October 28, 2025
Samuel Chapuis
BDMA - Master Student
samuel.chapuis@student-cs.fr

Personal Data Vis

BDM-02 - Visual Analytics

1 Data Collection

Dataset description

This report summarizes a personal dataset extracted from my GitHub commit history. The dataset contains 459 records and 9 attributes (columns). Key columns include `repo_full_name`, `commit_day`, `commit_hour`, `message`, and `is_merge`.

What is the data about?

This dataset captures metadata about my software development activity on GitHub. Each row represents one commit event and includes the repository name, the commit date (day-level), the commit hour (UTC), whether the commit is a merge, and the raw commit message. Personally, this data reflects my work rhythm, focus across projects, and habits over time.

Column names, data types, and data size

The table below lists the columns and their inferred data types:

Column	**Dtype**
`repo_full_name`	`object`
`repo_private`	`bool`
`repo_language`	`object`
`repo_stars`	`int64`
`repo_forks`	`int64`
`commit_day`	`object`
`commit_hour`	`int64`
`message`	`object`
`is_merge`	`bool`

Motivation & Questions

I chose this dataset because I believe that my commit history can reveal interesting patterns about my overall work habits and productivity cycles.

- Am I more productive at certain times of day?
- How much does the money matters in my work (private vs public repos)?

- Are my favorite programming languages reflected in my commit activity?
- Is there any seasonality (e.g., exam periods, internships) visible in the data?
- Was my life healthier when I was in the USA (comparing local commit time between France and California)?

Data collection process

The dataset was collected using a Python script that queries the GitHub API for repositories I own or contribute to and retrieves commit metadata since 2023. The script normalizes timestamps to derive `commit_day` and `commit_hour` (UTC), identifies merge commits through parent count, and writes a deduplicated CSV. Sensitive fields (e.g., emails, access tokens) were omitted to preserve privacy.

Once collected, the dataset was cleaned and enriched through additional Python preprocessing steps. First, a temporal filter was introduced to distinguish commits made within a specific time window (January–June 2025). This binary variable, named `is_in_the_USA`, was computed using a conditional statement. This step allows later comparisons between different phases of my activity, such as before, during, or after a specific period.

Secondly, the raw `message` field from each commit was parsed into three semantic components using a custom regular-expression function called `split_message`. This function separates the text that appears:

- *before parentheses* — indicating a possible action type or prefix (e.g., “fix”, “add”, “update”);
- *between parentheses* — capturing contextual tags (e.g., affected module, file, or feature);
- *after a colon or parentheses* — retaining the descriptive part of the message.

The extracted components were assigned to three new columns (`message_type`, `message_argument`, and `message_message`), replacing the original raw message column. This transformation simplifies later text analysis and classification tasks, such as identifying recurring commit patterns or estimating coding effort.

Overall, this two-step cleaning process makes the dataset more structured and interpretable while preserving its personal and chronological character.

Next steps

The first exploratory charts already reveal interesting temporal patterns: one showing the concentration of commits by hour, another by weekday, and a third comparing repositories by activity volume.

In a future iteration, I plan to expand the analysis by:

- comparing activity across seasons or academic periods to detect productivity cycles (e.g., exams or internships);
- analyzing the difference between personal and collaborative repositories;
- classifying commits by type (*feature*, *fix*, or *documentation*) using message keywords;
- correlating repository language and commit frequency to observe shifts in focus across programming ecosystems;
- integrating quantitative measures such as lines of code added or deleted to evaluate effort rather than frequency alone;
- exploring time zones to distinguish local vs. remote work habits across projects.

These next steps would transform the dataset from a simple chronological record into a more complete reflection of my working patterns, project diversity, and long-term evolution as a developer.

Notes on ethics and privacy

Only metadata required for the analysis is kept. The dataset does not include code content or private message bodies beyond commit messages, and it can be further anonymized by hashing repository names or redacting message text if needed.

