyzed, as it is mostly encompassing the other 2. Certain algorithms had to be introduce to make the difference between appearance and character and quantity.

## 1.2 The final groups.

As disscussed in 1.1.3.2.1 we have 5 tokens of similarity that we wish to pursue, meaining 5 methods of comparison between 2 files. Each token can be tough of as a parameter to the final comparison script resulting in n groups that are very simmilar in respect to that token.

Let's take as an example : token fruits. The resulting groups will be 3 groups, group A has HTML files that all feature apples elements in one form or another; group B has only of html files that feature banana elements and maybe some yellow canoe; group C ... and so on.

Using the fruit token we managed to group the files based on the fruits they posses.

If we were to apply this to our similarity tokens in the sections above, the output will not be as clear cut since the tokens are much more abstract, but the logic should transfer- we will get groups which have similar appearances,character etc.

This logic maps well on the primary tokens, as well as the agregate tokens i.e. (u.p. average and global average). A grouping by the global average will create groups that have very similar coeficients on all primary tokens. The details remain to be determined in a special section.

### 1.2.1 The initial verdicts

The first step toward forming final groups is to perform an initial evaluation of the files.

In this phase, our **jurors** (see **1.1.3.3.2**) examine the evidence and issue a verdict- well, almost.

In practice, we store a combined opinion of two automated jurors in the **verdicts** folder.

The two jurors are **Hashing** and **Deeplearn**.

The decision logic is as follows:



Hashing:

```
Compare every file with every other file
Record verdict
```

Deeplearn:

```
Compare every file with every other file
IF my verdict is very favorable:
    Reduce severity of the hashing verdict
ELSE:
    My vote is ignored
```

All final algorithms have been crafted through extensive testing on sample data. Initially, the hashing approach alone yields over **95% accuracy**, especially since the data mostly consists of near-duplicate files.

However, certain **edge cases** slip through the cracks—where the hashing verdict is slightly too harsh, and files that should belong together end up separated.

To address this, we use a second layer of verification: a deep learning model. Although this model has lower overall accuracy, it's incredibly useful due to its **near-zero false positive rate**. It may miss some matches (false negatives), but when it says two files are similar, it's almost always right.

A highly favorable verdict from the deep learning model is a **strong signal** that two files belong in the same group.

While hashing struggles to grasp the "intent" or semantic meaning of a page, the deep learning model excels at it.

For instance, two betting websites might differ just enough in structure or aesthetics to fool hashing—but deep learning picks up on their shared identity.

By combining both verdicts, we achieve approximately **99% initial grouping accuracy**.

### 1.2.2 The initial groups

Initial groups are created by parsing the verdict file.

A verdict entry has the following format:

```
file1.html → file2.html = <verdict_float>
```

The grouping logic is straightforward:

```

For file<i> in all files:
    For each comparison involving file<i>:
        If the verdict passes the threshold:
            Assign file<i> to the first available matching group

```

Naturally, as more groups are created, their density decreases, and toward the end, many groups contain only one file.

### 1.2.3 The reconversion

A key question in solving this problem is:

**How many groups should we aim for?**

There is no definitive answer.

Technically, one group per file or a single group for all files are both valid—but neither is ideal.

So, a good rule of thumb is: **fewer groups are better**, but only as long as we maintain meaningful distinctions.

This leads to a core challenge:

What constitutes a *different* file?

How much difference justifies creating a new group?

There's no universally correct answer.

Too few groups, and unrelated files get lumped together. Too many, and many groups contain only a single file, defeating the purpose of classification.


My balanced approach is to **reassign files from singleton groups**—those with just one file.

The reassignment process stops under two conditions:

- All singleton groups have been processed, **or**
- The total number of groups has reached a minimum threshold.

The idea is simple:

A group with multiple files is more likely to represent a meaningful category.

Interestingly, many of the 2-file groups consist of files that differ only by a  suffix.

While this suggests they are versions of the same file, I chose not to merge them unless supported by concrete evidence, as we must treat each file as unique unless clearly shown otherwise.