Bonus, Visisual insight with the dataset

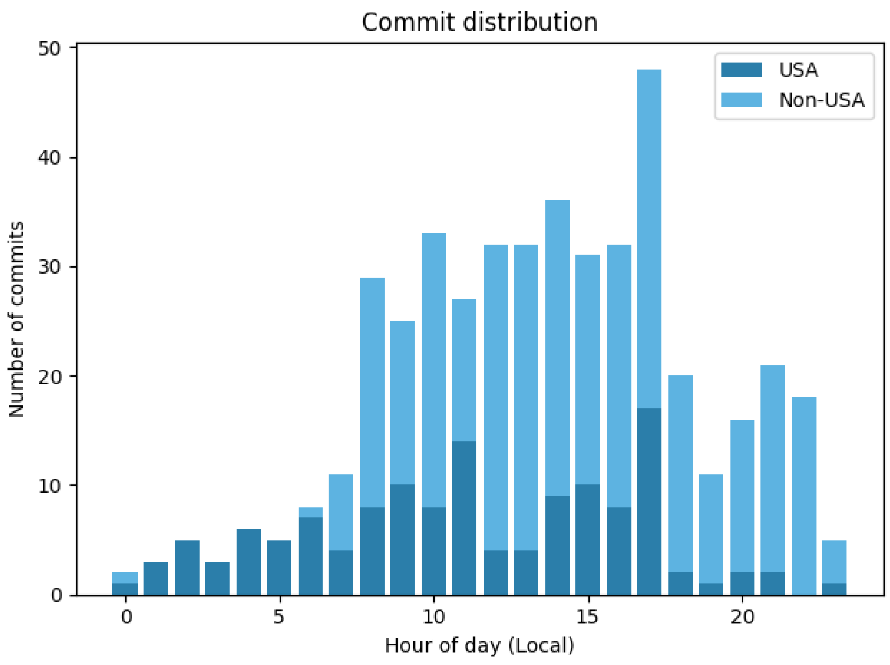


Figure 1: Local hours of commit

This graph shows the commits distribution by hour in local time, however it is possible to have 1 or 2 hours of drift because I computed the local time from the UTC hour only, without taking into account the daylight saving time. Still it clearly shows that I tend to code late at night especially when I was in the USA where there is a lot of commits after 3 PM UTC (which is 8 PM in California).

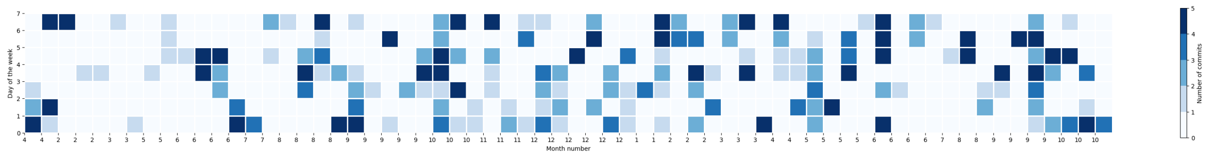


Figure 2: GitHub activity by repository

This graph is a reproduction of the GitHub contribution activity graph, showing the number of commits per day. It is possible to see a tendance of more activity over time with some peaks during rush periods like publications (may 2025). It is also possible to see a lower activity during vacation periods (june 2025 and 2024).

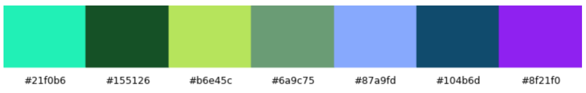
2 Choosing Colors

Categorical column

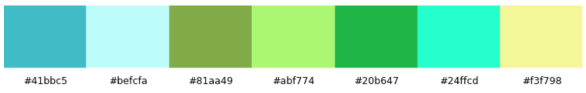
For this color-design exercise, I chose the column `repo_language`, which contains the main programming language of each repository. This variable is categorical and non-ordered, with about 5–7 dominant languages in my data. It is therefore ideal for a categorical color palette design task.

Colorgorical Palettes

Two color palettes were generated using the **Colorgorical** tool (`http://vrl-v2.cs.brown.edu/color`). Each palette was produced with 7 colors, corresponding to the top 7 languages in my dataset. I experimented with different parameter values for *Perceptual Distance*, *Pair Preference*, and *Hue Filtering*.



Palette B



Palette A

3. Palette Evaluation

The two palettes were evaluated according to four criteria: *Distinctiveness*, *Harmony*, *Semantic Fit*, and *Accessibility*. The following table summarizes my observations.

Criterion	**Palette B**	**Palette A**
Distinctiveness	The colors are clearly distinguishable: all ΔE_{2000} values exceed 20, ensuring perceptual separation. Visual inspection confirms good category discrimination.	Some colors (green–yellow) show smaller ΔE values ($\approx$10–15), making certain pairs slightly less distinct.
Harmony	Cool tones (blue–green) create a coherent, professional, and calm look consistent with tech-oriented visuals.	Warmer tones bring variety and contrast but appear less balanced overall.
Semantic Fit	Blue evokes reliability (Python), green suggests growth or development (Kotlin, Go), and yellow fits front-end languages.	Colors chosen mainly for aesthetics; weaker semantic links between hues and categories.
Accessibility	Simulated under deuteranopia and protanopia (Coblis): all hues remain distinguishable; two greens appear similar but separable.	Green-green confusion occurs under color blindness, reducing legibility.