This is where additional jurors—**SSIM**, **ORB**, and **Histogram**—join the analysis:

From the last group:

WHILE the group contains only one **file** AND the number of groups **is** above the minimum threshold:

For each earlier group:

Compare the singleton **file with** the first **3** files **in** the group:

Get SSIM score

IF ORB throws an error:

Skip this group

Get HISTOGRAM score

Compute the average of SSIM **and** HISTOGRAM

Identify the group **with** the best average score

Move the **file** to that group

Delete the now-empty original group

This ultimately leads to over 99% accuracy in the final groups.

### 1.2.4 Pros and Cons of This Approach

After extensive consideration of how best to combine the available algorithms, I believe the current solution strikes the most accurate balance.

However, there is a major drawback: **it is extremely slow**. While I knew from the outset that speed wasn't the primary focus of this challenge, the execution time is still problematic.

For the full dataset (tiers 1-4), the verdict generation process takes approximately **7 hours**, the majority of which is consumed by the deep learning algorithm. If we remove that component, the runtime drops to around **2 hours**, though we sacrifice some accuracy—particularly for edge cases.

Improving performance is a promising direction for future development.

## 1.3 The Code & Solution

This solution was developed in **Python**, as it naturally supports the image comparison algorithms I needed. The development environment was **Linux**.

---

### 1.3.1 Project Structure

SHELL

```
.
├── requirements.sh
├── solve.sh
├── ss_maker.py
├── initial_verdicts.py
├── initial_grouping.py
├── final_groupings.py
├── groups
│   ├── tier1
│   ├── tier2
│   ├── tier3
│   └── tier4
├── jurors
│   ├── hashing.py
│   ├── orb.py
│   ├── histogram.py
│   ├── deeplearn.py
│   └── ssim.py
├── screenshots
│   ├── tier1
│   ├── tier2
│   ├── tier3
│   └── tier4
├── tier1
├── tier2
├── tier3
├── tier4
└── verdicts
    ├── tier1
    │   └── hashing
    ├── tier2
    │   └── hashing
    ├── tier3
    │   └── hashing
    └── tier4
        └── hashing
```

### 1.3.2 How to Use

#### Folders:

- **groups/**: contains the final **.txt** group result files for each tier
- **jurors/**: contains the implementations of all image comparison algorithms
- **screenshots/**: stores all rendered screenshots
- **verdicts/**: contains verdict outputs
- **tier1-4/**: the dataset directories

#### Key Scripts:

- `ss_maker.py`: handles screenshot generation. I chose Playwright for rendering due to its robust, built-in functionality.
- `initial_verdicts.py`: processes screenshots and saves initial verdicts to the `verdicts/` directory (see algorithm in section 1.2.1).
- `initial_groupings.py`: creates preliminary groupings in the `groups/` directory (see section 1.2.2).
- `final_groupings.py`: determines final groupings (see section 1.2.3).

### Important

You can selectively run the scripts on specific tiers by passing the tier name as an argument.


For example: `python initial_verdicts.py tier3`

### Helper Scripts:

- `solve.sh`: runs the entire pipeline by assembling the Python scripts into one execution flow.
- `requirements.sh`: installs all necessary Python packages (note: Torch packages are quite large).

### Note:


The algorithm is **non-deterministic**, meaning the initial and final groupings may vary slightly between runs. If results are not satisfactory, consider re-running the initial and final grouping steps for the affected tier.

 if the script doesn't work manually run the group scripts. Before this go into the groups folder and delete the text files.

### PYTHON

```
python initial_groups.py tier1
python initial_groups.py tier2
python initial_groups.py tier3
python initial_groups.py tier4
```

```
python final_groupings.py tier1
python final_groupings.py tier2
python final_groupings.py tier3
python final_groupings.py tier4
```

 The screenshots have been deleted so the github upload is possible.

Verdicts and final version of groups are present.  
Before deleting verdicts remember the very high run time.