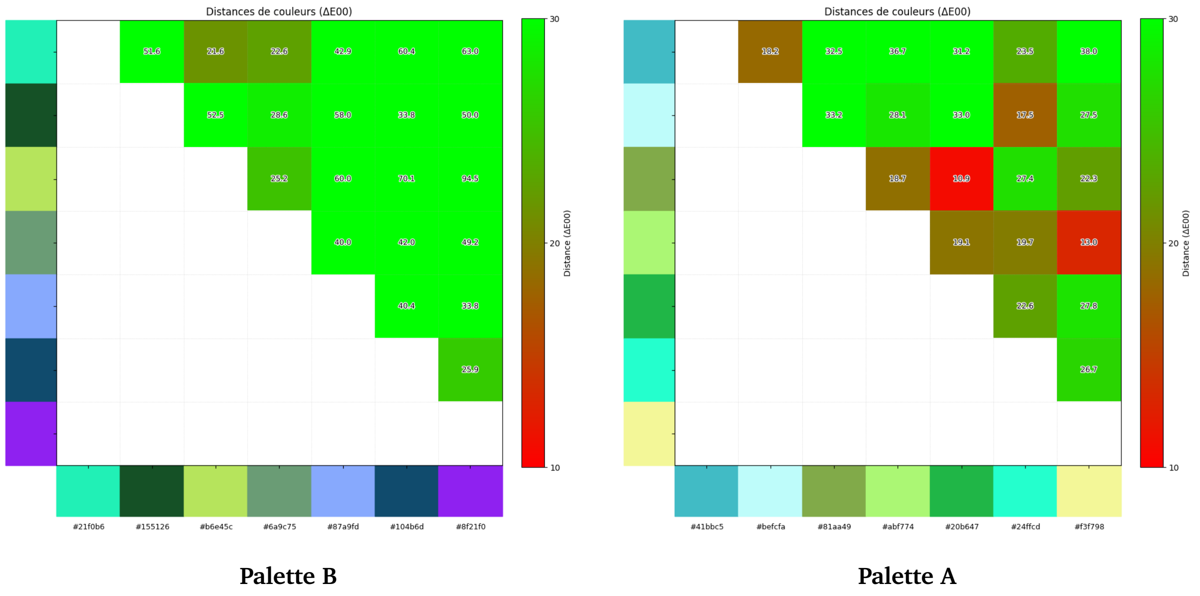


Figure 3: Confusion matrix for color

4. Applying the Chosen Palette

The selected palette (Palette B) was applied to visualize the number of commits per programming language. Each color represents a distinct language category.

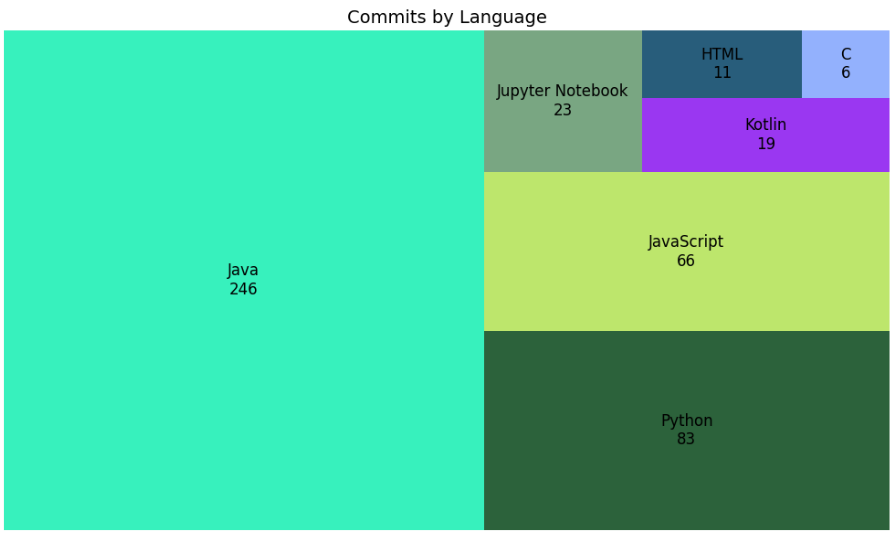


Figure 4: Commits by language using the selected categorical palette.

The chart shows the distribution of commits per programming language in my GitHub dataset Each color encodes a unique language category using the custom palette generated through Colorgorical. The palette was chosen to maximize perceptual distance while maintaining visual harmony through cool blue–green tones. Accessibility testing confirmed that color distinctions remain clear under red–green color-blind conditions. Color therefore effectively communicates categorical information in this dataset while preserving a balanced, professional aesthetic.

Reflexion

Designing the palette through Colorgorical highlighted how perceptual metrics can guide visual clarity. The tool's "perceptual distance" parameter was especially helpful in maintaining ΔE_{2000} values above 20, ensuring that all colors remain distinct even for users with mild color vision deficiencies.

The interface was intuitive, and generating multiple palettes was simple. However, the main challenge was finding the right balance between *aesthetic coherence* and *functional clarity*. Palettes with high perceptual distance often appear unbalanced or too contrasted, while those with closer hues look harmonious but reduce category distinction.

Because my data represents programming languages, I also considered *semantic fit*, for this I helped myself with the github color for the langages, where python is green, C is blue, Kotlin purple.

For accessibility, both palettes were tested using the Coblis simulator. Although Colorgorical ensures perceptual contrast, visual testing confirmed that subtle greens may still appear similar for deuteranopia users. This underlines that accessibility validation requires both computational and perceptual evaluation.

Overall, Colorgorical was highly effective for producing perceptually grounded palettes, but aesthetic and semantic adjustments still relied on human judgment. The final palette achieves a satisfying trade-off between clarity, harmony, and inclusivity, demonstrating the practical balance between data-driven color selection and design intuition.